

Towards Transparent Hot-Swapping Support for Producer-Consumer Components *

Nico Janssens, Sam Michiels, Tom Mahieu, Pierre Verbaeten
DistriNet, Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A
3001 Leuven, Belgium
{Nico.Janssens@cs.kuleuven.ac.be}

October 2002

Abstract

Unanticipated software adaptations are becoming increasingly important in the domain of computer networks. Due to the performance and availability requirements of computer networks, these adaptations need to be enforced at runtime (by means of hot-swapping). However for dynamic changes to yield valid systems, a safe state for reconfiguration of the involved software modules must be enforced. This paper presents a method to impose such a safe state for hot-swapping with minimal interference to the rest of the system, and with minimal contribution from the programmer. We believe that a wide range of producer/consumer based systems can take advantage from the presented solution.

1 Introduction

Software evolution in computer networks is becoming increasingly important with the proliferation of *unanticipated adaptations*. Examples of such adaptations range from security patches to application specific alterations (both functional and non-functional). However, the protocol stacks that belong to these computer networks (and that are subject to software adaptations) are highly critical systems. They cannot be brought down easily or be switched off for a long time because of availability and performance requirements. As such, software adaptations of protocol stacks should be accomplished *dynamically* [3], without stopping those parts of the stack that are unaffected by the change (hot-swapping).

When zooming in on the type of services that are typically presented by protocol stacks, it catches the eye that most of these services are producer/consumer based. As an example, we refer to a fragmentation service (such as used by IP), consisting of a "fragmenter" component (breaking up each packet into a number of fragments) at the sending protocol stack and a "reassembler" (restoring the original packet) as its consuming counterpart, located at the receiving stack. In general, the producer/consumer model consists of two *tightly coupled* components that are cooperating (in the same program, or in a distributed environment) to implement some service. In addition, the correct functioning of both the producer and consumer does not depend on external services offered by the system they belong to, and vice versa. Stated differently, we assume independent communication between the producer or consumer on the one hand, and the rest of the system on the other hand.

Due to their interdependencies, the *cooperation* between producer and consumer is required to successfully perform the service. This cooperation is defined by means of a *protocol transaction*, consisting of a sequence of one or more asynchronous *interactions* between the producer and consumer (as illustrated in figure 1). Referring to the fragmentation service, a protocol transaction to fragment and reassemble a packet encapsulates a number of interactions, each realizing the transfer of one fragment from the

*Research for this paper has been carried out with financial support of the Fund for Scientific Research Flanders, Belgium – F.W.O. RACING # G.0323.01

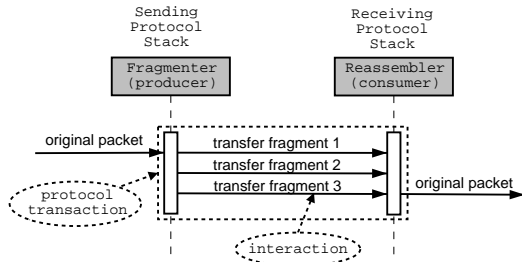


Figure 1: Protocol transaction between **fragmenter** and **reassembler**

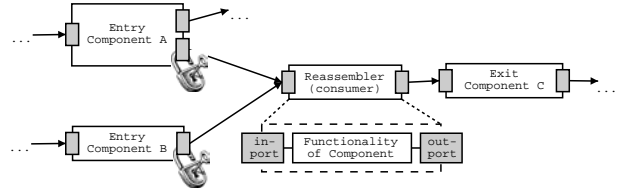


Figure 2: Communication between software modules

fragmenter to the **reassembler**. This cooperation implies that, from a reconfiguration point of view, both the producer and the consumer are only consistent after termination of a protocol transaction.

In case of producer/consumer dependencies, two kinds of reconfigurations can be enforced:

- **isolated adaptations.** In this case, replacing the functionality of the producer will not break the semantic consistency with the consumer (and vice versa), nor will the communication protocol be changed. An example of such adaptation is the replacement of the **fragmenter** by a new version extended with additional non-functional support (such as logging). This will not affect the correct functioning of the existing **reassembler**.
- **structural adaptations.** Both the producer and the consumer need to be replaced in order to preserve semantic consistency. As an example the **fragmenter** could be replaced by a new counterpart that is extended with error-correction support. It is obvious that (for the fragmentation service to work correctly) also the **reassembler** will have to be replaced by a new version that as well can handle error-correction.

In the remainder of this position paper, we focus on the first type of run-time adaptations. We are concerned with imposing a *safe state* for **consumer reconfigurations**, taking into account the dependencies (formalized by the protocol transaction) between producer and consumer. Since we consider this problem from a software engineering perspective, a safe state should be obtained with minimal interference to the rest of the system and with minimal contribution from the programmer.

In order to make minimal interference to the rest of the system imperative, we make the following assumptions:

- A consumer is reactive: it can only accept and serve requests. A consumer will never autonomously initiate requests (that is, for a purpose different from servicing a request).
- There is no opportunity to interfere in the functioning of the producer (e.g. by stopping its activities). This restriction emerges when the producer is part of a black-box application that has been developed without hot-swapping support in mind (such as a fragmenter that belongs to a legacy protocol stack), or when performance issues exclude adaptations of the producer.

2 Architectural Support for Safe Reconfiguration

Kramer and Magee [6] have stated that achieving safe software reconfigurations requires the software modules (in this context the consumer or **reassembler**) to be both *consistent* and *frozen*. When software modules are consistent, they don't include results of partially completed services (or protocol transactions). By forcing software modules to be frozen, state changes caused by new protocol transactions are impossible. Kramer and Magee describe this required consistent and frozen state as the *quiescence* of a component.

Starting from this definition, a quiescent state is acquired when a software module is not participating in a protocol transaction and has no pending protocol transactions, which it must accept and service. Kramer and Magee propose a mechanism to impose such a quiescent state by means of a configuration manager, which is employed to recognize and deactivate (or freeze) the relevant transaction initiators [6].

These transaction initiators are the components in the system that are able to start protocol transactions capable of causing state changes on the components that are targeted for reconfiguration.

Illustrated through the fragmentation service, the **reassembler** will automatically become quiescent when it has received all fragments and has restored the original packet. Applying the solution of Kramer and Magee to impose this state over the **reassembler**, would result in deactivating the **fragmenter** when its protocol transaction is completed (that is, when all fragments are sent out). However, as stated in the introduction, this is not a feasible solution since we have excluded the opportunity to interfere in the functioning of the producer.

In addition, this solution also causes inconvenience to the programmer of software modules, who has to *embed extra code* for deactivation in order to acquire a quiescent state for other software modules. When referring to the fragmentation service, the **fragmenter** should become extended with support to deactivate itself after completing its service.

2.1 Decoupling Functionality from Blocking Support

The problems stated in the previous section are caused by a high coupling between the functionality of software modules (offered by a programmer), and additional support to deactivate the software module. For that reason, we propose a means to separate both aspects.

This separation is accomplished by extending software modules with communication ports (as illustrated in figure 2). Two kinds of communication ports are provided: the *inport*, which is the interface of the software module, and the *outports*, of which each defines interactions with another software module (by using its inport). Both kinds of communication ports are separated from the functional behavior of the component, and offer indirections that can be used to manipulate interactions (that are part of protocol transactions) initiated between modules.

As an example, we refer to figure 2. In order to force the **reassembler** to become quiescent, all interactions directed to that module are intercepted. This is done by holding up all outgoing interactions of adjacent outports that are directed to the **reassembler**. When the reconfiguration is completed, the execution of these blocked interactions will be resumed.

2.2 Advantages

This approach offers a number of advantages:

First of all, the functional behavior of a module is decoupled from support to block interactions that cause state changes on other components. As such, changing the way of holding up interactions at the outports will not interfere with existing module functionality and vice versa. As an example, the possibility to choose between two different blocking strategies is illustrated in figure 3. To obtain safe reconfiguration, one could decide to *block* the execution thread in which the outgoing interactions are initiated, using the **ThreadBlockingStrategy**. An alternative could be to *queue* outgoing interactions without interrupting the execution thread by selecting the **PacketQueueingStrategy**. Such a change can be achieved by only adapting the outports of the modules that are involved.

Secondly, the impact of a blocking operation on a software module can be made more fine-grained. Instead of stopping all interactions initiated by a module (e.g. by interrupting the execution thread of that software module), only the outports initiating interactions that engage components which need to become quiescent should get blocked (illustrated by **Entry Component A** in figure 2).

Finally, minimal interference to the rest of the system can be guaranteed. Interrupting interactions in a composition can be restricted to those locations where an actual reconfiguration is needed. Instead of freezing the producer (as proposed in [6] and [1]), only the adjacent modules that may cause state changes of the consumer need to get blocked (as illustrated in figure 2). As such, there is no need to (unnecessarily) drain the complete pipe between the producer and the consumer.

3 State Consistency Issues

Separating the functional behavior of a module from potential support to block its outgoing interactions is not enough to enforce safe reconfiguration. Since there is no knowledge about the state of the consumer at

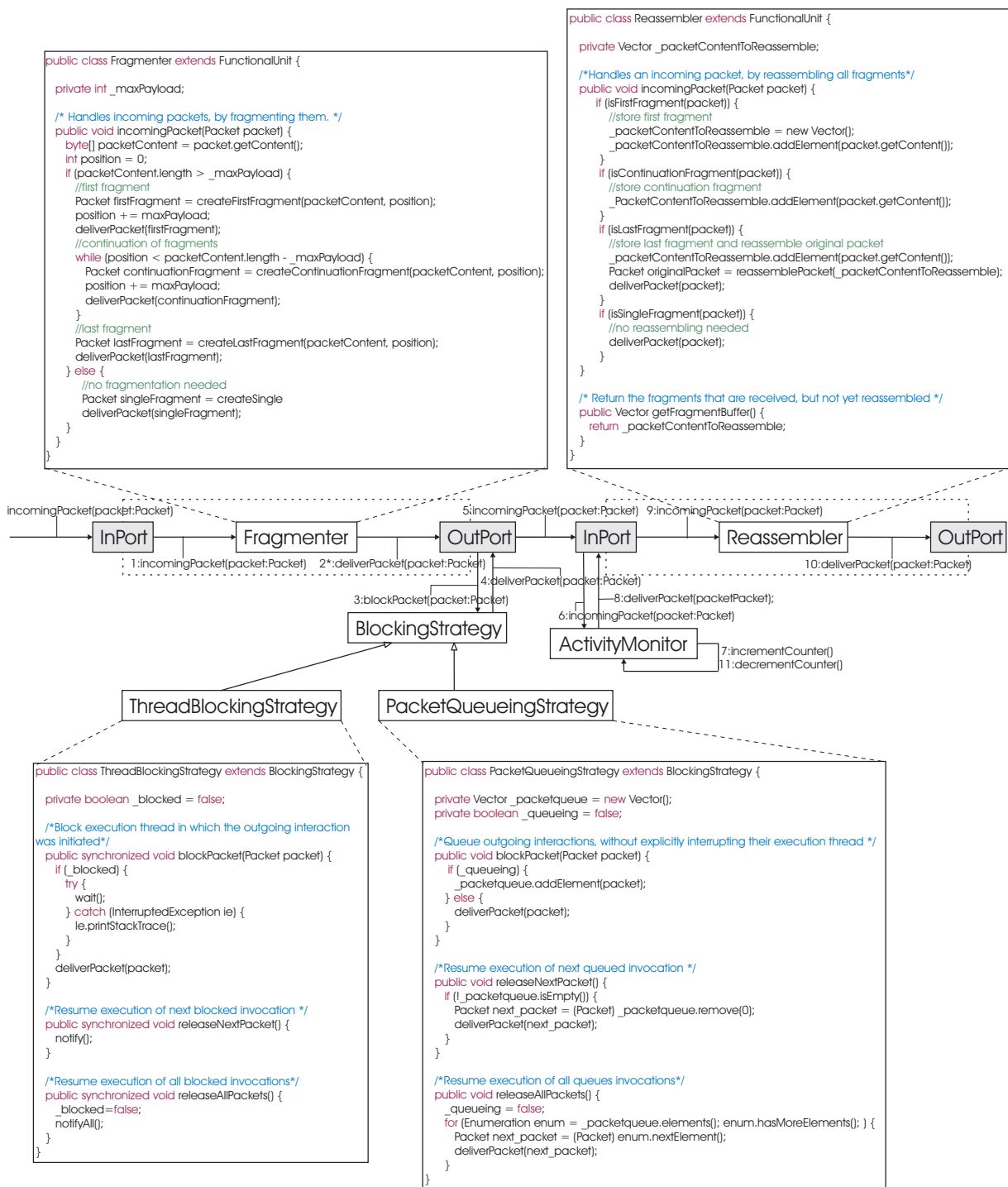


Figure 3: This example represents a simplified setup of the fragmenter/reassembler service: a **Fragmenter** is directly connected to a **Reassembler**, rather than cooperating in a distributed environment. In order to impose a quiescent state on the **Reassembler**, all interactions to that module will get intercepted. This results in extending the outport of the **Fragmenter** with a **BlockingStrategy**. Selecting a **ThreadBlockingStrategy** or **PacketQueueingStrategy** does not bring about any changes in the original code of the **Fragmenter**.

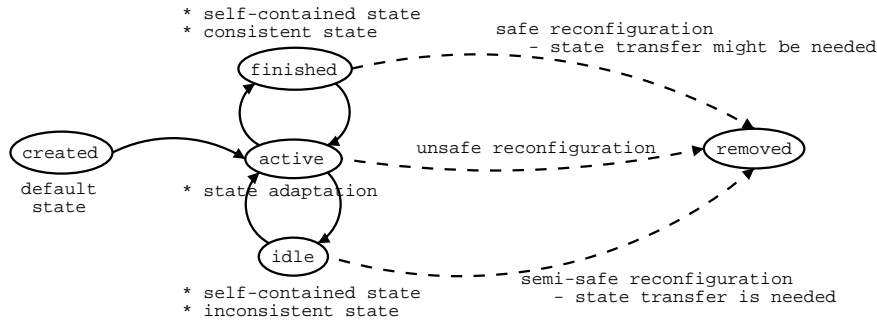


Figure 4: Component execution state machine

the moment interactions are blocked, reconfiguration may lead to inconsistency (caused by replacing the consumer when protocol transactions are only partially completed). When referring to the fragmentation service, replacing the **reassembler** at the moment when it has not yet received all fragments (and as such could not reassemble the original packet) will break the consistency between producer and consumer (and in that way, the correct functioning of the fragmentation service).

This flaw can be corrected by providing "controlled" interaction blocking support. After blocking the outputs that are directing interactions to the consumer, it should be possible for the reconfiguration system (which conducts the actual reconfiguration) to check whether *safe reconfiguration* of the consumer is achievable. When this is not the case, blocked interactions are resumed one by one until the required safe state for reconfiguration is attained (e.g. by using the `releaseNextPacket()` of the blocking strategy as illustrated in figure 3).

3.1 Component Execution State

Checking whether safe reconfiguration of a consumer module is achievable requires interaction with that module. For that purpose, we propose to extend consumer modules (that are eligible for reconfiguration) with monitoring code to reflect their current execution state. This state should become abstracted into a set of generic execution states, representing the execution model of consumer components that participate in protocol transactions. Instead of only an active (during processing of a protocol transaction) and a passive (after finishing a protocol transaction) state, as described in [6], we propose an execution model consisting of three states: *active*, *idle* and *finished* (see figure 4)

- **active**. A software module is active when it is *engaged in servicing an interaction*. Stated differently, a software module is active when it is processing an invocation initiated by another software module. Illustrated by means of the fragmentation service, a **reassembler** is active while engaged in the processing of a received fragment. At this point, the state of a software module is *not self-contained*. As such, a reconfiguration would break the consistency of the software modules involved in the reconfiguration. It is obvious that safe reconfiguration is impossible at this point.
- **idle**. A software module is idle when it is *not engaged in servicing an interaction* but is still *participating in the protocol transaction*. Applied to the **reassembler**, its execution state is idle when it is expecting but not currently processing a new fragment. At this point, the software module is *self-contained*. However, since there is no guarantee about termination of the protocol transaction, reconfiguration could endanger the consistency. As such, safe reconfiguration can only be guaranteed by means of *additional consistency recovery* support, which requires software modules to capture and reinstate module specific application-state at runtime [2]. Referring to the intended **reassembler** replacement, safe reconfiguration can only be accomplished when all the received fragments are transferred from the old **reassembler** to its new counterpart.
- **finished**. A software module is finished when it is *not engaged in a protocol transaction* (or stated differently, when a complete protocol transaction is terminated). Referring to the **reassembler**, the finished state is reached when all fragments are received and the original packet is reassembled.

At this point, the state of a software module is both *self-contained* and *consistent*. As such, *safe reconfiguration* is achievable.

3.2 Towards Automatic Component Extension

As stated in the introduction, it is one of our goals to obtain a safe state for reconfiguration with absolute minimal contribution from the programmer of the consumer component. Hence, extending modules with monitoring code to reflect their current execution state should become automated by appropriate tool support.

3.2.1 Activity monitor

Due to the reactive behavior of the consumer, monitoring code to check whether the consumer is *active* or *idle* can (automatically) be added by only extending its communication ports. In case of concurrent interactions, activity inside a software module can be monitored by means of a counter located at its inport, which is incremented on invocation and decremented on return [7] (illustrated by means of the `ActivityMonitor` in figure 3). When only sequential interactions are used, a counter can be replaced by means of a boolean flag. This reduces the monitoring overhead for each interaction.

3.2.2 Component invariant

However, to check whether an *idle* software module is *finished*, input from the programmer of the software module is required. This input is limited to the definition of a *component invariant*, indicating the termination of the protocol transaction. Stated differently, the component invariant describes the internal state of the consumer for which safe reconfiguration can be achieved. In case of replacing the `reassembler` (containing the functionality as described in figure 3), the component invariant asserts that the reassembler does not contain fragments that are not yet reassembled into the original packet.

When extending component functionality to check the invariant is impossible, the component invariant should be examined at the communication inports by means of additional monitoring code. However, this approach brings along a number of important drawbacks. On the one hand, extending the monitoring code at the communication ports comes at the cost of performance reduction, since for each interaction the current component execution state is recalculated. On the other hand, writing external monitoring code that reflects the internal state of the component is liable to be a difficult task and might result in essentially replicating the component's code. This limits the ability to (automatically) extend components that with monitoring code for checking whether safe reconfiguration is achievable.

In order to overcome these limitations, we have opted for checking the component state by means of introspection, rather than monitoring its interactions. Current technologies such as Aspect-Oriented Programming[5] provide a useful means to extend existing code with such state checking support in a modularized way. To illustrate this, listing 1 presents an AspectJ[4] extension of the Reassembler (as illustrated in figure 3) defining the monitoring code for checking whether the module is finished based on the definition of the component invariant. The reconfiguration system (which conducts the actual reconfiguration) will only check this state in the face of an actual reconfiguration when the targeted component is idle.

```
public aspect ReassemblerFinishedStateAspect {

    public boolean Reassembler.isFinished() {
        return getFragmentBuffer().size() == 0; // implementation of
                                                // component invariant
    }
}
```

Listing 1: an AspectJ extension of the Reassembler defining the monitoring code for checking its invariant

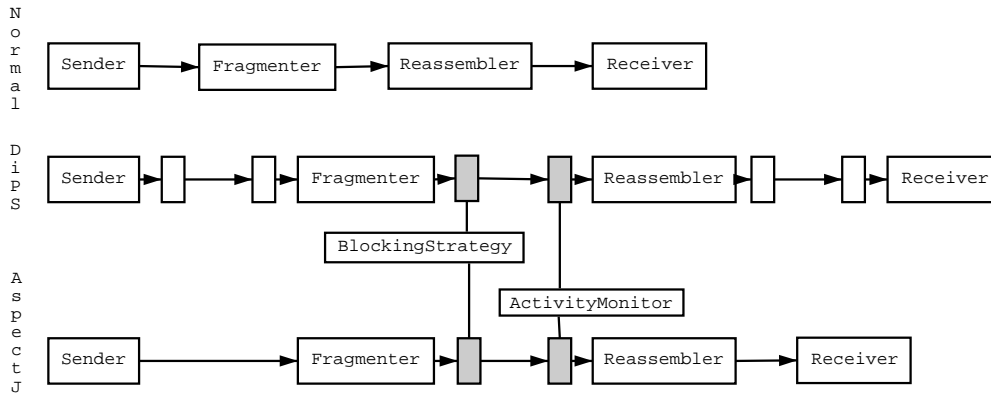


Figure 5: Performance Test Configuration

4 Performance Evaluation

Extending software modules with communication ports that are equipped with monitoring code (to check whether a component is active or not) comes at the cost of performance reduction. Additional indirections as well as the employed monitoring functionality impose a performance penalty on each interaction. In order to evaluate this performance impact, a number of tests have been carried out.

To measure the performance overhead caused by communication ports extended with monitoring code, three implementations of the fragmenter/reassembler composition we have tested (as described in figure 5). The first one only contains the basic functionality of both a fragmenter and reassembler, furthermore stripped from all communication ports and monitoring code. The second implementation uses DiPS [3, 9], a component framework for building customizable protocol stacks. A DiPS component can be seen as a wrapper, surrounding each functional module with first-class in- and outports. Finally, the last implementation uses AspectJ to weave monitoring functionality around existing code where needed. For all three implementations, the average delay between sending and receiving a packet has been measured. To compensate throughput variance due to code initialization or the Java garbage collector, 100000 packets are sent for each implementation. Furthermore, each packet was split up by the fragmentation service into exactly 10 fragments. The results of these tests are shown in table 1 and indicate that performance overhead caused by communication ports and monitoring code is limited.

	Average Delay (ms)	Performance overhead(%)
Normal Setup	0.10796	NA
DiPS	0.114453	6
AspectJ	0.112316	4

Table 1: Performance tests are executed on an Intel Pentium 4 CPU 1.70GHz with 512 MB RAM running Linux 2.4.19. All software is written in Java, compiled with JVM from SUN (1.4.1).

5 Conclusions and Future Work

In this position paper, we have described a mechanism to impose a safe state for unanticipated reconfiguration of producer/consumer based systems. The presented solution requires minimal contribution from the programmer and causes minimal interference to the rest of the system.

To achieve this goal, we have proposed to separate the functional behavior of software modules from additional support to block interactions. However, since there is no knowledge about the state of both the producer and the consumer at the moment interactions are blocked, reconfiguration may lead to inconsistencies. As a solution to this problem, we have suggested to (automatically) equip each component that

is eligible for reconfiguration with monitoring code to checks its component execution state: active, idle or finished. By using Aspect Oriented Programming (AOP), it should become possible to automatically extend components with such monitoring code. The only contribution required from the programmer of the involved component is the definition of a component invariant, which describes when a finished state is reached.

Future research involves the investigation of three important sub-tracks:

First of all, scheduling support is needed when several outports are holding up interactions (that are part of the same protocol transaction) directed to the same software module. An appropriate scheduling scheme will be essential for resuming blocked interactions to impose a safe state for reconfiguration.

Secondly, extending a software module with a single execution state machine will not be sufficient when more complex situations are envisaged. When several transactions are concurrently affecting the same software module in parallel (without affecting each other), separated execution state machines for each transaction are needed.

Finally, imposing a safe state for reconfiguration significantly affects the performance of the system. Holding up outgoing interactions causes a performance bottleneck when these interactions are resumed after the reconfiguration has been accomplished. To reduce this performance impact, concurrency management architectures like DMonA [8] will be further investigated.

References

- [1] K. M. Goudarzi and J. Kramer. Maintaining node consistency in the face of dynamic change. *Proc. of 3rd International Conference on Configurable Distributed Systems (CDS '96), Annapolis, Maryland, USA*, IEEE Computer Society Press:62, 1996.
- [2] C. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Department of Computer Science, University of Maryland, January 1994.
- [3] N. Janssens, S. Michiels, T. Mahieu, and P. Verbaeten. Towards Hot-Swappable System Software: The DiPS/CuPS Component Framework. In *Proceedings - The Seventh International Workshop on Component Oriented Programming*, 2002.
- [4] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 2001.
- [5] Gregor Kiczales et al. Aspect-Oriented Programming. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP97)*, volume 1241, pages 220–242. Springer-Verlag, 1997.
- [6] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.
- [7] Dylan McNamee, Jonathan Walpole, Calton Pu, Crispin Cowan, Charles Krasic, Ashvin Goel, Perry Wagle, Charles Consel, Gilles Muller, and Renauld Marlet. Specialization tools and techniques for systematic optimization of system software. *ACM Transactions on Computer Systems*, 19(2):217–251, 2001.
- [8] S. Michiels, L. Desmet, N. Janssens, T. Mahieu, and P. Verbaeten. Self-Adapting Concurrency: The DMonA Architecture. 2002.
- [9] S. Michiels, F. Matthijs, D. Walravens, and P. Verbaeten. DiPS: A Unifying Approach for Developing System Software. In A. D. Williams, editor, *Proceedings - The Eighth Workshop on Hot Topics in Operating Systems*, page 175. University of Karlsruhe, IEEE Computer Society, 2001.