

Support for Distributed Adaptations in Aspect-Oriented Middleware

Eddy Truyen Nico Janssens Frans Sanen Wouter Joosen

DistriNet, Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven, Belgium

{eddy.truyen,nico.janssens,frans.sanen,wouter.joosen}@cs.kuleuven.be

Abstract

Many aspect-oriented middleware platforms support run-time aspect weaving, but do not support coordinating distributed changes to a set of aspects at run-time. A distributed change entails weaving or unweaving multiple inter-dependent aspects that are logically or physically distributed. Coordinating such multiple weavings inside the application layer is a complex and difficult task for the application developer, because global state consistency, structural integrity and other safety properties have to be preserved. In this paper, we present the DyReS framework that offers the required coordination support on top of existing aspect-oriented middleware platforms. The framework is customizable towards application-specific requirements to achieve improved performance and reconfiguration semantics. We have validated our approach by delivering and examining two implementations of the DyReS framework: one on top of JBoss AOP and a second one for Spring AOP.

Categories and Subject Descriptors C.2.4 [Computer-Communication Networks]: Distributed Systems—Distributed Applications; D.2.11 [Software Engineering]: Software Architectures—Domain-specific architectures; D.2.13 [Software Engineering]: Reusable Software—Reusable libraries

General Terms Management, Experimentation

Keywords Distributed aspects, run-time aspect weaving, dynamic reconfiguration, DyRes, Spring AOP, JBoss AOP.

1. Introduction

Distributed applications are typically built on middleware that offers a component model and execution environment for these applications. Practical middleware platforms nowadays have to support complex compositions of application components and have to support a broad range of services that deal with non-functional concerns in a distributed application. Aspect-Oriented Middleware (AOM) has contributed to improving the modularization of such complex applications, by supporting an aspect-component model that offers aspect-oriented composition alongside traditional composition of provided and required interfaces [28, 10]. The core

concept in aspect-oriented composition is the aspect [17]: a coherent abstraction that encapsulates one specific (often crosscutting) concern in a separate software module. An aspect defines behavior that can be executed and defines composition logic to describe where and when this behavior should be executed. AOM typically separates aspect behavior and composition logic from each other. The aspect behavior is represented within traditional components as methods whereas the composition logic is specified as a set of advice bindings.

To accommodate the increasing need for dynamically adaptive software, most AO frameworks support compositional adaptation [21] at run-time. As a result, distributed applications and middleware can perform dynamic changes to their composition of aspects and components. Known examples of middleware and application-specific functionality that benefit from this run-time compositional adaptation include caching [12], translation between networking protocols [6], transaction support [18], security policies [33], self-optimization [8], and message compression and fragmentation functionalities to deal with quality-of-service variations in the network [7]. Many of these adaptations in aspect-oriented middleware are distributed - either logically or physically. For example, adding message fragmentation involves adding multiple inter-dependent advice bindings: one that weaves in a component for fragmenting outgoing messages, and one that weaves in a component for reassembling incoming fragments. An example of a logically distributed change - in fact executed on a single node - is the replacement of an authorization aspect with a more advanced authorization strategy, where the latter depends on another aspect for producing specific authentication tokens inside the application layer.

Managing such run-time changes to ensure system-wide consistency, structural integrity and other safety properties is a complex task that consists of multiple phases: specifying change, verifying change, and eventually implementing change [27]. This paper addresses a specific issue with change implementation: how to implement a distributed change to a set of aspects in a coordinated way such that two safety properties - structural integrity and global state consistency - are preserved.

State-of-the-art AOMs lack support for preserving both of these safety properties. In particular, hardly any attention has been given to removing distributed aspects in a safe way. First, if the unweaving of multiple inter-dependent aspects is not performed in a coordinated way, each aspect is independently unwoven. This may lead to a situation where *structural integrity* is compromised: the one aspect is still woven, but the other aspect is not. Second, components containing the aspect behavior may maintain state and other components may depend on this state. For example, the aspect for fragmenting messages typically maintains a buffer for messages wait-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD'08, March 31 - April 4, 2008, Brussels, Belgium.
Copyright © 2008 ACM 978-1-60558-044-9/08/0003...\$5.00

ing to be fragmented. If this component is removed without taking into account this state, *global state consistency* may be compromised.

Additional coordination protocols are therefore necessary to enforce that distributed adaptations are performed in a safe way. Implementing these coordination protocols inside the application layer, however, can be complex and error prone if the application developer is not an expert in carrying out safe distributed reconfigurations. Moreover, implementing these protocols in the application layer intertangles software-adaptive concerns with the functional concerns of the application. This obstructs the independent analysis and evolution of these concerns.

This paper presents the DyReS framework¹ that provides the necessary coordination support. As a proof of concept, the DyReS framework has been implemented on top of the JBoss AOP [1] and Spring AOP [2] frameworks.

This paper makes two contributions. First of all, DyReS offers various reconfiguration operations for (1) forcing components and aspects to reach a safe state, (2) adding, removing and replacing aspects, and (3) coordinating the execution of these actions on different nodes. The framework implements these reconfiguration operations on top of the underlying middleware using aspect-oriented and object-oriented programming techniques. Some reconfiguration operations are implemented fully by the framework, whereas the implementation of other reconfiguration operations are abstract aspects or classes that must be specialized for the concrete application. By means of a domain-specific scripting language, the application developer can then implement a distributed adaptation in a separate adaptation script that executes the reconfiguration operations in a particular order. This approach allows the developer to reason about a distributed adaptation without being bothered by the underlying implementation details. This should reduce the complexity inherent in defining correct distributed adaptations.

Secondly, we observed that existing dynamic reconfiguration frameworks typically support one generic coordination protocol for implementing distribution adaptations [4, 23, 19]. The scripting language of DyReS, on the contrary, enables the user to develop customized coordination protocols that are tailored to application-specific requirements and thus perform more efficiently. To prove the relevance of supporting such customizations, we have compared different ways of implementing a distributed adaptation in a concrete application. This will show that a customized coordination protocol generates much less performance overhead than a generic coordination protocol.

The remainder of this paper is structured as follows. Section 2 discusses a detailed problem statement by means of a case study. Section 3 presents the DyReS framework and discusses how the framework supports different strategies for implementing a distributed adaptation. Section 4 illustrates how DyReS supports the creation of custom-made adaptation scripts for different adaptation scenarios. Section 5 then presents our quantitative analysis to prove the relevance of supporting these customizations. Section 6 discusses related work. We conclude in Section 7.

2. Refined problem statement

In this section we refine the problem statement of this paper by means of a case study. Subsequently we overview the key requirements that DyReS must support. Finally, we present our analysis that state-of-the-art AOMs lack support for coordinating distributed changes to a set of aspects.

¹DyReS stands for Dynamic Reconfiguration Support.

2.1 Case study

Our running example is based on an existing case study on dynamic adaptation which we have adapted for our purposes. The case study concerns a multi-protocol instant messaging service (IMS) for personal communication on a variety of protocols such as Web PC-based Internet chat, Short Message Service (SMS), etc. [35]. Figure 1 shows an overview of the IMS architecture. The distributed execution environment of the service consists of a typical three-tier architecture. End users access the service through a facade which redirects all client traffic to several replicas of the IM server components. These replicas operate on a set of middle-tier nodes. Some replicas provide additional facilities which handle access to the service through specific protocols such as SMS. All replicas share a common relational database that allows replicas to operate as a collective service in an undifferentiated way, supporting single sign-on, session migration from one user device to another, etc.

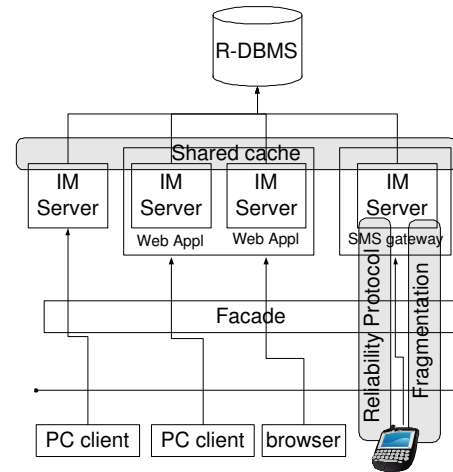


Figure 1. Overview of the architecture of the IMS platform.

2.1.1 Application-specific requirements

The goal of the case study is to use dynamic adaptation for achieving service optimization with respect to the overall Quality-of-Service perceived by the end users. The QoS requirements we focus on in this case study include *availability*, *responsiveness* and *reliability* of the communication service. High availability means that end users should experience no temporal service disruptions. The overall experienced responsiveness of the platform should also be kept permanently at an acceptable level. The third requirement is about dealing with omission failures in the network links: every message should be sent only once and message ordering in the scope of a user session should be guaranteed.

Besides service optimization, it is also important that the IMS platform can dynamically evolve to new or changing requirements without bringing the platform off-line. For example, security risks will increase when e-trading is supported by an extended set of services. Security measures such as encryption and authentication will have to be dynamically added then to the IMS platform.

2.1.2 Adaptation scenarios

We consider three crosscutting concerns (caching, message fragmentation and reliability) in the IMS architecture that are subject to dynamic adaptation (see Figure 1). We consider these concerns in two adaptation scenarios.

In a first scenario, the network link between the middle-tier IMS components and the database becomes overloaded. This has a

negative effect on the overall responsiveness of the IMS platform. In this case, all IMS replicas may be augmented with a distributed cache service. Data retrieved from a query to the database is then stored locally in this cache. The cache is implemented as an aspect that has an instance at every IMS replica. Data updates are kept synchronized between the cache instances through a two-phase commit protocol. Data updates are not immediately stored in the database, but they are sent in block to the database².

Another adaptation scenario is when the SMS network becomes overloaded. This may lead to massive packet loss and re-ordering. In this case a reliability and message fragmentation service may be added on top of the SMS protocol. The reliability service consists of a retransmission component at the PDA client and a message ordering component at the SMS gateway. The fragmentation service splits large messages into smaller ones so less bandwidth is consumed in the case of potential re-sends. Its implementation consists of a fragmenting component on the PDA client and a reassembling component on the SMS gateway.

These adaptations should only be deployed under certain conditions in the network however, hence the need for dynamic adaptation. For example, when the load of the network link to the database does not exceed a certain threshold, the responsiveness of the IMS platform is worse with the distributed cache in place than if the IMS components directly send SQL queries to the database. This is because the management of the distributed cache also introduces overhead. Therefore, if the distributed cache was already deployed in this situation, it should be dynamically removed.

An important complexity arises with the issue of *how to implement this dynamic removal process* because this also introduces performance overhead in some way. The problem is that this performance overhead may negatively effect or even conflict with the important QoS requirements of the IMS platform. For example, when removing the distributed cache, it must first be synchronized with the database. This may make the data unavailable for a substantial amount of time, which has a negative effect on the overall availability and responsiveness of the IMS platform. Similarly, removing the fragmentation service from the SMS protocol should not lead to a sudden drop in responsiveness.

2.2 Key requirements

How to implement a distributed adaptation has been researched extensively in the field of dynamic reconfiguration of component-based distributed systems. Goudarzi, for instance, describes in [23] that a dynamic software reconfiguration yields a correct system if after completing the reconfiguration process:

1. the system satisfies its *structural integrity* requirements,
2. the affected entities in the system are in a *reconfiguration-safe execution state*, and
3. the *application state invariants* hold.

We briefly explain by means of a pedagogical example (i.e., removal of the fragmentation service) how these three safety properties apply to run-time aspect weaving. Assume that in the IMS platform at some point of time the load of the SMS network has exceeded a certain threshold. As a result, the fragmentation service has been woven across the PDA client and the SMS gateway. After some time, however, the load of the SMS network drops to a normal level again, and therefore, decision logic triggers the removal of the fragmentation service.

In JBoss AOP and Spring AOP, fragmentation would be implemented by means of two interceptors: a fragmenting intercept-

²Notice that the two-phase commit protocol is not as a bottleneck as the database because the access pattern on the database is dominated by reads and not by writes.

```
Aspect classes:
public class MessageFragmenter
    implements MethodInterceptor {
    protected Thread thread;

    public MessageFragmenter() {
        queue = new LinkedList<MethodInvocation>();
        thread = new Thread(this);
        thread.start();
    }

    public Object continueInvocation(MethodInvocation mi)
    throws Throwable {queue.add(mi); }

    public synchronized void run() {
        while (running) {
            if (!queue.isEmpty()) {
                MethodInvocation mi = queue.removeFirst();
                MethodInvocation[] fragments = fragment(mi);
                for (int i=0; i< fragments.length; i++)
                    fragments[i].proceed();
            } }
        ...
    }
}
public class MessageReassembler
    implements MethodInterceptor { ... }
```

```
./springconfig.xml:
<!-- declaring application components as beans -->
<bean id="stub" class="Stub" .../>
...
<!-- declaring Messagefragmenter as a bean -->
<bean id="fragmenter" class="MessageFragmenter"/>
<!-- declaring advice binding -->
<bean id="fragmentingBinding"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="advice">
        <ref local="fragmenter"/> </property>
    <property name="patterns">
        <value>.send.*</value> </property>
    </bean>
<!-- create proxies to apply advice -->
<bean class="org.springframework.aop.support.BeanNameAutoProxyCreator">
    <property name="beanNames">
        <value>stub</value> </property>
    <property name="interceptorNames">
        <value>fragmentingBinding</value> </property>
    </bean>
```

```
Client main program:
public static void main(String[] args) {
    //Loading the client application.
    //MessageFragmenter gets woven because
    //it is specified in the springconfig.xml file
    ApplicationContext ctx = new
        FileSystemXmlApplicationContext("springconfig.xml");
    ...
    //dynamically removing the MessageFragmenter
    Advised advised = (Advised) ctx.getBean("stub");
    Advisor advisor =
        (Advisor) ctx.getBean("fragmentingBinding");
    advised.removeAdvisor(advisor);
    ...
}
```

Figure 2. The top box illustrates the implementation of the fragmenting aspect in Spring AOP. A configuration file (shown in the middle box) specifies how the aspects and application components are composed. The third box at the bottom illustrates how the client application is started up at the PDA. At some point of time, the client application decides to unbind the fragmenting interceptor. This unbinding must be coordinated however with the unbinding of the reassembling interceptor at the SMS gateway.

tor at the PDA client and a reassembling interceptor at the SMS gateway. These interceptors mutually depend on each other in the sense that they implicitly interact with each other based on a well-defined collaboration. Figure 2 gives a detailed overview of part of the implementation of this collaboration in Spring AOP. For each text message to be fragmented, the fragmenting interceptor starts up a new instance of the collaboration. To complete processing a collaboration, both interceptors have to maintain state – e.g., the reassembling interceptor maintains a queue of fragments waiting to be reassembled into a full text message. In the rest of this example, we use the term *collaboration instance* to refer to the complete processing of *one* text message by the fragmentation service.

2.2.1 Structural integrity

A first requirement for safe dynamic software reconfiguration relates to the system’s software structure: after completing a reconfiguration, the system must still satisfy its structural integrity requirements. These structural integrity requirements constrain the structure of a system in terms of the relationships between collaborating components and the ways in which these modules must be put together [4].

For distributed changes to sets of aspects, preserving structural integrity involves respecting all causal dependencies in the course of ongoing collaboration instances. Unweaving the reassembling interceptor at the SMS gateway, while the fragmenting interceptor is still woven at the PDA client, for instance, obviously breaks the causality between the fragmenting and reassembling interceptor. This, in turn, compromises the correct functioning of the overall IMS platform.

2.2.2 Safe state

A second requirement for safe dynamic software reconfiguration relates to the execution state of the system. After completing a reconfiguration, the whole system must be left in a state which allows normal operation as if no reconfiguration had occurred.

The approach we currently use to reach such a safe state is based on Kramer and Magee’s definition of “quiescence”. In [19], Kramer and Magee have indicated that software modules reach a reconfiguration-safe state when they are both *consistent* and *frozen*. If software modules are consistent, they do not contain results of partially completed collaboration instances. By freezing software modules, new collaboration instances cannot start to execute and thus cannot cause state changes. Kramer and Magee define this consistent and frozen state as the *quiescence* of a software module.

If we apply this definition to the two fragmentation interceptors, then quiescence comes about when (1) the fragmenting interceptor is prevented from being invoked to fragment new text messages, and (2) all fragmented text messages in transit have been reassembled in their original form. If both conditions are fulfilled, the fragmenting and reassembling interceptors can be safely removed without compromising the correct functioning of the system.

2.2.3 Application state-invariants

A last requirement for safe dynamic software reconfiguration relates to the application state-invariants. These invariants define the predicates for a reconfiguration to be legal, each expressed over the state of (a subset of) the components in the system [23]. A typical example to illustrate this involves the replacement of a fragmenting interceptor that generates unique ID’s for each fragment. To preserve this invariant, the new fragmenting interceptor must be initialized in a state which prevents it from producing ID’s that were already generated by the old module. Optionally, this may require transferring the old interceptor’s state towards the new one.

2.3 Limitations of existing AO frameworks

As indicated in Figure 2, Spring AOP offers an API for adding and removing advice bindings at run-time. This API is local however in the sense that each Spring AOP instance only maintains information about interceptors that are deployed in the local VM. Furthermore, Spring AOP does not provide any coordination mechanisms when multiple inter-dependent interceptors must be woven together. JBoss AOP lacks support in the same way. The philosophy of the AO frameworks is that it is the responsibility of the application developer to correctly coordinate the execution of weaving and unweaving actions.

3. The DyReS framework

This section gives an overview of the DyReS framework and its architecture.

3.1 Overview of our approach

DyReS extends existing aspect-oriented frameworks with run-time support for implementing distributed adaptations in a more easy way. In DyReS, the implementation of a distributed adaptation is structured as a sequence of reconfiguration tasks, where each task consists of one or more reconfiguration actions. For example, in the above case study, replacing an already deployed MessageFragmenter interceptor with another MessageFragmenter interceptor typically involves the following reconfiguration tasks:

Installation: Creation of a new interceptor instance.

Finishing: Shutting down the old interceptor by first blocking messages, then waiting till the old interceptor reaches a safe state.

Activation: Performing the reconfiguration itself by unbinding the old interceptor and binding the new interceptor. Then, resuming the execution by unblocking the queued messages.

Removal: Removal of the old interceptor instance.

When implementing a distributed adaptation to a set of aspects, reconfiguration tasks at different nodes also have to coordinate with each other. For example, the MessageReassembler interceptor at the SMS gateway can only be finished after the corresponding MessageFragmenter interceptor at the PDA client has been finished. This is because the MessageFragmenter depends on the MessageReassembler for completing ongoing fragmentation processing of messages.

DyReS offers reconfiguration support for implementing and coordinating the above reconfiguration tasks at a certain level of abstraction. This is achieved by offering a generic set of *reconfiguration operations* and *synchronization primitives* that encapsulate certain details about the implementation of the reconfiguration tasks. A developer can use the reconfiguration operations and primitives to implement a custom-made distributed adaptation without being bothered by underlying implementation details. This separation of concerns thus promotes the ability to reason about distributed adaptations at a higher level of abstraction.

DyReS’s reconfiguration support provides the following nine reconfiguration operations and two synchronization primitives:

- **create** involves loading a specified aspect into the running application – that is, without binding it to the application.
- **interrupt** supports to drive components and aspects into a safe state by intercepting messages that prevent them from reaching a safe state.
- **impose_safe_state** detects when an aspect or component instance reaches a safe state and supports transferring the persistent state of those instances.

- **deactivate** is used to stop active objects that execute in their own thread of control. Notice that the fragmenting interceptor is an example of such an active object: it fragments messages in its own thread of control.
- **unbind** removes a specified binding from the application.
- **bind** is responsible for connecting an aspect to the application using a particular advice binding.
- **activate** is used for starting up an active object.
- **resume** resumes all intercepted invocations after the reconfiguration has been completed.
- **remove** deletes an aspect instance from the system.
- the **synch_wait** primitive blocks the current reconfiguration task until it receives a specified synchronization message from a specified node. This is needed when multiple aspects must be woven across multiple nodes in the distributed system.
- the **synch_notify** primitive sends a specified synchronization message to a specified node.

Detailed discussion of these operations and primitives has already been described elsewhere [15]. The reconfiguration operations are inspired by the well-known dynamic change management model from Kramer and Magee [19]. The Kramer and Magee model distinguishes between six reconfiguration primitives (create, remove, link, unlink, passivate, activate). We have split some of these primitives (unlink, link and passivate) into separate reconfiguration operations in order to better separate certain coordination concerns from each other.

As the problem of dynamic reconfiguration is very complex, the semantics of the reconfiguration operations cannot be completely defined by the framework. An operation like **bind** is implemented in full by the framework, but an operation like **impose_safe_state** or **interrupt** can not given precise semantics in advance. These reconfiguration operations are more like locations in the reconfiguration process where user-specified code (in an imperative general-purpose programming language) can be inserted in order to fully implement these operations. As such, DyReS implements these reconfiguration operations as abstract classes with hooks for user-specified code.

To support the developer, the DyReS framework already supports a library of common implementation strategies that the user can plug into these hooks. For example, the framework provides two variant implementation strategies for the **interrupt** operation. One for queuing synchronous invocations by blocking the current thread, one for queueing asynchronous invocations by storing them in a waiting queue. The user can specify in an adaptation script which implementation strategy to select, overriding the default strategy of the framework.

3.2 Adaptation scripting language

DyReS enables the developer to implement a distributed adaptation as a set of adaptation scripts, one for each node involved in the adaptation. These adaptation scripts will then be interpreted by DyReS.

As already stated above, a single adaptation script is structured as a sequence of reconfiguration tasks where each task consists of one or more reconfiguration actions. Each reconfiguration action is logically connected to the execution of a specific reconfiguration operation. The specification of a reconfiguration action consist of two parts: (1) the particular reconfiguration operation (or synchronization primitive) that must be executed, and (2) any required data (parameters, properties, user-specified code) that the DyReS framework needs for executing the reconfiguration operation. Note that

not all required data elements must be specified by the developer; in this case a default will be used by the DyReS framework.

Currently, in DyReS, an adaptation script is specified in XML. The concrete XML schema of the language depends on the concrete underlying AO framework; our goal is to keep this language as closely as possible to the aspect deployment style of the underlying AO framework. This enables a higher acceptance of DyReS by developers³. For example, for the DyReS implementation on top of Spring AOP, the developer needs to deploy at each node three XML files: two Spring configuration files (similar to the one in Figure 2) that respectively define the original and the new composition of aspects and components at that node, and a third XML file that specifies the adaptation script itself for bringing the system from the old to the new composition. For the purpose of this paper, we will use a more abstract syntax (see Figure 3) that hides the differences between the adaptation script languages for JBoss and Spring.

[Reconfiguration operation]	
[Required data]	[Description of required data]
create	
class	name of aspect class to be woven into the system
ID	unique name for aspect in the running system
scope	scope of aspect instances (perVM=default)
props	properties for initializing aspect instance(s) (default=none)
interrupt	
interruptor	name of aspect class that contains implementation of interrupt operation (default='ThreadBlockingInterruptor')
messages	pointcut representing messages to be intercepted
ID	unique name for representing interruptor in the system
impose_safe_state	
finisher	name of class that contains implementation of impose_safe_state operation (default='DelayFinisher')
props	properties for configuring finisher class (default=none)
deactivate	
deactivator	name of class that contains implementation of deactivate operation (default='ThreadDeactivator')
props	properties for configuring specified deactivator class
unbind	
bindingID	name of the advice binding to be removed
bind	
adviceID	name of advice to be bound
pointcut	pointcut for composing the advice
ID	unique name for representing the advice binding
activate	
activator	name of class that contains implementation of activate operation (default='ThreadActivator')
props	properties for configuring specified activator class
resume	
interruptorID	name of interruptor to release intercepted messages
remove	
aspectID	name of the aspect instance to be removed
synch_notify	
msg	text message used as synchronization message
dest	IP address of node to be notified
port	port to send message to
synch_wait	
msg	String representing synchronization message
source	IP address of node which is expected to send message

Figure 3. The figure gives an abstract overview of the specification of the different reconfiguration actions. Each reconfiguration action is specified as an XML element that specifies various elements necessary for the DyReS framework to execute the reconfiguration action.

³ The full implementation of DyReS on top of JBoss AOP and Spring AOP is available at <http://www.cs.kuleuven.be/~distrinet/projects/DyReS/>.

3.3 Framework architecture

Figure 4 gives an overview of the layered architecture of DyReS. It consists of three parts: (1) the core of the framework for specifying and executing adaptation scripts, (2) the AO framework specific instantiation of DyReS, and (3) hooks for plugging in customized implementation strategies for the reconfiguration operations.

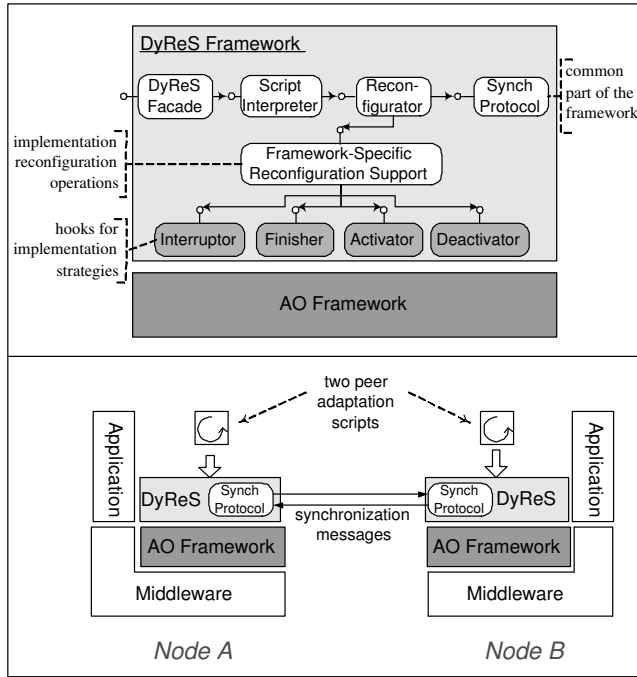


Figure 4. Overview of the architecture of DyReS. The top of the figure shows the layered architecture of DyReS. The bottom of the figure shows the architecture from a deployment view

A distributed adaptation is started by deploying adaptation scripts across the nodes involved in the adaptation. At each node, the DyReS Facade component accepts the adaptation script and forwards it to the Script Interpreter component. The Script Interpreter component executes each script by sequentially iterating over the reconfiguration actions found in the script. The execution of each reconfiguration action is delegated to a component that implements the Reconfigurator interface. This component uses the Framework-Specific ReconfigurationSupport component which contains the AO framework-specific implementation of all the reconfiguration operations. For example on top of Spring AOP, the GenericApplicationContext, Advisor, Advised, MethodInterceptor and XMLBeanFactory API from Spring are used. On top of JBoss AOP, these reconfiguration operations are implemented using the AspectManager API.

As already stated above, for some reconfiguration operations, user-specified code must be plugged into predefined hooks. DyReS already supports several implementation strategies. As indicated in Figure 3, the application developer has to select these strategies during the specification of the adaptation script. We now give, for each relevant reconfiguration operation, an overview of the strategies currently supported by the framework (see also Figure 5).

The **interrupt** and **resume** operations are implemented by weaving in an aspect that implements the DyReSInvocationInterruptor interface. This aspect offers operations for blocking and resuming messages. A DyReSInvocationInterruptor aspect must

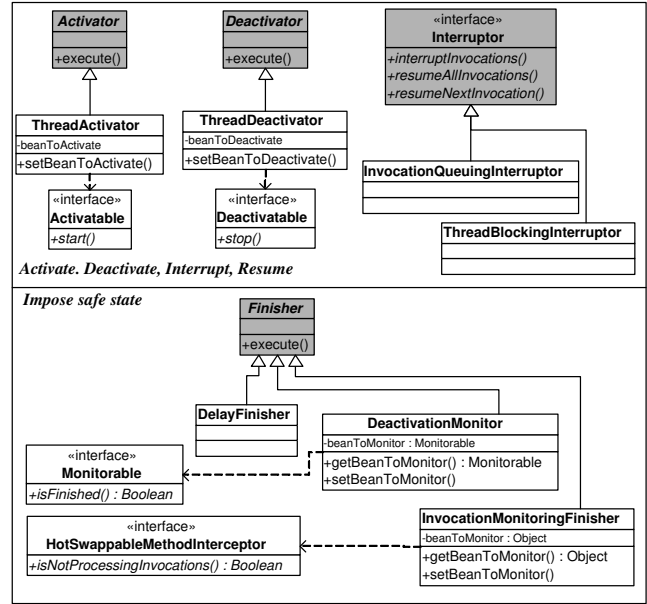


Figure 5. Overview of the alternative implementation strategies for reconfiguration operations.

intercept those messages that start up new collaboration instances. The framework supports the following implementation strategies.

1. **ThreadBlockingInterruptor** (default): A **ThreadBlockingInterruptor** disables intercepted messages by blocking their execution thread. To resume the execution of these messages afterwards, their threads are activated again.
2. **InvocationQueuingInterruptor**: An **InvocationQueuingInterruptor** disables intercepted messages by storing them in a buffer and returning control to the invoking client. This **DyReSInvocationInterruptor**, therefore, can only be used to interrupt the execution of asynchronous invocations. To resume the execution of interrupted calls afterwards, DyReS iterates over the list of stored messages and continues their execution. So, resumed calls are executed by a DyReS thread instead of by the original application thread.

The **deactivate** and **activate** operations must be implemented as two classes that implement the **Deactivator** and **Activator** interfaces respectively. Currently, there is one implementation strategy provided by the classes **ThreadDeactivator** and **ThreadActivator**. These are used for stopping and starting active objects that run in their own threads. The active objects have to implement the **Activatable** and **Deactivatable** interface for starting and stopping their threads respectively.

The **impose_safe_state** operations is supported by implementing the **Finisher** interface. A **Finisher** is responsible for driving the aspects that are subject for reconfiguration to a safe state. The following implementation strategies are currently provided by the DyReS framework.

1. **DelayFinisher** (default) waits for a specific time period. After that it is assumed that aspects and components subject for reconfiguration are in a safe state. This dummy implementation works for light-weight aspects (e.g., message encryption) that do not need a lot of time for completing their ongoing collaboration instances.

- InvocationMonitoringFinisher waits until the aspect is not processing accepted messages anymore. The aspect has to implement the HotSwappableInterceptor interface for this.
- DeactivationMonitor waits until the aspect is in a self-declared safe state. The aspect has to implement the Monitorable interface for this.
- StateTransferObject gets from the aspect a state object that represents the current state of the aspect instance. This state object must be used to initialize the new aspect. The aspects have to implement the StateTransferable interface for this. This only works for replacement and if a state mapping is possible.

4. Customized implementation of distributed adaptations

Existing dynamic reconfiguration frameworks typically support one generic coordination protocol for implementing distribution adaptations [4, 23, 19]. The scripting language of DyReS, on the contrary, enables the user to develop customized coordination protocols that are tailored to application-specific requirements and thus perform more efficiently with respect to these requirements. Our proposed framework allows to produce a custom-made adaptation script in two different ways. The first manner is by changing the order in which reconfiguration actions are performed. The second manner includes that for some reconfiguration operations, a user-specified implementation strategy can be selected.

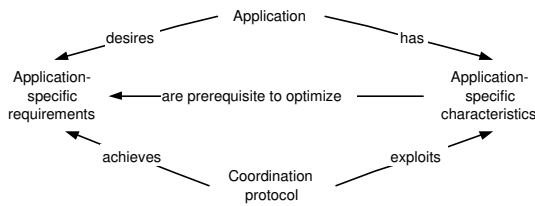


Figure 6. Application-specific requirements and characteristics.

Figure 6 sketches our approach. Key design decisions about how to implement a distributed adaptation to a set of aspects often depend on application-specific requirements on the one hand and application-specific characteristics on the other hand:

Application-specific requirements Different applications may desire different reconfiguration semantics and performance requirements. Figure 7 gives some concrete examples of these application-specific requirements. For example, in the IMS case study, it is very important that the reconfiguration process for replacing the fragmentation service does not affect the responsiveness of the application. A coordination protocol that first finishes the old fragmentation service before activating the new fragmentation service is not desired, because it blocks all ongoing service request for some time. The end user experiences this as a temporary service disruption, therefore affecting responsiveness of the IMS platform. Instead, a more optimized coordination protocol activates the new service before shutting down the old service while ensuring that all messages get processed consistently by the same service.

Application-specific characteristics The application and the aspects may have specific characteristics that can be exploited to change the implementation of the coordination protocol into a more optimal variant. For example, switching the order of finishing and activation is only possible when the fragmentation service does not have external state dependencies to the invoking client component.

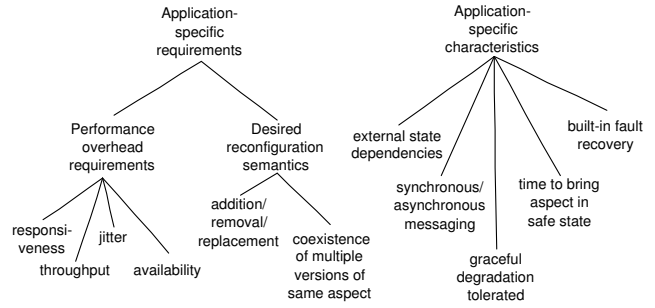


Figure 7. Examples of application-specific requirements and characteristics.

4.1 Revisiting the case study

In this section we show – by means of the IMS case study in section 2.1 – how developers can create their own custom-made adaptation scripts to achieve application-specific requirements. Before we zoom in on the scripts, we first summarize the relevant characteristics of the aspects and components involved in the IMS platform; thereafter we will consider different reconfiguration semantics (removal/addition/replacement).

Remember that the specific performance requirements of the IMS platform are high availability, responsiveness and reliability (see section 2.1). The implementation of a distributed adaptation (e.g., the run-time removal of the distributed cache or fragmentation service) potentially has a negative effect on these application-specific requirements. This is because the affected aspects may have specific characteristics that lead to additional overhead. Figure 8 gives an overview of the relevant characteristics of the aspects involved. For example, it may take a long time to bring the distributed cache into a safe state, among others because data updates must be synchronized between the distributed cache instances and the cache must be synchronized with the relational database. The same rationale, although less severe, applies to fragmenting. Bringing the fragmenting aspect into a safe state involves waiting until its internal queue of messages has been fully processed.

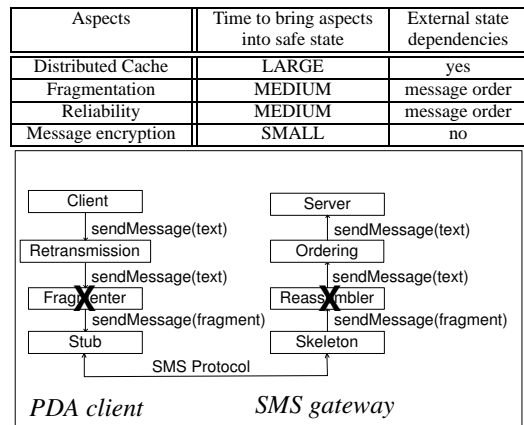


Figure 8. The table presents the characteristics of the aspects that have a potential negative impact on availability, reliability and responsiveness of the IMS platform, when removing these aspects. The figure presents an overview of a basic adaptation scenario where the reliability and fragmentation service have to be removed. Because the fragmentation service depends on the reliability service, the fragmentation service has to be removed first.

The application itself also has some interesting characteristics. First, the reliability service supports local message ordering in the scope of a session. This local message ordering support is necessary because the fragmentation service may affect the order of messages sent by one user. As such, the fragmentation service depends on the reliability service. Second, the IMS application uses asynchronous messaging. After marshalling a text message onto the SMS network, the underlying SMS protocol immediately returns control to the Stub component. We will show that these application-specific characteristics can be exploited to implement the removal of the fragmentation service in a more optimal way.

The remainder of this section presents various alternative adaptation scripts for implementing the removal, addition and replacement of the fragmentation service and distributed cache.

4.2 Removal of fragmentation service

We now discuss a basic adaptation scenario: the removal of the fragmentation service for an ongoing session between a PDA node and the SMS gateway. We assume that for each session between a PDA node and the central SMS-gateway a separate instance of the fragmenting and reassembling aspects are deployed. Thus within a single session, two aspect instances must be removed (single PDA-case). We later also consider the case where the fragmentation service is removed from all ongoing sessions simultaneously (multi-PDA case).

We will first discuss the basic distributed adaptation as supported by DyReS. This adaptation does not take into account application-specific requirements and characteristics. Subsequently we will show how this basic variant can be customized by using marking and dispatching support.

4.2.1 Variant 1: Impose Safe State before Resume

This variant essentially finishes the fragmenting and reassembling aspects before unbinding them from the application. In other words, the mechanism to preserve global state consistency is used to preserve structural integrity. To implement this distributed adaptation, two adaptation scripts must be specified: one for the removal of the fragmenting aspect and one for the removal of the reassembling aspect (see Figure 9). The scripts implement three kinds of reconfiguration tasks at each node: finishing the old aspects, activating the system without the aspects, and removing the aspects. Different tasks at different nodes have to coordinate with each other through the synchronization primitives. An example of such a synchronization is that the PDA node must finish its Fragmenter aspect before the SMS gateway can finish the corresponding Reassembler aspect. In order to express this, the SMS gateway waits (**synch.wait**) for a notification (**synch.notify**) from the PDA node.

The reconfiguration task of finishing the fragmenting aspect consists of the following actions. First, the application is interrupted by means of the **interrupt** action. This action blocks all invocations of the Stub component – as these invocations start up new instances of the fragmentation collaboration. The **interrupt** action implicitly uses the default `ThreadBlockingInterruptor` class for implementing the blocking of invocations. Secondly, the fragmenting aspect is brought in a safe state by means of an **impose_safe_state** action. This action uses the `DeactivationMonitor` class which waits until the fragmenting aspect has processed all accepted messages. The `DeactivationMonitor` class gets a reference to the fragmenting interceptor through the *props* element of the **impose_safe_state** action. Thirdly, the thread inside the fragmenting aspect is deactivated using the **deactivate** action. After that, the SMS gateway can finish the reassembling aspect by imposing its safe state (in this case it is sufficient to use the default delay finisher).

After the finishing tasks are completed at both nodes, the two aspects are unbound (independently from each other) from the appli-

cation through two **unbind** actions. After that (again this requires synchronization), the activation task at the PDA client restarts the application through the **resume** action. Finally in the removal task, both aspect instances are removed.

```
<reconfiguration <!--removalFragmenter-->
<!--Finishing of fragmenter aspect-->
<interrupt <!--default interruptor is used-->
  messages="call(Stub.send*(..))"
  ID="interrupt_1" />
<impose_safe_state finisher="DeactivationMonitor"
  <props beanToMonitor="fragmenter" /> />
<deactivate <!--default deactivator is used-->
  <props beanToDeactivate="fragmenter" /> />
<synch_notify msg="SafeStateAtClientOK"
  dest="SMS_gateway" port="11000"/>

<!--Activation of system without fragmenter aspect-->
<unbind bindingID="fragmentingBinding" />
<synch_wait msg="ServerDeactivated" source="SMS_gateway" />
<resume interruptorID="interrupt_1"/>

<!--Removal of MessageFragmenter interceptor-->
<remove aspectID="fragmenter" />
/>
```

```
<reconfiguration <!--removalReassembler-->
<!--Finishing of reassembler aspect-->
<synch_wait msg="SafeStateAtClientOK" source="PDA_node" />
<impose_safe_state /> <!--default finisher used-->

<!--Activation of system without reassembler aspect-->
<unbind bindingID="reassemblingBinding" />
<synch_notify msg="ServerDeactivated"
  dest="PDA_node" port="10000" />

<!--Removal of MessageReassembler interceptor-->
<remove aspectID=reassembler />
/>
```

Figure 9. Adaptation scripts for removing the fragmenting and reassembling aspect at the PDA node and SMS gateway respectively.

A potential performance problem with this implementation of the distributed adaptation is that the user may experience a sudden deterioration in the responsiveness and availability of the SMS service. This is because the Stub component is frozen during the entire time interval between the execution of the **interrupt** action and the execution of **resume** action. This interval is long because of the **impose_safe_state** actions. Only after a safe state has been reached at both nodes, the Stub component will be unblocked by means of the **resume** primitive. Section 5 describes a set of performance tests that underline this performance problem.

Multi-party coordination is required when more than 2 nodes are involved in a distributed adaptation. For example, consider the situation where the Fragmenter aspect has to be removed from multiple PDA clients. To implement such a multi-party coordination protocol in DyReS, the adaptation script for the fragmenter must be deployed at each PDA client. Moreover, the adaptation script at the SMS gateway must now synchronize with all PDA clients. For example, it can only remove the reassembler aspect after *all* PDA clients have finished their Fragmenter aspect instances. Thus it must perform *multiple synch_wait* actions, one for each PDA.

4.2.2 Variant 2: Resume before Impose Safe State

The distributed adaptation discussed in the previous section is very simple and therefore robust, but it introduces a temporary service disruption of the SMS service. Now we present a variant that does not introduce a service disruption, but is more complex because it depends on marking and dispatching support.

This variant basically puts the SMS service of the IMS application in a *two-version mode*. This means that an old and a new version of the IMS application run in parallel next to each other. The old version represents the IMS application with the fragmentation service and the new version is without the fragmentation service.

For the sake of structural integrity, it is important to ensure that existing collaboration instances of the fragmentation service are still consistently processed by both the Fragmenter and Re-assembler aspects. To ensure this, additional reconfiguration support is installed by DyReS using two additional aspects: a Marker aspect and a Dispatcher aspect. The Marker aspect, to be installed at the PDA node, attaches a label to new text messages that are not anymore processed by the Fragmenter aspect. The Dispatcher aspect, installed at the SMS gateway, inspects this label to distinguish fragments from non-fragmented messages. It will then forward fragments to the Reassembler aspect, and forward non-fragmented messages directly to the Stub component.

Figure 10 shows the scripts that implement this variant. First, the dispatching and marking support is dynamically installed in the application using the **interrupt**, **create** and **bind** actions. The dispatcher gets a reference to the Reassembler aspect through the *props* element of the **create** action. After that, the aspects of the fragmentation service are unbound, and the application is resumed. Subsequently, the fragmentation service is finished through the **impose_safe_state** and **deactivate** actions as in variant 1. Finally, the fragmentation aspects and the marking and dispatching aspects are removed again.

The important difference with variant 1 is that, at the PDA node, the order of the **resume** and **impose_safe_state** actions is reversed. This resolves a large part of the service disruption introduced by variant 1. Consequently, end users will experience much less service interruption (hence fulfilling the responsiveness requirement of the IMS application). Variant 2 also exploits the fact that the SMS application only uses asynchronous messaging. This is shown by looking at the **interrupt** action which uses the the *InvocationQueuingInterruptor* aspect instead of the (default) *ThreadBlockingInterruptor* aspect.

A final issue is how to deal with potential message re-orderings. During the execution of the **impose_safe_state** action, new text messages from end users may overhaul existing text messages that are still being processed by the fragmentation service. In this scenario however, it is possible to exploit another application-specific characteristics of the IMS platform: the deployed reliability service already implements a message ordering protocol on top of the fragmentation service. As such, the reliability service will re-establish the correct order. A prerequisite, however, is that the reliability service should not be removed before the fragmentation service is removed.

4.3 Addition and Replacement

For replacement and addition of the fragmentation service it is possible to write adaptation scripts in the same way as variant 1 and variant 2. We will shortly discuss how the various reconfiguration tasks for replacing a fragmentation service differs from removal. In the installation task, the aspects of the new fragmentation service must be created as well using the **create** operation. The finishing task may need to be extended with an additional **impose_safe_state** action that uses the *StateTransferObject* class for transferring persistent state from the old aspects to the new aspects. The activation task needs to be extended with **bind** actions for binding the new aspects to the application. Finally, the removal task remains the same. The dispatcher and marker aspects are generic in the sense that they can deal with removal, addition and replacement. The **create** action only needs to configure these aspects properly through the *props* element.

```
<reconfiguration <!--removalFragmenter-->
<!--Installation of marker-->
<create class="MarkerInterceptor" ID="marker"/>
<synch_wait msg="DispatcherInstalled"
  source="SMS_gateway"/>
<interrupt interruptor="InvocationQueuingInterruptor"
  messages="call(Stub.send*(..))" ID="interrupt_1"/>
<bind adviceID="marker"
  pointcut="execution(Stub.send*(..))"
  ID="markingBinding"/>

<!--Activation of system mode without fragmenter-->
<unbind bindingID="fragmentingBinding"/>
<resume interruptorID="interrupt_1"/>

<!--Finishing of fragmenter aspect-->
<impose_safe_state finisher="DeactivationMonitor"
  <props beanToMonitor="fragmenter"/> />
<deactivate <!--default deactivator is used-->
  <props beanToDeactivate="fragmenter" /> />
<sync_notify msg="SafeStateAtClientOK"
  dest="SMS_gateway" port="11000"/>

<!--Removal of marker and fragmenter aspects-->
<synch_wait msg="DispatcherRemoved" source="SMS_gateway"/>
<interrupt interruptor="InvocationQueuingInterruptor"
  messages="call(Stub.send*(..))" ID="interrupt_2" />
<unbind bindingID="markingBinding"/>
<resume interruptorID="interrupt_2"/>
<remove aspectID="marker"/>
<remove aspectID="fragmenter"/>
/>
```

```
<reconfiguration <!--removalAssembler-->
<!--Installation of dispatcher-->
<create class="DispatcherInterceptor" ID="dispatcher"
  <props oldAspect="reassembler"/> />
<interrupt interruptor="InvocationQueuingInterruptor"
  messages="call(Server.send*(..))" ID="interrupt_3"/>
<bind adviceID="dispatcher" pointcut=
  "execution(Server.send*(..))" ID="dispatchBinding"/>

<!--Activation of system mode without reassembler-->
<unbind bindingID="reassemblingBinding"/>
<resume interruptorID="interrupt_3"/>
<sync_notify msg="DispatcherInstalled"
  dest="PDA_node" port="10000" />

<!--Finishing of reassembler aspect-->
<synch_wait msg="SafeStateAtClientOK" source="PDA_node"/>
<impose_safe_state/> <!--default finisher used-->

<!--Removal of dispatcher and reassembler aspects-->
<interrupt interruptor="InvocationQueuingInterruptor"
  messages="call(Server.send*(..))" ID="interrupt_4"/>
<unbind bindingID="dispatchBinding"/>
<resume InterruptorID="interrupt_4"/>
<sync_notify msg="DispatcherRemoved"
  dest="PDA_node" port="10000" />
<remove aspectID="dispatcher"/>
<remove aspectID="reassembler"/>
/>
```

Figure 10. Variant 2.

4.4 Removal of distributed cache

The dynamic removal of the distributed cache from the IMS platform requires to drive the distributed cache to a safe state. This involves three actions: (i) interrupting new service requests at the Facade component of the IMS platform through an **interrupt** action, (ii) waiting till all ongoing instances of the 2-phase commit protocol for updating the distributed cache are completed through an **impose_safe_state** action and (iii) synchronizing the cache to the database through another **impose_safe_state** action. To imple-

ment the last action, the DyReS framework has to be extended with an application-specific implementation strategy for the **impose_safe_state** reconfiguration operation.

Another variant, that keeps service disruption as low as possible, is using two dual stacks with output control. This mechanism is inspired by a well-known fault tolerance technique, called active replication. All requests for updating data are blocked. However data queries are sent to both the distributed cache and the database. Outputs of both components are monitored. When these outputs converge, it can be assumed that the distributed cache has reached a safe state and thus can be removed safely. This mechanism can be implemented as a separate aspect that is dynamically installed in the application using the **create** and **bind** operations, similarly to adding the marking and dispatching support.

4.5 Implementation issues

There are some subtle differences between Spring AOP and JBoss AOP that forced us to tweak the presented adaptation scripts for JBoss AOP. Currently in JBoss AOP, the weaving of a single interceptor may cause errors in the execution of already woven interceptors at the same join point. Although this is caused by a bug⁴, resolving the bug requires a major refactoring of the JBoss AOP class loader. As a consequence, an **interrupt** action must ensure that the `InvocationQueuingInterruptor` aspect is woven at a join point where no other aspects are bound. In this particular case study, the interruptor has been bound to operations of the Client component instead of the Stub component (see Figure 8).

Furthermore, in JBoss AOP, when unbinding the fragmenting interceptor at run-time, not only the binding is immediately removed (as would be expected) but also the interceptor instance itself. This means that text messages stored in the waiting queue of this interceptor are lost after unbinding it. In other words, this means that an `Impose Safe State` action always must be executed before an `Unbind` action. Hence, the `Resume before Impose Safe State` strategy is not an option for JBoss AOP.

5. Evaluation

In this section we will evaluate the performance overhead of the DyReS framework on top of Spring AOP for different types of distributed adaptations. Due to lack of space, we do not discuss the overhead of DyReS on top of JBoss AOP, but similar results can be expected there. We also elaborate on our claims that these dynamic reconfigurations can be optimized for different performance requirements. To do so, we investigate the actual cost incurred by a number of distributed adaptations. This cost will be evaluated in terms of

1. the service disruption that is caused,
2. the total reconfiguration time, and
3. the overhead during normal operation.

The first metric quantifies the period of time in which the nodes that are affected by the distributed adaptation are unable to start any new collaboration instances. Obviously, this period of time should be as small as possible because of its major impact on availability and responsiveness. The second metric captures the time that it takes to complete an adaptation. The third and last one indicates the overhead of the framework during normal application operation, i.e. with no reconfigurations going on.

5.1 Test setup

The test setup we used for measuring the cost of adding, replacing and removing the fragmentation service represented the different

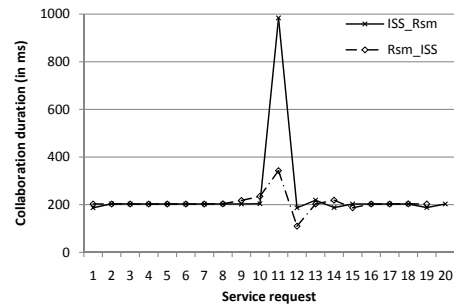
⁴ <http://jira.jboss.org/jira/browse/JBAOP-380>

adaptation scenario's from Section 4. We defined a call chain between four beans that were physically distributed over two different virtual machines: bean A (client) and B (stub) were located in one virtual machine, bean C (skeleton) and D (server) in another one. A third virtual machine was used to coordinate the distributed adaptations. We carried out our experiments on a Sony VAIO laptop with a 1.73 GHz processor and 1024 MB RAM with Java 1.6.0, Spring 2.0.6 and JBoss AOP Tools 1.1.2 installed.

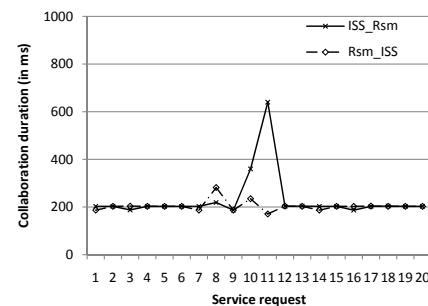
5.2 Service disruption

To determine the service disruption, we measured the effect of a distributed adaptation by quantifying over the duration of subsequent collaboration instances.

Figure 11 presents an overview of the collaboration durations for different variants of the addition and removal of a fragmentation service respectively⁵. First of all, we notice the highest collaboration duration (close to 1000 ms) when a fragmentation service is added using the `Impose_Safe_State before Resume` variant (*ISS_Rsm*). This is due to the fact that the complete application needs to be brought into a safe state, which is much more complex than only one or two aspect components as in a removal or replacement scenario. Secondly, we clearly can observe in both graphs that the `Resume before Impose_Safe_State` variant (*Rsm_ISS*) causes much less service disruption. This confirms the claims made in section 4 that this variant less affect the responsiveness of the application.



(a) Adding fragmentation service



(b) Removing fragmentation service

Figure 11. Overview of service disruption during addition and removal of a fragmentation service.

⁵ The replacement experiment resulted in similar findings as the removal experiments, and is left out of the paper due to lack of space.

5.3 Adaptation time

Figure 12 lists some indicative times that it took to complete the distributed adaptations that were discussed above. The minimum time to execute a reconfiguration at the fragmenter side was 750 ms while the maximum time here was 1094 ms. At the reassembler side, completion times varied between 1094 ms and 1531 ms. This variation can be explained by the different amount and/or ordering of reconfiguration steps that have to be undertaken for each distributed adaptation. Total adaptation time also is influenced by the application-specific context, such as e.g. the time it takes to finish certain components or the number of nodes that are involved in a specific reconfiguration (which increases the time that is needed to synchronize between all the nodes).

Reconfiguration	Time fragmenter		Time reassembler	
	ISS_Rsm	Rsm_ISS	ISS_Rsm	Rsm_ISS
Add	1094 ms	750 ms	1531 ms	1094 ms
Remove	938 ms	953 ms	1312 ms	1329 ms
Replace	937 ms	968 ms	1328 ms	1344 ms

Figure 12. Overview of adaptation times

5.4 Overhead during normal operation

We used the same test setup in order to assess the overhead of distributed adaptations during normal operation. As one should expect, there is limited to zero overhead when there are no ongoing reconfigurations. Only when a distributed adaptation is started, the necessary DyReS plumbing gets loaded into the application and is put into play. As soon as the adaptation is completed, this plumbing can be removed again. The only overhead during normal execution is introduced by preparing the code for interception, such as for instance with the proxy-based implementation of Spring AOP. The performance penalty of introducing such a proxy in Spring AOP to enable us to intercept the call is only in the range of 200 to 400 nano seconds [36].

6. Related work

Three categories of related work are considered: AO frameworks that support atomic weaving, AO frameworks that support remotely distributed AOP, and work that has applied existing coordination protocols on top of AO middleware.

Atomic weaving Existing AO frameworks such as Prose [25], Wool [29], and DAC++[3] support atomic weaving of a single aspect in a local execution environment. Also some context-oriented programming languages [14] provide dynamically scoped adaptations with support for preserving structural integrity. It is expected that this kind of support will become an integral part of virtual machine support for AOP[5]. Coordinated weaving support, as provided by DyReS is however different in the sense that DyReS also ensures that multiple aspect weavings are performed in an atomic way.

Lasagne [34] and Arachne [9] are AO frameworks that offer the possibility for atomic weaving of multiple aspects in a distributed environment. Other related work in this context includes a framework for policy-driven adaptation of aspects [12] and a middleware for coordinated deployment of crosscutting QoS features [37, 38]. None of the above frameworks support preserving global state consistency, while DyReS does.

AO frameworks that support remotely distributed AOP AO frameworks such as JAC [28], DJCutter [26], AWED [24], DReflex [32] and DyMAC[20] support the ability to express dynamic compositions that depend on the evaluation of available context information about the distributed infrastructure. As a result, complex

composition with remote events can be expressed more concisely and in a more localized way. The integration of DyReS on these platforms has not been considered yet. It can be expected though that some reconfiguration primitives of DyReS can be implemented more elegantly on top of these platforms than on top of Spring AOP or JBoss AOP.

AWED [24] is an aspect language with explicit distributed programming mechanisms. To implement this language, Navarro et al. used the DJAsCo distributed AOP architecture. This framework supports dynamic weaving of stateful distributed aspects. No support is provided for reaching a safe state before unweaving an aspect however. In the case of replacements, mutually consistent execution states may be preserved by using DJAsCo’s state sharing support. This support seems to enable initializing the new aspect with the execution state of the old one, which can be employed as an alternative for reaching quiescence to preserve system-wide consistent execution states.

Tanter and Toledo’s versatile kernel for AOP supports runtime link manipulation to dynamically deploy/undeploy distributed aspects [32]. Its remote consistency framework maintains (structural) consistency between changes made to links in different hosts. Reaching a reconfiguration-safe state to safely remove aspects, however, seems not to be supported.

Applying existing coordination protocols to aspects DyReS has largely been inspired by previous research on dynamic change management of component-based systems [19, 23, 4, 16]. In our previous position paper [15], we argued that the existing coordination protocols developed in this space can be largely reused for distributed aspect weaving. Our supporting hypothesis is that aspect modules do not fundamentally differ from component modules, but only employ a different binding structure. In this vein, Surajbali et al. [31] have also ported existing coordination protocols from the component-based Gridkit platform [11] in order to coordinate distributed weaving and unweaving of aspects in OpenCOM. A similar approach has been explored in the Fractal Aspect Component model [30].

The novelty of DyReS in this respect is that it can be customized to achieve improved performance and reconfiguration semantics in a specific application context. This ability to customize the implementation of how to coordinate a distributed adaptation is important when a trade-off must be made between performance requirements on the one hand and safety properties on the other hand.

Even in the space of component-based reconfiguration, existing dynamic change management systems, except for the framework of Hillman[13], lack the ability to customize the reconfiguration process. The difference between DyReS and Hillman’s framework is that DyReS uses a domain-specific language for building custom-made reconfiguration algorithms, whereas in Hillman’s approach developers have to use object-oriented framework specialization techniques which is at a lower level of abstraction.

7. Conclusion

In this paper, we have presented the DyReS framework that offers coordination support for distributed adaptations in aspect-oriented middleware. Currently, the framework is implemented on top of Spring AOP and JBoss AOP. Both middleware platforms encapsulate run-time aspect weaving technology, but do not support coordinating distributed changes to a set of aspects at run-time. Coordinating the weaving or unweaving of multiple inter-dependent aspects is a very tedious and error-prone task because global state consistency, structural integrity and other safety properties need to be ensured. To raise the level of abstraction, DyReS offers a limited set of operations for forcing aspects and application compo-

nents into safe states; adding, removing, and replacing multiple aspects; and coordinating changes between different nodes. As our first main contribution, a distributed adaptation can be specified in an adaptation script that is nicely separated out of the application layer. Secondly, DyReS is customizable towards application-specific requirements to achieve improved performance and reconfiguration semantics. As a result, an application developer has the choice between variant strategies for implementing the distributed adaptation. We have performed a quantitative performance analysis that proves the relevance of supporting these customizations.

Acknowledgments

We would like to thank the reviewers for their insightful comments that helped to improve this paper.

References

- [1] JBoss AOP homepage. <http://labs.jboss.com/jbossaop>.
- [2] Spring website. <http://www.springframework.org/>.
- [3] S. Almajali and T. Elrad. Coupling availability and efficiency for aspect-oriented runtime weaving approaches. In *Proceedings of DAW 2005*.
- [4] João Paulo A. Almeida, Maarten Wegdam, Marten van Sinderen, and Lambert J. M. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *Proceedings of DOA 2001*, Rome, Italy, 2001.
- [5] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04*, 2004.
- [6] Rémi Douence, Thomas Fritz, Nicolas Lorient, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with arachne. In Mezini and Tarr [22].
- [7] Gary Duzan, Joseph Loyall, Richard Schantz, Richard Shapiro, and John Zinky. Building adaptive distributed applications with middleware and aspects. In *Proceedings of AOSD 2004*, New York, NY, USA, 2004.
- [8] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In Mezini and Tarr [22].
- [9] T. Fritz, M. Ségura, M. Südholt, E. Wuchner, and J. Marc Menaud. An application of dynamic aop to medical image generation. In *Proceedings of DAW 2005*.
- [10] L. Fuentes, M. Pinto, and P. Sánchez. Dynamic weaving in cam/daop: An application architecture driven approach. In *Proceedings of DAW 2005*, 2005.
- [11] P. Grace and G. Coulson. A distributed architecture meta model for self-managed middleware. In *Proc. 6th Workshop on Adaptive and Reflective Middleware (ARM 2006)*, 2007.
- [12] Phil Greenwood and Lynne Blair. A framework for policy driven auto-adaptive systems using dynamic framed aspects. 4242, 2006.
- [13] J. Hillman and I. Warren. An open framework for dynamic reconfiguration. In *ICSE*, 2004.
- [14] Robert Hirshfeld, Pascal Constanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology (JOT)*, March-April 2008. <http://www.jot.fm>.
- [15] N. Janssens, E. Truyen, F., and W. Joosen. Adding dynamic reconfiguration support to JBoss AOP. In *MAI '07: Proceedings of the 1st workshop on Middleware-Application Interaction*, New York, NY, USA, 2007.
- [16] Nico Janssens, Wouter Joosen, and Pierre Verbaeten. NeCoMan: Middleware for Safe Distributed-Service Adaptation in Programmable Networks. *IEEE Distributed Systems Online*, 6(7), 2005.
- [17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. L., and J. Irwin. Aspect-Oriented Programming. In *ECOOP'97—Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, 1997.
- [18] Jörg Kienzle and Samuel Gélineau. Ao challenge - implementing the acid properties for transactional objects. In *Proceedings AOSD 2006*.
- [19] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.*, 16(11), 1990.
- [20] B. Lagaisse and W. Joosen. True and Transparent Distributed Composition of Aspect-Components. In *Proceedings Middleware '06*, volume 4290 of *Lecture Notes in Computer Science*, 2006.
- [21] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *Computer*, 37(7), 2004.
- [22] Mira Mezini and Peri L. Tarr, editors. *Proceedings of the 4th International Conference on Aspect-Oriented Software Development, AOSD 2005, Chicago, Illinois, USA, March 14-18, 2005*, 2005.
- [23] Kaveh Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. PhD thesis, Imperial College, London, 1999.
- [24] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvé. Explicitly distributed aop using awed. In *Proceedings of AOSD 2006*.
- [25] Angela Nicoara and Gustavo Alonso. Dynamic aop with prose. In *Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005)*, 2005.
- [26] M. Nishizawa, S. Chiba, and M. Tsubori. Remote pointcut: a language construct for distributed aop. In *Proc. AOSD'04*.
- [27] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3).
- [28] 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*, 2001.
- [29] Yoshiki Sato, Shigeru Chiba, and Michiaki Tsubori. A selective, just-in-time aspect weaver. In *GPCE '03*, 2003.
- [30] L. Seinturier, N. Pessemier, L. Duchien, and T. Coupaye. A component model engineered with components and aspects. In *CBSE*, 2006.
- [31] B. Surajbali, G. Coulson, P. Greenwood, and P. Grace. Augmenting reflective middleware with an aspect orientation support layer. In *Proc. 6th Workshop on Adaptive and Reflective Middleware (ARM 2007)*, 2007. to appear.
- [32] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed aop. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes on Computer Science*, Bologna, Italy.
- [33] 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 Engineering (ICSE'01)*, 2001.
- [34] Eddy Truyen and Wouter Joosen. Run-time and atomic weaving of distributed aspects. *Transactions on Aspect-Oriented Software Development II*, 2006.
- [35] Giuseppe Valetto and Gail E. Kaiser. A case study in software adaptation. In *WOSS*, 2002.
- [36] A. Vasseur. Aop benchmark. <http://docs.codehaus.org/display/AW/AOP+Benchmark>.
- [37] E. Wohlstadter, S. Tai, T. A. Mikalsen, I. Rouvellou, and P. T. Devanbu. Glueqos: Middleware to sweeten quality-of-service policy interactions. In *ICSE*, 2004.
- [38] Eric Wohlstadter, Stoney Jackson, and Premkumar Devanbu. DADO: enhancing middleware to support crosscutting features in distributed, heterogeneous systems. In *ICSE 2003*, 2003.