Explicitly distributed AOP using AWED

Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno de Fraine, Davy Suvée

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Explicitly distributed AOP using AWED

Luis Daniel Benavides Navarro and Mario Südholt

and Wim Vanderperren and Bruno De Fraine and Davy Suvée

N° 5882

Avril 2006

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5882--FR+ENG

Explicitly distributed AOP using AWED

Luis Daniel Benavides Navarro and Mario Südholt
and Wim Vanderperren and Bruno De Fraine and Davy Suvée

Thème COM — Systèmes communicants
Projets OBASCO et SSEL (VUB)

Rapport de recherche n° 5882 — Avril 2006 — 33 pages

Abstract: Distribution-related concerns, such as data replication, often crosscut the business code of a distributed application. Currently such crosscutting concerns are frequently realized on top of distributed frameworks, such as EJBs, and initial AO support for the modularization of such crosscutting concerns, *e.g.*, JBoss AOP and Spring AOP, has been proposed.

Based on an investigation of the implementation of replicated caches using JBoss Cache, we motivate that crosscutting concerns of distributed applications benefit from an aspect language for explicit distributed programming. We propose AWED, a new aspect language with explicit distributed programming mechanisms, which provides three contributions. First, remote pointcut constructors which are more general than those of previous related approaches, in particular, supporting remote sequences. Second, a notion of distributed advice with support for asynchronous and synchronous execution. Third, a notion of distributed aspects including models for the deployment, instantiation and state sharing of aspects. We show several concrete examples how AWED can be used to modularly implement and extend replicated cache implementations. Finally, we present a prototype implementation of AWED, which we have realized by extending JAsCo, a system providing dynamic aspects for Java.

Key-words: Aspect-Oriented Programming, distributed programming, AWED, DJAsCo

This work has been supported by AOSD-Europe, the European Network of Excellence in AOSD (`www.aosd-europe.net`).

This work has been done in collaboration from members by OBASCO project team (4 rue Alfred Kastler, 44307 Nantes cedex 3, France) and SSEL group at Vrije Universiteit Brussel (Pleinlaan 2, 1050 Brussels, Belgium).

Programmation par aspects à distribution explicite

Résumé : Les préoccupations liée à la distribution, comme la réplication de données, sont souvent entrelacées avec le code métier d'une application distribuée. Actuellement ces préoccupations transversales sont fréquemment implémentées à l'aide de canevas distribués, tels que les EJB. Un support initial à l'aide de la programmation par aspects pour la modularisation de telles préoccupations transversales a été proposé, par ex., JBoss AOP et Spring AOP.

Basé sur une analyse de l'implémentation du cache répliqué JBoss Cache, nous motivons que ces préoccupations transversales d'applications distribuées profitent d'un langage d'aspects pour la programmation distribuée explicite. Nous proposons AWED, un nouveau langage d'aspects apportant des mécanismes de programmation distribuée explicites. Concrètement, ce langage réalise trois contributions. D'abord, les constructeurs de coupes distantes sont plus généraux que ceux déjà proposés et nous introduisons, en particulier, un notion de séquences distantes. Ensuite, nous proposons une notion d'actions distribuées qui peuvent être coordonnées d'une manière synchrone ou asynchrone avec l'application de base. Finalement, AWED réalise une notion d'aspects distribués comprénant des moyens pour le déploiement, l'instanciation et le partage de données entre aspects répartis. Par plusieurs exemples concrets, nous montrons comment AWED peut être utilisé modulairement pour réstructurer et étendre les implémentations de caches répliqués. Finalement, nous présentons une implémentation de prototype, que nous avons construite en étendant JAsCo, un système d'aspects dynamiques pour Java.

Mots-clés : Programmation par aspects, programmation distribuée, AWED, DJAsCo

Contents

1 Introduction

Distributed applications are inherently more complex to develop than sequential ones because of the additional requirements brought about by a partitioning of the software system across the network (*e.g.*, handling of communication and synchronization between system components, network failures, management of load balancing, ...). Previous research has shown that traditional programming languages do not allow to separate well distribution concerns from standard functional concerns [35]. Web caching [28, 10] and unit testing of distributed applications [24], for instance, have been shown to be subject to serious crosscutting problems. Techniques developed in the field of Aspect-Oriented Software Development (AOSD) [22] should be useful to separate distribution concerns. However, despite its increasing popularity for sequential applications, relatively few AO approaches address the development of distributed software.

By now, a number of distributed middleware solutions have been developed that offer features for aspect-oriented development, such as JBoss AOP [1] and Spring AOP [3]. However, these solutions in essence apply a non-distributed AO system to an existing framework for distribution, such as J2EE. More specifically, they can only modify the distribution behavior of a base program by introduction (or removal) of distribution-related code expressed in terms of the underlying framework. Such approaches are therefore inherently limited to the framework's capabilities. In particular, they do not provide general support for explicit distribution in the aspect language or weaver technology, for instance, to support automatic deployment, state sharing and remote execution.

Few AO approaches, most notably D [35], JAC [25] and DJcutter [24], have support for explicit distribution in aspects. However, these approaches provide rather limited support for aspects which are triggered on conditions of programs running on remote hosts. Mechanisms to quantify over sets of hosts are lacking, and advice execution cannot be distributed over remote hosts. Sequence aspects, for instance, have recently been proven useful for system-level implementation of web caches [16], in particular protocol modifications, but none of the AO languages for distributed aspects support remote sequences. A larger set of distributed abstraction should support, in particular, a larger set of distributed applications which may require diverse distributed implementation strategies.

To resolve these shortcomings, this paper introduces AWED, an aspect-oriented programming language with explicit support for distribution. AWED has three main characteristics. First, it offers remote pointcuts that can match events on remote hosts, including support for remote sequences. Second, it allows for distributed advice execution. Third, it provides a model of distributed aspects which addresses deployment, instantiation and data sharing issues. Furthermore, we present an implementation of AWED built on top of the dynamic AOP system JAsCo [30]. This implementation allows to dynamically apply distributed aspects and apply compiler optimizations such as just-in-time compilation and hot swapping. Finally, to motivate and illustrate our approach, we present a detailed analysis of crosscutting concerns in the context of the application server JBoss [2], using AWED, in particular, to address modularization issues of the JBoss Cache component.

This paper is structured as follows. Sec. 2 motivates crosscutting problems concerning replication and transaction concerns in JBoss cache. Sec. 3 presents the AWED aspect language. In Sec. 4 we show how AWED can be applied to different distribution problems, in particular distributed caching schemes. Sec. 5 introduces DJAsCo, our publicly-available prototype implementation of AWED on top of JAsCo. In Sec. 6 we evaluate our approach qualitatively and quantitatively. Sec. 7 discusses related work and Sec. 8 gives a conclusion and presents future work.

2 Motivation: cache replication and JBoss cache

As a motivating example we consider distribution problems arising in the context of the JBoss application server [2], which is built on J2EE, the Java-based middleware platform. Concretely, we consider replication in the JBoss Cache subsystems [8]. Replicated caches that provide a fast store close to client applications are a common solution to speed up distributed applications. The cache implementation of JBoss can be used to replicate data within a set of machines and thus ensures that all machines can access that data locally.

JBoss cache is a replicated transactional cache, *i.e.*, the coherence of hte data in the caches forming a common cluster is ensured by guarding updates using transactions. A JBoss cache can either be configured to be local, in which case no data is replicated to other caches on other machines, or it can be global, which means that all changes are replicated to all the other caches (on all other machines) that are part of the cluster. Transactions in JBoss Cache are implemented using pessimistic locking. Once a transaction is finished on the local machine, a two phase commit protocol is initiated to replicate transactions. If all the caches can acquire the necessary local locks and make the modifications, a commit message is sent to finalize the transaction, otherwise the transaction is rolled back on all nodes.

In the following we focus on two modularization problems which JBoss Cache is subject to. (Note that we do not investigate if JBoss, which we consider a legacy application, could have been restructured to avoid these problems in the first place.) First, as we will analyze in detail below, the replication concern in JBoss Cache is crosscutting: it is scattered over large parts of its implementation and tangled with other scattered concerns. Second, modification of the standard behavior of JBoss Cache is also hindered by this crosscutting. It is, for instance, not possible to replicate specific data only in subsets of the caches from the cluster, once a cluster has been initialized.

JBoss Cache comes equipped with two AO-related mechanisms which could potentially be useful to overcome the modularization issues mentioned: an interception mechanism (package `jboss.cache.interceptors`) and JBossCacheAOP. With regard to the first mechanism, we show below that these interceptors do not resolve the crosscutting issues mentioned above. The second mechanism, JBossCacheAOP, is an Aspect-Oriented extension of JBoss Cache implemented using JBoss AOP, which allows to use JBoss Cache together with standard Java objects ("POJOs") in a transparent manner. JBossCacheAOP enables developers to simply put an object in a cache, and automatically handles the mapping of the object to the

```
1  public Object put(Fqn fqn, Object key, Object value)
     throws CacheException {
3   GlobalTransaction tx=getCurrentTransaction();
    MethodCall m =
5     new MethodCall(putKeyValMethodLocal,
          new Object[]{tx, fqn, key, value, Boolean.TRUE});
7   return invokeMethod(m); }
```

Figure 1: Low-level transaction handling in class `TreeCache`

cache; the programmer can continue using POJOs as usual. This mechanism thus does not match our goals either: JBossCacheAOP is solely used to facilitate the use of JBoss Cache in an application but does not address modularization or extension of JBoss Cache's cache replication functionality.

Technically, the JBoss Cache implementation stores data in a tree data structure. The main tree data class is augmented with code to support a chain of interceptors, partially separating crosscutting concerns (*e.g.*, replication, transactions, eviction policies and automatic access to backend data stores) in classes called interceptors. Each method invocation to a `TreeCache` object is then processed by the elements of the interceptor chain. The `TreeCache` class, however, is still subject to crosscutting by the code for replication and transactions. For instance, each low-level cache manipulation method, such as `put` has to get the transactional context, modify it if necessary and pass it along explicitly as shown in Figure 1, where the object of type `MethodCall` is used for replication purposes.

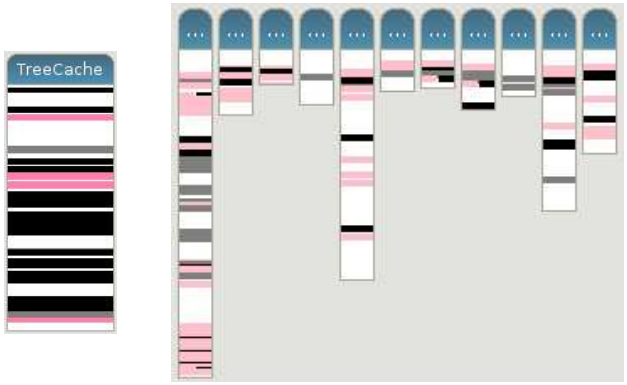

Figure 2: Crosscutting concerns in JBoss: class `TreeCache` (left), interceptor package (right)

Figure 2 illustrates these crosscutting concerns by showing the scattering of replication and transaction code in class `TreeCache` (left diagram in the figure) in the interceptor package (right). Replication code is colored black in both subfigures, transaction code is marked dark gray and calls into the `TreeCache` method from the interceptor package are colored light gray. The figures clearly exhibit crosscutting of replication code, its tangling with

transaction code and the interdependence between the `TreeCache` class and the interceptor package (the large number of calls from `TreeCache` into the interceptor package are not shown because they are frequently located near or part of the crosscutting code in the `TreeCache` class.)

More precisely, the `TreeCache` class includes 188 methods and consists of 1741 lines of code (LOC): the scattered code relevant for replication amounts to more than 196 LOC; the code for transactions accounts for more than 228 LOC. The situation for the interceptor framework is similar: it includes 9 classes consisting of 1263 LOC altogether; the (lower-bound) line counts for code relevant for replication, transactions and calls to `TreeCache` respectively are 30, 41, and 73. This provides strong evidence that the interceptor mechanism is not sufficient to solve the crosscutting problems.

Hence, understanding the replication and transactional behavior of the JBoss Cache implementation, which uses the interception mechanism, is far from trivial. Even simple modifications to the policies are difficult because of the crosscutting concerns. This applies, *e.g.*, if the replication policy is to be changed to one where replication is done only when a cache is interested in some specific data and only within the subgroup of hosts that are also interested in the same data instead of replicating always between all members of a cluster. Finally, note that AspectJ-like languages are not appropriate in this context: as shown by Nishizawa et al. [24] they are subject to limitations, in particular, requiring inadequately complex aspect definitions in the context of distributed crosscutting functionalities.

3 The AWED language

Modularization of crosscutting concerns for distributed applications using an aspect language, *i.e.*, in terms of pointcut, advice and aspect abstractions, suggests support for the following issues: (i) a notion of remote pointcuts allowing to capture relationships between execution events occurring on different hosts, (ii) a notion of groups of hosts which can be referred to in pointcuts and manipulated in advice, (iii) execution of advice on different hosts in an asynchronous or synchronous way and (iv) flexible deployment, instantiation, and state sharing models for distributed aspects.

AWED provides such support through three key concepts at the language level. First, *remote pointcuts*, which enable matching of join points on remote hosts and include remote calls and remote cflow constructs (*i.e.*, matching of nested calls over different machines). As an extension of previous approaches AWED supports remote regular sequences which smoothly integrate with JAsCo's stateful aspects [34] but also include features of other recent approaches for (non-distributed) regular sequence pointcuts [14, 15, 16, 4]. Second, support for *distributed advice*: advice can be executed in an asynchronous or synchronous fashion on remote hosts and pointcuts can predicate on where advice is executed. Third, *distributed aspects*, which enable aspects to be configured using different deployment and instantiation options. Furthermore, aspect state can be shared flexibly among aspect instances on the one hand, as well as among sequence instances which are part of an aspect on the other hand.

3.1 Syntax and semantics

AWED's syntax is shown in Fig. 3 using EBNF formalism (*i.e.*, square brackets express optionality; parentheses multiple occurrences, possibly none; terminal parentheses are enclosed in apostrophes).

3.1.1 Pointcuts

Pointcuts (which are generated by the non-terminal Pc) are basically built from **call** constructors (**execution** allows to denote the execution of the method body), field getters and setters, nested calls (**cflow**) and sequences of calls (non-terminal Seq).

AWED employs a model where, upon occurrence of a join point, pointcuts are evaluated on all hosts where the corresponding aspects are deployed. Pointcuts may then contain conditions about (groups of) hosts where join points originate (term **host**(Group)), *i.e.*, where calls or field accesses occur. Furthermore, pointcuts may be defined in terms of where advice is executed (term **on**(Group)). Advice execution predicates may further specify a class implementing a selection strategy (using the term **on**(Group, Select)) which may, *e.g.*, act as an additional filter or define an order in which the advice is executed on the different hosts. Groups are sets of hosts which may be constructed using the host specifications **localhost**, **jphost** and `adr:port`, which respectively denote the host where a pointcut matches, the host where the corresponding join point occurred and any specific host (Note that in general several applications may share the same port; we do not consider this case, which is rare in practice anyway). Alternatively, groups may be referred to by name. (Named groups are managed dynamically within advice by adding and removing the host which an aspect is located on, see Sec. 3.1.2 below.)

Finally, pointcut definitions may extract information about the executing object (**target**) and arguments (**args**), and may test for equality of expressions (**eq**), the satisfaction of general conditions (**if**), and whether the pointcut lexically belongs to a given type (**within**). Pointcuts may also be combined using common logical operators.

As a first example, the following simple pointcut could be part of a replicated cache aspect:

call(void initCache()) && **host**("adr1:port1")

Here, the pointcut matches calls to the cache's initCache method that originate from the host that has the specified address. The advice will be executed on any host where the aspect is deployed (possibly multiple ones) as there is no restriction on the advice execution host. The following example restricts the execution hosts to be different from the host where the joinpoint occurred:

pointcut putCache(Object key, Object o):
call(* Cache.put(Object,Object))
&& !**on**(**jphost**) && **args**(key, o)

Here, the pointcut matches calls to the cache's put operation on hosts other than the joinpoint host and binds the corresponding data items. (Note that in this case the clause !**host**(**localhost**) could replace !**on**(**jphost**) to achieve exactly the same effect of matching

```
// Pointcuts
Pc          ::= call(MSig)  | execution(MSig)  | get(FSig) | set(FSig)
              | cflow(Pc) | Seq  | host(Group) | on(Group[, Select])
              | target({Type}) | args({Arg})
              | eq(JExp, JExp)  | if(JExp)
              | within(Type)
              | Pc || Pc  | Pc && Pc  | !Pc
Seq         ::= [Id:] seq({Step})  | step(Id,Id)
Step        ::= [Id:] Pc [→Target ]
Target      ::= Id  | Id || Target
Group       ::= { Hosts }
Hosts       ::= localhost  | jphost  | "Ip:Port"  | GroupId
GroupId     ::= String
Select      ::= JClass

// Advice
Ad          ::= [syncex] Pos({Par}) : PcAppl '{' {Body} '}'
Pos         ::= before  | after  | around
PcAppl      ::= Id({Par})
Body        ::= JStmt  | proceed({Arg})  | addGroup(Group) | removeGroup(Group)

// Aspects
Asp         ::= [Depl] [Inst] [Shar] aspect Id '{' {Decl} '}'
Depl        ::= single  | all
Inst        ::= perthread  | perobject  | perclass  | perbinding
Shar        ::= local  | global  | inst  | group(Group)
Decl        ::= [Shar] JVarD  | PcDecl  | Ad
PcDecl      ::= pointcut Id({Par}) : Pc


// Standard rules (intensionally defined)
MSig, FSig  ::= // method, field signatures (AspectJ-style)
Type        ::= // type expressions
Arg,Par     ::= // argument, parameter expressions
Id          ::= // identifier
Ip,Port     ::= // integer expressions
JClass      ::= // Java class name
JExp        ::= // Java expressions
JStmt       ::= // Java statement
JVarD       ::= // Java variable declaration
```

Figure 3: AWED language

non-local joinpoints.) If the corresponding advice puts the item in the local cache, a condition on the aspect type (named, *e.g.*, `ReplCache`), such as **!within**(`ReplCache`), can be used to avoid triggering the pointcut during the advice execution.

Sequences Sequences (derived by the non-terminal Seq) are supported by two constructions on the pointcut level. First, the term [Id:] **seq**({Step}) allows to define sequences which may be named using an identifier and consist of a list of (potentially named) steps (non-terminal Step). A step may define the steps to be executed next (non-terminal Target).[1] A sequence matches if the sequence is completed, *i.e.*, if the current joinpoint matches the last step and previous joinpoints of the execution trace matched the previous steps in order. Second, the term **step**(seq,step) matches if the step named step of the sequence named seq matches. This allows advice to be triggered after a specific step within a sequence using a term of the form s: **seq**(... l: logout() ...) || **step**(s, l). Note that this last term matches the complete sequence besides the specified step; this can, if necessary, easily be ruled out. This sequence definition allows for a smooth integration of sequences and the finite-state aspects of non-distributed JAsCo.

To illustrate the use of sequence pointcuts, consider the following pointcut, which could be part of a simple cache replication protocol:

pointcut replPolicy(Cache c):
replS: **seq**(s1: initCache() && **target**(c) → s3 || s2 || s4,
s2: cachePut() → s3 || s2 || s4,
s3: stopCache() → s1
s4: cacheInconsistency())

(Here, identifiers like initCache denote undefined pointcuts specifying corresponding call pointcuts.) The pointcut above defines a sequence of four steps. An initialization step which may be followed either by a put operation (s2), termination of the cache (s3) or an error step (s4). A put operation (s2) may be repeated, followed by a cache termination (s3) or result in a cache inconsistency (s4). After cache termination, the cache may be initialized once again. Finally, a cache inconsistency terminates the sequence (and may be reacted upon by advising the pointcut).

Note that the above definition does not enforce that the steps are taking in the context of the same cache. This is, however, simple to achieve by binding the targets of the different steps **target**(c), and use **eq** or **if** pointcuts to ensure the appropriate relationships at different steps in a sequence.

A step may be referred to in pointcuts and advice as exemplified in the following example which shows how to provide a special pointcut for the second step in the previous sequence (and how to bind the variables used in that step) so that advice can later be attached to it:

pointcut putVal(Cache c, String key, Object o):
step(replS, s2) && **target**(c) && **args**(key, o)

[1] Note that while our sequences obviously encode finite-state automata, many applications of regular structures, in particular communication protocols [16], are effectively sequence-like, *i.e.*, of a one-dimensional directed structure, so that we decided to use the more intuitive terminology for AWED.

3.1.2 Advice

Advice (non-terminal Ad) is mostly defined as in AspectJ: it specifies the position (Pos) where it is applied relative to the matched join point, a pointcut (PcAppl) which triggers the advice, a body (Body) constructed from Java statements, and the special statement **proceed** (which enables the original call to be executed).

In an environment where advice may be executed on other hosts (which is possible in AWED using the **on** pointcut specifier), the question of synchronization of advice execution with respect to the base application and other aspects arises. AWED supports two different synchronization modes for remote advice execution: by default remote advice is executed asynchronously to the calling context. In this case synchronization, if necessary, has to be managed by hand. In contrast, **syncex** marks remote advice for synchronous execution. Local advice is always executed synchronously. The semantics of the **proceed** statement is "localized": the last around advice invokes the original behavior on the local host. The return value of the around advice is sent back to the original joinpoint host and processed in the regular around advice chain on that joinpoint host in case of a synchronous advice execution. Asynchronous advices are executed in parallel and there is thus no guarantee with respect to advice precedence. We have opted for this semantics because it provides for an intuitive yet efficient remote advice execution semantics.

Advice is also used to manage named groups of hosts: **addGroup** adds the current host to a given group, **removeGroup** allows to remove the current host from a group.

To give an example of basic advice functionality, the following advice definition is useful in the context of collaborating replicated caches:

```
around(String k, Object o): putCache(k, o) {
Object obj = getNewRemoteValue(k);
if (obj != null) { proceed(k, obj); }
else { proceed(k, o); } }
```

This advice first tests whether a new value is present remotely for a given key. If this is the case, the new value is stored in the cache.

3.1.3 Aspects

Aspects (non-terminal Asp) group a set of fields as well as pointcut and advice declarations. Aspects may be dynamically deployed (Depl) on all hosts (term **all**) or only the local one (term **single**).

Furthermore, aspects support four instantiation modes (Inst): similar to several other aspect languages, aspects may be instantiated per thread, per object, or per class. However, aspect instances may also be created for different sets of variable bindings arising from sequences (term **perbinding**) as introduced in [16, 4]. In this last case, a new instance is created for each distinct set of bindings of the variables in the sequence, *i.e.*, of the variables declared as arguments of a sequence pointcut or fields used in the sequence pointcut.

Finally, AWED allows distributed aspects of the same type to share local state (Shar): values of aspect fields may be shared among all aspects of the same type (term **global**), all

```
1 all aspect CacheReplication{
    pointcut cachePcut(Object key, Object o):
3    call(* Cache.put(Object,Object))
        && args(key,o) && !on(jphost) &&
5    !within(CacheReplication);

7   before(Object key, Object o): cachePcut(key,o){
     Cache.getInstance().put(key, o); }
9 }
```

Figure 4: Cache replication as an aspect

instances of an aspect which have been created using the instantiation mechanisms introduced before (term **inst**), all aspects belonging to the same group (term **group**(Group)) or all aspects on the one host (term **local**; note that these possibly belong to different execution environments, such as JVMs). Sharing modifiers can be given for an aspect as a whole or individual fields (Decl), if both are given, the latter have priority.

4 Applications

In this section, several applications of AWED are presented. We illustrate how replicated, cooperating distributed caches can be modularly implemented, and how distribution and execution clustering concerns can be introduced concisely into an existing non-distributed application.

4.1 Caching revisited

In Sec. 2 we have presented distributed caching and, in particular, its support through the JBoss Cache OO framework, as a motivating example for crosscutting in distributed applications. Fig. 4 shows how an aspect for cache replication can be implemented using AWED which accounts for all places where cache elements are requested and replicated to all other caches in a cluster, *i.e.*, an essential part of the functionality of replication within JBoss Cache's `TreeCache` class.

The aspect declaration in line 1 indicates that the aspect will be distributed globally and that a singleton instance (AWED's default instantiation mode) is created on each host. The pointcut defined in lines 2–5 matches calls putting elements in the cache; the term **!on**(**jphost**) limits advice execution to aspects which are not deployed on the host where the join point matched. The advice (lines 7–8) simply puts the element in the cache. As the pointcut assured that only aspects which are remote to the matching join point perform this advice, replication is thus achieved.

As a more intricate example, we consider an example of the large number of replication strategies for caches that use hierarchical, cooperative and adaptive caching strategies [9, 19]. Such strategies typically do not distribute data over whole clusters but replicate objects only to caches in the cluster that explicitly request them. Furthermore, cooperative behavior is

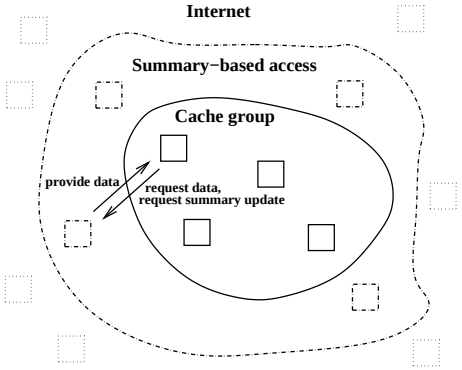

Figure 5: Adaptive cache behavior

useful, *e.g.*, looking for a copy in neighboring caches before (slowly) accessing farther caches or a centralized server holding the master copy of the data at hand. This kind of behavior is not part of the current JBoss Cache specification and would be very difficult to graft on its implementation without the use of AO techniques due to the crosscutting problems of its replication code.

In the following we present the heart of a summary-based cooperative cache strategy using AWED. Summary-based caching strategies (see, *e.g.*, [18]) use "summaries", *i.e.*, small digests of the cache contents of neighboring caches. The summaries can be used to test whether a cache contains a value with high probability. They can therefore be used to guide the decision which neighboring caches to contact and thus reduce network traffic.

Fig. 5 schematically illustrates a summary-based caching strategy used in the context of a cooperative replicated cache scheme. In the following, we present an aspect `CollaborativeCachePolicy` (see Fig. 6) realizing cache groups which replicate data among them as introduced in the previous example, but which also uses summaries to selectively get data from farther caches outside the cache group. These two sets of hosts are represented by groups cacheGroup and summaryHosts, respectively (line 3). Summary information is shared between hosts of the cache groups and the farther hosts at the border using AWED's group sharing feature. This provides for a concise integration of summaries and is appropriate because summary-based caching algorithms only infrequently update summaries. A simpler (and at times more inefficient) sharing mechanism than for the cached data itself can therefore be used for them.

Overall, the aspect consists of a three step remote sequence `replPolicy` (line 20), which first matches the cache initialization, followed by repeated cache accesses and cache replication operations. An explicit equality test ensures that the put operation in the third step concerns the same cache objects (identified through their keys) as looked up in the second step (assuming that the cache object does not change).

Concretely, at cache initialization time (see the pointcut at line 8) the two host groups are set up as well as initial summary information (advice at line 25). A cache access using

```
all aspect CollaborativeCachePolicy {

  group(cacheGroup, summaryHosts) SummaryT summaries;
  group(cacheGroup, summaryHosts) int cacheMisses = 0;
  final int THRESHOLD = 1000;
  ...

  pointcut initCache(Cache c):
    call(* Cache.init()) && host(localhost) && target(c);

  pointcut getCache(Cache c, String key):
    call(* Cache.get(String)) && host(cacheGroup)
      && target(c) && args(key);

  pointcut putCache(Cache c, String key, Object data):
    call(* Cache.put(String, Object)) && target(c)
      && args(key, data) && !on(jphost) && on(cacheGroup)
      && !within(CollaborativeCachePolicy);

  pointcut replPolicy(Cache c):
    replP: seq(s0: initCache(c) -> s1
               s1: getCache(c, k1) -> s2,
               s2: putCache(c, k2, val) && eq(k1, k2) -> s1);

  after(Cache c): step(replP, s0) && target(c) {
    if (c.isDomain(cache)) addGroup(cacheGroup);
    if (c.isDomain(border)) addGroup(summaryHosts);
    initializeSummaries(); }

  around(Cache c, String key): step(replP, s1) && args(c, key) {
    Object obj = c.get(key);
    if (obj == null) {
      obj = proceed();
      if(obj != null) {
        c.put(key, obj);
        cacheMisses++; } }
    return obj; }

  syncex around(Cache c, String key):
      step(replP, s1) && args(c, key)
        && on(summaryHosts, awed.combination.and(
                              awed.targets.filter(summaries),
                              awed.result.getFirst)) {
      if (cacheMisses > THRESHOLD)
         updateSummaries(summaries.getHosts())
      return c.get(key, o); }

  before(Cache c, String key, Object o):
      step(replP, s2) && args(c, key, o) {
    c.put(key, o); }
}
```

Figure 6: Aspect-based cooperative cache

```
1 pointcut distribution(Facade f):
    target(f) && call(* *(..)) &&
3     && !host("Serveripadr:port") && on("Serveripadr:port");

5 syncex Object around(Facade f): distribution(f) {
    return proceed(Facade.getInstance()); }
```

Figure 7: Distribution as an aspect

getCache (line 11) first looks up the value locally, and, if not found in the cache group attempts to acquire it from the outside. The advice at line 30 first accesses the local cache. If the data is not found, it calls **proceed** to trigger a synchronous remote advice (line 39) which (because of the **on** clause) queries hosts of group summaryHosts: this is achieved using a filter selecting hosts whose summaries indicate that the value should be present. The remotely executed advice body returns the query result and requests updates of the summaries if a threshold number of cache modifications has been exceeded (this accounts for the basic property of infrequent updates of summary information, see [18]). The replication within the cache group is achieved as above by matching put operations using the pointcut putCache (line 15). This pointcuts triggers the advice at line 48 which executes a put operation on all hosts in the cache group which are different from the host where the original put occurred.

4.2 Distribution and Clustering

In [29], Soares *et al.* illustrate how AOP techniques can be employed to explicitly introduce distribution within existing, non-distributed applications. For this, AspectJ is employed to automatically insert the required RMI code fragments. Their proposal requires two types of aspects: one aspect for handling server distribution concerns and one aspect for handling client distribution concerns. At the server side, a remote interface is generated for each object that should be distributable and the server-side aspect declares the remote objects to implement these generated interfaces. Additional methods are introduced by means of the server aspect to implement various technical details to support references to server objects (by default RMI sends a copy of the server object to the client). The client side aspect is responsible for capturing and redirecting the local method calls and declaring these methods to throw remote exceptions. In addition, each method specified in the remote interface requires a dedicated redirection advice, as AspectJ does not allow to change the target object in a *proceed* statement which is required to redirect the calls to the remote objects.

AWED allows for a more elegant solution, which does not require the overhead of introducing the required RMI-specific code. AWED allows to solve this distribution problem using a single aspect which is illustrated in figure 7. The `distribution` pointcut selects all calls to `Facade` methods on the client and makes sure that the accompanying advice is only executed at the server side. By employing negation of the `host` designator, calls on

```
pointcut clustering(Facade f):
  target(f) && call(* *(..)) && !host("Servergroup")
    && on("Servergroup", awed.hostselection.RoundRobin);

syncex Object around(Facade f): clustering(f) {
  return proceed(Facade.getInstance()); }
```

Figure 8: Clustering as an aspect

the server side will not match the pointcut themselves. The redirection behavior is encapsulated in a synchronous around advice. As the around advice gets executed on the server host, the `getInstance` method of the `Facade` class will retrieve an instance which is local to the server host (this could be generalized in order not to rely on a single object.) The `proceed` expression will invoke the original behavior on that Facade instance located on the server host. The AWED solution improves on the AspectJ-based solution: first, there is no need for RMI specific code to be injected in the server classes, which is a tedious process that is not always possible, and secondly, only one aspect with one pointcut and advice suffices while in the AspectJ solution at least two aspects and a pointcut-advice pairs for each method in the Facade class are necessary. Note that AWED shares this advantage with other middleware-based AOP approaches, such as DJCutter [24] and DAOP [26].

When multiple servers are available to handle remote requests, one can choose to cluster these servers together such that incoming requests can be dispersed, *e.g.*, to balance the server load. Again, AWED provides an elegant solution and allows this clustering behavior to be encapsulated in a single aspect. Figure 8 illustrates this clustering pointcut. All available servers are part of the `ServerGroup` and the `on` designator specifies that the accompanying advice should be executed only on a server that is part of that specific group. As only one specific host should be the target of the redirection, a *Round Robin* load balancing mechanism is employed, which assigns a server host on a rotating base.

5 Implementation

Two of the main features the AWED language requires from its underlying middleware implementation are the ability to intercept joinpoints from other hosts and the capability to execute advice on other hosts. A static aspect compiler, as for instance employed by AspectJ [21], is not well suited to facilitate a flexible distributed AOP platform, as this setup requires all aspects to be present on all the applicable hosts at compile or weave time. As such, all hosts need to be known and fixed in advance, which removes a lot of flexibility. A dynamic AOP approach however, allows to dynamically add/remove hosts and aspects, which is an important feature for large-scale distributed systems.

Therefore, we have chosen the JAsCo dynamic AOP framework as an implementation platform for the AWED language. JAsCo can be easily extended and provides highly efficient advice execution through its Hotswap and Jutta systems [33]. Furthermore, JAsCo already

natively supports a model of stateful aspects based on finite state machines [34], which can be extended to support distributed sequences as well. In the remainder of the paper we present DJAsCo, an extension of the base JAsCo system for explicit distribution. We first briefly introduce the JAsCo run-time architecture and its optimizations. Afterwards, the DJAsCo run-time architecture and our prototype implementation are discussed in detail. The DJAsCo extension has been made publicly available as a part of the regular JAsCo distribution [20].

5.1 JAsCo run-time infrastructure

The JAsCo run-time infrastructure is based on a central *connector registry* that manages the registered connectors[2] and aspects at run-time (see figure 9). This connector registry serves as the main addressing point for all JAsCo entities and contains a database of connectors and instantiated aspects. Whenever a connector is loaded into or removed from the system at run-time, the connector registry is notified and its database of registered connectors and aspects is automatically updated. The left-hand side of Figure 9 illustrates a JAsCo-enabled class from which the joinpoint shadows are equipped with traps. As a result, whenever a joinpoint is triggered, its execution is deferred to the connector registry, which looks up all connectors that are registered for that particular joinpoint. The connector on its turn dispatches to the applicable aspects.

The connector registry has an open plugin-based architecture that allows to easily extend the joinpoint interception, aspect lookup (pointcut evaluation) and advice execution parts, in particular for composition and optimization purposes.

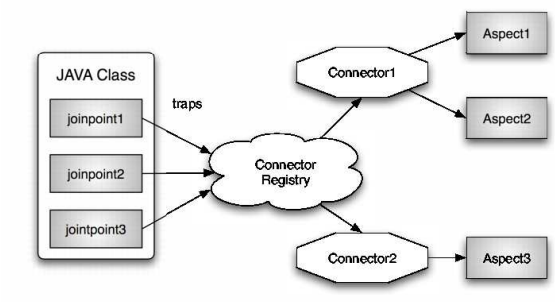

Figure 9: JAsCo run-time architecture

In addition to the connector registry, the run-time architecture consists of two other systems: HotSwap and Jutta. HotSwap allows to dynamically install traps only at those joinpoint shadows that are subject to aspect application. When a new aspect is deployed, the

[2]JAsCo introduces explicit *connectors* that instantiate and deploy aspects onto a concrete (component) context. In our AspectJ-based language, the aspect construct is responsible for both. An AspectJ-like aspect can easily map onto a connector-aspect pair in JAsCo.

applicable joinpoints shadows are hot-swapped at run-time with their trapped equivalents. Likewise, the original byte code is reinstalled when an aspect is removed and no other aspects are applicable on the joinpoint shadow at hand. Jutta on the other hand, is a just-in-time compiler for aspects that allows to generate a highly optimal code fragment for every joinpoint shadow. By caching these code fragments, an important performance gain is realized. The current version of the JAsCo run-time weaver, based on HotSwap and Jutta, is able to compete performance-wise with statically compiled aspect languages such as AspectJ, while still preserving dynamic AOP features [13]. A major concern of the DJAsCo design is to preserve compatibility of the weaver with these two tools, in particular to enable the optimization of remote pointcuts.

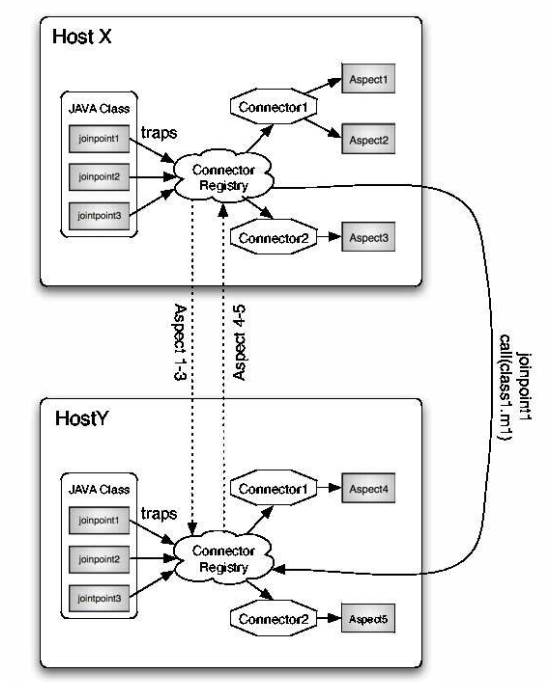

Figure 10: DJAsCo distributed run-time architecture. The aspects are distributed to all relevant hosts. Joinpoint1 occurs in Host X and is also sent to Host Y in order to trigger aspects on that remote joinpoint.

5.2 DJAsCo run-time architecture

The JAsCo run-time architecture can be distributed using two different strategies: either a single connector registry is kept for all hosts or each host separately maintains a dedicated

connector registry. The first solution has the advantage that a single registry is responsible for the aspect execution, whereas the second solution requires the distributed connector registries to be synchronized. In general, a central entity is considered to be a problematic solution in a distributed setting, as it inherently does not scale and can become a performance bottleneck. Therefore we choose to deploy a separate connector registry at each host (see figure 10).

Every connector registry is responsible for the locally intercepted joinpoints and its locally deployed aspects. In order to allow aspect execution on remote joinpoints, the intercepted joinpoints need to be sent to the other hosts. Likewise, in order to allow aspect execution on remote hosts, the aspects need to be distributed as well. The following sections explain these issues in more detail.

5.2.1 Remote pointcuts

In order to execute advice that trigger on remote joinpoints, the joinpoint information should be distributed to all interested hosts. To this end, a plugin for the connector registry has been implemented that: 1) intercepts all joinpoints, 2) prepares them for transmission and 3) sends them to the remote hosts. Joinpoints need to be prepared before transmission as not all joinpoint information might be transmittable. Our current system uses Java Serialization to transmit objects from one host to another. For this, the joinpoint is first stripped from all contextual information (e.g. callee, actual arguments) that is nor serializable nor primitive. Furthermore, the advice implementation cannot access advice variables nor reflectively query joinpoint information that is not serializable. Although this is an important limitation, it is typical in distributed environments. For instance, arguments of Java RMI method invocations need to be serializable or primitive as well.

In a last step, the joinpoint information is sent to the remote hosts. In order to locate and send this information to other interested hosts, the JGroups framework is employed [7]. JGroups is a well-known toolkit for reliable multicast communication. In addition, JGroups supports a wide range of network protocols, which makes our system independent of specific network technologies.

JAsCo's stateful aspects, *i.e.*, finite-state based sequences [34], have been extended to a distributed setting in order to implement AWED's distributed sequences. This is a rather straightforward process, because the state of a sequence pointcut is not managed by the JAsCo run-time infrastructure (deployed locally, see figure 10), but by the aspect itself. The aspect intercepts remote joinpoints, matching its stateful pointcut description in a similar way as for joinpoints matching regular pointcuts. Afterwards, the internal state is updated by firing the relevant transition(s) in the internal state machine.

5.2.2 Aspect Distribution

In order to execute advice on remote hosts, the aspects themselves should also be distributed to the host(s) in question. One solution would be to force an administrator to manually deploy the aspects on every applicable host. However, as an advice execution host sometimes

depend on complex expressions with several variables, it might be difficult to manually deploy the aspects onto remote hosts in an optimal fashion. For instance, deploying aspects to hosts where they can never be applicable is useless and wastes the system resources of those particular hosts. Hence, the DJAsCo extension automatically distributes the aspects to all remote hosts that might be applicable. When a new host joins, the DJAsCo run-time infrastructure detects this event, and the host automatically receives the possibly applicable aspects. Likewise, when new aspects are deployed at a particular host, they are automatically deployed at the relevant remote hosts. It is possible to avoid this automatic deployment of aspects on remote hosts by marking them with the `single` modifier.

Technically, JGroups is again employed to transfer the aspects and to be informed of changes in the network setup such as newly joined hosts. In contrast to joinpoints, aspects are class-based entities and it suffices to send the class byte-code to the remote hosts. Hence, possible serialization problems are avoided.

5.2.3 Synchronous Advice Execution

The default mode for advice execution is asynchronous with respect to advice executions on remote hosts. However, when an advice is marked with the `syncex` modifier, it needs to be executed synchronously, *i.e.*, the host where the joinpoint occurs, waits for this advice to be executed. As such, the specified aspect precedence on the joinpoint host is still guaranteed. The return value of the around advice is sent back to the original joinpoint host and processed in the regular around advice chain. A proceed to the original behavior is however still a local proceed (likewise to proceed in an asynchronous advice), which means that the joinpoint proceeds on the host where the advice is executed and not on the joinpoint host.

DJAsCo implements a synchronous advice execution by generating a proxy aspect at the joinpoint host. This proxy aspect is automatically generated and dynamically weaved when an aspect, defining a synchronous advice execution, is deployed. The proxy aspect's advice delegates to the appropriate execution host where the original advice is executed. The JGroups framework is again employed for the synchronous communication between the involved hosts.

5.2.4 State Sharing

The AWED language supports state sharing between different instances of the same aspect type regardless of the location and/or VM where they are executed. In order to implement `local` sharing, DJAsCo generates one master field on every host for each locally shared aspect field. All aspect instances of the aspect type on that host automatically refer to that field using Java RMI. Field queries and updates are automatically redirected to the shared field. This redirection takes place by employing another AWED aspect that is dynamically generated and weaved when an aspect, defining a shared field, is being deployed. Visibility modifiers for the fields (such as `private`) do not hinder the sharing implementation because they can be overridden at run-time.

```
all aspect StateSharing {
  pointcut stateChanged(Object value):
   set(myaspectname.myfieldname) && args(value) && !on(jphost);

  after(Object value): stateChanged(value) {
      myaspectname aspectinstance = myaspectname.apectOf();
      aspectinstance.myfieldname=value; }
}
```

Figure 11: State sharing as a AWED aspect

The `global` state sharing could be implemented in a similar fashion, *i.e.*, having one globally shared field. However, this solution suffers from a serious robustness problem as all aspect instances of the same type would rely on one specific host that holds the shared field. Therefore, the local master fields are explicitly synchronized using yet another AWED aspect that is automatically generated at deployment time. Figure 11 illustrates a simplified version [3] of this global state sharing aspect. The after advice is triggered for every state change of the `myfieldname` field of the `myaspectname` aspect. The advice is executed on every host except on the one that triggered the joinpoint. As such, the state change is propagated to all other hosts. The advice implementation first fetches an aspect instance of the given type on the host where it is executing and than changes the value to the newly assigned value. Because all aspect instances on that host refer to the same field, the new value is immediately propagated to all aspect instances of the `myaspectname` type on the host at hand.

5.2.5 Optimization of remote pointcuts

The JAsCo HotSwap and Jutta systems are compatible with the DJAsCo architecture. Because all aspects are present at every applicable host (even aspects that might execute their advice elsewhere), the local HotSwap system still knows where to insert traps. Aspects that do not define pointcuts relevant for the local host are not deployed and are of no interest to the HotSwap system as they do not induce newly trapped joinpoints. Remote joinpoints are represented similarly to local joinpoints. Hence, the Jutta system is still able to generate and cache a code fragment for executing the joinpoint locally. As such, apart from the network delay and serialization/deserialization cost, no additional overhead is required for remote pointcuts and distributed advice executions. Note that hotswapping and code caching properties of our system are therefore inherited from JAsCo.

[3] Notice that this aspect has been simplified for presentation purposes. For example, it does not cope with the fact that aspect types might not be present on every host.

6 Evaluation

In this section we present results on a qualitative and quantitative evaluation of our approach. First, we show how aspects with explicit distribution can be used for refactoring and extension of the standard replication strategy of JBoss. Second, we present initial performance evaluations of our approach.

6.1 Refactoring and extending JBoss replication

In order to qualitatively evaluate our approach we present two experiments, which show how AWED can be used to improve JBoss standard replication strategy. The first experiment illustrates how existing OO code can be refactored to achieve better modularization of distributed concerns, while the second presents an extension of JBoss cache replication by a selective cache policy.

6.1.1 JBoss replication code

We first present an analysis of features of the standard JBoss cache transaction-guarded replication mechanism that are essential to our experiments. Figure 12 shows three methods from three different classes that respectively participate in the coordination and replication of the prepare phase, of the two phase commit protocol used, and in the host that is starting the replication of the transaction. First, the `commit` method is called in the class implementing the `Transaction` interface (i.e. `DummyTransaction`) this method uses the before-completion and after-completion idiom: if the `beforeCompletion` method of any of the listeners has failed, the `afterCompletion` method used to roll back the transaction. When a before-completion method is called all the listeners are notified, in particular the listener `SynchronizationHandler`. When `SynchronizationHandler`'s `beforeCompletion` method is called the `runPreparePhase` method of the class `ReplicationInterceptor` is invoked. Using reflection and the `Jgroups` API, the remote calls are distributed to all the participating hosts, waiting for an answer in the case of synchronous distribution. Note that this behavior corresponds only to the code executed in the host from which the transaction originates. Once a remote host receives a message to replicate, the `TreeCache` class invokes the `replicate` method that belongs to the class `ReplicationInterceptor`. In that class, the transaction is decomposed, each call is then replicated using a local call inside a local transaction, using the normal interceptors chain, see Sec. 2). This code excerpt shows that replication code gets scattered in multiple classes and tangled with other functionalities as well as infrastructure support code (*e.g.*, code implementing interceptors and listeners).

6.1.2 Refactored solution using AWED

Figure 13 shows a re-implementation of the basic mechanism of replication proposed by JBossCache. The pointcut definition matches a call to the method `_put` in the class

```
//Class DummyTransaction
public void commit() throws ... {
    ...
    try {
        boolean outcome=notifyBeforeCompletion();
        ...
        notifyAfterCompletion(doCommit? Status.STATUS_COMMITTED :
                              Status.STATUS_MARKED_ROLLBACK);
        ....
    }
    finally {
        // Disassociate tx from thread.
        tm_.setTransaction(null);
    }
}

//Class SynchronizationHandler
public void beforeCompletion() {
    TransactionEntry entry=tx_table.get(gtx);
    ...
    // REPL_SYNC only from now on
    try {
        int status=tx.getStatus();
        switch(status) {
        ...
        case Status.STATUS_PREPARING:
        try {
            MethodCall prepare_method;
            prepare_method=new MethodCall(TreeCache.prepareMethod,
               new Object[]{gtx, modifications, (Address)cache.getLocalAddress(), Boolean.FALSE});
            runPreparePhase(gtx, prepare_method, (Address)cache.getLocalAddress(), modifications, false);
         }catch(Throwable t) {
             log.warn("runPreparePhase() failed. Transaction is marked as rolled back", t);
             tx.setRollbackOnly();
             throw t;
        }
        break;}
    }catch(Throwable t) {
        throw new NestedRuntimeException("", t);
    }
}

//Class ReplicationInterceptor
protected void runPreparePhase(GlobalTransaction tx, MethodCall prepare_method, Address coordinator,
                               List modifications, boolean async) throws Exception {
    List rsps;
    int num_mods=modifications != null? modifications.size() : 0;
    // this method will return immediately if we're the only member (because exclude_self=true)
    if(log.isTraceEnabled()) log.trace(...);
    rsps = cache.callRemoteMethods(cache.getMembers(), TreeCache.replicateMethod,
                                   new Object[]{prepare_method},
                                   !async, // sync or async call ?
                                   true, // exclude self
                                   cache.getSyncReplTimeout());
    if(!async && rsps != null) checkResponses(rsps);
       // throws an exception if one of the rsps is an exception
}
```

Figure 12: JBoss cache transaction replication implementation

```
all aspect TreeCacheTransactionReplication {
    pointcut cachePut(GlobalTransaction gtx, String fqn, Object key, Object value):
     call(* org.jboss.cache.TreeCache._put(GlobalTransaction , Fqn , Object , Object, boolean))
         && args(gtx, fqn, key, value, boolean) && !on(jphost)
         && !within(TreeCacheTransactionReplication);

before(GlobalTransaction gtx, String fqn, Object key, Object value):
        cachePut(gtx, fqn, key, value){
    testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
    if(gtx==null){
        System.out.println("Replicating without tx..");
        cp.getTc().put((Fqn.fromString(fqn), key, value);
    }
}
```

Figure 13: Refactoring the JBoss replication code (principle)

`TreeCache` on all the hosts that are not the joinpoint-host. Once the joinpoint is matched, a replicating advice is executed only if the matched joinpoint was not inside a transaction.

Figure 14 shows a re-implementation of the transaction-guarded replication strategy of JBoss Cache. AWED allows to cleanly modularize the transaction-guarded replication protocol in a single aspect. The first advice is executed when a local `commit` method is invoked, the code is used to extract information from the context and invoke an auxiliary method used to trigger the replication protocol. The aspect then defines two pointcuts that represent a `RemoteCommit` and a `RemoteRollBack` respectively. With those pointcuts definitions in place, three synchronous `around` advices are defined to implement the two phase commit protocol. The first `around` advice is used to rollback the transaction in the joinpoint host if an error occurs during the prepare phase. The second `around` advice executes replication handling at other nodes through a remote call to `preparePhase()` and raises an exception in case of errors. The third `around` advice commits the transaction. Note that two different selection classes are used during the protocol execution. The class `dhamaca.hostselection.AllSuccessfull` is used to execute the synchronous advice in all hosts and returns an exception if any of the hosts reports an exception. The class `dhamaca.hostselection.All` is used to execute the advice in all hosts. Finally an advice that matches any remote rollback is put in place to assure local rollback of transactions when the prepare phase fails in any of the remote hosts.

6.1.3 Extension of the JBoss replication strategy

As a second evaluation, we realized an extension to the JBoss standard replication strategy: data should only be replicated to nodes that have explicitly requested it, *i.e.*, new objects inserted in a cache group are not replicated spontaneously.

Figure 15 shows an aspect definition using AWED that enables lazy replication in a Jboss-Cache. The aspect defines two piontcuts `startSelectiveMode` and `finishSelectiveMode`, respectively used to define the start and end events of the selctive replication mode. The implementation uses a sequence pointcut `replS` which implements a distributed protocol:

```
all aspect LocalTransacCommit {
  pointcut localCommit():
      call(* org.jboss.cache.transaction.DummyTransaction.commit()) && on(jphost)
      && !within(org.emn.djasco.cache.TransacReplication);

  before(): localCommit() {
      testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
      GlobalTransaction gtx=cp.getTc().getCurrentTransaction();
      TransactionEntry entry= cp.getTc().getTransactionTable().get(gtx);
      List modifications= new LinkedList(entry.getModifications());
      ReplicationHelper.getInstance().txReplication(gtx, modifications);
  }

  pointcut remoteCommit(GlobalTransaction gtx, List modifications):
      call(* org.emn.djasco.cache.ReplicationHelper. txReplication(GlobalTransaction, List))
      && args(gtx, modifications) && !within(org.emn.djasco.cache.TransacReplication);

  pointcut remoteRollBack(GlobalTransaction gtx, List modifications):
      call(* org.emn.djasco.cache.ReplicationHelper. rollBack(gtx, modifications))
      && args(gtx, modifications) && !within(org.emn.djasco.cache.TransacReplication);

  //Local advice roolbaks if something fails
  //in the two phase commit protocol
   syncex around(GlobalTransaction gtx, List modifications):
      remoteCommit(gtx, modifications) && on(jphost){
      testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
      try{
          proceed();
      }catch(Exception e){
          ReplicationHelper.getInstance().rollBack(gtx, modifications);
          throw e;
      }
  }
  //Execute the prepare phase in all the hosts
  syncex around(GlobalTransaction gtx, List modifications):
      remoteCommit(gtx, modifications) && !on(jphost, dhamaca.hostselection.AllSucessfull){
     testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
     try{
         ReplicationHelper.getInstance().preparePhase(gtx, modifications);
     }catch(Exception e){
         ReplicationHelper.getInstance().rollBack(gtx, modifications);
     throw e;
     }
     proceed();
  }
  //commits definetly in all the hosts
  syncex around(GlobalTransaction gtx, List modifications):
      remoteCommit(gtx, modifications) && !on(jphost, dhamaca.hostselection.All){
     testDjascoCachePolicyExtension cp = testDjascoCachePolicyExtension.getInstance();
     try{
         ReplicationHelper.getInstance().commitPhase(gtx, modifications);
     }catch(Exception e){
        //don't do nothing if fails commiting locally
     }
     proceed();
  }

  before(GlobalTransaction gtx, List modifications):
    remoteRollBack(gtx, modifications) && !on(jphost){
        //roll back the transaction
    }
}
```

Figure 14: Refactoring the JBoss replication code (detailed excerpt)

```
all aspect lazyReplication {
    pointcut cacheGet(String fqn):
     call(* org.jboss.cache.TreeCache.get(String))
     && args(fqn) && on(jphost) && !cflow(* lasyReplication.*(*));

    pointcut cacheReplicate(MethodCall method_call):
     call(* org.jboss.cache.interceptors.ReplicationInterceptor. replicate(MethodCall))
     && args(method_call) && on(jphost);

    //this pointcut can be matched by any execution on
    // any host, is not host restricted
    pointcut startLasyModeEvent():
     call(* AWED.utils.LazyMode.start());

    //finish lazy mode event
    pointcut finishLasyModeEvent():
     call(* AWED.utils.LazyMode.end());

    pointcut replPolicy(String fqn, MethodCall method_call):
       replS: seq(s1:startLasyModeEvent() -> s4 || s3 || s2 ,
                  s2: cacheGet(fqn) -> s4 || s3 || s2,
                  s3: cacheReplicate(method_call) -> s4 || s3 || s2,
                  s4: finishLasyModeEvent() -> s1)

    around(String fqn):
       step(replS, s2){
          IamInterestedIn(fqn);
          return proceed();
       }

    around(MethodCall method_call): step(replS, s3){
        Method meth=method_call.getMethod();
        if(meth.equals(TreeCache.prepareMethod) ||
            meth.equals(TreeCache.commitMethod) || meth.equals(TreeCache.rollbackMethod)) {
            return proceed();
        }
        else if(amIInterested(method_call)){
        return proceed();
    }
}
```

Figure 15: Extending the JBoss replication strategy

this protocol starts selective replication mode, followed by local interception of get operations (which explicitly indicate interest in some data). Then the aspect intercepts local replication operations, replicating only when some interest to the data has been registered, until selective mode is terminated. Data which is to be replicated is selected through the advice applied before get operations (step 2 in the sequence `replS`) which registers interest to the data.

6.1.4 Comparison to a JBoss-only solution

To conclude this evaluation, let us compare the two previous examples to an implementation based only on JBoss. JBoss Cache implements transaction replication using three main classes `TreeCache`, `ReplicationInterceptor` and `SynchronizationHandler`. Reflection is used to (un)marshal messages sent between nodes in the cluster. This architecture is augmented with a before-completion and after-completion idiom that is present in different classes. Methods include code to hold a special state, which is used to coordinate protocol stages. Tracing the default replication behavior and integration of the proposed algorithm is far from trivial due to the crosscutting these functionalities are subject to.

Contrary to the tangled implementation of transaction and replication code in JBoss Cache, our aspect refactoring clearly separates replication and transactions (transactions appear, apart from their setup, only in exception handlers). Concretely, we have been able to refactor the scattered replication functionality and the corresponding transaction handling which amounts to around 500 LOC in JBoss into one aspect of around 100 LOC. Furthermore, our aspect does not require any particular transaction management but reuses the default transaction management of JBoss Cache. Finally, the second example shows how extensions can be easily integrated using AWED, in particular, because distributed protocols can concisely be expressed using sequence aspects.

6.2 Performance Evaluation

Finally, we present the results of a small set of experiments conducted[4] to evaluate the run-time performance of the DJAsCo implementation. The main objective of this micro-benchmark is to evaluate the run-time overhead of triggering a joinpoint on a local host and executing a corresponding around advice on a remote host. Unfortunately, little comparison is possible with existing AOP approaches that also incorporate distribution facilities. Although JAC, CaesarJ [23] and PROSE [27] provide the ability to remotely deploy aspects, interaction between local/remote joinpoints/advices remains impossible. DJCutter on the contrary, provides a more expressive distribution model which includes some of the features of Dhamaca. However, at the time of writing, DJCutter is being refactored towards an extension of the abc platform [6] and no implementation is publicly available. The set-up of the proposed micro-benchmark is quite straightforward: it features one method that adds two integer input values and returns their sum. On this method, a single around advice

[4]Pentium II 348 MHz, 128MB RAM, Debian Linux 3.1 (Sarge), Sun Java 1.5-b05

	Base method	AspectJ Around	DJAsCo Around
Non-distributed	0,0026 ms	0,0037 ms	0,0129 ms
	RMI method	**DJAsCo Sync Ex.**	**DJAsCo Async Ex.**
Distributed	23,1912 ms	78,7917 ms	16,7374 ms

Table 1: Overview of benchmark results

is deployed that, instead of returning the sum, returns the multiplication of the two integer input values. For each conducted experiment, the result represents the execution time (expressed in milliseconds) of the base *sum* method on which the *multiplication* aspect has been deployed. Table 1 presents the results of the conducted experiments.

In order to have a reference value to compare to, the *multiplication* aspect is first deployed in a non-distributed manner. For this, we employ both AspectJ and DJAsCo. AspectJ induces only a small run-time overhead compared to the base method execution time. As DJAsCo is a dynamic AOP approach, it already generates a certain performance overhead as the weaving of the aspects takes place at run-time. However, the fact that DJAsCo is three times slower then AspectJ, can mainly be attributed to a non-optimized implementation of the around advice. When before/after advices are employed, the run-time performance of DJAsCo comes very close to the one of static AOP approaches such as AspectJ [13].

In a second set of experiments, the *multiplication* aspect is deployed in a distributed manner. Both synchronous and asynchronous DJAsCo advice executions have been considered. The execution time of the asynchronous DJAsCo advice execution mainly consists in the time required by the JGroups framework to broadcast the triggered joinpoint to all involved remote hosts. The execution time of the synchronous DJAsCo advice execution illustrates the time required to trigger the execution joinpoint, transfer it to a remote host, execute the around advice (of the *multiplication* aspect) and return this multiplication result to the host that triggered the joinpoint. In fact, a synchronous DJAsCo advice execution comes close to calling an RMI method on a remote host. Therefore, we simulate the synchronous advice execution by employing a Java RMI client-server setup. The results illustrate that DJAsCo is about three times slower compared to using Java RMI. This overhead can mainly be attributed to the use of the JGroups framework, which is, although being very reliable, rather slow. In case of the DJAsCo asynchronous advice execution for instance, the overhead of JGroups mounts up to 16,7298 ms which is 99% of the total execution time. In the future, we plan to investigate other communication frameworks that help us at optimizing the run-time performance for distributing joinpoints and aspects.

7 Related Work

D [35] has been the first aspect language with means for explicit distribution. This approach includes, in particular, COOL, a synchronization sublanguage, and RIDL, a sublanguage for the definition of remote interfaces. The former essentially supports the declarative definition

of mutual exclusion relations between objects. The latter allows to specify the semantics of remote method invocation between different execution spaces, in particular the argument passing semantics. However, D does not provide equivalents for most features of AWED, in particular, remote sequence aspects, distributed advice and distributed aspects.

JAC [25] is a dynamic AOP-framework, which, in contrast with other AOP approaches, does not introduce a dedicated aspect language, but describes aspects in terms of regular OO-abstractions. Their proposed framework is extended with the notion of a distributed pointcut definition. This pointcut definition extends a regular pointcut with the ability to specify a named host that delimits the context in which the joinpoint should be detected. In contrast with AWED, JAC does not provide support to delimit the context in which distributed advices should be executed, nor does it support remote sequences and instantiable distributed aspects. In order to manage the distributed deployment of aspects, JAC replicates its Aspect-Component manager, which is similar to the DJAsCo connector registry, on the involved remote hosts. A consistency protocol makes sure that whenever aspects are weaved at one specific host, the same aspects are also weaved at the other involved hosts.

DJcutter [24] is an extension of the AspectJ language, which extends a subset of AspectJ's constructs to make them behave as remote pointcuts. Similar to JAC and AWED, DJcutter introduces an explicit *host* pointcut designator that allows to delimit the context in which joinpoints should be detected. In addition, DJcutter supports distributed *cflow* pointcuts if the base application is implemented using a custom socket implementation. Similar to AWED, the state information of a joinpoint is exposed to an advice by using the *args* and *target* designators, from which by default, the values are remotely transferred by copy. At the run-time level, DJcutter proposes a centralized aspect-server. This server gathers joinpoint information of remote pointcut definitions and executes the related advices local to the aspect-server. The advice server constitutes a bottleneck in a large distributed systems and makes the implementation of, *e.g.*, cache replication much more complex compared to AWED, because the aspect implementer is responsible to manage and execute the appropriate methods on the distributed hosts.

There are some approaches which have essentially developed for the sequential case but provide some mechanisms for AOP in distributed environments. CaesarJ [23] and PROSE [27] are part of this class of approaches. They mainly provide a sequential aspect model based on collaborations between aspects and classes but allow aspects to be created on remote hosts. Concretely, CaesarJ allows the host where an aspect is to be executed to be explicitly specified when the aspect is created. In contrast to our approach no further means to explicitly refer to remote aspect interactions are provided: they do not provide mechanisms for sharing of aspect state, management of distributed aspect instances as well as distributed advice execution.

Another important category of approaches applying AOP to distributed applications, *e.g.*, JBoss AOP [1], Spring AOP [3] and several research approaches, such as that proposed by Duclos *et al.* [17], essentially allow non-distributed aspects to manipulate applications implemented using an existing framework for distribution. While they integrate some features which can be found in our system (such as dynamic aspect application), they do not pro-

vide features for explicit specification of distribution issues in aspects. In contrast to our approach, replicating data in remote caches can thus only be done, for instance, by an aspect which repeatedly calls existing methods for put operations on remote caches. Similarly, Cohen and Gil [11] essentially preserve a non-sequential AOP framework. They however propose an extension to directly express aspects involving J2EE-based crosscutting concerns between servers and clients through so-called tier-cutting concerns using a `remotecall` construct. They do not support any of our more advanced concepts, such as remote sequences, distributed advice and distributed aspects. Taking an application-oriented view, Colyer and Clement [12] investigate the benefits of using AspectJ, also a sequential AOP approach, to modify interfaces of Websphere, IBM's J2EE-based application server. All of these approaches (as well as several AO frameworks for distributed middleware, such as DAOP [26] and Lasagne [32]) do not propose features for explicit distribution in the aspect language, which are useful, as we have shown in this paper, also for EJBs and similar frameworks.

Inspired by our approach, Tanter and Toledo have recently proposed a kernel for distributed AOP, ReflexD [31]. The kernel consists of a general framework for the implementation of distributed AOP languages. It incorporates remote pointcuts but does not provide explicit support for remote sequences. It also includes distributed advice but does not allow asynchronous execution of advice. A rich model for distributed deployment is introduced, but requires a more complex infrastructure than our DJAsCo based implementation. In contrast to AWED ReflexD supports different modes of distributed parameter passing.

Finally, there are some AOP approaches which investigate concurrency issues and thus are also directly relevant to some of the distribution issues we have considered. To cite one example, Andrews [5] considers aspect weaving in a process calculus relying on asynchronous communication, essentially by aggregating fine-grained concurrent executions into coarser-grained ones. While this technique should be in principle applicable to distributed aspect languages, his work does not provide explicit means to support distribution-specific concerns, such as data replication.

8 Conclusion and Future Work

In this paper we have provided evidence that current systems for AOP with explicit distribution are limited. We presented AWED, a language including distributed sequence pointcuts, remote advice execution, and distributed advice. We also introduced DJAsCo, an implementation of AWED based on the JAsCo AOP system. Finally, we provided evidence that our concepts of explicit distribution in aspects can be usefully applied to cache replication problems.

At the language level, AWED constitutes a first proposal for a comprehensive aspect language for explicit distribution. However, a number of issues remain to be explored, in particular, integrated means for synchronization of asynchronous advice and finer-grained control of aspect deployment, instantiation and data sharing. The AWED implementation as a JAsCo extension should be improved by performance optimizations specific for the distributed context.

Acknowledgements. We thank Jean-Marc Menaud for our fruitful discussions on replicated caches and the anonymous reviewers for their valuable remarks.

References

[1] *JBoss AOP*. http://jboss.com/products/aop.

[2] *JBoss home page*. http://jboss.com.

[3] *Spring AOP*. http://www.springframework.org/.

[4] C. Allan et al. Adding trace matching with free variables to AspectJ. In R. P. Gabriel, editor, *Proc. of OOPSLA'05*. ACM Press, Oct. 2005.

[5] J. H. Andrews. Process-algebraic foundations of aspect-oriented programming. In *Proc. of the 3rd International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 187–209, 2001.

[6] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhotak, O. Lhotak, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In P. Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 87–98. ACM Press, Mar. 2005.

[7] B. Ban. JGroups, a toolkit for reliable multicast communication. http://www.jgroups.org/, 2002.

[8] B. Ban and B. Wang. *JBossCache Reference Manual V. 1.2*. JBoss Inc., 2005.

[9] G. Barish and K. Obraczka. World wide web caching: Trends and techniques. *IEEE Communications Magazine*, May 2000.

[10] S. Bouchenak, A. Cox, S. Dropsho, S. Mittal, and W. Zwaenepoel. AOP-based caching of dynamic web content: Experience with J2EE applications. Technical Report RR-5483, INRIA, 2005.

[11] T. Cohen and J. Gil. AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In *Proc. ECOOP '04*, volume 3086 of *LNCS*. Springer-Verlag, 2004.

[12] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proc. of AOSD'04*. ACM Press, 2004.

[13] B. De Fraine, W. Vanderperren, D. Suvée, and J. Brichau. Jumping aspects revisited. In R. E. Filman, M. Haupt, and R. Hirschfeld, editors, *Dynamic Aspects Workshop*, pages 77–86, Mar. 2005.

[14] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proceedings of GPCE'02*, volume 2487 of *LNCS*, pages 173–188. Springer-Verlag, Oct. 2002.

[15] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. AOSD'04*. ACM Press, Mar. 2004.

[16] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Ségura-Devillechaise, and M. Südholt. An expressive aspect language for system applications with arachne. In *Proc. AOSD'05*. ACM Press, Mar. 2005.

[17] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In *Proc. of AOSD'02*, pages 65 – 75. ACM Press, 2002.

[18] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[19] ICP. *Internet Cache Protocol*. http://icp.ircache.net/.

[20] JAsCo. *JAsCo website*. http://ssel.vub.ac.be/jasco/.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[22] G. Kiczales, J. Lamping, A. Mendhekar, et al. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. ECOOP 1997*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[23] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *Proc. ESEC/FSE'04*, pages 127–136, 2004.

[24] M. Nishizawa, S. Shiba, and M. Tatsubori. Remote pointcut - a language construct for distributed AOP. In *Proc. of AOSD'04*. ACM Press, 2004.

[25] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in Java. In *Proceedings of Reflection'01*, volume 2192 of *LNCS*. Springer-Verlag, Sept. 2001.

[26] M. Pinto, L. Fuentes, M. Fayad, and J. Troya. Separation of coordination in a dynamic aspect oriented framework. In *Proc. of AOSD'02*. ACM Press, 2002. short paper.

[27] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In G. Kiczales, editor, *Proc. AOSD 2002*, pages 141–147. ACM Press, Apr. 2002.

[28] M. Ségura-Devillechaise, J.-M. Menaud, G. Muller, and J. L. Lawall. Web cache prefetching as an aspect. In *Proc. of AOSD'03*. ACM Press, 2003.

[29] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of OOPSLA'02*, pages 174–190. ACM Press, 2002.

[30] D. Suvée and W. Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In M. Akşit, editor, *Proc. AOSD 2003*, pages 21–29. ACM Press, Mar. 2003.

[31] E. Tanter and R. Toledo. A versatile kernel for distributed AOP. In *Proceedings of the 6th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, LNCS. Springer-Verlag, June 2006.

[32] E. Truyen et al. Dynamic and selective combination of extensions in component-based applications. In *Proc. ICSE 2003*, May 2001.

[33] W. Vanderperren and D. Suvée. Optimizing JAsCo dynamic AOP through HotSwap and Jutta. In R. Filman, M. Haupt, K. Mehner, and M. Mezini, editors, *DAW: Dynamic Aspects Workshop*, pages 120–134, Mar. 2004.

[34] W. Vanderperren, D. Suvee, M. A. Cibran, and B. De Fraine. Stateful aspects in JAsCo. In *Proc. of Software Composition (SC'05)*, volume 3628 of *LNCS*. Springer-Verlag, Apr. 2005.

[35] C. Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399