

Adding Dynamic Reconfiguration Support to JBoss AOP

Nico Janssens, Eddy Truyen, Frans Sanen and Wouter Joosen
DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven, Belgium

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

ABSTRACT

The majority of aspect-oriented middlewares supporting dynamic aspect weaving fail to preserve important safety properties while weaving or unweaving a distributed aspect at runtime. This position paper looks in particular at the safety properties of structural integrity and global state consistency.

Preserving these two safety properties in the presence of dynamic change has already been extensively addressed in the space of dynamic reconfiguration of component-based distributed systems. As will be argued in this position paper, existing coordination protocols developed in this space can be largely reused for distributed aspect weaving provided that some small adaptations are made to account for the aspect-oriented composition mechanisms.

To demonstrate results and as a proof-of-concept, we describe how we have ported the NeCoMan dynamic reconfiguration support on top of the JBoss AOP framework. As a result, system-wide consistency can be preserved in JBoss when weaving or unweaving a distributed aspect at runtime.

Keywords: dynamic distributed aspect weaving, aspect-oriented middleware, dynamic reconfiguration of distributed systems, global state consistency, structural integrity

Categories and Subject Descriptors

D.2.11 [Software Engineering]: Software Architectures—*Domain-specific architectures*

1. INTRODUCTION

Since the traditional role of middleware is to hide resource distribution and platform heterogeneity, it is a logical place to deal with nonfunctional application concerns such as quality-of-service (QoS), fault tolerance and security. Because of their very nature, the implementation of these nonfunctional concerns often crosscuts multiple classes and components if one has to rely purely on object-oriented and component-based software development technologies [6].

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.

MAI '07, March 20, 2007 Lisbon, Portugal

Copyright 2007 ACM 978-1-59593-696-7/07/0003 ...\$5.00.

Besides, most of the nonfunctional concerns in distributed applications also crosscut the boundaries of a single computer node. Support to compress remote invocations, for instance, obviously needs to become integrated on at least two nodes of a distributed system (the sending and the receiving node). Aspect-oriented middleware (AOM) such as Lasagne [14], JAC [12], Prose [11], CAM/DAOP [3] and DyMAC [8] have therefore been developed to support the creation, deployment and execution of aspects for a distributed environment¹.

To enable building adaptive systems and accommodate the unanticipated evolution of highly available systems, a number of AOMs provide support for *dynamic* aspect weaving and unweaving at different nodes of a distributed system. This way, one can change at runtime which aspects are to be employed without shutting down and restarting the affected distributed applications. The majority of AOMs supporting dynamic aspect weaving, however, have focused primarily on providing the enabling technology to weave and unweave the behavior of a distributed aspect on-the-fly, but they do not assure that *system-wide consistency* will be preserved while weaving or unweaving a distributed aspect. Consequently, these AOMs do not guarantee that the system will not fail when dynamic aspect weaving is in progress. Since industrial middleware platforms such as JBoss and Spring are leveraging dynamic aspect weaving techniques, addressing this problem is of utmost importance for the sake of system integrity and robustness in general.

As we explain in more detail in Section 2, preserving system-wide consistency during dynamic reconfiguration includes maintaining two important safety properties: *structural integrity* and *mutual state consistency*. The achievement of both safety properties has already been addressed extensively in the space of dynamic reconfiguration of component-based distributed systems [7, 9, 1, 5]. This position paper argues that 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. Hence, existing coordination protocols (only) need to be adapted for going about with this different binding structure.

To illustrate this, we present a coordination protocol to conduct safe distributed aspect weaving. This protocol originates from one of the coordination protocols that the NeCoMan dynamic reconfiguration support employs [5, 4]. As

¹We refer to an aspect as a modular unit that implements a crosscutting concern.

Section 3 further explains in more detail, NeCoMan conducts (among others) safe distributed recompositions in programmable networks. To validate the resulting protocol, we extended JBoss AOP² with additional support for carrying out safe distributed aspect weaving.

As a second contribution, this paper indicates that existing coordination protocols can be reused if there is a clear separation between the *reconfiguration algorithms* (that implement the coordination protocols) on the one hand, and the primitive *reconfiguration operations* (that are used to link, unlink, activate, finish, etc. the affected units of reconfiguration in a platform-specific manner) on the other hand. By separating both concerns, we could reuse NeCoMan’s distributed reconfiguration algorithm to extend JBoss AOP with dynamic distributed aspect weaving support. In fact, we only had to provide a JBoss AOP implementation of the eight reconfiguration operations that the algorithm uses.

The remainder of this paper is structured as follows. Section 2 defines the distributed aspects that are subject for reconfiguration. Besides, this section also analyzes the safety properties that must be fulfilled to preserve system-wide consistency while dynamically weaving and unweaving distributed aspects. Next, Section 3 presents how NeCoMan conducts safe dynamic distributed reconfigurations of component-based network nodes. Section 4 then specifies how NeCoMan has been adapted to carry out safe distributed aspect weaving in JBoss AOP. After that, Section 5 compares our approach with related research. Finally, Section 6 summarizes this paper and discusses future work.

2. DYNAMIC DISTRIBUTED ASPECT WEAVING

We first briefly specify the properties of the distributed aspects that are subject for reconfiguration. Next, we present the safety properties that must be fulfilled to preserve system-wide consistency while weaving and unweaving these distributed aspects dynamically. These safety properties result from previous research to support dynamic reconfiguration in component-based distributed systems. Finally, we investigate what is needed for the current dynamic weaving support of JBoss AOP to fulfill these safety properties.

2.1 Distributed aspects

In general, distributed aspects enable to modularize concerns that crosscut the boundaries of different computer nodes. In the context of this research, we have focused on distributed cross-cutting concerns such as reliability, compression, fragmentation, encoding and encryption. These concerns each consist of *tightly coupled* functionalities located on different nodes, which need to collaborate remotely. Fragmentation, for instance, involves remotely cooperating functionalities responsible for fragmentation and reassembling, as illustrated in Figure 1.

Furthermore, the distributed aspects we focus on adopt a *client-server* collaboration model. According to [2], this implies that the client processes of the aspect initiate service activity, while the server processes wait for requests to be made and then reacts to them. This can again be illustrated with the fragmentation service, where the fragmenting functionality (hosting the client process) initiates service activity by invoking remote reassembling functionality (providing

the server process).

2.2 Consistency preservation

How to preserve system-wide consistency in the face of dynamic reconfiguration has been researched extensively in the field of component-based distributed systems. Goudarzi, for instance, describes in [9] 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 how these safety properties apply to dynamic distributed aspect weaving.

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. As Almeida and his colleagues correctly state in [1], these structural integrity requirements constrain the structure of a system in terms of the relationships between collaborating software modules³ and the ways in which these modules must be put together.

For distributed dynamic aspect weaving, preserving structural integrity involves respecting all causal dependencies in the course of ongoing client-server collaborations. Unweaving the reassembling behavior of a fragmentation aspect at the server-side, while the client is still using fragmenting functionality, for instance, obviously breaks the causality between the aspect’s fragmenting and reassembling behavior. This, in turn, compromises the correct functioning of the affected application. Most distributed aspects therefore must be bound atomically – that is, in an all or nothing fashion – as binding these aspects partially may result in inconsistent application behavior.

2.2.2 Reconfiguration-safe execution state

A second requirement for safe dynamic software reconfiguration relates to the execution state of the system being reconfigured. 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 reconfiguration-safe state is based on Kramer and Magee’s definition of “quiescence”. In [7], 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 collaborations. By freezing software modules, new collaborations are prevented from executing 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 distributed aspects, then quiescence comes about when (1) all ongoing collaborations between the distributed behavior of the aspect have completed,

³A software module may be a procedure, object, component, aspect, etc., depending on the employed programming model.

²<http://labs.jboss.com/jbossaop>

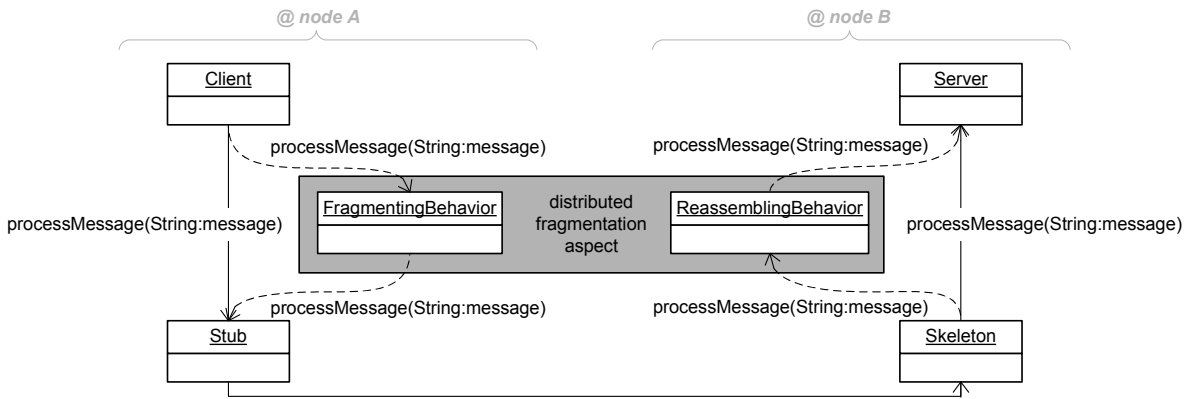


Figure 1: This figure presents an example of a distributed aspect. When the fragmentation aspect is woven, the execution of the application follows the dashed lines. If not, the regular invocation path is followed. Note that we will use this pedagogical example through the remainder of this paper.

and (2) no new collaborations will be initiated until after the reconfiguration has terminated. For the fragmentation aspect depicted in Figure 1 this implies that quiescence comes about when (1) all fragmented invocations in transit have been restored in their original form and (2) the fragmenting behavior is prevented from being invoked to fragment new invocations. If both pre-conditions are fulfilled, the fragmentation aspect can safely be 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 software modules in the system [9]⁴ We pre-assumed, however, that aspects do not have application-specific invariants, but depend on the base application for this through a well-defined interface. Hence, preserving application state-invariants will not be tackled in this paper.

2.3 Dynamic weaving of distributed aspects in JBoss AOP

We analyzed the dynamic weaving support of JBoss AOP with respect to the safety properties outlined above. We first indicate how we have implemented distributed aspects on top of JBoss AOP. Then we present our conclusions.

Currently, on top of JBoss AOP, distributed aspect behavior is implemented as a bundle of JBoss AOP Interceptors (e.g. one fragmenting and one reassembling bundle)⁵. These interceptors are then bound to the application by an appropriate set of AdviceBinding instances. Technically speak-

⁴A typical example to illustrate this involves the replacement of a module that generates unique ID's. For this replacement to be legal, the new generator should not repeat ID's that were already produced by the old version. To preserve this invariant, the new generator must be initialized in a state which prevents it from producing ID's that were already generated by the old module.

⁵Note that, as AOMs differ in the programming model that they offer, there are also other ways for implementing distributed aspects. For instance, JBoss AOP does not support the concept of remote pointcut as JAC [12] or DyMAC [8] do.

ing, an AdviceBinding instance is an object that pairs a JBoss AOP Interceptor to a pointcut specification. As remote pointcuts are not supported, separate AdviceBinding instances must be deployed on the different nodes of the distributed application.

JBoss AOP offers an AspectManager API for adding and removing advice bindings at run-time. The AspectManager abstraction is local however in the sense that AspectManager entities which are located on different hosts do not synchronize with each other to ensure coordinated weaving of the various advice bindings. As such, we observed that JBoss AOP does not support any coordination protocol for safe dynamic weaving of distributed aspects. A number of preliminary tests have furthermore indicated that the current version of JBoss AOP does not maintain consistency either when weaving multiple aspect bindings in the context of a single node.

The philosophy of JBoss AOP is here that it is the aspect developer's responsibility to correctly coordinate the (local and distributed) execution of weaving and unweaving actions. Our previous experiences with safe dynamic software reconfiguration, however, have indicated that writing such ad-hoc consistency checks can be very complex and error-prone.

We therefore argue that specific reconfiguration support must be layered on top of the AspectManager API of JBoss AOP to assist a developer in safely weaving and unweaving distributed aspects. This reconfiguration support must (1) coordinate the correct weaving and unweaving of the various Interceptors across the distributed system, and (2) conceal the complexity of these reconfigurations from the developer. To accomplish this, we started from the reconfiguration support that NeCoMan provides, as NeCoMan tackles a similar problem in the context of programmable networks. We therefore briefly describe this middleware in the next section.

3. THE NECOMAN DYNAMIC RECONFIGURATION SUPPORT

In short, we previously developed the NeCoMan reconfiguration support to coordinate the addition, replacement and removal of network services that are similar to the services

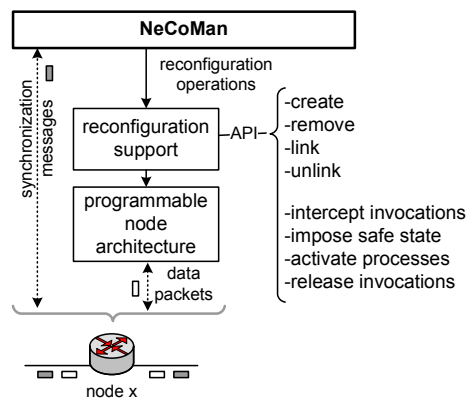


Figure 2: High-level overview of the NeCoMan architecture.

encapsulated by the distributed aspects this paper focusses on [5, 4]. The emphasis of NeCoMan is on the dynamic reconfiguration and evolution of *component-based* applications. Components have well-defined ports (i.e. interfaces) and connectors help to decouple components from one another. Based on the hypothesis that aspects do not fundamentally differ from components (except for using a different composition structure), we have investigated to what extent NeCoMan’s reconfiguration support can be reused for conducting safe distributed aspect weaving. Before elaborating on the latter, we briefly sketch NeCoMan’s architecture as well as the algorithm that has been transformed for conducting safe distributed aspect weaving.

3.1 NeCoMan architecture

To support reuse, NeCoMan’s reconfiguration support has been separated from all functionality directly related to (1) the affected network service and (2) the node architecture being recomposed. To be precise, NeCoMan only coordinates the execution of a reconfiguration by instructing the underlying node to carry out (the node-specific implementation of) a number of *reconfiguration operations*. Each node therefore has to offer dedicated “reconfiguration support” to assist NeCoMan in carrying out a distributed reconfiguration (see Figure 2). This separation of concerns allows to reuse NeCoMan’s reconfiguration support for different architectures (such as JBoss AOP) by only replacing the employed reconfiguration support.

We analyzed that, to achieve this separation of concerns, a node’s reconfiguration support must provide eight *reconfiguration operations* (see again Figure 2). Four of these operations serve to change the affected composition. As Kramer and Magee proposed in [7], changes to a composition should be expressed in terms of its structure. Hence, to assist NeCoMan in modifying a composition, a node’s reconfiguration support must offer *create*, *remove*, *link* and *unlink* operations which implement the following behavior:

- **Create** involves loading the specified component into the system – that is, without connecting its communication ports.
- **Remove** deletes a disconnected component from the system.
- **Link** is responsible for connecting a specified commu-

nication port of a component.

- **Unlink** disconnects a specified communication port of a component.

Besides assisting NeCoMan in changing a composition, a node’s reconfiguration support must also enable NeCoMan to control a component’s execution state in a generic way. This can be accomplished by the providing the following four operations: *intercept invocations*, *impose safe state*, *activate processes* and *release invocations*.

- **Intercept invocations.** Reaching a reconfiguration-safe state requires that components can be frozen. This is achieved by intercepting invocations – that is, to prevent these invocations from invoking the affected component. The “intercept invocations”-operation is responsible for providing this functionality.
- **Impose safe state.** Besides freezing, additional support is needed to drive a component to a safe (quiescent) state. This includes, among others, monitoring the affected component until a safe state comes about and/or transferring this component’s state-information towards its new counterpart. This support is offered by the “impose safe state”-operation.
- **Activate processes.** To activate a new component, all its active objects (if any) must be started. The “activate processes”-operation assists NeCoMan in accomplishing this.
- **Release invocations.** Finally, once a recomposition has completed, all intercepted invocations must be resumed. This is covered by the “release invocations”-operation.

The next subsection presents the algorithm that we have transformed for conducting safe distributed aspect weaving. As will be illustrated soon, this algorithm coordinates the distributed execution of the eight reconfiguration operations listed above⁶.

3.2 Reconfiguration algorithm

Figure 3(a) depicts a Petri net model of this algorithm⁷. Each transition in this model represents the execution of a single *reconfiguration action*. These reconfiguration actions each implement a specific reconfiguration subtask, such as loading a new component into a node’s composition, activating this component’s active objects, disconnecting an old component, etc. To implement these specific subtask, each reconfiguration action combines the execution of the reconfiguration operations listed above (as depicted in Figure 3(c)).

We illustrate this algorithm in Figure 3(b) with the replacement of an old fragmentation service (represented by fragmenting component FC_{old} and reassembling component RC_{old}) with a new version (represented by FC_{new} and RC_{new}). This reconfiguration begins by installing FC_{new} and RC_{new}

⁶A thorough description of this algorithm can be found in [4], Chapter 5.

⁷Note that the mathematical representation of Petri nets will not be employed in this paper. Instead, we use Petri nets only as a modeling language to visualize the distributed execution of the reconfiguration actions

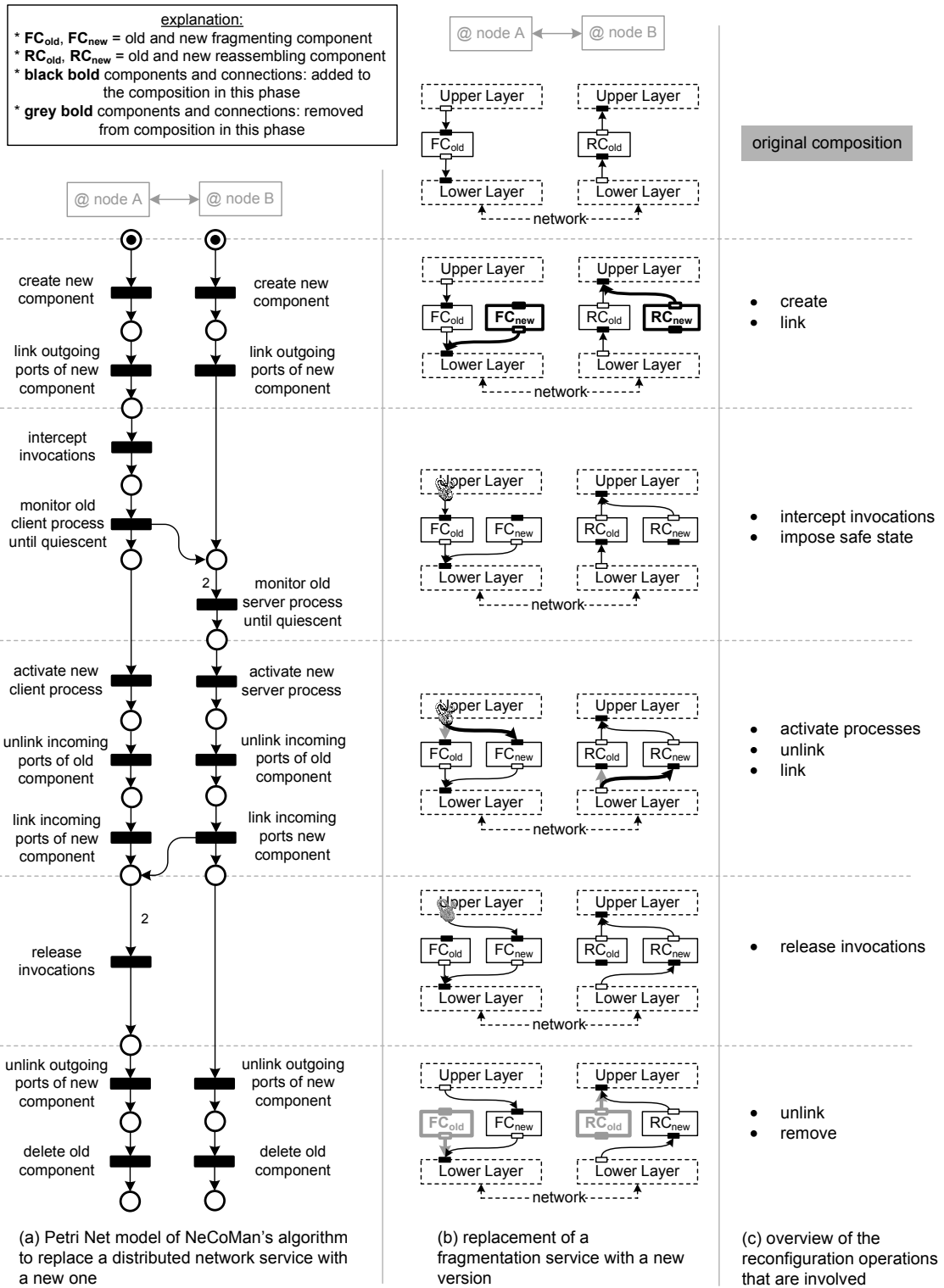


Figure 3: NeCoMan conducting a distributed recomposition in a programmable network.

on the sending and receiving node, respectively. To accomplish this, NeCoMan instructs both nodes (independently) to create these components. Next, NeCoMan invokes both nodes to connect FC_{new} and RC_{new} into their composition. Because the new fragmentation service should not be acti-

vated in this phase, the latter is limited to connecting the outgoing ports of both components⁸.

⁸According to the adopted component model, a component delivers an invocation through its *outgoing port* to the *in-*

Next, NeCoMan drives FC_{old} and RC_{old} to a reconfiguration-safe (quiescent) state. To accomplish this, NeCoMan first instructs the sending node to intercept invocations directed to FC_{old} . After that, NeCoMan instructs the same node to impose a safe state over FC_{old} . As a result, the sending node monitors FC_{old} until all accepted invocations are fragmented. Once this occurs, NeCoMan transmits a synchronization message towards the receiving node to impose a safe state over RC_{old} as well. The receiving node then monitors RC_{old} until all fragmented invocations that are still in transit have been reassembled. At this point, quiescence is reached.

As soon as FC_{old} has reached a reconfiguration-safe state, NeCoMan instructs the sending node to start FC_{new} 's active objects (if any), to unlink the incoming ports of FC_{old} and to link those of FC_{new} . As a result of these link and unlink actions, invocations will be delivered to FC_{new} instead of to FC_{old} after releasing intercepted invocations. Simultaneously, NeCoMan instructs the receiving node to do the same for RC_{new} once RC_{old} has reached a reconfiguration-safe state. After linking the incoming ports of RC_{new} , NeCoMan sends a synchronization message from the receiving node towards the sending node. When this message has arrived and FC_{new} 's incoming ports are linked, NeCoMan instructs the sending node to release all intercepted invocations. At this point in the reconfiguration process, the new fragmentation service processes all invocations in transit.

Finally, NeCoMan removes FC_{old} and RC_{old} from both nodes. This involves first disconnecting both components by unlinking their outgoing ports. Next, NeCoMan instructs the affected nodes to delete FC_{old} and RC_{old} . Since the old fragmentation service is quiescent, the removal of these components will not compromise the correct network operation.

4. NECOMAN AND JBOSS AOP

Reusing NeCoMan to conduct dynamic distributed aspect weaving in JBoss AOP required the following two adaptations. First, we had to provide a JBoss AOP-specific implementation of the eight reconfiguration operations that NeCoMan employs. This will be discussed briefly in the next subsection. In addition, we also had to adapt the employed reconfiguration algorithm to deal with the different binding structure (which is inherent in aspect-oriented composition). We elaborate on this in subsection 4.2.

4.1 Reconfiguration operations

The *create*, *link*, *unlink* and *remove* operations are implemented by using JBoss' AspectManager API. Create and remove, for instance, have been implemented by using the AspectManager's support to add and remove AspectDefinitions. Link and unlink, for their turn, involve simply adding and removing a specified AdviceBinding.

When NeCoMan invokes the *intercept invocations* operation, it passes the AdviceBinding where invocations must be intercepted. A dedicated interceptor then becomes woven dynamically conform the pointcut of the specified binding. This dedicated interceptor blocks invocations by adding them to a buffer (which is implemented as a mixin). To *release invocations* afterwards, all invocations from this buffer are invoked again, and the Interceptor becomes dynamically unwoven. Note that we are currently also experimenting

coming port of another component.

with a different interceptor that blocks the invoking execution thread instead of queueing its invocations.

Finally, the implementation of the *impose safe state* and *activate processes* operations depends very much on the semantics of the affected aspect. It is the developers responsibility, therefore, to provide code for reaching a safe state and activating an aspect's active objects. Note that we currently use a dummy implementation for the impose safe state operation. Instead of monitoring until an interceptor reaches a quiescent execution state, we wait for 500 ms to make sure that all invocations have been processed. As this pragmatic approach compromises the efficiency of a reconfiguration, however, a developer can also override the *impose safe state* operation to employ its own (customized) strategy for reaching a safe state (e.g. by monitoring some variables of the affected aspect).

4.2 Reconfiguration algorithm

Besides providing a JBoss AOP implementation of the eight reconfiguration operations, we also had to slightly adapt the employed algorithm. When conducting a distributed recomposition, the (original) algorithm distinguishes between incoming and outgoing ports. When binding an interceptor, however, this distinction becomes irrelevant. The reconfiguration actions to link a new component's incoming and outgoing ports, therefore, have been combined to a new action "bind new interceptor". Similarly, the actions to unlink an old component's incoming and outgoing ports have been combined to a new action "unbind old interceptor". To illustrate the effect of these changes, Figure 4(a) depicts a Petri net model of the resulting algorithm.

We explain this algorithm in Figure 4(b) with the replacement of a distributed fragmentation aspect (represented by fragmenting interceptor FI_{old} and reassembling interceptor RI_{old}) with a new version (represented by FI_{new} and RI_{new}). NeCoMan starts this reconfiguration by instructing both nodes to create FI_{new} and RI_{new} . Note that in contrast to a distributed recomposition, this does not include any binding operations. Next, NeCoMan instructs both nodes to reach a reconfiguration-safe (quiescent) execution state. This is similar to distributed recompositions.

The activation of the new aspect, however, is slightly different. Besides starting the new interceptor's active objects (if any), NeCoMan instructs both nodes to *unbind* the old interceptor and to *bind* the new one. This bind action is the equivalent of unlinking both a component's incoming and its outgoing ports. Similarly, the unbind action is the equivalent of unlinking both a component's incoming and outgoing ports. When conducting a distributed recomposition, in contrast, NeCoMan only instructs the affected nodes to link and unlink *incoming ports*⁹ during activation (see Figure 3(a)). This is because when conducting recompositions, a component's outgoing ports become linked and unlinked after creating and before deleting the affected components, respectively.

Finally, when the new fragmentation aspect is brought into use, the old one can safely be removed. This involves only deleting the affected interceptors as their bindings have already been removed when activating the new aspect.

⁹instead of linking and unlinking the outports as well

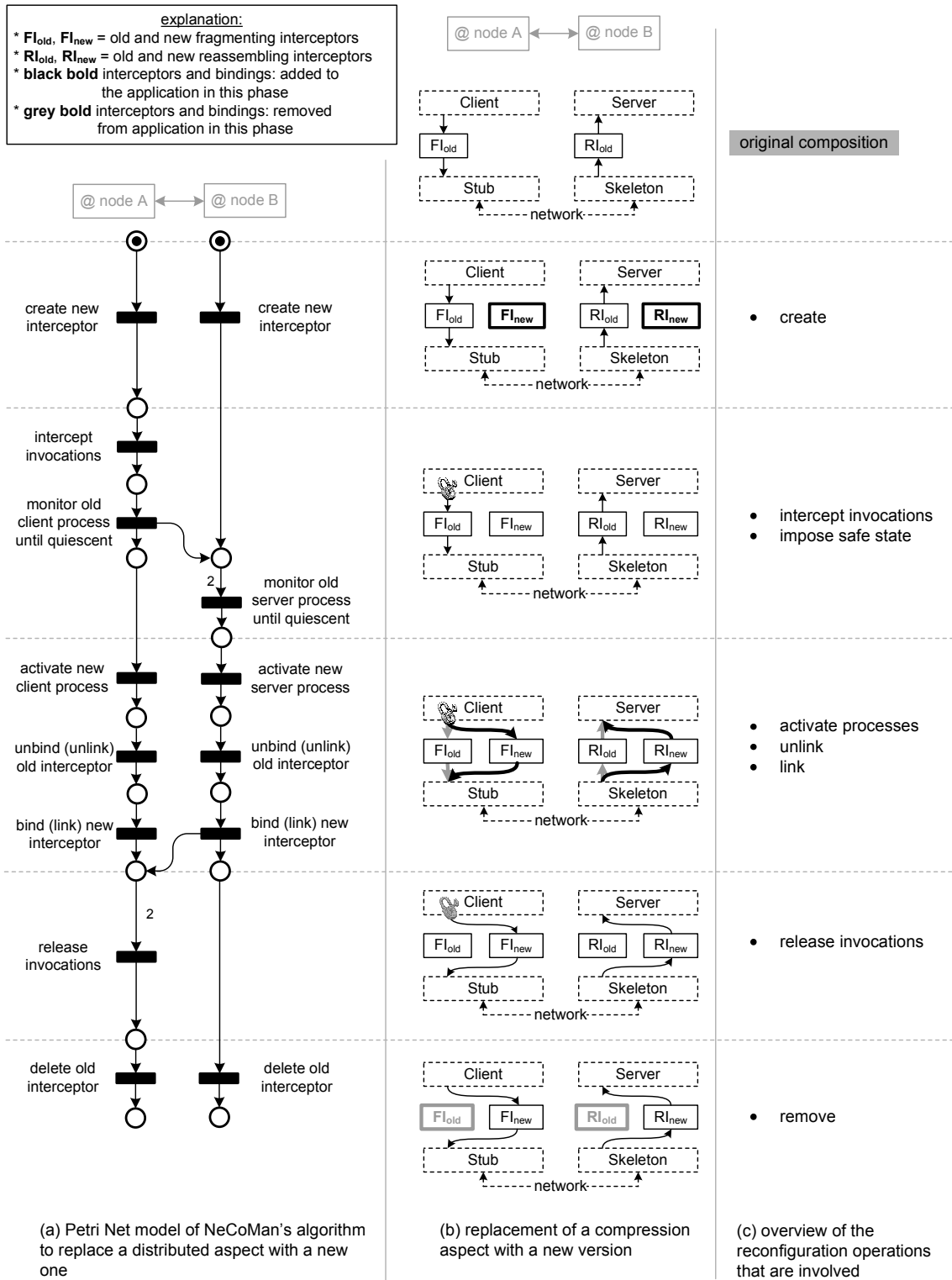


Figure 4: NeCoMan conducting distributed aspect weaving in JBoss AOP.

5. RELATED RESEARCH

Before summarizing this paper, we briefly compare our research with Tanter and Toledo's versatile kernel for AOP [13], AWED [10], Lasagne [14] and Prose [11].

Tanter and Toledo's versatile kernel for AOP supports runtime link manipulation to dynamically deploy/undeploy distributed aspects [13]. Besides, the "remote consistency framework" of the employed ReflexD library maintains (structural) consistency between changes made to links in different

hosts. Reaching a reconfiguration-safe state to safely undeploy stateful aspects, however, seems not be supported.

AWED [10] 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. Similar to Tanter and Toledo's versatile kernel for AOP, no support is provided to reach a reconfiguration-safe state before unweaving an aspect. Mutually consistent execution states can be preserved, however, 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.

To preserve all causal dependencies during dynamic aspect weaving (and unweaving), we have earlier presented a model and an architecture for middleware, called Lasagne, that supports run-time weaving of distributed aspects in an atomic way [14]. Once an invocation becomes tagged with a so called "aspect identifier", the latter propagates with the message flow of the entire collaboration. On the nodes where needed, Lasagne activates the aspect that the identifier specifies. Hence, Lasagne supports in-band reconfigurations of stateless aspects, whereas the work presented in this paper is targeted at our-of-band reconfigurations that may involve stateful aspects.

Finally, also Prose [11] provides dedicated run-time weaving support. Again this support does not deal with maintaining system-wide consistent execution states.

6. SUMMARY AND FUTURE WORK

In this paper, we analyzed the safety properties that must be fulfilled to preserve system-wide consistency while dynamically weaving and unweaving distributed aspects. Furthermore, we presented a coordination protocol to conduct safe dynamic distributed aspect weaving. This protocol results from NeCoMan's support to conduct safe dynamic distributed recompositions in programmable networks. By separating the employed reconfiguration algorithm from the primitive reconfiguration operations, we could reuse this support to extend JBoss AOP with dynamic distributed aspect weaving support. Future work involves a thorough validation of this approach.

This validation will focus (among others) on performance overhead. A drawback we experienced with Lasagne, is that it increases the processing time during normal execution. This has been one of the motivations to research the use of NeCoMan to support dynamic distributed aspect weaving. We therefore plan to investigate to what extent out-of-band distributed aspect weaving (as supported by the NeCoMan approach) may perform better than in-band distributed aspect weaving (as applied by Lasagne).

Besides, we also plan to investigate possible performance optimizations of the presented algorithm (as we already did in the context of distributed recompositions [5, 4]). If an aspect has no external state dependencies towards invoking clients, for instance, we could activate the new interceptors before driving the old ones to a reconfiguration-safe state. By doing so, we can reduce the service disruption caused by reaching quiescence, as the latter will occur while the new aspect is already brought into use.

7. REFERENCES

- [1] J. P. A. Almeida, M. Wegdam, M. van Sinderen, and L. J. M. Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *Proceedings of the 3rd International Symposium on Distributed Objects and Applications (DOA 2001)*, pages 197–207, Rome, Italy, September 2001. IEEE Computer Society.
- [2] G. R. Andrews. Paradigms for process interaction in distributed programs. *ACM Computing Surveys*, 23(1):49–90, 1991.
- [3] L. Fuentes, M. Pinto, and P. Sánchez. Dynamic weaving in cam/daop: An application architecture driven approach. In *Proceedings of the Dynamic Aspect Workshop in conjunction with AOSD 2005*, Mar. 2005.
- [4] N. Janssens. *Dynamic Software Reconfiguration in Programmable Networks*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, December 2006.
- [5] N. Janssens, W. Joosen, and P. Verbaeten. NeCoMan: Middleware for Safe Distributed-Service Adaptation in Programmable Networks. *IEEE Distributed Systems Online*, 6(7), July 2005.
- [6] 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*. Springer, 1997.
- [7] J. Kramer and J. Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.
- [8] 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.
- [9] K. Moazami-Goudarzi. *Consistency preserving dynamic reconfiguration of distributed systems*. PhD thesis, Imperial College, London, March 1999.
- [10] L. D. B. Navarro, M. Südholt, W. Vanderperren, B. D. Fraine, and D. Suvé. Explicitly distributed aop using awed. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 51–62. ACM Press, 2006.
- [11] A. Nicoara and G. Alonso. Dynamic aop with prose. In *Proceedings of International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005)*, June 2005.
- [12] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Reflection 2001*, volume 2192 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2001.
- [13] É. Tanter and R. 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*, pages 316–331, Bologna, Italy. Springer-Verlag.
- [14] E. Truyen and W. Joosen. Run-time and atomic weaving of distributed aspects. *Transactions on Aspect-Oriented Software Development II*, pages 147–181, 2006.