True and Transparent Distributed Composition of Aspect-Components

Bert Lagaisse and Wouter Joosen

Dept. of Computer Science, K.U.Leuven, Belgium
{Bert.Lagaisse, Wouter.Joosen}@cs.kuleuven.be

Abstract. Next-generation middleware must support complex compositions that involve dependencies between multiple components residing in different contexts and locations in the network.

In this paper we present DyMAC, an aspect-oriented middleware platform that offers an aspect-component model to support such complex distributed compositions by means of advanced remote pointcuts, transparent remote advice and distributed instantiation scopes for aspects. The remote pointcuts can evaluate on calls and executions of remote method invocations and can also evaluate on the distributed context. The remote advice can be executed transparently in a remote environment while still respecting the full semantics of existing types of advice, including around advice. The component model unifies aspects and components into one entity with one interaction standard.

To our knowledge, DyMAC middleware is the first AO middleware platform that distributes the concepts of aspect-oriented composition completely and transparently.

1 Introduction

The environments in which distributed software applications must execute have become very dynamic and heterogeneous. As a result, software must be dynamically composed and even be adapted at runtime. This is for instance the case in ubiquitous computing environments. Typical enterprise applications expose similar needs.

Distributed applications are typically built on middleware, the software that sits between lower level system software (such as the OS) and the distributed programming platform. A dominant trend in (typical) enterprise middleware is the fact that specialized servers combine many middleware functions with specific component frameworks (e.g. J2EE, .NET etc.). The value of such a middleware component framework is twofold [1,2]: first, a specific component model enables the construction of applications from independently developed third party components - at least in principle - and second, built-in services facilitate covering non-functional requirements of a distributed application. The presence of built-in (container managed) services is often the basis for acceptance of a specific middleware platform. However, this critical success factor is at the same time the basis for the limitations of such a middleware platform. Built-in services support modularized, declarative composition of concerns of a cross-cutting nature:

M. van Steen and M. Henning (Eds.): Middleware 2006, LNCS 4290, pp. 42–61, 2006.

these are concerns that cannot easily be addressed without creating code that is scattered over the application and middleware artifacts. However, these built-in services are hard to modify or customize. Solutions for these shortcomings have been proposed in [3,13]. But, the services also cannot be used in complex compositions. This observation characterizes the limited flexibility that is supported by state-of-the-art middleware: more complex compositions must be enabled. The types of compositions that should be supported are very broad. Many of these compositions involve dependencies between multiple components residing in different contexts and locations in the network. This is extremely hard if one has to rely purely on state-of-the-art software development technologies - i.e. object-oriented and component based software engineering.

To address this problem, aspect-oriented software development (AOSD[7]) often has been put forward as a possible solution. AOSD addresses this shortcoming by focusing on the systematic identification, modularization, representation and composition of (often crosscutting) concerns or requirements throughout the entire software development process. The core concept in AOSD is an aspect [4,5]: a coherent entity that addresses one specific concern and that has the properties of a module that can be changed independently of other modules. An aspect defines behaviour that can be executed (so called advice) and defines composition logic to describe complex and dynamic dependencies between this behaviour and the rest of the software system. This composition logic is expressed using a joinpoint model. A common definition of a joinpoint refers to well-defined places in the structure or execution flow of a program [5,7]. In any case, joinpoints represent dynamic, runtime conditions that arise during program execution. The occurrence of such a condition is an event that can trigger the execution of aspect behaviour (advice). A set of joinpoints can typically be specified with pointcut designators that address and describe the kind and context of the joinpoints [7]. By the *kind* of a joinpoint we mean for instance a method call or a field access, etc. By the *context* we refer to additional information that can be made available to constrain the condition, such as the method signature, type and identity of the caller or callee of a method, further credentials and properties of the caller etc. The statically decidable conditions of a pointcut can be evaluated at compiletime or loadtime. The dynamic conditions are evaluated at runtime.

In state-of-the-art AOP, context information is essentially limited to local information, managed in a single VM. In a realistic distributed application however, joinpoints must refer to context information that is inherently distributed. Relative to state-of-the-art AOP, *distributed* joinpoints are advanced in that they transparently distribute the basic concept of a joinpoint: they refer to sophisticated conditions in distributed systems that are required to express composition in a distributed application. In general, the context information that is needed must refer to (potentially multiple) components that are not local, and possibly to distributed infrastructure. In particular, to express these runtime conditions and express compositions, we need support for aspects with three key features.

1. We need remote pointcuts, that can evaluate on calls and executions of remote method invocations and also can evaluate on distributed context.
2. Remote advice is required, which can be transparently executed in a remote environment (different from the pointcut evaluation), while still respecting the full semantics of existing types of advice, including *around* advice[1].
3. To be part of a mature middleware concept, aspects need component semantics, including clearly defined interfaces, supporting third party composition and interaction with other aspects.

Many initiatives in the domain of aspect-oriented middleware (AO middleware) have started to support creation, deployment and execution of distributed aspects for a distributed environment [9,15,18,17]. However, so far none of these research results have defined and illustrated a *complete* solution to the above mentioned challenges. In this paper, we present DyMAC, a middleware architecture that offers true and transparent distributed composition of aspect-components. Its component and composition model offers a solution for the three key challenges.

The rest of this paper is structured as follows. In section 2 we refine the problem statement and motivate why true and transparent distributed composition of aspect-components will be an important feature of next-generation middleware. Section 3 describes the model, architecture and implementation of our solution, DyMAC middleware. Section 4 evaluates our prototype. We discuss related work in section 5 and we conclude in section 6.

2 Problem Refinement

We use an example of a banking application that manages checking accounts and offers support to perform transactions on the checking accounts. The core business component is *BasicBanking*, which is a component offering operations to create new checking accounts, and execute transactions: withdrawals, deposits and transfers. This component is located at the application server. The employees at the branch offices use the *EmployeeClient* component at their workstations to handle the requests of the customers to create a new checking account or perform a transaction. The clients send the requested operations to the BasicBanking service at the application server. We have depicted the deployment architecture of the application in Figure 1. It also describes the set of additional middleware services that are part of the application:

1. A client-side, local component asking the employees for credentials, before the EmployeeClient component starts up.
2. An authentication service to authenticate the credentials of the employees and add an authentication token to their execution context. This authentication service is located at the central authentication server and is called after the client has provided his credentials.

[1] Around advice replaces the advised method invocation.

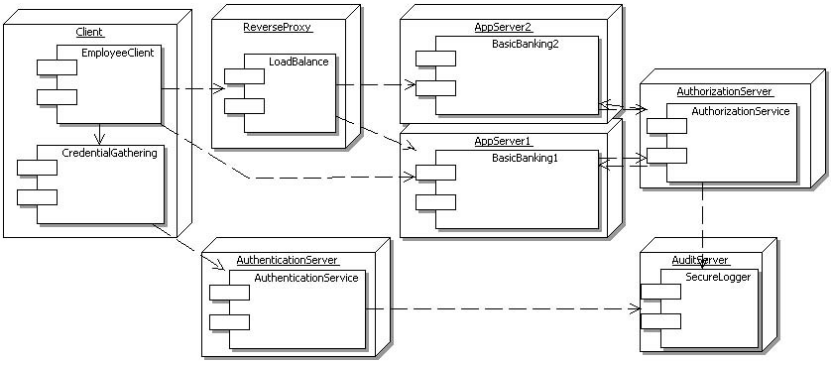

Fig. 1. The deployment architecture of the banking application

3. An authorization service at a central authorization server that verifies the application-level rights associated with the authenticated user. Before the execution of a remote method invocation at the BasicBanking service, this authorization server is called.
4. A load-balancing and fail-over service that delegates the calls of the clients to one of the application servers, based on the load or availability of the servers. This middleware service is located at a dedicated server (called reverse proxy-server).
5. A secure logging component at the central audit server that keeps track of all authentication and authorization attempts and the results.

The composition policies of these middleware services should solve the problem of crosscutting calls to the services in the core business components, and should allow a clean separation of concerns. Therefore the component and composition model needs to support the three key features from section 1. We motivate and refine the three key features with illustrations from the example.

Advanced remote pointcuts

1. Remote pointcuts that are able to refer to joinpoints before and after calling and executing remote method invocations in a distributed system. This notion extends the *kind* of the joinpoint for distributed systems. E.g. for the load balancing service: whenever a client machine calls the BasicBanking service, before that call load balancing advice should be called.
2. Remote pointcuts that can evaluate over the contextual properties about the components involved: the calling and receiving component name, the interface of the receiving component or the dependency name [2] of the sending component. E.g. for the authorization service: it has to be called each time a method is executed at the BasicBanking service. But this service is duplicated for load balancing as different components and has different names. By

[2] A dependency defines a required interface to be fulfilled by another component.

using the interface of the component as contextual property of the receiving component, all duplicated BasicBanking components can be captured in one pointcut.

3. Remote pointcuts that allow to evaluate over the contextual properties about the distributed location of sending an receiving component. E.g. for the load balancing service: the composition policy above uses the contextual property *hostgroup*[3] of the calling component. This scales better when other client-side components are using the BasicBanking service.

Transparent remote advice with full semantics

4. Transparent remote execution of advices, based on the deployment specification of an advising component. E.g. for the authorization service. The deployment location of that service should be transparent when defining the composition.
5. Remote before, after and around advice with full remote semantics. This allows to capture remote behaviour that is associated with calling and executing remote invocations.
   - E.g. for the validation at the authentication server of the given credentials, remote after advice is needed after the client has provided the credentials.
   - E.g. for the evaluation of the load balancing policy, remote around advice is needed: based on the load or availability of the application server it can *replace* the original call with a call to a backup server or it can just let the original call continue.

Unified aspect-component model

6. Unification of the entities. Components are aspects and aspects are components. E.g. the logger entity to log executing messages. This should be a reusable component, offering support for aspect-oriented composition in its interface. And it should be reusable and deployable in third party compositions and deployment infrastructures.
7. Unification of interaction: advices and methods are considered normal remote method invocations and are both subject to aspect-oriented composition. E.g. the remote authentication advice for validating the credentials is remotely advised by the logging component at the audit server.

State-of-the-art aspect-oriented middleware fails to define the complete range of compositions as sketched above. For instance, the contextual information provided for pointcut expressions is too limited and remote around advice with full semantics is not supported. We further discuss the shortcomings and the consequences in the related work section.

[3] A name for a group of hosts in the application, e.g. workstations.

3 DyMAC Middleware

In this section we discuss the DyMAC middleware platform. First, we briefly describe the basic concepts of the component model. Second, we explain the support for aspect-oriented composition in a distributed environment. Third, we explain how the middleware supports the concepts of the component and composition model. We explain the basic architecture of the middleware and describe the important subsystems that support distributed joinpoints, remote pointcuts and the coordination of remote advices. We also motivate the architectural decisions in the middleware that manage performance overhead. Fourth, we present a brief description of the .NET implementation of DyMAC.

The component model has been inspired by two evolutions: the first one is the evolution to distributed object-based component technology, such as EJB [8]. The second one is the evolution in AOSD towards the concept of an aspect with component semantics[10,11,15,14], and even towards a unified concept of aspect and component [19]. A pure object-based approach to the latter is currently supported for single-process applications [23]. Aspects are reduced to normal Java classes and advices are reduced to normal object methods, having a special signature. The resulting programming model offers one entity with one interaction standard. The aspect-oriented composition itself is defined in a separate specification file. These specifications define a pointcut, and which class and advice to bind. This way the advising entities become more reusable in third party compositions, an essential property for component based software development. DyMAC leverages the unified, object-based approach to aspects to the level of distributed object-based component models.

3.1 DyMAC's Basic Component Model

DyMAC components are object-based entities that separate interfaces and implementation. DyMAC components declare their required interfaces by means of dependencies. Components are composed in an application and are deployed on a distributed infrastructure. We explain these different concepts.

Components have two interfaces: a create-interface that specifies how to instantiate a component, and an instance-interface that specifies which methods are offered by a component instance. We illustrate this for the BasicBanking component in listing 1.1.[4]

The implementation of the component implements the members of the two interfaces as follows. First, it implements a constructor for each create-method specified in the create-interface. Second, it explicitly implements the instance interface. Client components implement the predefined interface IExecutable, which defines a main method, as an entry point for execution. Each component implementation also inherits from *ComponentInstance*, which binds a component to the DyMAC framework. Listing 1.2 illustrates the implementation of the BasicBanking component and the EmployeeClient component. It also illustrates

[4] The examples are implemented on the DyMAC.NET prototype and use plain C#.

Listing 1.1. BasicBanking interfaces

```
interface IBasicBankingCreate{
 IBasicBanking Create();}
interface IBasicBanking{
 void CreateAccount(string id);
 void WithDraw(string account, double amount);
 void Deposit(string account, double amount);
 void Transfer(string from, string to, double amount, string msg);}
```

Listing 1.2. Component implementations

```
class BasicBankingImpl : IBasicBanking, ComponentInstance {
 public BasicBanking(){}
 public void WithDraw(string account, double amount){...}
 ...}
class EmployeeClientImpl : IExecutable, ComponentInstance{
 public void Main(string[] args){
  ...
  //create an instance
  IBasicBanking ibb = Factory.Create("mybanking") as IBasicBanking;
  ...
  ibb.WithDraw(accountid, amount); //call an instance method
}}
```

how to create instances and how to do remote method invocations using the instance interface.

The component descriptor defines the component name, the provided interfaces, the implementation file and the dependencies of the component. A dependency is defined by a dependency name and the interfaces that are expected of the component that will be bound to the dependency. This is similar to the approach that is used in the EJB component model.

The application descriptor first defines a name for the distributed application, then it defines the set of components that is used, by referring to their descriptor. Then the compositions of the components are defined. These compositions can be normal compositions between dependencies and components, as well as aspect-oriented compositions.

Deployment descriptor. A distributed DyMAC application consists of a set of components that are deployed across a distributed infrastructure. This distributed infrastructure has a hierarchical structure: a hostgroup contains multiple hosts, (E.g. the client workstations), a host can contain multiple framework instances, which are processes. A framework instance can host multiple applications, each in an application domain. Multiple application domains are typically used on web servers and application servers to host multiple applications in one process. An application domain is a contextual unit of isolation for an application. The isolation guarantees that an application can be independently stopped. Furthermore an application cannot directly access code or resources in another application. A fault in an application cannot affect other applications. Processes with multiple application domains are only used for server processes; for the client tier of distributed applications, each application starts a different process

at the client, containing one application domain with the client tier components. For one distributed application, the deployment infrastructure consists of a set of application domains that are located on multiple processes and hosts. The deployment descriptor of an application defines those deployment locations of the components. For a remotely accessible component, this location is unique in the infrastructure and is defined by the name of a host, framework instance and application domain. A component can also be deployed as a local component, then it is deployed in every application domain of the distributed application. When components are deployed locally, then an instantiation call always creates a local instance of the component.

3.2 Support for Aspect-Oriented Composition

In DyMAC, aspect-oriented compositions are defined in the application descriptor. They consist of two main parts.

1. The component providing the aspect behaviour and its instantiation scope.
2. A set of bindings in which each binding defines a pointcut, refers to an advice-method of the advising component, and specifies an advice type (before, after or around).

We first explain the pointcut expressions, then we explain advice and the instantiation scopes of an advising component. For each concept, we emphasize how it supports AO composition in a distributed environment. Finally we illustrate AO compositions in DyMAC by means of the example from section 2.

Pointcut expressions. Pointcuts are logical expressions that evaluate over the kind and context properties of the joinpoints. The kind of the joinpoint can be restricted to calling and executing remote method invocations. The pointcut expressions to support that are a *call and execution* pointcut. Pointcuts can further evaluate over two sets of contextual properties. First, the *component-related properties* of the joinpoint model: the message signature, the dependency name of the sending component, the interface of the receiving component, and the names of the sending and receiving component and the name of the distributed application they belong to. Second, the *infrastructure-related properties* of the joinpoint model: the host names of sending and receiving component, their hostgroups, their framework and the application domain they belong to. If pointcuts do not specify a value for a certain property, it has the default value *all*. This breakdown of the contextual properties implies that pointcuts can be *remote in an implicit and explicit way.* Pointcuts evaluating on the component-related contextual properties only, are implicit remote pointcuts. They allow to refer to distributed components, without being aware (containing no information) of their distributed location. Notice that in this way the concept of a pointcut is transparently remote. Pointcuts that evaluate on the infrastructure-related contextual properties are explicitly aware of remote locations of components in the infrastructure. But, even with explicit remote pointcuts, AO compositions in applications can be made reusable in third party deployment scenarios by means of hostgroups. These are deployment-independent

groups of hosts, defined by the application. The deployment descriptor defines which hosts belong to the groups.

Advice. First we describe briefly the different types of advice that are supported in DyMAC. Second we describe the definition of advice methods in the interfaces of components. Third, we explain the implementation of advice and the joinpoint API. Last, we discuss the execution semantics of advice.

Types of advice. In DyMAC, three types of advice are supported: before, after and around. Before and after advices are respectively called after and before the call or execution of a remote method invocation. Around advice replaces the actual invocation it advises, but a *proceed* operation can be called in the advice to continue with the original remote method invocation. In case multiple advices have to be called on a certain joinpoint, the proceed call continues with the next advice in the advice chain. In case the advice is terminal in the chain, the call or execution of the original method invocation continues. After the execution of the advices later in the chain, the control flow returns to the rest of the around advice where the proceed was called.

Specifying Advice Methods. Methods, defined in the interface of a component, that are used as advice in an aspect-oriented composition, need to have a special signature. The advice can also be annotated with the kind of the joinpoint that the advice *supports*, *requires* or *prohibits* to be composed with. The possible advice kinds are BeforeCall, AfterCall, AroundCall, BeforeExection, AfterExecution and AroundExecution. Multiple *prohibits* and *supports* annotations can be defined. Only one requires annotation can be defined. These annotations are part of the interface (and thus the contract of the component [1]) because they express explicit requirements of the component when composed with other components. These need to be fulfilled to guarantee correct behaviour of the component. We specify the log method of the secure logger as an example. It only supports to be composed after an execution or call, because it needs the return message to check for exceptions.

```
[Supports(Kind.AfterExecution)]
[Supports(Kind.AfterCall)]
void Log(RuntimeJoinPoint rjp);
```

Advice implementation and the joinpoint API. The implementation of a component implements the behaviour of the advice methods specified in the interface. In this implementation the joinpoint API can be used to reflect on the current joinpoint. In DyMAC the current joinpoint is accessed using the RuntimeJoinPoint parameter of the advice. It contains information about the kind and context of the joinpoint. The joinpoint API contains contextual properties about the remote method invocation that is being advised and the calling and executing component instance of that invocation. Component properties like component name, interface and dependency names are read only. Infrastructure properties are read only too. Arguments of the method invocation can be altered. In case the joinpoint's kind is after a call or execution, the return message can also be inspected and altered. The joinpoint API also contains the proxies to the sending

and receiving component instance. Remote method invocations can be called on those remote instances out of the advices. This can be required to pull state of the component instances, that is needed in the advice, for example to evaluate an authorization policy using application-level domain knowledge. Another advantage of those proxies is that additional application-level behaviour can be called out of the advice.

Execution semantics of advice. Advice is considered a normal remote method invocation when it is called from the joinpoint context and executed at the receiving component. It is advisable like any other method invocation. The sending component of an advice is the component instance in whose context the advised joinpoint is situated. In case of a call, the sending component instance of the advice is the sending component of the invocation that currently is being advised. In case of an execution, the sending component instance of the advice is the receiving component of the invocation that currently is being advised.

Defining Distributed Instantiation Scopes. Instances of advising components are created implicitly. Aspect scopes [5,7] define the creation moment and usage scope of the instance. Typical instantiation scopes in single-process AOP systems are: per joinpoint, per class, per instance, per thread, per VM. The per instance instantiation scope, for example, means that there is one instance of the advising aspect for each object instance that is advised. For every new object a new aspect instance is created. That instance is reused for all advised method invocations on the advised object. DyMAC supports distributed instantiation scopes for components that are used to remotely advise in a distributed system, and thus includes scopes beyond process boundaries.

- Singleton : one instance in the distributed system.
- Per hostgroup : one instance per group of hosts.
- Per host : one instance per host.
- Per application domain : one instance per application domain.
- Per application : one instance per distributed application
- Per component type : one instance per component type in the distributed system
- Per component instance : one instance per component instance in the distributed system
- Per logical thread : one instance per logical thread (or distributed thread), which is used for remote control flows.

Example. We define the AO compositions of the load balancer and the secure logger in detail. They illustrate true remote around advice and after advice, but also the supported kinds of pointcuts and their evaluation on contextual properties. The load balancing composition in listing 1.3 expresses first that the LoadBalancing component is used as a singleton for this composition. The pointcut refers to all calls from the *workstations* hostgroup to the components with the *IBasicBanking* interface. They are advised by the *LoadBalancer* using the method *Balance* as around advice. Based on load and availability of the

application servers, the load balancer decides to proceed with the message or it calls one of the backup servers itself, thus replacing the original call of the client. We assume that the BasicBanking service is stateless, and therefore, all remote invocations to the instances can be redirected to another instance on another server. The location of the LoadBalancer is transparent in this composition. It is defined in a separate deployment descriptor.

The composition of the SecureLogger in listing 1.4 specifies that advice *log* is called after each execution on the authentication server. This log method uses the joinpoint API to check the return message. If it contains an exception, an authentication failure is logged with the contextual properties of the caller in the joinpoint. The binding with the authorization server is similar.

The concrete syntax of the application descriptor is XML based in the .NET implementation of DyMAC, but for readability and conciseness we use the Java configuration file syntax. The structure or abstract syntax of the composition is the same in both notations.

Listing 1.3. LoadBalancer

```
ao-composition{
 AdvisingComponent: LoadBalancer;
 Scope: Singleton;
 Binding{
  Pointcut{
   Kind: call;
   MethodMessage: * * (..);
   Caller{
    Hostgroup: workstations;}
   Callee{
    Interface: IBasicBanking;}}
  Advice{
   Kind: around;
   MethodMessage: Balance;
  }}
}
```

Listing 1.4. SecureLogger

```
ao-composition{
 AdvisingComponent: SecureLogger;
 Scope: Singleton;
 Binding{
  Pointcut{
   Kind: execution;
   MethodMessage: * *(..);
   Callee{
    Host: AuthenticationServer;}}
  Advice{
   Kind: after;
   MethodMessage: Log;
  }}
 Binding{
   //authorization logging
 }}
```

3.3 The Middleware Architecture

In this description of the middleware architecture we first describe briefly the top-level global architecture and its distributed deployment on the network. Then we focus on the essential subsystems that support the key features of the component and composition model: the aspectbinder and interception core that process the remote pointcuts at loadtime and at runtime, the distributed joinpoint architecture that supports advanced remote pointcuts with acceptable performance overhead, and the advice coordination infrastructure supporting multi-threaded, remote around-advices. In addition we discuss instance management, especially the component factory and instance registry for distributed instantiation scopes of advising components.

The top-level architecture. Each DyMAC framework can host multiple application domains for different distributed DyMAC applications. The framework middleware offers a remote interface *framework facade* to deploy, startup, stop

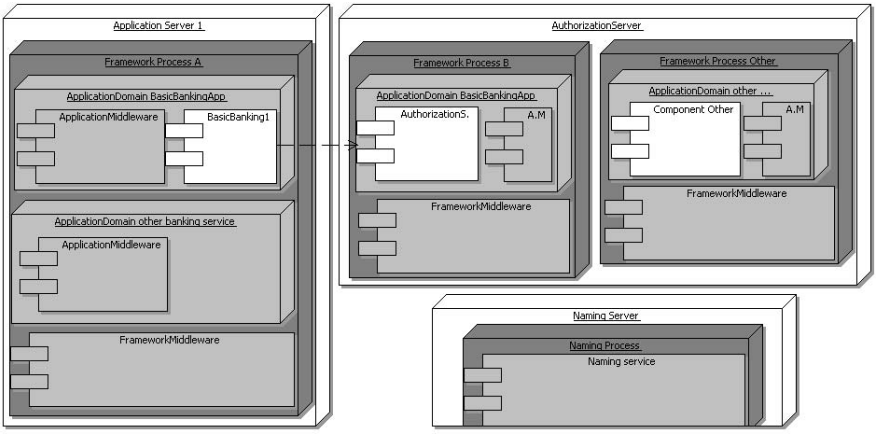

Fig. 2. DyMAC deployment view

and undeploy applications on the framework. The *deployer* distributes the application binaries and descriptors to the framework middleware to store them on the *application repository*. Then the distributed application can start up. The framework middleware first instantiates an application domain and loads the application middleware into it. The *application middleware* loads and manages the different components that are deployed in its application domain. A deployment scenario for the framework middleware, application domains, application middleware and some application components is depicted in Figure 2 . We now focus on the different subsystems of the middleware. An overview of these subsystems is depicted in Figure 3. The following subsystems are involved in the loading process. First the application descriptor is handled by the *ApplicationParser* and an *ApplicationSpec* model is built. After parsing successfully, the application domain loads the binaries and the *ApplicationVerifier* checks the ApplicationSpec model to verify if it conforms to the component model, and whether all binaries referenced are loaded. If the ApplicationSpec is sound, the application builder builds an application metamodel (*ApplicationType*). This model contains a component type for each component spec in the application spec model. Each component type contains a list of dependencies with a list of method definitions. The component type also contains a list of provided method definitions. In this step of the loading process all method definitions have an empty advice chain. The *AspectBinder* then handles the bindings defined in the AO compositions.

The AspectBinder. We first explain the common approach for call-pointcuts as well as execution-pointcuts in the bindings. Then we refine the explanation for each kind of pointcut. *For each binding,* the properties of the pointcut that are known at loadtime are evaluated. If the loadtime known properties of a method definition match, an *advice thunk* is inserted into the advice chain of the method. Advice thunks define a set of properties to evaluate at runtime (the *pointcut residue*[6]) and an advice method to be called when the runtime properties

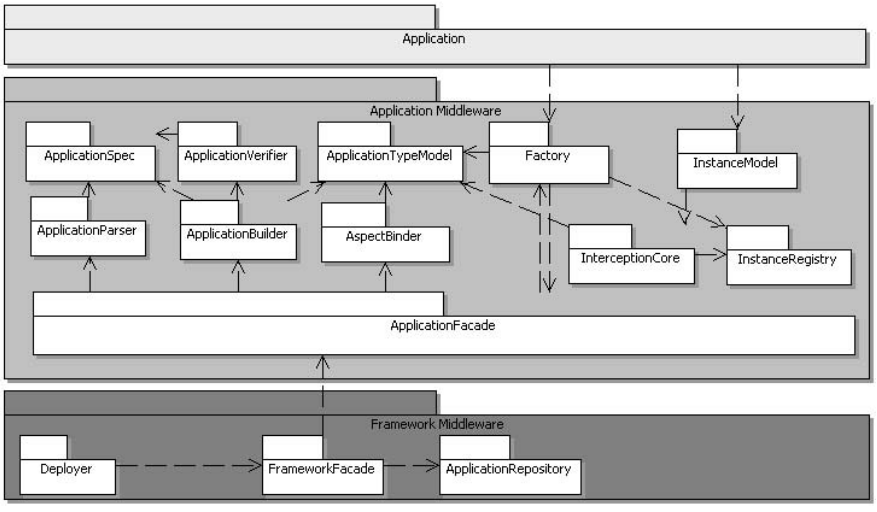

Fig. 3. DyMAC subsystem view

evaluate to true. *In case of an execution pointcut,* the method signature and the properties of the callee in the pointcut are evaluated at loadtime. The properties of the caller can only be evaluated at runtime when a method message arrives at the component. *In case of a call pointcut,* the method signature and the properties of the caller in the pointcut are evaluated at loadtime. The properties of the callee are evaluated at runtime when the message is sent to the component instance that is bound to the dependency. This binding can be different at runtime because of possible runtime changes to the component satisfying the dependency, such as its location.

Once the application is running, the *interception core* processes the remote invocations between components. This interception core has two important services. The first one is the distributed joinpoint infrastructure, that manages the runtime representations of distributed joinpoints. The second service is the remote advice coordinator. This service selects and evaluates the advice thunks for a joinpoint, iterates the resulting advices and handles the execution of them.

The distributed joinpoint infrastructure. Joinpoints in DyMAC contain runtime information about calling or executing method invocations between component instances in the distributed infrastructure. Four different kinds of joinpoints are distinguished at runtime : before a call, before an execution, after an execution and after a call. The distributed joinpoint infrastructure creates a runtime representation of these joinpoints, that *localizes all information* that is needed to select and evaluate the advice thunk.

Before a remote invocation is called a before-call joinpoint is constructed using the following information: (1) the contextual properties of the sending component, which are locally stored in the component type model, and (2) the contextual properties of the receiving component, stored in the proxy to the receiving

component instance. The properties of the sending component are added to the *call context* of the method message. The call context containing the caller properties is serialized as a piggy back on the remote invocation. *Before a remote invocation is executed* at the destination component, a before-execution joinpoint is created. The call context of the message is deserialized and the caller's contextual properties are added to the execution-joinpoint. The callee's contextual properties are stored local in the typemodel of the receiving application middleware platform and are added to the joinpoint *After the execution* the kind of the before-execution joinpoint is changed to after-execution, and the return message is added to the joinpoint. *After the call,* the call-joinpoint's kind is changed to after-call and the received return message is added to the joinpoint.

Architectural decisions about the management of the contextual properties are incorporated to avoid chattiness. Chattiness could have occurred when the information that is remote to the location of a joinpoint is pulled from the remote host *by need*. This could have occurred during call-joinpoint evaluation, when information about the callee is needed and during execution-joinpoint evaluation when information is needed about the caller. It could have occurred when contextual information about caller and callee is accessed using the joinpoint API at the execution location of the remote advice.

Therefore, when a component is instantiated, its contextual properties are stored in its proxy. Every remote client of the component receives the contextual properties along with the proxy. This does involve an initial transport overhead when the proxy is created, but the properties of the callee are always local for the call-joinpoint. To achieve locality of the caller-properties for an execution-joinpoint, the properties of the caller are added as a piggy back on the remote message to the callee. This omits a call-back of the callee to the caller to query contextual information during runtime evaluation. When the remote advice methods access the contextual properties of the joinpoint parameter, that information is local. The joinpoint object is a composite value object, and therefor a complete local copy is available in the execution context of remote advice methods.

The remote advice coordination infrastructure. The interception core of DyMAC coordinates remote advice with the advice handler. The advice handler of DyMAC is instantiated when advice needs to be executed for a certain joinpoint. The list of advices matching the joinpoint is handed over to the new advice handler instance. In case the advice chain contains a remote around advice, the advice coordinator is a remotely accessible instance. This remote access is necessary to give back the control flow in case a *proceed* is called in the remote around advice. In case the advice bindings do not contain remote around advices, the advice handler instance is a local object. This avoids expensive instantiation of a remotely accessible advice handler. In case the advice is local around advice, the proceed call to the advice handler is also local, and no remote advice handler is needed. In case the advice is remote before advice or remote after advice, the control flow returns to the local advice handler automatically after the execution of advice. We show the message flow for the execution of the withdraw operation in figure 4. Only the around advice of the load balancer and the AdviceHandler at the client are illustrated.

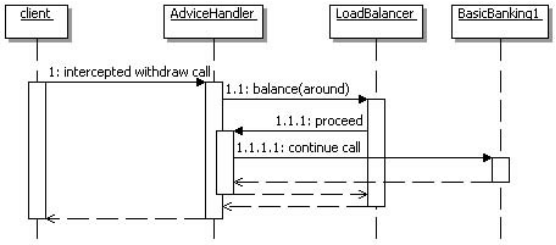

Fig. 4. Remote AdviceHandler for Remote Around Advice

Component Factory. Instantiation of components is supported by the DyMAC factory. This service provides a create-operation with as first argument a dependency name and then a variable number of arguments. When the create-operation is called, the factory looks up the dependency name, binds a component to it and creates an instance of the component, using the appropriate constructor in the implementation. This instance can be remote, depending on the deployment location of the component. The remote interface of the application middleware containing the component is called to create the remote instance.

Distributed instance registry. The instance of an advising component is implicitly created when it is needed in an AO composition. It is registered in the aspect registry at the deployment location of the component. If a new instantiation request arrives at the instance registry, the registry checks if there is already an instance bound to the requested instantiation context. This can be deducted from the caller properties that are a piggyback on the instantiation call. If such an instance exists, a proxy to the existing instance is returned to the requesting advice binding.

The application middleware of an application domain has a local cache of the aspect registry. It contains the proxies to the remote instances that are related to the application domain. This avoids expensive remote lookups before advice is executed, and also reduces chattiness. Concretely, the application middleware has a hash-based cache structure that is divided into substructures for the different instantiation scopes : group, host, framework, application domain, distributed application, component type and component instance.

For the logical thread instantiation scope, DyMAC supports another optimization for the distribution of the instance proxy along the distributed call flow. The proxy is piggy backed with a method message whenever the call flow is transferred to the next application middleware instance. This again avoids a lot of chattiness. Chattiness could have occurred due to lookups for the remote component instance handling the advice. In fact, these lookups would be performed at every application domain.

3.4 Prototype Implementation

The DyMAC middleware platform has been prototyped on .NET 2.0. It is implemented as a framework and does not involve any language extension. Component

interfaces and implementation can be defined in any CTS/CLS compliant language. Remote method invocations are normal method calls and do not involve any meta object protocol as in some other AO middleware approaches [15]. This implies that components and their interactions are statically verified by a production .NET compiler.

The InterceptionCore in the middleware is built on the Context Bound Object technology to intercept remote messages. This interception happens in the .NET CLR. The CLR then activates the DyMAC message sink of the InterceptionCore, which creates a runtime joinpoint and starts the DyMAC AdviceCombiner. This way of dynamic interception does not require byte code weaving on components and does not require the CLR to run in debug mode.

The remote method invocations between component instances are implemented on .NET remoting. Piggy backs added by the DyMAC middleware are stored in the *call context* of a remote message. This is a hashtable associated with the call flow and .NET remoting serializes this information along with the remote method invocation.

4 Evaluation

We evaluate the runtime overhead introduced by the middleware platform to support the features of the component model. This runtime overhead is evaluated in terms of three kinds of resource usage: increased data access, increased network usage and increased usage of computation resources. We compare the runtime overhead of a DyMAC based application with an application based on state-of-the-art distributed component technology. For this comparison we use the .NET implementation of DyMAC. The distributed aspect-component model of DyMAC.NET is built on top of .NET remoting. So the first version of the application has been built on DyMAC, the second version of the application uses only the .NET remoting infrastructure.

Performance analysis. The runtime overhead of the DyMAC is evaluated in terms of increased data access, increased number of remote messages, increased size of remote messages, and increased usage of computation resources. DyMAC does not introduce additional data access at runtime. The application descriptors and configuration files are loaded at startup time of the application. So in case data access is involved in an application operation, that latency is the main performance bottle neck, because it is orders of magnitude larger than the networking and computing overhead. If we ignore data access, then the network overhead is the next important performance overhead. We evaluate network overhead in the next paragraph in detail. The overhead of calculating advice activation caused by the middleware layer is neglectable in comparison with the latencies of the network access. However, when using the framework for pure single-process, single-user AOP applications, without network communication or data access, the framework's overhead is significant. Offering features like client-specific middleware extensions and dynamic adaptability does involve a large computation overhead, due to the use of runtime interception, reflection

and access of data-structures containing component metadata. That overhead is acceptable for large-scale distributed enterprise applications, and even unavoidable for the features required in that domain. [5]

Network overhead. The network overhead of DyMAC has to be evaluated on two properties : the number of additional messages involved, and the increased message size. If a remote before or after advice is used in stead of an ordinary method message, the number of messages stays the same. If an around advice is used with a proceed call, the proceed call seems to involve an additional remote call: i.e. the call-back to the advice coordinator. But, implementing the behaviour of around advice in the pure .NET remoting environment also involves two normal method messages: one to query if the remote message should be executed, and then another message to execute the behaviour that normally would come after the proceed call. The second element to compare is the overhead caused by increased message sizes. A method message in DyMAC carries additional information about the caller in its call context. This causes overhead in the transmission time. The parameter of advising method messages in DyMAC is a value copy of the joinpoint object, which contains kind and distributed context information. This also introduces a transmission overhead. To measure the network overhead, we compare the execution time of four kind of messages:

1. A pure .NET remoting method message with empty body, no parameters and no return value. We call this a minimal method message.
2. A DyMAC method message with the caller properties piggybacked.
3. A DyMAC advising method message, with caller properties piggybacked, but without the joinpoint object (the parameter is null).
4. A DyMAC advising method message with the joinpoint object.

First, the messages are sent between two processes on the localhost and second, between two different hosts on a 100MBit network. The first test simulates an optimal network. The second one compares messages in a real-life network. Timing started after 10 calls, to avoid delays by the .NET JIT compiler, because that would scale down the overhead. The timing results for each 1000 executions are:

```
Localhost:              100 Mbit:
1: 1.1093750 sec        1: 1.3785850 sec
2: 1.1875000 sec        2: 1.4935000 sec
3: 1.1875000 sec        3: 1.5175150 sec
4: 1.4075000 sec        4: 1.7695000 sec
```

The caller properties add a 6 to 7 % overhead maximally, on a minimal method message. The runtime joinpoint object adds another 18-20 % compared to the minimal method message. These overheads are calculated for minimal method messages and thus are an upper limit. As the method message gets more or

[5] For AOP systems focusing on single-user, single-process applications, different requirements exist. It is one of the key features to minimize overhead of activation of advice added by aspects. The application domain of one of the first AOP papers was the implementation of computation intensive algorithms for image processing[4]. In that domain, the performance overhead of activating added advice is crucial. Appropriate AOP tools, focusing on optimal advice weaving should be used then.

larger parameters, the relative overhead gets substantially smaller. In distributed enterprise applications, the facade pattern [8] is applied for remote components. In this pattern, remote messages have more parameters, and also have larger value objects as parameters or as return value. The average overhead will be smaller in that type of applications.

Based on our initial measurements, we claim that the runtime overhead of DyMAC is acceptable in a distributed application. We have illustrated that the penalty of fully distributed aspect-oriented middleware can be limited to network (message size) overhead as no additional messages are required to support remote pointcuts and remote advices. Recall that the figures presented above present an upper limit. Moreover we are working on optimizations that leverage pro-active distribution of useful context information, again without additional messages.

5 Related Work

This section focuses on existing AO middleware technologies that support to a certain extent distributed aspects with a notion of remote pointcut and remote advice: JAC[9], CAM/DAOP[15,16] and AWED[18].

JAC (Java Aspect Components [9]) is a Java-based framework that offers an aspect model to advice objects locally. Using this aspect model, a lot of internal middleware services of the framework itself are developed as aspects on the framework. The framework also supports distribution of components and distributed deployment. Both services are even implemented as aspects on the framework. JAC simulates the semantics of remote advice by executing local advice on a local copy of the aspect (aspects are replicated on each host). The states of the aspect instances are synchronized at each state change. The example from section 2 cannot be modeled using JAC. For instance, the security of the authentication service would be broken by duplicating private keys. Moreover, the workaround in JAC causes a lot of extra communication (chattiness). Finally, JAC has a limited join point model when it comes to evaluating distributed context information.

CAM/DAOP[15,16] is a framework for distributed applications that offers aspect-components and regular distributed components supporting broadcasting, events, synchronous and asynchronous messages. The aspect-components can offer remote before and after advice on sending and receiving messages. The CAM component model supports defining provided and required interfaces for components and aspect-components. DAOP does not offer remote around advice. The example from section 2 needs remote around advice for security services and for load balancing. The security services could be simulated with critical aspects, in which the before advice has to evaluate to true in order to let the message continue. But for the load balancing service this is not a solution. DAOP also has very limited pointcut expressions that do not allow to evaluate on contextual properties apart from the sending and receiving component role. Support for context properties concerning the distributed location or other component-related properties like interfaces are not supported.

AWED[18] is a language for distributed AOP and offers explicit remote pointcuts and explicit remote advice. The pointcut language offers the keyword host() to evaluate on the host of the joinpoint. Remote advice is explicitly remote using the on() keyword to specify the host on which the advice should execute. AWED's approach to distributed aspects and remote around advice does not offer the abstraction of around advice in a fully distributed way. The proceed statement in the remote around advice has local semantics. The original intercepted message is executed at the host where the around advice is executed. This approach only works if the destination component is a static class or singleton deployed in every VM. In the example of section 2, the effect of the proceed call is that the messages to the application server are executed on the reverse-proxy server. AWED deploys each aspect and each class on every host to realize this. AWED also doesn't support transparent execution of remote advice: the host on which the advice should be executed is explicitly stated in the pointcut using the on() construct. Defining the deployment location of the advising component in the pointcut destroys the separation of composition and deployment. This also mixes up the separation of the specification of pointcut and advice. A pointcut should express the events in the system that need to be advised. The host of the advice itself is not part of that, but part of the advice's deployment specification.

Other approaches have been proposed to integrate AOP into middleware, but without integrating the component model, only the class model of the programming language: JBOSS AOP[12,21], AspectWerkz[23], Spring AOP[22]. This design choice is clearly reflected in the kind and context of joinpoints that can be specified in pointcut expressions. The above mentioned approaches do not offer true support for distributed aspect composition anyway: they do not support a distributed joinpoint model, remote pointcuts or remote advice. AspectJ2EE[20] however, has the same approach offering a local AOP framework on Java classes, but with the difference that it offers one special kind of pointcut: remotecall, to advise remote calls from clients to EJBs. The use of contextual properties about location or components is not supported.

6 Conclusion

Complex compositions cannot be expressed effectively in state-of-the-art middleware and AOSD is a promising technology that can assist in improving the situation. In this paper we presented DyMAC middleware. The AO middleware platform offers true and transparent distributed composition by means of advanced remote pointcuts that can evaluate on distributed context, transparently remote advice with full semantics and a unified distributed component model. To our knowledge, this is the first middleware architecture that transparently and completely extends the power of aspect composition (join point model and advice execution) in a distributed context. We have prototyped DyMAC in a .NET environment and initial benchmarks show promising performance results.

References

1. C. Szyperski. Component software: beyond object-oriented programming. Second Edition. ACM Press/Addison-Wesley.
2. G. Heineman & W. Councill. Component-based Software Engineering. Addison-Wesley.
3. G. Blair, G. Coulson, et al. The design and implementation of OpenORB version 2. IEEE Distributed Systems Online Journal, 2(6), 2001
4. G. Kiczales. Aspect-Oriented Programming. In Proc. ECOOP'97.
5. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold. An Overview of AspectJ. In Proc. ECOOP'01.
6. E. Hilsdale & J. Hugunin. Advice Weaving in AspectJ. In proc. AOSD'04.
7. R. Filman, T. Elrad, S. Clarke & M. Aksit. Aspect-Oriented Software Development. Addison-Wesley, 2004.
8. Sun Microsystems. Inc. Enterprise Java-Beans (EJB) Specification v2.0, 2001.
9. R. Pawlak, L. Seinturier, L. Duchien & G. Florin. JAC: A Flexible Solution for Aspect-oriented Programming in Java. In Proc. Reflection'01.
10. D. Suvée, W. Vanderperren & V. Jonckers. JAsCo: An aspect-oriented approach tailored for component-based software development. In Proc. AOSD'03.
11. M. Mezini & K. Ostermann. Conquering Aspects with Caesar. In Proc. AOSD'03.
12. M. Fleury & F. Reverbel. The JBoss extensible server. In Proc. Middleware 2003.
13. E. Truyen, B. Vanhaute, B.N. Jorgensen, W. Joosen & P. Verbaeten. Dynamic and Selective Combination of Extensions in Component-Based Applications. In Proc. ICSE'01.
14. B. Lagaisse & W. Joosen. Component-Based Open Middleware Supporting Aspect-Oriented Software Composition. In Proc. CBSE'05.
15. M. Pinto, L. Fuentes & J.M. Troya. A Dynamic Component and Aspect-Oriented Platform. The Computer Journal,2005.
16. M. Pinto, L. Fuentes, M.E. Fayad & J.M. Troya. Separation of coordination in a dynamic aspect oriented framework. In Proc. AOSD'02.
17. M. Nishizawa, S. Chiba & M. Tatsubori. Remote pointcut: a language construct for distributed AOP. In Proc. AOSD'04.
18. L.D.B. Navarro, M. Südholt, W. Vanderperren, B. De Fraine & D. Suvée. Explicitly distributed AOP using AWED. In Proc. AOSD'06.
19. D. Suvée, W. Vanderperren, D. Wagelaar & V. Jonckers. There are no Aspects. In Proc. Software Composition 2004.
20. T. Cohen & J.Y. Gil. AspectJ2EE = AOP + J2EE: Towards an Aspect Based, Programmable and Extensible Middleware Framework. In Proc. ECOOP'04.
21. JBoss AOP homepage, http://labs.jboss.com/jbossaop
22. Spring Framework website. http://www.springframework.org/
23. AspectWerkz homepage, http://aspectwerkz.codehaus.org/