Client-Side Component Caching

A Flexible Mechanism for Optimized Component Attribute Caching

Christoph Pohl and Alexander Schill

Technische Universität Dresden
Institut für Systemarchitektur
Lehrstuhl Rechnernetze
D-01062 Dresden, Germany
{pohl,schill}@rn.inf.tu-dresden.de
phone: +49-351-463-38457
fax: +49-351-463-38251

Abstract. Locality of referenced data is an important aspect for distributed computing. Caching is commonly employed to achieve this goal. However, when using current component-oriented middleware client application programmers have to take care of this non-functional aspect by themselves, without direct support from middleware facilities or design tools. The paper at hand describes a novel approach to disburden them from this non-trivial, error-prone task by transparently integrating caching as an orthogonal middleware service using interceptors which are preconfigured at design-time using standard UML extension mechanisms. An advanced mechanism for dynamic adaptation of the caching service to changing access characteristics is introduced in the second part.

Keywords: Caching, distributed component-based middleware, Enterprise JavaBeans, adaptivity, reflection

1 Introduction

Todays middleware platforms, no matter whether procedural, object-, or component-oriented, apparently provide the means for transparent distribution by allowing remote procedure calls or method invocations to be as easily integrated as their local equivalents. However, a closer look reveals the caveats of this approach, at least if it's naively used: Every single remote call results in at least one network round trip which slows the code down by magnitudes.

With platforms like CORBA Components [1] or Enterprise JavaBeans [2], this issue is typically tackled at application level by streamlining remote interfaces, i.e. reducing necessary interactions between remote nodes, or by implementing caching frameworks at the same level. In their pursuit for the best possible locality of reference, distributed programs usually try to collocate data and process. This is either done by patterns like *value object* [3] that transfer bundled object attributes to the client, or by patterns like *session facade* [3]

J.-B. Stefani, I. Demeure, and D. Hagimont (Eds.): DAIS 2003, LNCS 2893, pp. 141–152, 2003.

that place computation-intensive logic at server side. Nevertheless, all of these workarounds violate the principle of transparent distribution, a non-functional aspect that should not bother the application programmer.

Our approach is based on the assumption that most component attributes are more often read than written which makes them suitable for caching. This additional meta-information can already be attached to the application model at design time. Generator tools use this information to preconfigure the caching logic. Integrated into the middleware itself via *interceptors* as reflectional mechanism, it avoids unnecessary network round trips unnoticed by the application programmer, thus forming a transparent proxy layer that completely hides the complexity of attribute storage and retrieval from the client programmer.

Possible use cases include interactive multimedia applications, e.g. eLearning scenarios, where object-oriented "fat client" programs need to communicate frequently with server-side data models. But the concept is also practicable for distributed server-side processing, e.g. web servers or Servlet containers accessing components on application servers. Generally, it's usable wherever component clients and servers are distributed over multiple network nodes.

The first prototype as explained in Sec. 2 relies on explicit, *static* attribute mark-ups at deployment time to enable an augmented container to generate necessary caching functionality. In contrast, our current activities aim at a self-learning concept that relieves the deployer or component assembler from the burden of classifying attributes by *dynamic* run-time adaptation to changing attribute access characteristics. The extensive usage of client-/server-side interceptor pairs also allows for the integration of more sophisticated centralized cache invalidation or update propagation schemes. This approach is introduced in Sec. 3.

2 Static Approach

As mentioned introductorily, our first prototype follows a static approach, i.e. cachability of attributes has to be declared at deployment time. Once considered cachable, an attribute remains in that state. A reference implementation based on Sun's Enterprise JavaBeans (EJB) platform [2] and the open source EJB container JBoss [4] was developed to demonstrate the underlying concepts. One major goal is the explicit, separate handling of the orthogonal non-functional aspect "caching" throughout a component's life cycle.

2.1 Component Design

Considering the standard software development process, developers already get a fair notion about a component's usage scenarios and corresponding data flows at design time, right after thorough analysis. A component's attributes are the most suitable candidates for caching as they contain its actual data. There are basically three categories of attributes[1]:

[1] References to other components can be treated in the same way as attributes in respect to caching, although they are technically handled in a different way.

read-only – practically never changes, to be cached upon first access;
cachable – changes rarely[2], to be cached upon first access; appropriate invalidation / update propagation protocol required for consistency;
volatile – non-cachable attributes that are subject to frequent changes or that should only be accessed in a transactional context.

The OMG's Unified Modeling Language [5] provides the means for storing such additional information using so-called stereotypes which imply certain characteristics and roles that can be evaluated by code generators and other tools to deduce applicable algorithms and code segments.

2.2 Component Implementation

As the underlying EJB platform encapsulates component attributes as pairs of `get/setXyz` methods by convention, stereotypes have to be mapped to these representations, accordingly. The Java programming language provides tagged comments, i.e. *JavaDoc*, for storing additional information about language elements that can be evaluated by compiler-independent parsers and tools. Some UML modeling tools already make use of this language feature. Thus, a stereotype << *cacheable* >> for a component attribute becomes a `/** @stereotype cachable */` comment above the corresponding accessor method, or a component-level constraint *caching.policy=LRUCachePolicy* translates to `/** @invariant caching.policy=LRUCachePolicy */` above the Bean class. We decided to use this feature for our EJB prototype in conjunction with *XDoclet* [6], an open source code generator for EJBs.

Originally intended to bridge the disconnection between bean implementations and interfaces that often tend to get out of sync, XDoclet generates interfaces, deployment descriptors, and auxiliary classes from Bean classes. It allows the construction of arbitrary code segments depending on special JavaDoc comments at class / method level and special template files that actually control the code generation process.

2.3 Code Generation

A special XDoclet Template is used to generate a separate `caching.properties` file conveying this information for client deployment. There is no use to package this information in a XML file along with other deployment descriptors because our static caching approach necessitates no additional processing by the server-side component container. Additionally, the generated `jboss.xml` is adapted to include our `CachingClientInterceptor` in the client-side interceptor chain, as explained below.

2.4 Deployment

To understand the way our caching-enabled components are deployed, a few introductory words about interceptors should be said:

[2] The definition of "rarely" is application-specific!

The modular architecture of JBoss features an interceptor framework similar to the OMG's specification [7] but more flexible for caching purposes. Interceptors are a meta-programming facility for distributed middleware platforms. On both client and server side, interceptors can be hooked into the control flow of (remote) operation calls, basically to add parameters and to augment results, but generally to alter virtually any property of a call's context, even its semantics.

The main difference between JBoss interceptors and CORBA Portable Interceptors is the way they are chained and handled: The CORBA specification defines different types of Interceptors and certain access points during request processing when they have to be called. The ORB would typically keep configured interceptors in an array and invoke them sequentially, each interceptor returning control after its task has been accomplished. In contrast to that, JBoss defines a slightly different protocol, described in [8], based on a linked list of interceptors established by the container. Although this imposes on every single interceptor the responsibility to invoke its successor, this allows for greater flexibility, i.e. to cut short the interceptor chain by quickly returning cached results.

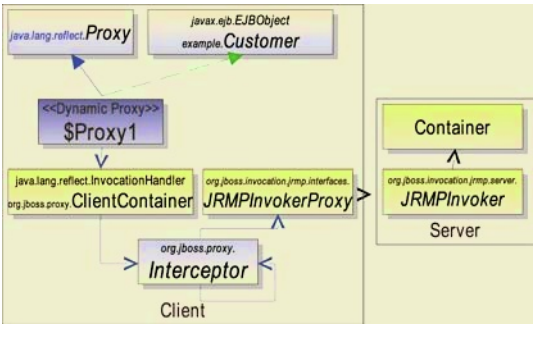

Fig. 1. Interceptors in JBoss Dynamic Proxies.

Client-side integration of these interceptors is shown in Fig. 1: Component proxies are transparently generated and instantiated using Java's Dynamic Reflection API. A `ClientContainer` passes each request through a chain of `Interceptors`, whose sequential order is determined by the bean provider or application assembler at deployment time. The last interceptor always hands the request to an `InvokerProxy` that finally calls the server. Server-side interceptors are stacked in a similar fashion. When a response returns from the server, it passes through the same interceptor chain in reverse order.

After code generation, the application assembler / deployer is given the opportunity to make certain manual adjustments to given descriptors, e.g. changing cachability of certain attributes, caching policy etc.

When a caching-enabled component is deployed, the interceptor chain is assembled as configured in the component's `jboss.xml` along with a `Proxy` as shown in Fig. 1. This conglomerate is transferred to the client upon its first JNDI lookup of the component. Interceptor instances are then created in the

client VM as needed by remote references. The first instance of a `CachingClientInterceptor` loads the information about cachable component attributes from the `caching.properties` located in the client class path and initializes the in-memory cache according to configured policies / replacement strategies.

2.5 Runtime

Prototypically, the cache back-end was implemented selectively using JBoss' `LRUCachePolicy` or `TimedCachePolicy` with component identity, method, and parameters as combined keys and results as values, i.e. $(i, m, \{p\}) \rightarrow r$. The basic granularity of cached data is per-attribute but as these are members of identifiable components, collective invalidation of attributes is still possible.

As we already elaborated in [9], *multiple reference handling* is also an important issue for caching services in distributed component middleware. Component references are typically passed around as marshaled objects, i.e. proxies / stubs, which makes it possible for a client to obtain a number of proxy objects for one and the same remote entity. This is counterproductive for memory consumption. `CachingInterceptor`s have also been leveraged to support efficient multiple reference handling by checking all returned remote references, i.e. proxies, for duplicates in the local cache, ensuring that at most one reference exists to a given component. The same is done with returned collections of references by querying their elements individually against the cache. Sun's EJB specification [2] explicitly discourages direct equality testing between entities using `equals()`, and `isIdentical()` may result in additional undesired remote calls, so `EJBHandle`s are held liable for component equality that is necessary for duplicate checking. Note that JBoss's proxy implementation already transfers handles upon initialization, hence no additional network round-trips are required. Calls to `remove()` methods require the interceptor's special attention because they imply the removal of all cache entries for keys $(i_r, m, \{p\})$ with a given identity i_r.

2.6 Performance and Usability Benefits

Our experiences with the framework showed general feasibility of the concept. The use of client-side interceptors is mandatory with JBoss, so the overhead for invoking yet another interceptor is quite low. Cache lookups turned out to be magnitudes faster than direct component attribute queries. A simple test scenario was set up with both client and server VM running on the same host[3] to eliminate the interfering influence of variable network delays. Preliminary results of cache miss times for queries to a component's *value object* were around $20ms$ per request, compared to $1ms$ and even less for cache hits. Depending on networking infrastructures, several more ms can be added for cache misses in non-local scenarios. More profound data is currently being collected in connection with the results of the following section.

[3] AMD Athlon™ XP1600+, 1GB RAM, Linux 2.4.21, Sun J2SE 1.4.1, JBoss 3.0.6.

However, the main advantage of our approach lies in the field of software engineering, formed by the usability benefits of the solution in comparison to traditional pattern-based solutions of the caching challenge.

3 Dynamic Approach

An obvious disadvantage of the above described static approach is necessity for component developers and deployers to precisely describe a component's cachability properties before deployment without any chance of later interference. This drawback gave the motivation for our current endeavors [10] to extend the framework to dynamically adapt cachability status of component attributes at runtime, i.e. whether a certain attribute should be considered for caching or not.

3.1 Server-Side Data Gathering

It has been anticipated in Sec. 2.4 that interceptors are also available at server side in quite a similar fashion. This enables us to centrally collect data about component access characteristics by implementing and chaining `CachingServerInterceptor`s. How the gathered information is evaluated and eventually used to dynamically adapt the `ClientCachingInterceptor`'s behavior is shown in Fig. 2.

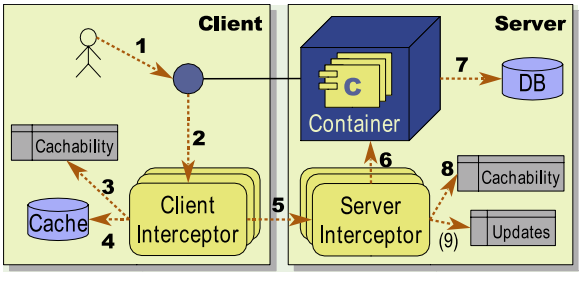

Fig. 2. Adaptive Caching Approach.

1. A container-generated dynamic proxy implementing the desired component's remote interface is called from somewhere within the client application code.
2. The proxy creates an `Invocation` object and passes this through the interceptor chain where a `CachingClientInterceptor` is installed to quickly answer invocations whose results it can anticipate from its cache contents.
3. If the invocation does not refer to an exceptional case like a `create` or `remove` method, the `CachingClientInterceptor` first checks its cachability table, a client-local singleton preconfigured from the `caching.properties` file (optional, if existent) and continuously updated by dynamic adaptation as described below. This table basically contains information about whether to cache certain component attributes based on accessor methods (`getXyz`).

4. Provided the afore mentioned test succeeded, the cache is eventually queried for valid invocation results as already explained in Sec. 2.5. Mutator methods (`setXyz`) lead to updates of their corresponding attributes before being passed on.
5. If the queried attribute was not found in the cache or is not to be cached for some reason, the invocation will finally be transferred to the server where it will pass another interceptor chain containing the `CachingServerInterceptor`.
6. At this moment, the interceptor will just hand off the invocation till it gets processed by the component container.
7. Depending on component type and state, the container may also query a database or other back-end information source to retrieve the desired data.
8. On its way back[4], the invocation is logged in a server-side cachability table for counting read / write accesses and deriving a cachability categorization based on the current read / write ratio for a given attribute. The last change time of an attribute's categorization is also logged in this table.
9. Successfully processed mutator methods receive special treatment: They are logged together with their time of occurrence and average time period in between for purposes of update propagation. The average time period is not calculated over all write accesses but rather only over the last five, which seems to be an appropriate heuristic, more economical in respect to memory consumption and more sensitive to rapid changings. These slots are initially set to a configurable default value.

3.2 Update Propagation and Client-Side Cache Adaptation

One Question remains unclear in the above presented procedure: How are clients notified about server-side updates concerning component state and cachability? Earlier experiments with event-based publish-subscribe middleware [11] scaled poorly for increasing numbers of clients due to the tremendous amount of status data and connections the server has to govern when using such an approach. Therefore, the decision was made for a client-driven "pull" strategy that relieves the server from the burden of direct "push" update propagation. The general proceedings of this approach can be described as follows:

– When a cachable attribute is accessed, the `CachingServerInterceptor` attaches the times of last modification and expiration according to the attribute's current average time span between changes as additional payload to the Invocation object, taken from step (9) above. The JBoss Proxy package uses `Invocation` objects to encapsulate all data belonging to a remote call. This object is passed through the interceptor chains where interceptors may attach and detach additional context information. The current cachability setting as explained in step (8) is also attached, accordingly.

[4] For concision and better readability, the return path has not been explicitly marked in Fig. 2. It basically follows the numbers in reverse order.

- Back on client side, the `CachingClientInterceptor` updates the attribute's cachability categorization if necessary and schedules a `java.util.TimerTask` with the given expiration time. This TimerTask will enqueue the attribute's identity together with its last modification time in a list of expired objects.
- The next remote call passing a `CachingClientInterceptor` picks up the value pairs from this expiration list and attaches them to the Invocation, thus preventing additional network traffic by this piggy-back strategy.
- Unmarshalled at server side, the expiration list is compared with the last modifications in the cachability and update tables, resulting in the creation of positive list containing a bit mask for changed cachability categorizations and value updates, which is transfered on the call's way back to the client.
- Updated attributes are then discarded from the client-side cache, implicating a normal retrieval upon next access that causes the described procedure to start over again. If an attribute turns out to be not cachable any longer, any of its possibly existing cache entries will also be discarded.

3.3 Performance Considerations

As the described dynamic approach for continuous adaptation of the caching service to changing access characteristics is still partially under development, it naturally contains a number of flaws. At the current status, the considerable overhead for update propagation limits the algorithm's efficiency to scenarios with larger attributes.

A number of optimization is currently being implemented. For instance, the mentioned *value objects* provide a convenient way for grouping attributes with similar access characteristics. Instead of caching individual attributes, single attribute queries can be mapped to cached value objects. Overhead for invalidation decreases as well. The current invalidation scheme could also be enhanced to support update propagation for certain attributes that require fast availability of changes at all replicating nodes, in other words: Instead of transmitting a list of updated attributes, the server-side interceptor could immediately send the changed values, thus saving a network round trip.

4 Related Work

Paradigms for distributed computing can basically be distinguished into distributed shared memory systems (DSM) and systems communicating via message queues or remote procedure calls (RPC). The former ones, typically found in high speed computing environments, share a mutual set of distributed memory pages whose consistency is maintained by the memory subsystem using multicast and similar technologies. Whereas in middleware of the latter category, the application programmer usually has to take care of data distribution and consistency by himself. If caching is used as a special form of partial replication, DSM systems typically rely on page-oriented caching strategies and procedural

or message-oriented middleware on object caching algorithms, respectively. As current distributed component middleware platforms like defined by [12] obviously belong to the latter kind, we will concentrate on the corresponding caching issues only.

4.1 Object Caching

A vast multitude of publications exists in the field of object caching, many of them on typical hypertext transfer issues. However, solutions for object-oriented middleware also date back a long way, e.g. to *Arjuna*, *Shadows* [13], and *Orca* [14], among others. Interesting parallels to "modern" patterns like *value object* [3] can already be found and the issues of invalidation versus update propagation are discussed there.

CASCADE [15], a CORBA caching service for applications in Wide Area Networks offers interesting insights on hierarchical cache management and staggered consistency levels, an aspect also taken up by other publications [16]. *Flex* [17] is a distributed caching system on top of *Fresco* and CORBA that also considers issues about caching object references, like object faulting and access detection, which shows the parallels to object-oriented databases. Although relevant, these publications have a somewhat different focus. Eberhard and Tripathi [18] proposed a transparent caching mechanism for Java RMI with configurable caching strategies and consistency protocols but they did not take changing access characteristics into account.

4.2 Adaptive Caching

The term "adaptive caching" is slightly overloaded, e.g. there are projects trying to combine the virtues of algorithms from page-oriented caching with object-oriented techniques. *ACME* (Adaptive Caching Using Multiple Experts) [19] uses machine-learning algorithms to dynamically weigh cache replacement strategies according to their success, which provides better performance for proxy cascades in web / hyper-media scenarios. *Divergence Caching* [20] illuminates the aspects of static and dynamic caching, i.e. fixed and variable refresh rates. Brügge and Vilsmeier [21] propose a caching strategy for CORBA Calls similar to the one presented in our paper, with focus on prefetching of attribute groups based on statistical evaluations of past invocations. However, no approaches to dynamically determine cachability are known to the authors.

4.3 Meta-programming

Meta-Programming or *reflectional programming* usually refers to coding on the abstract meta-level of a programming language that is used to describe the executed code itself, i.e. in terms of classes, methods etc. The basic mechanisms have not changed greatly since Smith's thesis about reflection [22]: programs or components should have a notion about their current context and (limited)

control over their interpretive environment. Interfaces to this meta-level are defined by *meta-object protocols* (MOP) like interceptors as they are used in the context of this paper. Several recent publications [23,24] leverage interceptors for building frameworks that more or less try to partly hide the complexity of meta-programming or to add a higher abstraction level. As we are using interceptors only as the means for integrating orthogonal functionality, current research on this field is slightly out of this paper's scope. The OMG's CORBA Portable Interceptor specification [7] is an attempt to standardize various research directions in this field, but offers less flexibility than the JBoss interceptor framework [8] used in our approach.

ArchJava [25] recently supports a wide range of connector abstractions for communication between components, including caches, but the focus is more on the ease of use provided by ArchJava's language extensions. In contrast to that, we completely resign the use of such extensions for the sake of transparency. An article of Filman et al. [26] is quite comparable to that but aims more at the shortcomings of Aspect-oriented Programming [27]; caching is also mentioned there as an example application of their approach.

5 Conclusion and Outlook

Our static approach to distributed component attribute caching showed the potential performance benefits and gave a first impression of the transparent integration into a component's life cycle. The proposed UML extension mechanisms are currently being summarized in a special UML profile for caching.

First directions for improvement have already been outlined in Sec. 3.3: *Value objects* provide a potential to decrease overhead by grouping attributes with similar access characteristics. Update propagation can be selectively used as an alternative to invalidation for rapidly changing, *volatile* attributes. The protocol should be easy to adapt in this direction.

Further investigations will include prefetching, i.e. possibilities to transfer data to client-side caches *before* it is queried, which especially suitable if cached components are organized in a hierarchical way. The greatest challenge in this connection will be the automatic detection of such data dependencies. Persistent caching will also be a focus of our future work because it provides an interesting feature for use cases like off-line client applications.

References

1. Object Management Group: CORBA Components. (2001) ptc/01-11-03.
2. DeMichiel, L.G., Yalçinalp, L.Ü., Krishnan, S.: Enterprise JavaBeans Specification Version 2.0. Sun Microsystems. Final release edn. (2001)
3. Sun Microsystems: Design Patterns Catalog. J2EE design patterns edn. (2001) `http://java.sun.com/blueprints/patterns/j2ee_patterns/catalog.html`.
4. JBoss Group: JBoss. (2003) Project homepage: `http://www.jboss.org/`.
5. Object Management Group: Unified Modeling Language, v1.4. (2001) formal/01-09-67.

6. Öberg, R., Schaefer, A., Abrahamian, A., Hellesøy, A., Colebatch, D., Harcq, V.: XDoclet. Project homepage: `http:// xdoclet.sourceforge.net/` (2003)
7. Object Management Group: CORBA Portable Interceptor Specification. (2001) ptc/01-03-04, formal/02-05-18.
8. Fleury, M., Reverbel, F.: The JBoss extensible server. In Endler, M., Schmidt, D., eds.: International Middleware Conference. Volume 2672 of LNCS., Rio de Janeiro, Brazil, ACM / IFIP / USENIX, Springer (2003) 344–373
9. Pohl, C., Schill, A.: Middleware support for transparent client-side caching. In: European conference on Theory And Practice of Software (ETAPS'02). Volume 65 of Electronic Notes in Theoretical Computer Science., Grenoble, France, Elsevier (2002) Software Composition Workshop.
10. Pohl, C.: Adaptively caching distributed components. In: Middleware2003 Companion, Rio de Janeiro, Brazil, PUC–Rio (2003) 325
11. Neumann, O., Pohl, C., Franze, K.: Caching in Stubs und Events mit Enterprise Java Beans bei Einsatz einer objektorientierten Datenbank. In Cap, C.H., ed.: Java-Informations-Tage JIT'99. Informatik Aktuell, Düsseldorf, Springer (1999) 17–25
12. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. Addison-Wesley (1997)
13. Caughey, S.J., Parrington, G.D., Shrivastava, S.K.: Shadows - a flexible support system for objects in distributed systems. In: 3rd International Workshop on Object Orientation and Operating Systems (IWOODS'93), Asheville, NC (USA) (1993) 73–82
14. Bal, H.E., Kaashoek, M.F., Tanenbaum, A.S., Jansen, J.: Replication techniques for speeding up parallel applications on distributed systems. Concurrency: Practice and Experience **4** (1992) 337–355
15. Chockler, G., Dolev, D., Friedman, R., Vitenberg, R.: Implementing a caching service for distributed CORBA objects. In: Middleware'00, Heidelberg, Germany, Springer (2000) 1–23
16. Krishnaswamy, V., Ganev, I.B., Dharap, J.M., Ahamad, M.: Distributed object implementations for interactive applications. In: Middleware 2000. (2000)
17. Kordale, R., Ahamad, M., Devarakonda, M.V.: Object caching in a CORBA compliant system. Computing Systems **9** (1996) 377–404
18. Eberhard, J., Tripathi, A.: Efficient object caching for distributed Java RMI applications. In Guerraoui, R., ed.: Proceedings of the International Middleware Conference (Middleware 2001). LNCS, Heidelberg, Germany, ACM / IFIP / USENIX, Springer (2001) 15–35
19. Ari, I., Amer, A., Gramacy, R., Miller, E.L., Brandt, S.A., Long, D.D.E.: ACME: Adaptive caching using multiple experts. In: Workshop on Distributed Data and Structures (WDAS 2002), Carleton Scientific (2002)
20. Huang, Y., Sloan, R.H., Wolfson, O.: Divergence caching in client-server architectures. In: Third International Conference on Parallel and Distributed Information Systems (PDIS '94), Austin, TX, IEEE (1994) 131–139
21. Brügge, B., Vilsmeier, C.: Reducing CORBA call latency by caching and prefetching. IEEE Distributed Systems Online (2003)
22. Smith, B.C.: Procedural Reflection in Programming Languages. PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, Massachusetts, USA (1982)
23. Blair, G.S., Coulson, G., Robin, P., Papathomas, M.: An architecture for next generation middleware. In: International Conference on Distributed Systems Platforms and Open Distributed Processing, London, IFIP, Springer (1998)

24. Truyen, E., Vanhaute, B., Joosen, W., Verbaeten, P., Jørgensen, B.N.: Dynamic and selective combination of extensions in component-based applications. In: International Conference on Software Engineering, IEEE (2001) 233–242
25. Aldrich, J., Sazawal, V., Chambers, C., Notkin, D.: Language support for connector abstractions. In: ECOOP 2003 Proceedings. Volume 2743 of LNCS., Darmstadt, Germany, AITO / ACM SIGPLAN, Springer (2003)
26. Filman, R.E., Barrett, S., Lee, D.D., Linden, T.: Inserting ilities by controlling communications. Communications of the ACM **45** (2002) 116–122
27. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: European Conference on Object-Oriented Programming (ECOOP'97). Volume 1241., Springer (1997) 220–242