An Aspect-Oriented Middleware Architecture for Run-time and Atomic Weaving of Distributed Aspects

Eddy Truyen,
Wouter Joosen,
Bert Lagaisse

Report CW 434, January 2006

Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

An Aspect-Oriented Middleware Architecture for Run-time and Atomic Weaving of Distributed Aspects

Eddy Truyen,
Wouter Joosen,
Bert Lagaisse

Report CW 434, January 2006

Department of Computer Science, K.U.Leuven

Abstract

There is an increasing need for dynamic and context-sensitive adaptation of distributed applications in order to dynamically cope with evolving requirements of the usage context. To support dynamic and context-sensitive adaptation, we propose an aspect-oriente architecture that weaves and unweaves aspects at run-time on demand of the usage context. The goal of this report is to explore and solve the atomicity issues that appear when distributed aspects are woven at run-time. Run-time weaving of distributed aspects, if performed without any support for atomicity, may endanger the global behavioral integrity of the application. Existing aspect-oriented middlewares lack support for atomic weaving of distributed aspects. This lack stems from the absence of appropriate internode coordination protocols at the level of the underlying runtime support of these middlewares. The contribution of this report is the design and implementation of a middleware, named Lasagne, that supports run-time weaving of distributed aspects in an atomic way. As a result, Lasagne is very well suited for coordinating cross-node and cross-layer adaptations in a distributed system. We present a detailed overview of Lasagne's run-time weaving model and the design of the Lasagne middleware.

An Aspect-Oriented Middleware Architecture for Run-time and Atomic Weaving of Distributed Aspects

Eddy Truyen, Wouter Joosen, Bert Lagaisse
Department of Computer Science
K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven, Belgium Eddy.Truyen@cs.kuleuven.ac.be

Project IWT 040116 “AspectLab”
Workpackage 3 - Deliverable d3.1

23th December 2005

Abstract

There is an increasing need for dynamic and context-sensitive adaptation of distributed applications in order to dynamically cope with evolving requirements of the usage context. To support dynamic and context-sensitive adaptation, we propose an aspect-oriente architecture that weaves and unweaves aspects at run-time on demand of the usage context. The goal of this report is to explore and solve the *atomicity* issues that appear when *distributed aspects* are woven at run-time. Run-time weaving of distributed aspects, if performed without any support for atomicity, may endanger the global behavioral integrity of the application. Existing aspect-oriented middlewares lack support for atomic weaving of distributed aspects. This lack stems from the absence of appropriate inter-node coordination protocols at the level of the underlying runtime support of these middlewares. The contribution of this report is the design and implementation of a middleware, named Lasagne, that supports run-time weaving of distributed aspects in an atomic way. As a result, Lasagne is very well suited for coordinating cross-node and cross-layer adaptations in a distributed system. We present a detailed overview of Lasagne's run-time weaving model and the design of the Lasagne middleware.

Contents

1 Introduction

The environments in which distributed software applications must execute have become very dynamic and heterogeneous. As a result, software must be dynamically adapted or even be able to adapt itself. This is for instance especially the case in ubiquitous computing environments and environments in which the availability of resources is variable and unanticipated. The types of adaptations that should be supported are very broad. Many of these adaptations require intrusive modification of multiple classes and components if one has to rely purely on state-of-the-art software development technologies - i.e. object-oriented and component based software engineering.

Aspect-oriented software development (AOSD) tackles this shortcoming by focusing on the systematic identification, modularization, representation and composition of (often non-functional) concerns – requirements – throughout the entire software development process. The core concept in AOSD is an aspect: a coherent entity that addresses one specific concern and that has the properties of a module that can be changed independently of other modules. At this stage it is intuitively clear that AOSD - when successfully supported - can simplify adaptations that emerge from changing concerns/requirements. This observation will be clarified in a motivation section (section 2). Based on this view, we consider the dynamic adaptation process of a distributed software application as a continuous process in which aspects are woven to (and unwoven from) the distributed application at run-time in order to accommodate evolving requirements.

However, many - if not most - non-functional concerns in distributed applications are distributed aspects. A distributed aspect is an aspect of which the behavior is woven (into multiple application components that are deployed) on different computer nodes of the distributed system. Aspect-oriented middleware (e.g. [38, 14, 41]) (AOM) supports the creation, deployment and execution of aspects for a distributed environment.

Weaving or unweaving of a distributed aspect at run-time implies adding (or removing) mutually dependent functionalities at different nodes of a distributed system. This may obviously endanger the global behavioral integrity of the software system. A coordination mechanism is required to ensure the behavioral integrity of the distributed application while the weaving is in progress. In fact run-time (un)weaving must be performed *atomically* from the perspective of the unaffected parts of the distributed application. Atomicity means that the mutually dependent functionalities must be added or removed in an all or nothing fashion.

Existing aspect-oriented middleware lacks appropriate support for atomic weaving of distributed aspects. While some systems such as Prose[40] provide support for atomic weaving of aspects within the boundaries of a single computer node, there exists to our knowledge no system that supports atomic weaving of distributed aspects. The main contribution in this report is that we define a new runtime support system (Lasagne middleware) that performs the atomic weaving in such a way that it will not require changes to the existing AO programming models. The design of the Lasagne middleware is essentially based on two guiding principles. First, weaving of distributed aspects is performed in two phases: a *deployment* phase and an *activation* phase. Thus the problem of atomicity becomes easier to solve. Secondly, we create a solution that pre-

serves the essence of an aspect-oriented execution environment. Therefore the atomic weaving mechanism must enable a coordinated manipulation of control flow across the boundaries of different computer nodes. We will argue that the application of these two key principles provides the basis for an elegant solution that supports run-time and atomic weaving of distributed aspects.

The rest of this report is structured as follows. In section 2 we articulate the scope and problem statement of this report and motivate why support for run-time aspect weaving will become an important feature of the future middleware. Section 3 describes the runtime weaving model of the Lasagne middleware. Section 4 presents the architecture and design of the Lasagne middleware. We further illustrate the value of our solution in section 5 by discussing the implementation, the applicability, performance trade-offs of Lasagne, and integration of Lasagne on top of standard middleware. We describe related work in section 6 and we conclude in section 7.

This report is an **AspectLab**[1] deliverable.

2 Scope and motivation

This section first illustrates the range of adaptations that Lasagne aims to support. Subsequently the problem statement and contribution of this report is revisited into more detail.

2.1 Scope of supported adaptations

Distributed software applications must be dynamically adapted and be able to self-adapt. We give an example to illustrate the versatility of the spectrum of adaptations. Consider an e-finance application that initially supports a limited service of delivering statements of the customer's checking and savings accounts[2]. An initial level of security for this straightforward and minimal application could deliver confidentiality by sending the messages over an encrypted network layer - the encryption or decryption facilities will obviously be implemented in the lower layer of the middleware architecture. Security risks will increase when online transactions are supported by an extended set of services. This will be even more the case when private banking services are added, for instance to support the e-trading of securities. The security measures that will be added to the application are no longer embedded in the middleware, in fact specific security measures will be added to the application code, to use advanced authorization for financial transactions, and to add a non-repudiation feature for specific types of transactions (typically with a higher risk level). Both the financial institution and the clients may wish to have a solid audit trail that effectively enables to proof that certain transactions have been ordered by the client and executed by the bank. As the service becomes more successful and the state-of-the-art of security technologies evolves, the bank may have to adapt the cryptographic building blocks that produce signatures (for instance when

[1] http://ssel.vub.ac.be/aspectlab/

[2] We will sketch the evolution of this minimal service towards a fully fledged e-banking service. This illustration is based on our experience with real life experience with that type of applications - in that sense, the sketched evolution matches the real history of an application. More info can be found in [26]

certain types of hashing functions become too weak to avoid the algorithms to be broken). Also, authentication tokens do evolve; sometimes different user types may choose for different tokens (smart cards that store X.509 certificates, vs. popular generators of one-time passwords, etc.). This picture includes a lot of extensions and alternatives for an evolving security solution. It therefore illustrates that adaptations can be

- *embedded in the middleware or tightly coupled to application* code (E.g. IP level encryption vs. application level authorization for transactions);
- *extending* the existing functionality (e.g. adding signature verification through interaction with the CRL - certificate revocation list - of a public key infrastructure), or *replacing* certain parts (e.g. when updating the signature generation techniques);
- *context sensitive*: user type and possibly the type of application functionality that is requested may determine the set (or one specific instance) of authentication techniques that are required.
- *anticipated or unanticipated*: for instance, the extension of non-repudiation techniques with an audit trail functionality may be triggered by regulations hat have or have not been expected by the designers of the system. Similarly, replacement of cryptographic functionality may be anticipated or not.

We clearly envisage a broad spectrum of adaptations. The common denominator for all these adaptations is that they are all behavioral, i.e. the adaptations require functional extensions and/or replacements. We do not address the management of globally consistent state neither do we focus on adaptations that are limited to the modification of configuration parameters (e.g. QoS parameters). We believe that the concept of aspects is well suited for modelling entities that are subject to complex behavioral adaptations.

Most importantly, the need for adaptations occurs not only at load-time but also at run-time. For each non-functional requirement, multiple alternative algorithms exist and each algorithm can be implemented as a separate aspect. Which solution is to be applied cannot be determined in advance. For example, when a drop of networking resources in the operating system layer occurs, many non-functional aspects in the system may have to be replaced with an alternative aspect that reduce the use of networking resources (at the price of some quality levels). Run-time weaving is required as this kind of decisions can clearly only be taken at run-time.

2.2 Distributed aspects and atomic weaving

The main contribution of this report is a solution for the atomicity problems that appear when distributed aspects are woven at run-time. The remainder of this motivation section first introduces the terminology used in this report. Then the atomic weaving problem is studied in detail. Finally, the contribution of this report with respect to existing aspect-oriented middleware is elaborated.

Terminology. Nowadays there is a lot of research on applying AOSD (Aspect-Oriented Software Development) to the development of distributed software applications: AOSD is typically applied across the full software lifecycle to achieve

a better modularization of software entities that address either functional or non-functional requirements. Three core concepts in AOSD are *concerns*, *aspects* and *weaving*. In the context of this report, it is sufficient to introduce these concepts at the level of AOP (Aspect-Oriented Programming)[23].

Concerns are similar to requirements in a broad sense of the word, ranging from high-level requirements that are articulated in an early stage of the software project[3], to additional - often detailed - requirements that are generated when performing detailed design and implementation[4]. At the programming level, an aspect is a modular unit that implements a concern. An aspect definition contains (a) behavior (code that must be executed) which is called *advice* and (b) a specification that expresses when, where and how to invoke the advice; this specification is called a *pointcut*. A pointcut is conceptually defined as a predicate that evaluates over *join points*. A join point is a well-defined place in the structure or execution flow of a program where additional behavior can be attached. Finally, *weaving* is the process of composing core functionality modules (typically application components) with aspects, thereby yielding a working system.

Our work targets an *aspect-component model* that supports aspect-based composition such that aspects and application components are independent from each other. Aspects therefore have to respect the well-defined interfaces of the application components to avoid undocumented coupling between aspects and application components. As such, we only target pointcuts that evaluate over *non-invasive* join points: join points which are visible in the interface of the components (typically: the declared required and provided operations and the invocation/execution of these operations). Furthermore, in order to keep the behavior of an aspect reusable, pointcuts are specified separately in an *aspect binding*. Consequently, behavioral adaptations can be represented by aspects that have the properties of components, enabling independent reuse.

Atomic weaving. Run-time weaving or unweaving of distributed aspects may endanger the global behavioral integrity of the system. A pedagogical illustration can be based on a simple example of encryption/decryption. Suppose a distributed producer-consumer application, in which a producer object produces data items that are sent over the network to multiple consumer objects that process the data. Now suppose that at a particular point of time, at the node of the producer the need for encrypting the messages is detected due to environmental changes[5]. As such, the advice for encrypting messages must be dynamically woven into the producer object. It is clear that this single, local weaving is not enough because the encryption advice depends on the associated decryption advice that is to be composed at the nodes of the consumers. Encryption and decryption must therefore be woven in an atomic fashion to preserve behavioral integrity. Finally notice that we have not yet discussed aspect replacement. This is a reconfiguration operation that dynamically replaces a specific aspect with another aspect by unweaving the former and weaving the

[3]For instance, "The application should ensure confidentiality when information is exchanged between two parties.".

[4]For instance, "decrypted messages should never be cached.".

[5]This can for instance be triggered by an intrusion detection system - techniques for detecting environmental changes and correlating them into adaptation decisions are obviously out of the scope of this report.

latter. Obviously, such a runtime replacement must be performed in an atomic manner as well.

Appropriate coordination mechanisms for atomic weaving, unweaving and replacement are non-trivial in a distributed system. For example, in the encryption/decryption example, atomic weaving does not mean that encryption and decryption advices must be injected at exactly the same point of time. At the point of time when the encryption advice is executed for the first time at the producer node, there may still be non-encrypted messages circulating somewhere in the network that still have to be processed by consumer objects. If an attempt to decrypt a non-encrypted message will result in an error, then the decryption advice at the consumer's nodes cannot be activated at the same time as the encryption advice at the producer's node. This atomicity problem becomes even worse when the need and types of encryption changes frequently: this is typically the case when context-sensitive adaptations have to be supported.

Contribution of the report. A lot of aspect-oriented middleware platforms have by now built-in support for run-time weaving but without covering atomic weaving for distributed adaptations. Yet many non-functional concerns in distributed applications are distributed aspects. It is important to stress that we observe this lack from the perspective of the runtime support, and not from the perspective of the programming model[6]. The reason for this is that state-of-the-art systems lack a protocol that coordinates the runtime interpretations of the pointcuts *across the different computer nodes* of the distributed system. We therefore aim for a runtime support system that performs the atomic weaving without changing existing join models.

We aim for a generic solution that can cover a broad spectrum of behavioral adaptations as introduced above. The assumed non-invasive join point model gives considerably more safety to the process of runtime weaving as compared to invasive join point models. Finally, we want to ensure that the atomic weaving support does not yield any service disruption.

3 A model of run-time and atomic aspect weaving

We argue that the underlying run-time weaving model of existing aspect-oriented middlewares are not fit to be enhanced with support for atomic weaving of distributed aspects, because such enhancements would require very complex inter-node coordination protocols. Consequently, we tackle this complexity of the atomic weaving problem at the place where it is born: at the level of the run-time weaving model itself. As will be shown, this approach allows us to keep the required inter-node coordination protocols quite simple.

Lasagne is based on a run-time weaving model that is divided into two phases: first, aspects are deployed into the distributed application. Once de-

[6] From the perspective of the programming model, support for distributed aspects is mainly concerned with an appropriate join point model for deploying and composing advices across multiple computer nodes in a system. Some AOM such as JAC[38] supports pointcuts that refer not only to execution points within the application logic, but also to the computer nodes in the distributed system. This is a straightforward way of modeling distributed aspects and we assume for the scope of this report that this is sufficient for most cases.

ployed, aspects can be dynamically activated on a per message basis. We refer to the former mechanism as *deployment* and to the latter as *activation*. The above separation between deployment and activation allows us to apply the 'divide and conquer' principle to the complex problem of atomic weaving, and thus allows us to deal with this complexity by means of functional decomposition. More specifically, we divide the atomic weaving problem into three sub-problems that are easier to solve on their own.

The remainder of this section is structured as follows. The main use cases of the run-time weaving model are presented in section 3.1. Then, the models of deployment and activation are respectively presented in sections 3.2 and 3.3. The approach behind these models is illustrated in the context of the producer-consumer application and the encryption-decryption aspect. Subsequently, section 3.4 sets out our functional decomposition approach to the atomic weaving problem.

3.1 Use cases

We describe a high-level overview of deployment and activation from the perspective of what are the main human actors involved and the corresponding use cases in which these human actors interact with the Lasagne middleware.

Deployment is performed by a human operator who is trusted within the organization and system-infrastructure boundaries of the distributed application. In the rest of this report, we refer to this human operator as the *application deployer*. The application deployer can see fit to deploy, un-deploy or replace aspects, at load-time or run-time. How an aspect must be bound with a particular application, is specified by the application deployer in an aspect binding. This aspect binding basically encapsulates one or more pointcuts for the advices of the aspect.

In the spirit of context-sensitive adaptation, activation (or deactivation) of an aspect is requested by a *contextual actor*. Contextual actors are software programs or devices, that often are based on interaction with human operators such as end users, system administrators, and domain experts (e.g. security experts who decide on the policies for access control). We describe two simple pedagogical examples of contextual actors and how contextual actors are able to request aspect-based adaptations to be performed.

One example of a contextual actor is a client application or integrated device that uses the services of the aforementioned e-finance application. The client uses activation as a mechanism to customize the service to its individual preferences. Simply by tagging so called *aspect identifiers* to its client requests, the client is able to activate aspects that contribute functional or non-functional extensions to the e-finance application that are desired by that client. Another example of a contextual actor, is a resource monitor system component, that monitors the availability of resources in the underlying distributed system infrastructure of the e-finance application. The resource monitor uses activation as a mechanism to adapt the e-finance application to undesired fluctuations in the resource availability or to system overload. For example, during system overload, the resource monitor could deactivate a resource-intensive aspect, and instead activate a cheaper variant of the aspect by tagging all client requests, that arrive at the server-side of the e-finance application, with the aspect identifier of the cheaper aspect.

3.2 Deployment

Deployment of an aspect assumes the existence of an aspect binding that specifies how the aspect must be bound with the distributed application. For example, Figure 3.2 illustrates how the `encrypt()` and `decrypt()` advices of the `encryption-decryption` aspect are bound with the Producer and Consumer components. For instance pointcut `messageSend()` binds the `encrypt()` advice with a call of the `consume()` operation from within any `Producer` object that is located on host `media`.

```
aspectspecification encryption-decryption  {
  classPath = "....";
  around advice encrypt(byte[] data);
  around advice decrypt(byte[] data);}
}

aspectbinding encryption-decryption {
  pointcut messageSend(data): call Consumer.consume(byte[]
    data) within Producer.produce() at media.cs.kuleuven.be;

  pointcut messageReceipt(data): execution Consumer.consume(byte[]
    data) at multi.cs.kuleuven.be;

  connect {
    messageSend(byte[])<- encrypt(byte[]);
    messageReceipt(byte[]) <- decrypt(byte[]);
 }
}
```

Figure 1: The specification of the encryption-decryption aspect and its binding to the producer-consumer application is shown. The aspect specification describes the signatures of the advice methods of the aspect and contains the location where the code can be downloaded. The aspect binding encapsulates one or more pointcuts and connects these pointcuts to the advices.

Deployment implies that the Lasagne middleware interprets the pointcuts to effectuate the binding of the aspect with the application. An important characteristic of Lasagne is that deployment does not directly inject the aspect code into the execution of the application. Instead, deploying an aspect, and therefore interpreting the associated aspect binding, affects a representation of the application at the metalevel. This representation is called *deployment metadata*. Deployment metadata describes a lower-level configuration of the application in terms of which advices of aspects must be woven at which join points of base components[7]. Deployment metadata is thus a lower-level representation of an aspect binding, such that the runtime weaver can interpret it more efficiently.

The deployment model of Lasagne is thus characterized as a continuous and incremental evolution of deployment metadata. Figure 2 gives an overview of the deployment model. The *application deployer* is responsible for specifying

[7]As argued in section 2 we only want to weave at non-invasive join points which, in a component-based setting, either correspond to an execution of a provided operation or a call to a required operation

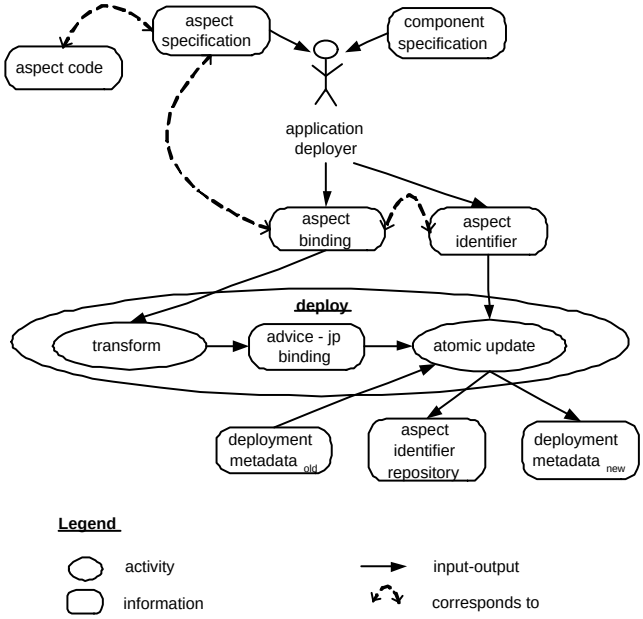

Figure 2: Deployment data flow. The application deployer gives to the deployment tool an aspect identifier and an aspect binding. The deployment tool first transforms the aspect binding into lower-level advice-to-joinpoints bindings and then atomically updates the deployment metadata with these bindings and the corresponding aspect identifier. The aspect identifier is stored in public accessible aspect identifier repository.

an aspect binding, which encapsulates one or more pointcuts. Hereby (s)he assumes that specifications of aspects and base components are available. These specifications describe the signatures of aspects' advices and components' provided and required interfaces, and the location from which the aspects' and components' code can be downloaded. Once the aspect binding is specified, the application deployer uses the Deployment API of the middleware to deploy the aspect. For example, the encryption-decryption aspect is deployed as follows:

```
lasagne.core.Deployment.deploy(
    "encryption-decryption",
    new AspectIdentifier("MessageConfidentiality"));
```

As shown in figure 2, the process of deployment is a consecutive process of transforming the aspect binding into lower-level updates of deployment metadata, so called *advice-to-joinpoint bindings*, and atomically applying these updates to the deployment metadata. An advice-to-joinpoint binding simply captures information that a specific advice must be executed at a specific join point. It thus conceptually corresponds with a tuple of 4 members: the signature of the advice method, the aspect identifier associated with that advice method, the (non-invasive) join point where the advice must be executed, and the order of execution (relative to other advices at that join point).

A peculiarity of the deployment model, as shown in Figure 2, is that the application deployer must also associate a unique name to the aspect. This

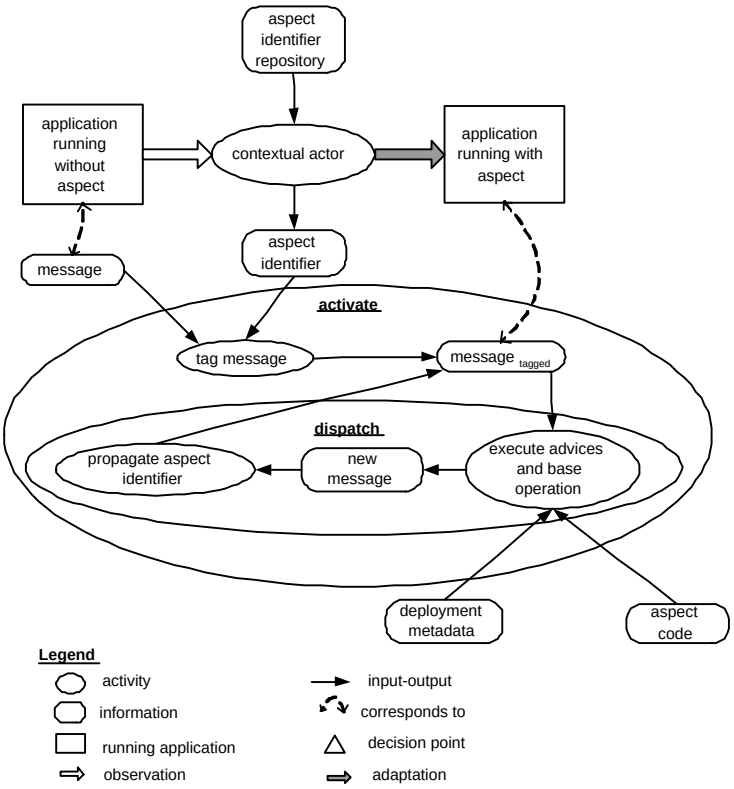

Figure 3: Activation data flow. The contextual actor observes directly or indirectly the behavior of the application and decides at a certain point in time to adapt the application by activating one or more aspects. To activate an aspect, the contextual actor dynamically tags aspect identifiers to messages which are on their way to the application. When processing a message, the runtime weaver consults the tagged aspect identifiers in order to know which advices to execute. If some aspect is not yet injected into the application, the runtime weaver first loads the aspect's code, and updates its internal data structures, based on the deployment metadata. New messages, resulting from the original message's execution, are tagged with the same set of aspect identifiers by the runtime weaver. As such, aspect identifiers propagate.

name, referred to as *aspect identifier*, is supposed to associate with the aspect a meaning that makes sense to contextual actors. In other words, the aspect identifier is part of the problem domain of what adaptations the contextual actor aims to achieve in a particular deployment environment. The aspect identifier may describe for instance a capability, feature or quality-of-service-level that the aspect contributes. For example, in the context of the producer-consumer application, the application deployer maps the encryption-decryption aspect to the aspect identifier "MessageConfidentiality" to indicate that this aspect offers a specific security property when transmitting messages. The aspect identifier is for future reference on behalf of contextual actors: they need it in order to activate the aspect.

Lasagne does not prescribe a specific way for distributing aspect identifiers to contextual actors. There are obviously different ways of achieving this goal and we do not subscribe exclusively to one of them. We simply assume that aspect identifiers are stored in some kind of *aspect identifier repository*. In the most simple way, the deployed aspect identifiers are made available in a public directory service and, optionally, they are associated with additional documentation about the behavior and quality of service provided by the aspect.

3.3 Activation

It is important to understand that deployment of an aspect does not involve "turning the switch" that effectively integrates the corresponding advices in execution space. Instead, turning the switch is under control of flow-dependent activation that allows to initiate the execution of aspects by tagging aspect identifiers to messages.

In the spirit of context-sensitive adaptation, activation (or deactivation) of an aspect is requested by a contextual actor. To support concurrent customization by simultaneous contexts, activation of an aspect will only affect the current execution context (i.e. the current message flow or invocation path in which the contextual actor is involved) and will not interfere with other ongoing flows of messages. It is therefore also called*flow-dependent* activation.

Figure 3 gives an overview of the activation model. In order to request the activation of an aspect, the contextual actors must know the corresponding aspect identifier. To request the activation of an aspect at a certain execution point, a contextual actor has to intercept the application-level message that corresponds with this execution point and tag it with the desired aspect identifier.

Of course, the contextual actor can tag multiple aspect identifiers to a message. Multiple aspects are thus combined and activated for the same message. Once the aspect identifiers are tagged with the message, they *propagate* with the invocation path of messages, succeeding the original message. As such, the aspect activation logic travels (as a set of aspect identifiers) with the message flow.

The runtime weaver of the Lasagne middleware consults the tagged aspect identifiers in order to know which advices to execute for a given message. When an aspect is activated for the first time, and therefore, the code of the aspect is not yet injected into the application, the runtime weaver has to consult the deployment metadata in order to know which advice must be woven at a particular join point.

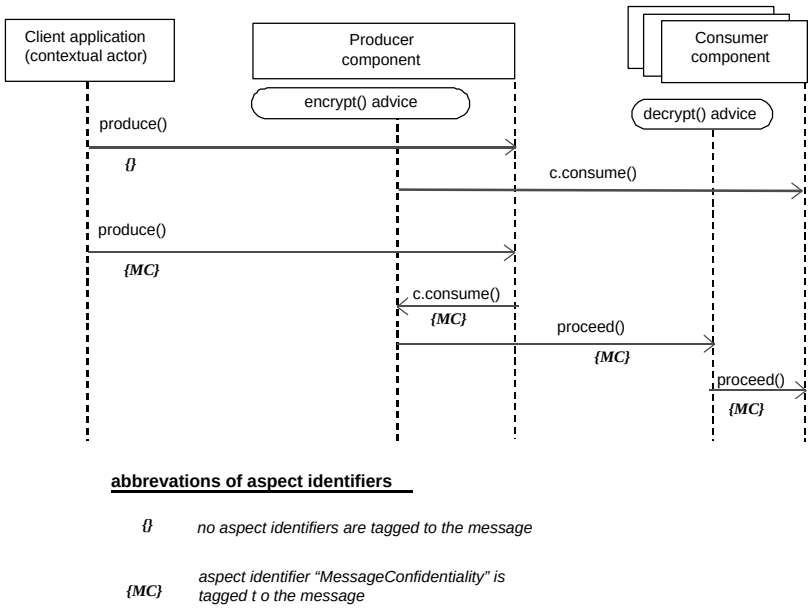

Figure 4: Flow-dependent activation of aspects

Figure 4 presents a message sequence diagram that illustrates the flow-dependent activation of the encryption-decryption aspect in the producer-consumer example. Here a client application, using the service of the producer-consumer application, plays the role of contextual actor. The figure illustrates that once the client application dynamically decides to tag the aspect identifier "Message-Confidentiality" to one of its client requests, this decision is propagated with the subsequent message flow such that the advices of the corresponding aspect are all dynamically woven and invoked at the appropriate join points.

3.4 Atomic weaving

The separation between deployment and activation enables us to decompose the problem of atomic weaving into three sub-problems:

Complete and correct deployment During deployment, Lasagne ensures that an update to the deployment metadata, due to aspect binding interpretation, is correctly and completely distributed across all computer nodes on top of which the distributed application executes.

Consistency in the activation of aspects During flow-dependent activation, Lasagne guarantees the consistency in the activation of aspects: once a combination of aspects has been activated for a specific message flow by tagging of aspect identifiers, this decision is consistently coordinated all over the subsequent execution of the message flow by means of propagating the aspect identifiers with the message flow. The tagging and propagation policies must however also succumb to certain constraints in order to fully ensure consistency within the whole distributed application. These constraints will be explained in detail in section 4.2.2.

Coordination between deployment and activation An important correlation between deployment and activation is that Lasagne only permits the activation of an aspect after the aspect is completely and correctly deployed. Thus, a successful completion of the deployment protocol is a prerequisite for initiating the activation of the aspect. The underlying rationale is of course that when the binding of an aspect is incompletely represented in deployment metadata, the execution of the aspect will also be incomplete, and thus result in inconsistencies in the system-wide behavior of the application. This phased transition from deployment and activation is realized by means of a coordination protocol.

The next section describes the design of the overall system architecture of the Lasagne middleware and the design of the runtime weaver in order to achieve the above three requirements.

4 Architecture and design of the Lasagne middleware

In this section we present the architecture and design of the Lasagne middleware. First we give an overview of the distributed systems architecture of Lasagne. Then we elaborate upon the approach taken to achieve atomic weaving. Finally we focus on the design of Lasagne's runtime system which is the core of the Lasagne middleware.

4.1 Distributed system architecture

Figure 5 gives an overview of the distributed system architecture of the Lasagne middleware. It is conceptually divided into a *runtime system*, which is the core of the middleware, and an underlying *deployment architecture*.

This section first discusses the architectural styles of distributed applications and underlying platform (operating system, programming language) that the Lasagne middleware is compatible with. Subsequently the architecture of the runtime system and the deployment architecture are presented by means of projecting the **deploy** and **activate** activities of the Lasagne's weaving model (see figures 2 and 3) onto the different components of these architectures.

Assumed architectural style of applications and platform. The Lasagne middleware is compatible with a broad range of application architectures and platform architectures. There are only a few assumptions.

With respect to the application architecture, it is assumed that the components of the distributed application are grouped in several operating system processes that are possibly allocated on different computing nodes of the distributed system. Furthermore, we assume it is possible to draw a clear and stable virtual boundary around the deployment space of the distributed application (see figure 5 for having an intuitive idea of what is meant with this virtual boundary). The concept of a virtual boundary conveys the image that the interior of the boundary, provides a service to something that is exterior. From a deployment perspective, the concept of a boundary implies a closed world assumption: aspects that are woven into the interior area, are not supposed to

be woven into the (unknown) exterior. As will be discussed in section 4.2.2, the concept of a virtual boundary is introduced for ensuring consistency in the activation of aspects: Lasagne only guarantees consistent activation for aspects that do not crosscut through the virtual boundary. In practice, this all means that Lasagne is well suited for client-server architectures where an exterior client process (e.g. the client of the e-finance application) or any intermediate proxies (e.g. the resource monitor) require dynamic adaptations of the interior server processes (the e-finance application itself). On the other side, Lasagne is less suited for peer-to-peer architectures, or applications of ad-hoc networks where the boundary between the exterior and interior cannot clearly be drawn.

With respect to the platform, we make no assumptions about the internal structure of a process (for example, a process may be multi-threaded or not, may be divided in distinct security domains or not, etc.). We do assume that intra-process as well as inter-process communication between components is based on messages. Finally, we assume that within each process there is a separate instance of the Lasagne runtime system running. For example, in Java-based applications[3], there would be a single instance of the runtime system per Java virtual machine.

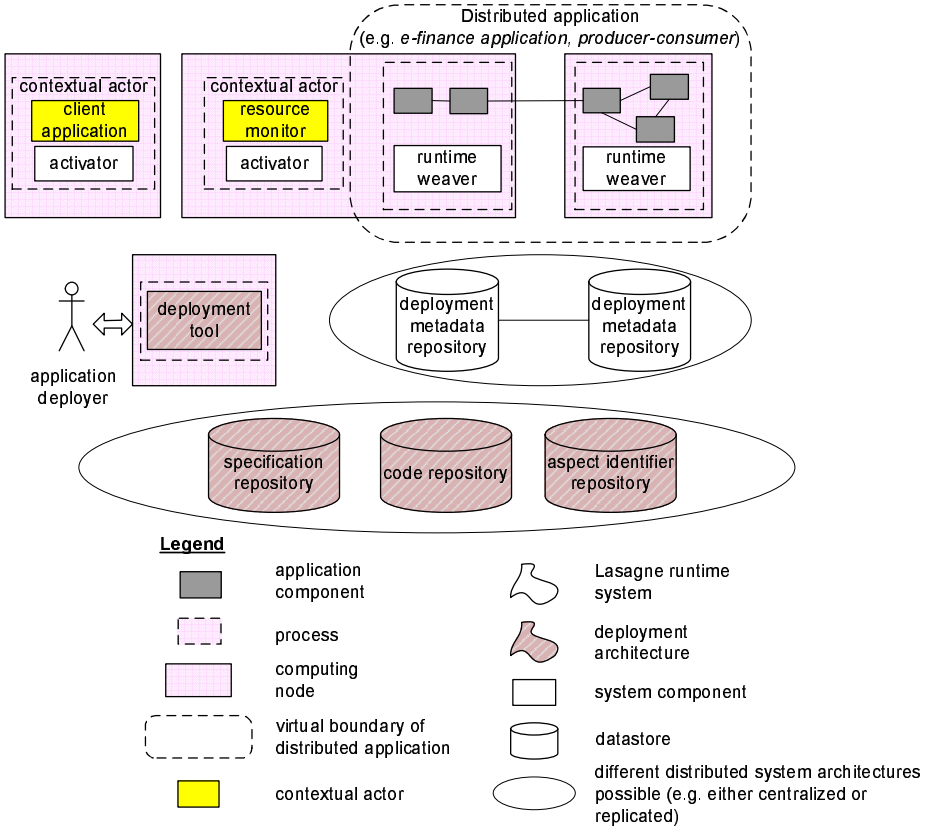

Figure 5: Distributed system architecture of the Lasagne middleware's runtime system and the surrounding deployment architecture.

Runtime system. The runtime system essentially consists of a *runtime weaver*, an *activator* and a *deployment metadata repository*.

The runtime weaver is the central component of the runtime system (there is a separate runtime weaver instance per process). Its main responsibility is to support the **dispatch** activity, as described in figure 3. Since aspect activation in Lasagne is message-driven, it is not strange to consider the Lasagne runtime weaver as a dispatch mechanism. Message dispatch is two-dimensional in Lasagne: not only the receiver field of messages but also the tagged aspect identifiers control where messages gets dispatched to. Similar to standards middleware, messages in Lasagne have an associated *invocation context*, in which properties about the ongoing computation can be stored. This invocation context is also used to tag aspect identifiers to messages. Propagation of the aspect identifiers with the message flow is simply achieved by copying the invocation context from one message to another.

The *activator* component is a plug-in for contextual actors that provides support for the **tag message** activity in figure 3. To support this, it provides an API that allows contextual actors to set aspect identifiers to the invocation context of messages. Notice further that aspect identifiers can be incrementally tagged to a flow of messages by different contextual actors.

Deployment metadata is stored in a *deployment metadata repository*. This repository may be centralized or replicated, depending on how local one wants to keep the deployment metadata to the runtime weaver instances, and depending on the size and topology of the distributed system. Clearly, there is a range of alternative system architectures possible. At one end of this range there is a single deployment repository instance for the whole distributed application; at the other end, there may be a replica at every computing node of the distributed application.

Deployment architecture. The deployment architecture consists of a *deployment tool* and a *data tier*.

The *deployment tool* component supports the **deploy** activity, as described in 2. It thus offers functionality for transforming an aspect binding into advice-to-joinpoint bindings and for atomically updating the deployment metadata repository with these bindings. This deployment tool also assumes there is a programming model or language construct for specifying aspect bindings.

The data tier conceptually consists of three kinds of data stores. The most important one is the *aspect identifier repository*. As stated in section 3.2, when deploying a new aspect, the application deployer is expected to create an associated aspect identifier. Once the aspect is completely and correctly deployed, the deployment tool will update the aspect identifier repository. Contextual actors can then query or browse the repository for the aspect identifier. The second type of datastore is the *specification repository*. It acts as a back-end for the deployment tool by storing aspect bindings, and aspect and component specifications. Finally, the *code repository* stores all the executable code artifacts of aspects and components. The Lasagne runtime weaver will eventually consult the code repository when the aspects' code must be loaded into memory. Similar to the deployment metadata repositories, there are several alternative architectures possible for the data tier.

4.2 Support for atomic weaving

This section presents the detailed approach of how the Lasagne middleware supports atomic weaving. As stated in section 3.4, we have divided this complex problem into three subproblems. For each of these problems, we describe our approach to solve it.

4.2.1 Correct and complete deployment

In order to enable correct and complete deployment of an aspect, atomicity and concurrency control must be provided at the level of updates to deployment metadata. The best way to implement this is by means of a conventional transaction protocol. Hence, we refer to the process of updating the deployment metadata of a distributed application as a *deployment transaction*. How to implement the deployment transaction depends on whether the deployment metadata repository is replicated or not:

1. A centralized repository is simply implemented as a transactional file service or database that employs a two-phase locking protocol to serialize concurrent updates.
2. In a decentralized approach deployment metadata is replicated at mulitple nodes of the distributed application. Here, an update of deployment metadata must be coordinated by means of a two-phase commit protocol. Concurrency control within each single repository is again provided by implementing it as a transactional file service. The advantage of the decentralized approach is that the Lasagne runtime weaver can consult the deployment metadata locally. This advantage may outweigh the additional costs of the two-phase commit protocol because, as will be seen below, the runtime weaver consults the deployment metadata quite often.

4.2.2 Consistency in the activation of aspects

As already explained in section 3.3, the main mechanism behind ensuring consistency is the propagation of aspect identifiers with the message flow. As a result, within a distributed application, the runtime weaver instances in different processes are implicitly coordinated by the tagged aspect identifiers. In other words, when two runtime weaver instances are requested to dispatch messages which logically belong to the same message flow, then these messages will be consistently dispatched to the same combination of aspects.

A second problem related to activation consistency has to do with the aforementioned concept of the virtual boundary around the distributed application: aspect identifiers may only be tagged to messages that cross the virtual boundary of the distributed application. This constraint is necessary to ensure that aspects are activated in an all-or-nothing fashion across the entire area inside this boundary.

There are two ways to enforce this constraint: by rigorous design at the level of the application architecture, or by run-time checks at the level of the middleware. At the level of the application architecture, one can enforce the constraint manually during architectural design by placing contextual actors at the rights places of the application architecture. For example, every application that provides a service to client applications has some process that acts as

a front-end. It is clear that tagging of aspect identifiers may only occur on messages that are exchanged between the client applications and this front-end (either at the client-side or at the server-side).

The second approach is to make the virtual boundary of a distributed application computable such that underlying middleware infrastructure can automatically detect when messages cross and do not cross that boundary. One way to achieve this, is introducing the concept of a *base identifier* which is specializing the concept of aspect identifier. A base identifier is a name that identifies the components that are inside the boundary of the distributed application. For example, the components of the e-finance application could all be tagged with the base identifier"E-Finance", whereas the client application is tagged with the base identifier "Client". Then, a message crosses the boundary when the sender and receiver component are associated to different base identifiers. Consequently, the constraint can automatically be checked by the activator component if the base identifiers are marshalled into the sender and receiver fields of the wired messages.

4.2.3 Coordination between deployment and activation

Having divided to conquer, one must reunite to rule[19]. Indeed, having separated weaving into a **deploy** and an **activate** activity to cope with the complexity of atomic weaving, one must bring these two activities into synchrony to effectively gain the desired atomicity property. To achieve this synchrony, a coordination protocol must be defined between the different system components of the Lasagne middleware.

The conventional two-phase distributed transaction protocol for updating deployment metadata and the dedicated runtime support for achieving activation consistency allows us to keep the required coordination protocol simple.

Three different coordination protocols are actually required, respectively for weaving, unweaving and replacement of aspects at run-time. These coordination protocols enable not only run-time activation but also run-time deployment, thereby enabling unanticipated adaptation. The remainder of this section presents the details of the coordination protocol for aspect weaving and unweaving. The details of the coordination protocol for aspect replacement can be found in the appendix of this report.

Coordination protocol for aspect weaving. Figure 6 illustrates the coordination protocol for aspect weaving by means of a concrete adaptation scenario. Before explaining the coordination protocol, let us first present the adaptation scenario.

The scenario is about unanticipated adaptation. A specific client application, which is already using the services of the distributed application, desires a feature that is not yet deployed in the application. To accommodate this request, the application deployer should deploy at run-time a new aspect that meets the feature as required by the client. We assume there is an off-line meeting about the requirements of the feature between the stakeholders of the client application and the distributd application. After that, the application deployer seeks to realize the feature by means of an suitable aspect implementation and specifies the binding of the aspect with the distributed application by means of an aspect binding. Once these steps are taken, the application deployers

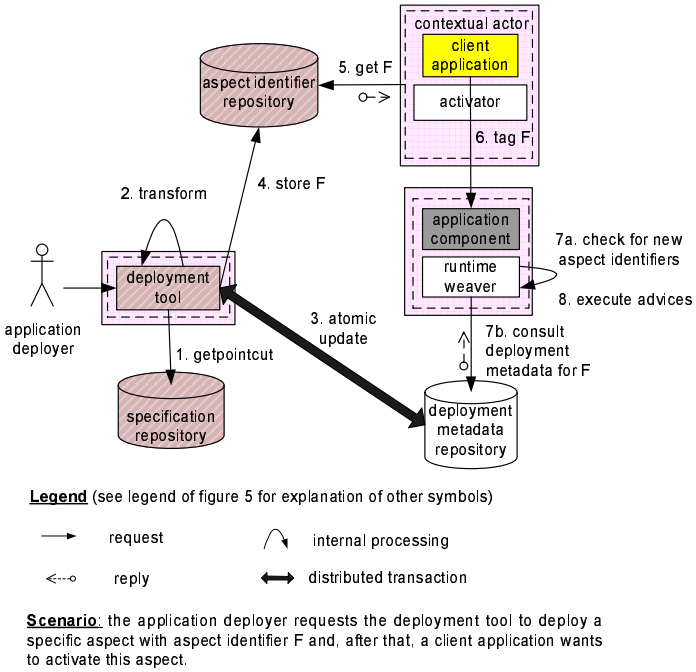

Figure 6: Coordination protocol for aspect weaving at run-time

requests the deployment tool to actually deploy the aspect and hereby he associates a new aspect identifier, say F[8] to the aspect, to indicate to the client application that the aspect implements the particular feature requested. After that, the coordination protocol for governing the actual run-time weaving of the aspect begins. Figure 6 indicates, by means of numbered arrows, what are the steps involved in this coordination protocol.

The coordination protocol consists of an deployment phase and an activation phase. The deployment phase consists of 4 steps (see figure 6):

1. the deployment tool retrieves the aspect binding of the aspect from the specification repository,
2. the deployment tool transforms the aspect binding into advice-to-joinpoint bindings,
3. the deployment tool updates the deployment metadata repository with the advice-to-jointpoint bindings in a deployment transaction,
4. the deployment tool inserts the aspect identifier F into the aspect identifier repository.

The activation phase consists of 4 steps:

5. the client application retrieves the aspect identifier F from the aspect identifier repository,
6. the client application tags aspect identifier F to its subsequent client requests,

[8]the 'F' from feature

7. the runtime weaver notices the new aspect identifier F on the first arrived message and consults the deployment metadata,
8. the runtime weaver updates its internal data structures to manage the new aspect and executes the appropriate advices, related to F.

Coordination protocol for aspect unweaving. Figure 7 presents the coordination protocol for unweaving an aspect at run-time. Aspect unweaving is useful for adaptation scenarios where the stakeholders of the distributed application want to remove a certain feature from the application because of various reasons, such as the feature is not often used by the clients, it is malfunctioning, or it is too heavy in terms of resource usage. In any case, these adaptation scenarios always involves a request from the application deployer to unweave an aspect with, say, aspect identifier F.

The coordination protocol consists of a deactivation phase and an un-deployment phase. The deactivation phase consist of 5 steps (see figure 7):

1. the deployment tool requests all runtime weaver instances to shutdown the execution of aspect F,
2. each runtime weaver instance configures itself such that it will block new messages from the outside (i.e. crossing the boundary of the distributed application),
3. the deployment tool removes aspect identifier F from the aspect identifier repository,
4. a client application sends a client request carrying the aspect identifier F,
5. the client application receives a runtime exception from the runtime weaver that aspect identifier F is not available anymore.

The un-deployment phase consists of 2 steps:

6. when there are no messages circulating anymore inside the distributed application that carry aspect identifier F, an asynchronously running thread will update the internal data structures of each runtime weaver instance such that all advice code and aspect instances, associated to F, can be released from memory,
7. each runtime weaver notifies the local deployment metadata repository, such that the latter can remove all relevant advice-to-joinpoint bindings from the deployment metadata.

4.3 Detailed design of the runtime weaver

The runtime weaver is the most critical component of the Lasagne runtime system since it is the crossroad where deployment and activation meet at runtime. Deployment metadata is the input for the runtime weaver to load aspects and to compose advices and base operations correctly. The aspect identifiers are input for the runtime weaver to dynamically dispatch messages to a selected subset of advices.

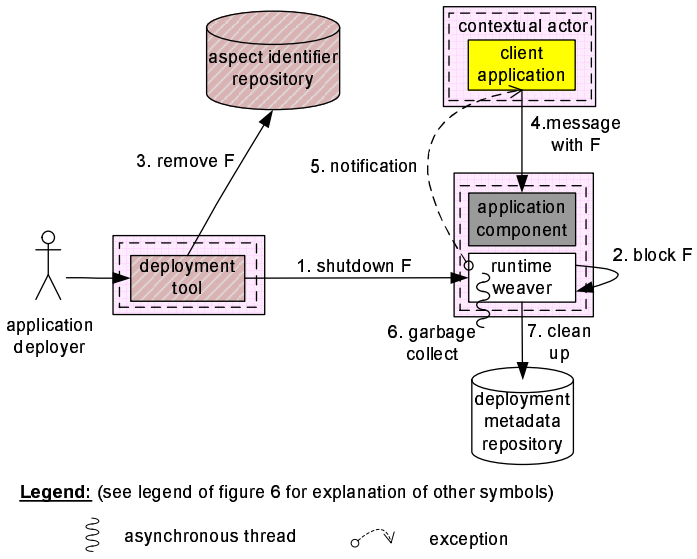

Figure 7: Coordination protocol for aspect unweaving at run-time

Structure of the runtime system. Figure 9 gives an overview of the internal structure of the runtime weaver. As stated above, the runtime weaver acts as a message-based dispatcher and is also designed in this way. From a decomposition perspective, it consists of a deployment manager component and an advice combiner component. The deployment manager component embodies the consultation and processing of deployment metadata, whereas the advice combiner embodies the message dispatch functionality for executing the appropriate advices.

To manage the aspect code and instances of an aspect, the deployment manager aggregates two data structures: a *method map* and an *aspect instance map*.

For each base component, the deployment manager has a separate method map. The method map is a table that has an entry for each provided and required operation of the component. Each entry stores an ordered list of advices. Each advice in the list is also labelled with the aspect identifier of its enclosing aspect. The advice combiner consults this method map to know which advices it may choose from to execute a received message.

For each base component *instance*, the deployment manager also manages a separate instance map. The instance map is a table that stores the aspect instances that the particular component instance is associated to. Each aspect instance is stored in the instance map with its aspect identifier as key. The base component instance is stored with its base identifier as key.

Load-time consultation of deployment metadata. During startup of the application, some deployment metadata may already exist because the application deployer is also free to deploy aspects before the application is started

up (hence, anticipated adaptation is also supported). This off-line specified deployment metadata will be processed by the deployment manager component at load-time. Thus, whenever a new base component is loaded, the deployment manager will consult the existing deployment metadata for this base component. To enable this, the deployment metadata repository offers a specific operation. The specification of this operation is presented in figure 8 as an UML model. As shown in the figure, the deployment manager receives a record structure of two elements. First, information related to where the aspects' implementations can be downloaded. Second, for every base operation, an ordered list of advice-to-joinpoint bindings is returned. When processing the deployment metadata for a specific base component, the deployment manager will transform the deployment metadata into reified pointers to advice methods in memory. These pointers are stored in the method map as advice method lists, according to the structure of the Advice-JPBindings entity in figure 8).

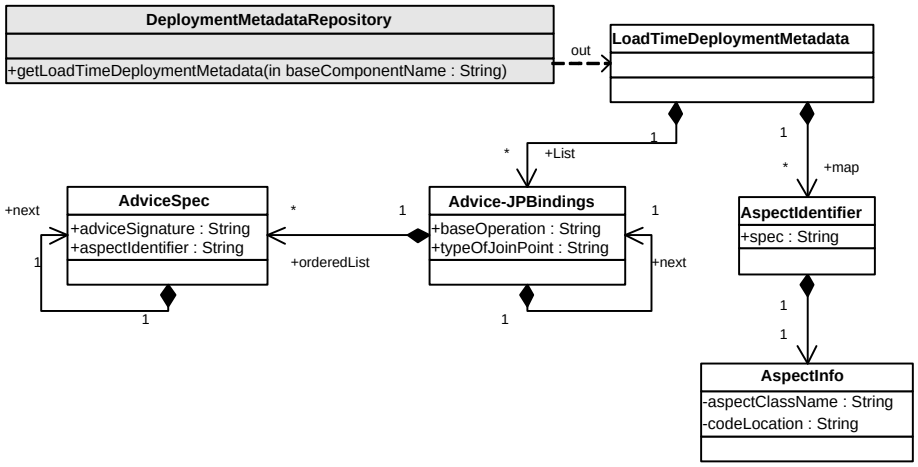

Figure 8: Deployment metadata repository interface for consulting deployment metadata at load-time.

Whenever a new instance of a base component is created, the deployment manager creates a separate instance map and creates a new entry in this instance map for every deployed aspect. When an advice is executed local to the base component instance, an instance of the enclosing aspect will be created and stored in the corresponding entry of the instance map.

Dispatching process. The flow-dependent activation mechanism is represented in the Lasagne runtime weaver by the advice combiner component. The advice combiner interprets the tagged aspect identifiers of each incoming message and will only redirect this message through the advices associated to these aspect identifiers. For every received message, the advice combiner thus selectively combines the advices that are bound to the invoked base operation. The tagged aspect identifiers determine to which advices the message will be dispatched.

Figure 9 illustrates in more detail how the deployment manager and the advice combiner interact among each other to perform the actual weaving. The

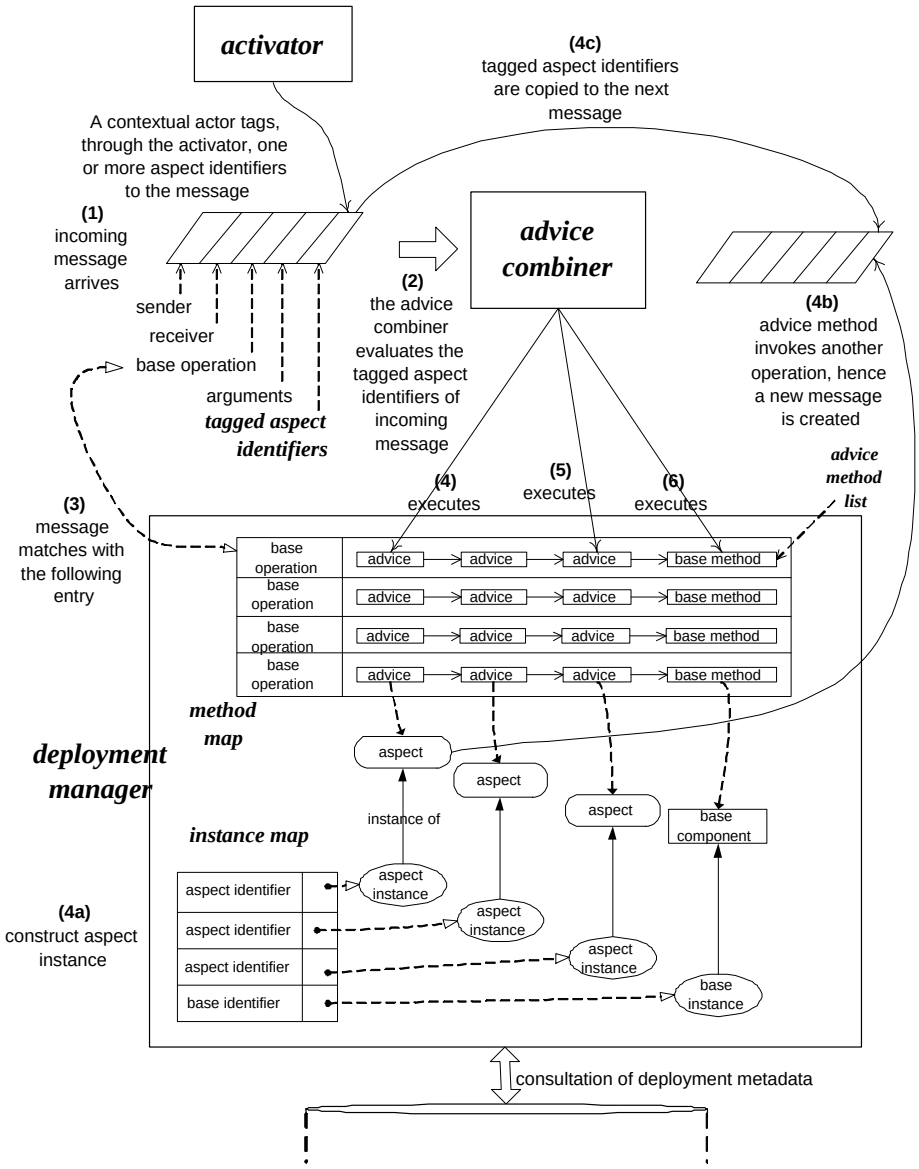

Figure 9: The interplay between the advice combiner and the deployment manager

figure actually depicts the situation that three aspects have already been loaded into memory. This is reflected in the deployment manager by the fact that advices from three different aspects are stored in the method map and by the fact that three entries have been created in the instance map to store an instance of the corresponding aspects.

To illustrate how the advice combiner and deployment manager interact, the figure shows a sequence of steps, numbered from (1) to (6).

An activator, shown at the top left corner of the figure, tags one or more aspect identifiers to a particular client request. The message flow, that represents the processing of this client request, propagates through the application. Step (1) in the figure indicates when the message flow has arrived at a base component instance of the application. The advice combiner must dispatch the incoming message to the appropriate combination of advices (step (2)). With this end in view, the advice combiner must

(3) determine the method list whose entry in the method map matches with the invoked base operation of the message, and
(4)(5)(6) execute those advices in that method list whose corresponding aspect identifier also appears tagged in the message.

When the advice combiner initiates the execution of the first advice method in the list (step (4)), it must first be checked whether the instance of the corresponding aspect has already been constructed(step (4*a*)). If the corresponding entry in the instance map is still empty, a new aspect instance must be created before the execution of the method can proceed. Of course, these two steps are repeated for the execution of the other methods in the method list as well (i.e. step (5) and (6)).

During execution, the advice methods or base method may invoke other operations. Thus, a new message is created for every invoked operation (step (4*b*)). To ensure consistency in the combination of aspects, the tagged aspect identifiers of the originally received message are copied into the new messages (step (4*c*)).

Run-time consultation of deployment metadata. After start-up of the application, the application deployer is able to deploy new aspects at run-time. This means that deployment metadata will be incrementally extended at run-time. As such, deployment metadata will be incrementally processed by the deployment manager as well, based on a lazy evaluation approach.

Only when the deployment manager receives a request from the method combiner to execute advice related to a newly deployed aspect identifier, the deployment manager consults the deployment metadata. The deployment metadata repository offers another operation for this run-time consultation. The specification of this operation is shown in figure 10. The most important difference with the load-time consultation is that only deployment metadata that is related to the new aspect identifier will be returned. The returned deployment metadata contains per base operation signature a specific advice, associated to the new aspect identifier, and the relative position at which the advice must be inserted in the existing advice method list. Only after these deployment steps are performed, the dispatching process can continue.

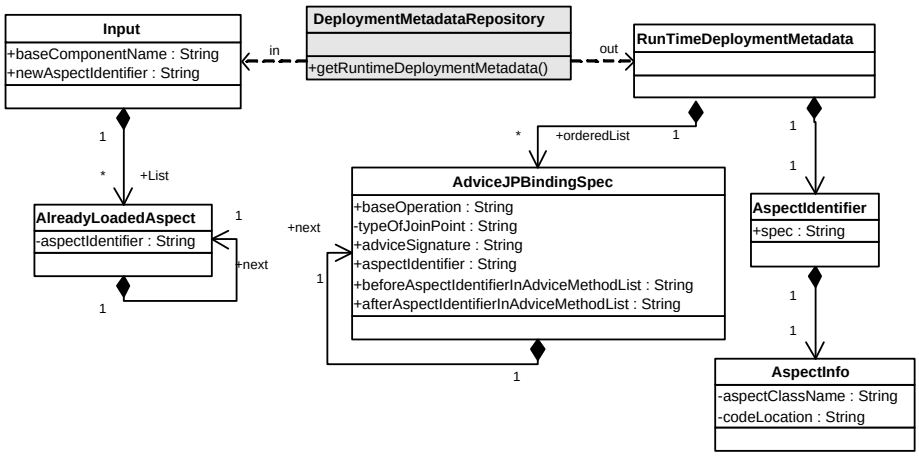

Figure 10: Deployment metadata repository interface for consulting deployment metadata at run-time.

As a conclusion, it is good to point out that a single specification of an aspect binding is thus transformed and interpreted across three layers of data structures. First the aspect binding itself is stored in the specification repository. Second the deployment metadata is stored in the deployment metadata repository. Third the internal structures of the runtime weaver contains pointers to advice methods.

5 Discussion

The discussion of this work will focus on four topics: the techniques used for implementing Lasagne, the integration of Lasagne on top of standard middleware platforms, the performance trade-offs in the current design of Lasagne, and broader applicability of Lasagne.

5.1 Implementation

To validate our run-time weaving model we have implemented a prototype of the Lasagne middleware on top of the programming language Correlate[42]. Needless to say that we focussed here on the key element that lies within the core of the Lasagne middleware: its runtime system.

Correlate is a concurrent object-oriented language with an execution environment that makes it easier to develop distributed applications. The execution environment is designed as a meta-level architecture that offers a so called *Meta-Object-Protocol* (MOP)[22]. Correlate's MOP essentially offers several meta-objects that allow to inspect and modify the internal semantics of the language and execution environment. Correlate is implemented as an extension of the Java programming language[3].

Figure 11 illustrates the implementation structure of the Lasagne middleware. As shown in the figure it consists of three layers:

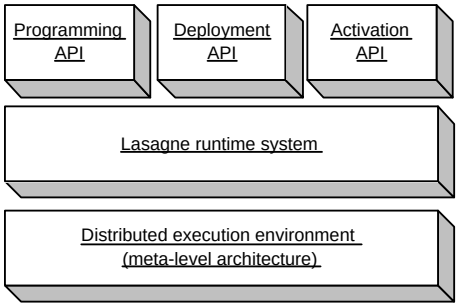

Figure 11: Overview of the Lasagne middleware

- the API layer that offers three separate programming interfaces for programming, deploying and activating aspects.
- the Lasagne runtime system
- A distributed execution environment that comes with the execution environment of Correlate.

The Lasagne runtime system is implemented as a modular plug-in of Correlate, using Correlate's MOP. The design of the runtime weaver and the details of the coordination protocols have already been presented in section 4.3 and 4.2.3. The deployment metadata repository is implemented as a set of files that are managed in a distributed file system[9]. Each file stores the deployment metadata of a given base component. The activator component is implemented using Correlate's MOP as well, by relying on the particular meta-object that allows to intercept method invocations and executions as first-class message objects. Its interface corresponds to the Activation API shown in figure 11.

Finally, the Programming and Deployment API provide basic support for programming and binding aspects to a distributed application. As stated in section 2, we do not require any extensions to state-of-the-art distributed pointcut constructs. Runtime deployment is currently supported by manually editing the deployment metadata files. As outlined in section 4.2.1, to ensure atomicity and concurrency control when updating the deployment metadata files, we only need to rely on basic transaction protocols.

5.2 Integration in state-of-the-art platforms

We compare Lasagne middleware to standards middleware such as OMG's CORBA, Enterprise Java Beans (EJB), and Microsoft .NET, and discuss how Lasagne could be integrated on top of these platforms.

As already stated in section 4.1, the runtime weaver is compared to a message dispatcher. Therefore, we believe the runtime weaver is situated at the same level as the Portable Object Adapter(POA) component in the OMG CORBA ORB model[50], or at the same level as the Container architecture in the Enterprise Java Beans (EJB)[33] and OMG CORBA Component model[51]. An important difference is however, that in traditional object-based middleware, message dispatch (or similar mechanism in container-based middleware) is only

[9]UNIX Sun OS 5.8 implementation of the Sun network file system[43].

driven by the receiver's object identity, as stored in the message. In Lasagne, message dispatch is driven by the tagged aspect identifiers as well.

The tagging of aspect identifiers can be commonly achieved in CORBA by means of portable interceptors[36] or in .NET by means of a special API that gives access to the logical call context[27].

In Lasagne, aspect identifiers are propagated with the message flow by storing the identifiers as part of the reified messages, and copying them along the invocation path. Another approach is storing the aspect identifiers in a thread local variable. We have not followed this approach in order to circumvent the problem of loosing *logical thread identity* when control crosses system boundaries, or when control is asynchronously handed over between two threads as in the active object model[35]. Some of the standard middleware platforms already provide a mechanism to cope with this problem such as distributed threads[9] in real-time CORBA or the LogicalCallContext class in .NET[27].

Finally, the aspect identifier repository can be compared with a standards component repository, with the difference that it enables lookup of (distributed) aspects, whereas a standards component service enables lookup of (distributed) components.

5.3 On performance trade-offs

The proposed design and implementation has clearly been focused on providing generic support for a broad range of behavioral adaptations that have to be performed in a distributed system. As we aimed for a solution that enables different context-dependent versions of the application to coexist, and that should enable distributed service offering without any disruption because of adaptations - it seems obvious that we have created an architecture which generates quite some overhead in achieving the proposed objectives. We analyze the costs by addressing (1) cost in terms of service availability, (2) overhead in terms of the cost of processing interactions, (3) overhead in terms of memory consumption.

(1) Our solution offers consistent behavioral adaptation of distributed entities without causing unavailability of any service or application feature.

(2) We have chosen to obtain the value of our solution at the cost of an increased processing time for invocations. The reason is threefold:

1. We want a solution that guarantees no service disruption when an (complex) adaptation is in progress. This is an inherently solid basis for mission-critical systems that must be permanently on-line.
2. We anticipate that a broad range of network applications (e.g. the typical case of web services) is relatively insensitive to processing delays as communication delays often dominates the response times in executing distributed services. We discuss this in further detail in subsection 5.4.
3. From a methodological viewpoint, we believe that it is a sound strategy to ensure that (a) a first (base level) architecture is offering a robust solution with absolute guarantees (no service disruption and atomicity) and that (b) optimizations - probably weakening other qualities (e.g. flexibility) - are pursued thereafter. This is exactly the scope of our ongoing and future work. For instance, Lasagne has already an option to limit deployment of aspects to load-time only, hereby only supporting anticipated adaptation.

Flexibility cuts are minor (aspects can still be dynamically activated and deactivated on a per message basis), while completely eliminating the performance overhead related to the coordination between deployment and activation.

(3) Our architecture is extremely economical when it comes to memory cost. Deploying distributed aspects implies the storage of a logical reference (aspect identifier) in the nodes where a specific aspect must be executed. Pro-actively loading aspects would lead to substantially larger memory footprints, as well as extra delays when triggering adaptation. In the current solution, aspect code is loaded on demand, i.e. when the first activation of the required behavior occurs.

5.4 Broader applicability

For several application areas, it has become increasingly more important that software can be dynamically adapted to evolving requirements of a given context. Our discussion will focus on three such application areas: application service provisioning, autonomic computing, and adaptive Quality of Service (QoS).

The proliferation of web services, and recent initiatives such as Microsoft's *Office Live*[10] indicate that software applications are gradually evolving from a desktop model to a *service provider* model. For example, Microsoft Office Live will offer expert business management applications as on-line services without that their customers need Office or Windows installed on their desktop computers. Moreover, the underlying system infrastructure is not only targeted at remote application execution, but also at the uniform deployment and integration of new applications. In other words, Mario Tokoro's vision[47], that system software such as compilers will some day in the future be offered as distributed service objects, seems closer at hand. This evolution gives the notion of a client request a completely new meaning. To run the compiler, for example, you send a client request.

When popular on-line software services will have a vast number of client requests on a daily basis, it becomes economically profitable to customize software services to the needs of individual customers. Existing software services are already personalized. Such personalization is however limited to the user interface and data that is offered to the customer. There lies also great opportunity in real customization scenarios where customers can dynamically select whole behavioral features to the software service itself. Needless to say that Lasagne is very well suited for providing this type of client-specific customizations.

Lasagne could also provide a complementary advantage to middleware support for autonomic computing and self-healing[15, 6], and adaptive QoS[4]. Research in these areas is targeted towards extending middleware with policies and techniques for context monitoring and dynamic selection of adaptation strategies. To our knowledge, these middleware extensions do not automate the task of coordinating adaptation operations across multiple processes and computer nodes; the coordination must be explicitly managed by the programmer. After all, these middleware extensions are integrated with an object-based platform, thereby constraining the representation of a single adaptation strategy as inherently non-crosscutting. We believe that Lasagne could be useful for automating some of the cross-node coordination requirements in QoS control and autonomic

computing systems. After all, Lasagne offers a solid service-enabling platform for the management and coordinated selection of cross-process and cross-node adaptation strategies.

6 Related work

Six categories of related work are considered in our research: recent aspect-oriented middleware advances, network software, dynamic software architecture, coordinated adaptation in distributed systems, reflective middleware, and programming languages. The discussion will focus on the first five categories.

Aspect-oriented middleware. The general relation to other aspect-oriented middleware platforms has already been discussed in the motivation of this report. To our knowledge, no other aspect-oriented middleware exists to date that provides support for atomic weaving of distributed aspects. The most important difference between existing AO middlewares and Lasagne is that their run-time weaving model does not support the distinction between deployment and activation.

DADO[52] (distributed aspects for distributed objects) middleware offers an innovative IDL languages for programming distributed aspects in heterogeneous environments. DADO has a mechanism, called Remote Multiple Contextual Invocation (RMCI), that provides a form of dynamic per-invocation selection of aspects as in Lasagne. However this mechanism is restricted within the scope of a single client-server interaction and, therefore, does not provide any guarantees with respect to system-wide consistency in the activation of aspects.

CAM/DAOP[14, 39] is a component and aspect based approach that combines the benefits of both CBSD and AOSD disciplines. An innovative aspect of CAM/DAOP is that it specifies the composition of components and aspects using an architectural description language (ADL)[44], named DAOP-ADL. This ADL-based approach provides an interesting complement to Lasagne (or any run-time weaving approach for that matter). After all, using DAOP-ADL, application deployers are able to comprehend the overall aspect-component composition, facilitating a better understanding and easier verification of the application *as a whole*. This is of course an important software engineering quality that improves the safety and robustness of deploying aspects at runtime.

Lasagne would, in the same vain, be nicely complemented by model-driven middleware (e.g. [16]) and aspect-oriented domain modelling (e.g. [18]). Combining these approaches leads to a very powerful concept. Design models of aspects and applications can be specified, composed and possibly verified. Once composed, these models can be automatically synthesized to deployment metadata for a specific middleware platform of choice.

Referring to our discussion on the applicability of Lasagne to adaptive QoS, Duzan et al. [13] have built an aspect-oriented middleware platform, called QuO, that allows to separate adaptive QoS concerns from the application. The middleware also provides support for coordinating adaptation operations across multiple processes and computer nodes. Lasagne could be useful for automating that coordination support as well.

Dynamic architectures for network protocols. Some of the problems related to ensuring consistency in the activation of aspects (see section 4.2.2 have already been addressed, at least to some extent, in networking protocols. It is common for messages to have dynamically computed headers in order for peers to process the message correctly by dispatching it through a series of protocols. Also at the client-side, the shift towards dynamic protocol stack composition architectures (e.g [34, 31]) makes that the protocol stack at the server-side must have the intelligence to deal with this correctly.

There are two important differences between Lasagne and network protocols however. First, Lasagne is more generic because it is aimed to support adaptations at different layers of software, including the application layer[49], middleware layer[48] and network layer[21]. Secondly, Lasagne's support is reusable across multiple application domains. In protocol stacks the support for message dispatching is wired inside the protocol stack architecture itself, whereas in Lasagne all support is externalized in a separate runtime system that can be applied to any family of application.

Another related technology in this field is programmable networking. Programmable networking stands for a whole spectrum of networking technologies that aim to introduce more openness and adaptability into today's networks by making network routers programmable. Historically there have been two main approaches to the provision of programmability in networks[12]: First, in the *active networking paradigm*[46] so called 'active packets' carry programs that execute on 'active' nodes. Secondly in the *open signaling approach* (see e.g [7]), routers export 'control interfaces' through which they can be remotely (re)configured by out-of-band, application-specific, signaling protocols.

Overall, the active networking approach is the most dynamic since it enables application-specific adaptations across the network in a very fine-grained scale. However, it is perceived as more vulnerable to security treats from malicious applications, and there is also the problem of domain mismatch[30]: application programmers, who wish to customize the underlying network to their preferences, are suddenly confronted with a domain (in this case, active networks) that is completely different from their familiar application domain. The open signaling approach does not suffer from these problems since it supports adaptations that are programmed by a trusted and informed third party. However, open signaling is an order of magnitude less dynamic.

We position Lasagne in the middle of these two approaches:

- Similar to the open signaling approach, adaptations are programmed by a trusted application deployer, who is an expert at his application domain (in this case, network protocols).
- Similar to the active network approach, once adaptations are deployed, they can be dynamically activated on a per message basis by contextual actors (which, in this case, act as the (client) applications wanting to customize the underlying network).

Comparatively, adaptations in programmable networking can be programmed more complex than in Lasagne, but this also leaves the door open for more intrusive changes that endanger safety. A second important difference is that Lasagne is more generic. The usage context of programmable networking is more like a niche, and of course, the approach is somewhat optimized for that niche.

Dynamic software architectures. In Regis[29] and ArchStudio[37], distributed applications are constructed from a number of components and connectors that encapsulate the interactions between these components. Here, the term "component" corresponds with the traditional view of component-based development.

The emphasis of these works is on the description, dynamic reconfiguration and evolution of the application's architecture. Connectors help to decouple components from one another and the systems use configuration languages to describe the configuration of the components. The runtime structure of the application is altered by applying a program written in the configuration language to the current architecture, thus generating a different arrangement of components and connectors. Our goal differs from these works in that Lasagne focusses on adapting an application to context-specific needs, instead of coping with runtime software evolution in general. Furthermore, Lasagne's adaptation process is based on a system-wide additive refinement of existing base components with crosscutting aspects, rather than replacing existing components with new ones and switching connectors.

Coordinated adaptation in distributed systems. In the context of the above component-based reconfiguration approaches there is a whole field of research (e.g. [25, 37, 17, 2, 8, 1, 20]) that looks at the issue of how to achieve coordinated adaptation in a distributed system: when a reconfiguration involves replacing multiple components, atomicity problems appear that are very similar to those that have been studied in this report.

The existing approaches to coordinated adaptation in a component-based setting can be classified according to

- their need for service disruption as either *disruptive* (e.g. [2]) or *non-disruptive* (e.g. [1]) reconfiguration.
- their ability to replace an existing component by a new version or to let both be used simultaneously as either *single-configuration-managed* (e.g. [2]), or *multiple-configurations-managed* (e.g. [17]) reconfiguration.

Lasagne clearly belongs to the multiple-configurations-managed and non-disruptive category[10]. Remarkably, we have not found any coordinated adaptation approach, in a component-based setting, that combines non-disruptive and multiple-configurations-management. Admittedly, the problem of coordinated adaptation has been mostly studied in a setting where adaptation is triggered out-of-band, rather than in-band as Lasagne does.

Another important difference is that some of the aforementioned approaches also employ techniques and protocols for preserving global state consistency[32] and application state invariants[2]. As stated in section 2, Lasagne has not been designed for global state management, given that Lasagne targets behavioral extensions to mostly service-oriented application areas (also see section 5.4). Web services, for example, make as much as possible tiers state-less, and incorporate the complete state in a back-end database or in an XML document that is part of the message.

[10]The first point may require some explanation. Lasagne supports multiple-configuration-managed reconfiguration because the flow-dependent activation mechanism implies that an application may be running in different versions simultaneously.

Reflective middleware. Other related work are projects (e.g. [24] and [5]) that use reflection[28] to achieve adaptive middleware. Generally, the use of reflective techniques allows for more powerful adaptations in middleware and distributed applications than aspect-oriented middleware does. However, employing reflection for context-sensitive adaptations yields often a too complex system that cannot be understood by the typical contextual actor (or more specifically, the human operators involved). In this respect, Lasagne offers a very simple adaptation model to the contextual actor. For example, by simply tagging aspect identifiers to its messages, (client) applications can dynamically adapt the underlying middleware to its preferences.

More recent research on reflective middleware [11] has focussed on combining reflection with component framework technology[45] in order to preserve behavioral integrity. Much of this work has focussed on enforcing semantically compatible compositions of middleware components. Lasagne could be useful for implementing some of these integrity requirements, especially those that relate to maintaining semantic compatibility across multiple nodes.

7 Conclusion

The main contribution of this report is the study and solution of the atomicity problems that appear when distributed aspects are woven into the distributed application at run-time.

We have argued that the underlying run-time weaving model of existing aspect-oriented middleware platforms are not fit to be enhanced with support for atomic weaving of distributed aspects, because such enhancements would require very complex inter-node coordination protocols. Consequently, we have tackled this complexity of the atomic weaving problem at the place where it is born: at the level of the run-time weaving model itself.

The key behind the run-time weaving model of Lasagne is that it operates according to a two-phased process that distinguishes between deployment and subsequent activation of aspects. An aspect is made ready for weaving during deployment, but it is only really woven into the application's execution when it is activated within a particular message flow. This approach allows us to keep the required inter-node coordination protocols quite simple.

We have presented the architecture and design of the Lasagne middleware. The Lasagne middleware ensures that distributed aspects are woven in an atomic manner such that behavioral integrity of the distributed application is not harmed.

We have motivated, positioned and discussed our approach. We believe that Lasagne middleware offers a solid service enabling platform for the management and run-time weaving of behavioral extensions that are of crosscutting nature – cross-node, cross-process – and that tackle both application-layer as middleware-layer adaptations. Finally, we believe that the genericity of Lasagne's run-time weaving model makes it applicable to a broad range of application domains.

References

[1] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: How to upgrade distributed systems. In *Ninth Workshop on Hot Topic in Operating Systems (HotOS-IX)*, 2003.

[2] J. P. A. Almeida, M. Wegdam, M. van Sinderen, and L. Nieuwenhuis. Transparent dynamic reconfiguration for corba. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA 2001)*, pages 197–207. IEEE Computer Society, 2001.

[3] K. Arnold and J. Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley Publishing Company, 1996.

[4] Klara Nahrstedt Baochun Li. A control-based middleware framework for quality of service adaptations. *IEEE Journal of Selected Areas in Communication, Special Issue on Service Enabling Platforms*, 17(9):1632–1650, September 1999.

[5] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas. An architecture for next generation middleware. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 191–206. Springer, 1998.

[6] Gordon S. Blair, Geoff Coulson, Lynne Blair, Hector Duran-Limon, Paul Grace, Rui Moreira, and Nikos Parlavantzas. Reflection, self-awareness and self-healing in openorb. In *WOSS '02: Proceedings of the first workshop on Self-healing systems*, pages 9–14. ACM Press, 2002.

[7] Prashant R. Chandra, Allan Fisher, Corey Kosak, T. Ng, Peter Steenkiste, Eiichi Takahashi, and Hui Zhang. Darwin: Customizable resource management for value-added network services. In *ICNP*, pages 177–188, 1998.

[8] W.-K. Chen, M. A. Hiltunen, and R.D. Schlichting. Constructing adaptive software in distributed systems. In *Proceedings – the 21st IEEE International Conference on Distributed Computing Systems*, pages 635–643. IEEE Computer Society, 2001.

[9] R. Clark, D. E. Jensen, and F.D. Reynolds. An architectural overview of the alpha real-time distributed kernel. In *Proceedings of the USENIX Workshop on Microkernel and Other Kernel Architectures*, April 1992.

[10] 2006 Microsoft Corporation. Microsoft office live. http://www.microsoft.com/office/officelive/default.mspx.

[11] Geoff Coulson, Gordon S. Blair, Michael Clarke, and Nikos Parlavantzas. The design of a configurable and reconfigurable middleware platform. *Distributed Computing*, 15(2):109–126, 2002.

[12] Geoff Coulson, Gordon S. Blair, David Hutchison, Ackbar Joolia, Kevin Lee, Jo Ueyama, Antônio Tadeu A. Gomes, and Yimin Ye. NETKIT: a software component-based approach to programmable networking. *Computer Communication Review*, 33(5):55–66, 2003.

[13] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building adaptive distributed applications with middleware and aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 66–73, New York, NY, USA, 2004. ACM Press.

[14] L. Fuentes, M. Pinto, and P. Sánchez. Dynamic weaving in cam/daop: An application architecture driven approach. In *Proceedings of the Dynamic Aspect Workshop in conjunction with AOSD 2005*, March 2005.

[15] A. G. Ganek and T. A. Corbi. The dawning of the autonomic computing era. *IBM System Journal*, 42(1):5–18, 2003.

[16] A. Gokhale, D. Schmidt, B. Natarajan, J. Gray, and N. Wang. Model-driven middleware, 2004.

[17] J. Gouveia, G. Koutsoukos, L. Andrade, and J. Fiadeiro. Tool support for coordination-based software evolution. In *TOOLS Europe 2001*. IEEE Computer Society Press, 2001.

[18] Jeff Gray, Ted Bapty, Sandeep Neema, Douglas C. Schmidt, Aniruddha Gokhale, and Balachandran Natarajan. An approach for supporting aspect-oriented domain modeling. In *Proceedings of the second international conference on Generative programming and component engineering*, pages 151–168. Springer-Verlag New York, Inc., 2003.

[19] M. Jackson. Some complexities in computer-based systems and their implications for system development. In *Proceedings of the International Conference on Computer Systems and Software Engineering*, pages 344–351, 1990.

[20] Nico Janssens, Wouter Joosen, and Pierre Verbaeten. NeCoMan: Middleware for Safe Distributed-Service Adaptation in Programmable Networks. *IEEE Distributed Systems Online*, 6(7), July 2005.

[21] B. N. Jørgensen, E. Truyen, F. Matthijs, and W. Joosen. Customization of Object Request Brokers by application specific policies. In *Middleware 2000*, volume 1795 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2000.

[22] G. Kiczales, J. des Riviers, and D. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP2001 – Object-oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[24] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, L. C. Magalhãs, and R. Campbell. Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB. In *Middleware 2000*, volume 1795 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2000.

[25] J. Kramer and J. Magee. The evolving philosopher problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[26] Bert Lagaisse, Bart De Win, and Wouter Joosen. SoBeNet: financial case study - Part 1: requirements and analysis. Report CW 404, Department of Computer Science, K.U.Leuven, Leuven, Belgium, 2005. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW404.abs.html.

[27] J. Lowy. Contexts in .NET: Decouple components by injecting custom services into your object's interception chain. *MSDN Magazine*, March, 2003.

[28] P. Maes. Concepts and experiments in computational reflection. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 147–155. ACM SIGPLAN Notices 22(12), 1987.

[29] J. Magee, N. Dulay, and J.Kramer. Regis: A constructive development environment for distributed programs. *Distributed Systems Engineering Journal*, 1(5), 1994.

[30] Frank Matthijs, Nico Janssens, and Pierre Verbaeten. Automatic service composition: a case for active networks usability. Report CW 356, Department of Computer Science, K.U.Leuven, Leuven, Belgium, January 2003. URL = http://www.cs.kuleuven.ac.be/publicaties/rapporten/cw/CW356.abs.html.

[31] Sam Michiels, Nico Janssens, Lieven Desmet, Tom Mahieu, Wouter Joosen, and Pierre Verbaeten. A component platform for flexible protocol stacks. In *Component-Based Software Development for Embedded Systems: An Overview of Current Research Trends*, volume 3778/2005 of *Lecture Notes in Computer Science*, pages 185–208. Springer-Verlag, GmbH, November 2005.

[32] K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems.* PhD thesis, Imperial College, London, March 1999.

[33] R. Monson-Haefel. *Enterprise JavaBeans, 3rd Edition.* O'Reilly, September 2001.

[34] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. The click modular router. In *SOSP*, pages 217–231, 1999.

[35] O. Nierstrasz. Composing active objects — the next 700 concurrent object-oriented languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 151–171. MIT Press, 1993.

[36] Object Management Group. *The Common Object Request Broker: Architecture and Specification — Version 3.0*, July 2003. Available at http://doc.ece.uci.edu/CORBA/formal/02-06-33.pdf.

[37] P. Oreizy, N. Medvidovic, and R. N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 177–186. IEEE Computer Society Press / ACM Press, 1998.

[38] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.

[39] M. Pinto, L. Fuentes, and J.M. Troya. Daop-adl: An architecture description language for dynamic component and aspect-based development. In *Proceedings of the Second International Conference on GPCE*, volume 2830 of *LNCS*, page pp. 118137. Springer-Verlag, September 2003.

[40] A. Popovici, G. Alonso, and T. Gross. Just in time aspects: Efficient dynamic weaving for Java. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, pages 100–109. ACM press, 2003.

[41] A. Popovici, A. Frei, and G. Alonso. A proactive middleware platform for mobile computing. In *Middleware 2003*, volume 2672 of *Lecture Notes in Computer Science*, pages 455–473. Springer, 2003.

[42] B. Robben. *Language Technology and Metalevel Architectures for Distributed Objects*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 1999.

[43] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the sun network filesystem. *Innovations in Internetworking*, pages 379–390, 1988.

[44] M. Shaw and D. Garlan. *Software Architecture: Perspective on an Emerging Discipline*. Prentice-Hall, 1996.

[45] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 1998.

[46] D. L. Tennenhouse, J. M. Smith, W. D. Sincoskie, D. J. Wetherall, and G. J. Minden. A survey of active network research. *IEEECM*, 35(1):80–86, January 1997.

[47] Mario Tokoro. The society of objects. *OOPS Messenger*, 5(2):3–12, 1994.

[48] E. Truyen, B. Nørregaard Jørgensen, , and W. Joosen. Customization of component-based Object Request Brokers through dynamic reconfiguration. In *Technology of Object-Oriented Languages and Systems – TOOLS 33*, pages 181–194. IEEE Computer Society, 2000.

[49] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of the 23rd International Conference on Software Engeneering (ICSE'01)*, pages 233–242. IEEE Computer Society, 2001.

[50] Steve Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, February 1997.

[51] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. Overview of the corba component model. *Component-based software engineering: putting the pieces together*, pages 557–571, 2001.

[52] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *ICSE 2003*, pages 174–186, 2003.

A Coordination protocol for run-time aspect replacement

Figure 12 presents the coordination protocol for replacing an aspect at run-time. Replacement means that the application deployer requests the deployment tool to replace an aspect, having aspect identifier F_{old}, with a new aspect identifier, having aspect identifier F_{new}.

An important property of the coordination protocol is that replacement is performed transparent for the contextual actors that have activated the old aspect. To achieve this transparency, the second phase employs automatic aspect identifier conversion.

The coordination protocol is a combination of the two previous coordination protocols, and therefore consists of three phases: deploying the new aspect, deactivating the old aspect and activating the new aspect, and un-deploying the old aspect.

Deploying the new aspect. This phase consists of 5 steps (see figure 12):

1. the deployment tools retrieves the aspect binding of the new aspect,
2. the deployment tool transforms the aspect binding into advice-to-joinpoint bindings,
3. the deployment tool updates the deployment metadata repository with the advice-to-jointpoint bindings in a deployment transaction,
4. the deployment tool inserts the aspect identifier F_{new} into the aspect identifier repository,
5. the deployment tool requests all runtime weaver instances to replace aspect F_{old} with aspect F_{new}.

Deactivating the old aspect; Activating the new aspect. This phase consists of 6 steps:

6. each runtime weaver instance configures itself such that it will transform new messages from the outside (i.e. crossing the boundary of the application), that carry the aspect identifier F_{old} into carrying F_{new},
7. the deployment tool removes aspect identifier F_{old} from the aspect identifier repository,
8. a client request carries the aspect identifier F_{old},

9. the client request is converted by the runtime weaver such that F_{old} is detached from the client request and instead F_{new} is tagged to it,
10. the runtime weaver consults the deployment metadata for F_{new},
11. the runtime weaver instance updates its internal data structures and executes the advices that are associated to F_{new}.

Un-deploying the old aspect. This phase consists of 2 steps:

12. when there are no messages circulating inside the distributed application that carry aspect identifier F_{old}, an asynchronously running thread will update the internal data structures of each runtime weaver instance such that all advice code and aspect instances, associated to F_{old}, can be released from memory,
13. each runtime weaver instance notifies the local deployment metadata repository, so that the latter can remove all relevant advice-to-joinpoint bindings, associated to F_{old}, from the deployment metadata.

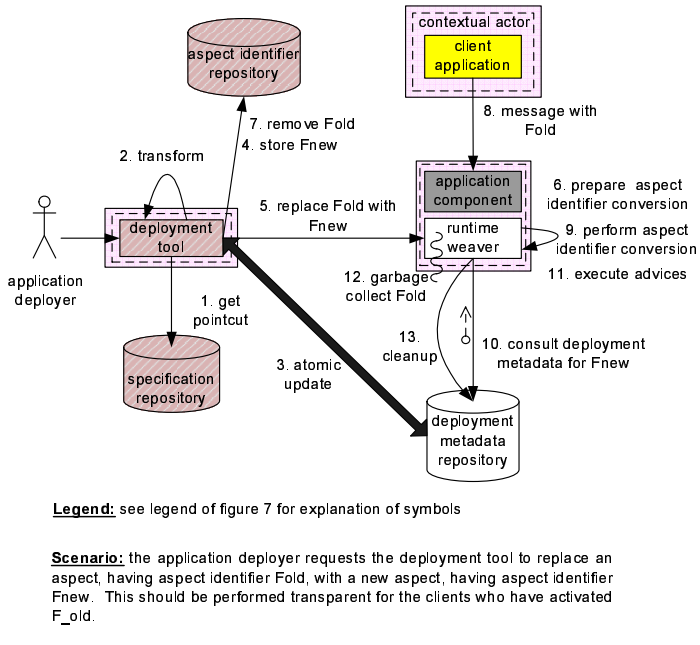

Figure 12: Coordination protocol for aspect replacement at run-time