

Caching Web Services: Aspect Orientation To The Rescue

Marc Ségura-Devillechaise and Jean-Marc Menaud

EMN, La Chantreterie, 4, rue Alfred Kastler. B.P. 20722,
F-44307 NANTES Cedex 3, France,
Phone: (+33) (0) 2 51 85 81 00
Fax: (+33) (0) 2 51 85 81 99
{msegura, jmenaud}@emn.fr

Abstract. Web caches are a software answer to the growing demand of bandwidth on the Internet. But the increasing number of dynamic Web objects like Web services decreases caches performances. This paper proposes to adapt dynamically caches to the cached objects. The required adaption is often dynamic, motivated by a variability that programmers could not anticipate before runtime. This paper discusses the possibility to use aspect orientation to achieve dynamic adaptation. While aspects are appearing as a suitable tool, dynamic adaptation adds some requirements to the aspect system that could be used. The purpose of this paper is to reach a clear and explicit understanding of these requirements.

Keywords: web caching, dynamic adaptation, aspect orientation, dynamic weaving, web services.

1 Introduction

The ever-increasing popularity of the Internet creates an urgent need to reduce the resulting traffic congestion. The problem is to improve the average response time. Researchers have explored two types of solutions. The first one consist in increasing the bandwidth of the network links. This raises a number of financial and technical problems. The second is to use caches over the Internet to replicate the most frequently accessed data. Although the latter is more affordable than the former, the actual benefit of using caches to improve response time over the Internet still remains negligible.

The maturation of Web-caching infrastructures raises hopes that clients and servers will both become more cache-friendly, for example, by using some H.T.T.P. and H.T.M.L. tags. This could increase both the fraction of Web content that is cacheable, and the fraction of Web content served through caches (rather than trough originating servers). Unfortunately, the World Wide Web is growing and changing rapidly. As shown in [1], the dynamic nature of Web data poses a

generic problem to information systems that either cache, summarize or index the Web.

Two key factors explain the growing number of dynamic objects on the Web. The first is a direct consequence of the web page lifecycle. More than 40% of pages in the .com domain change every day [2, 3]. Professionals maintaining commercial pages, update them frequently to provide timely information and attract more customers. The second source of dynamism is provided by the use of C.G.I.¹ scripts. Our study [4] on Internet traces shows that 45% of the different requests contain a question mark character (“?”) characteristic of these scripts. These particular requests are directly sent to C.G.I. scripts. Then, the script generates a dynamic H.T.M.L. response. Whatever the reasons, dynamicity decreases Web caching performance.

However, a new trend is emerging. Traditionally, the algorithms involved in the dynamic pages generation remains buried in the information system, in the Web server back-end. But now, the wide-spreading Web services are externalizing their algorithms in one or several components. Therefore, contrary to C.G.I. scripts, the generating components; Web services, are identified. This identification is done both in terms of server localization and semantic description of the offered functionalities. An appropriate answer is to apply caching techniques to Web services.

Unfortunately, caching Web services sounds difficult. First, downloading an entire Web service might outweigh the gain. This is a size problem. Secondly, most Web services are strongly coupled to the information systems they came from. Typically, they are interacting with the information system databases and potentially with other Web services. A cached Web service still requires these interactions. It is unclear if the gain in caching the Web service will not be outweighed by the amount of data exchanged during these interactions. These two problems lead to the idea that information systems should be responsible to provide specialized caches meeting their specific constraints.

Still, as shown in section 2, the previous idea is not easy. A cache remains a complex piece of software. As for Web services, the size of a specialized cache might prevent it to be downloaded. A solution, then, is to cope only with the specific parts of the specialized cache. The goal then is to download only these specific parts and to adapt a generic cache.

However, the practical experiments described in section 2 show that traditional modularization approaches are insufficient to allow this type of adaptation. The roots of the problem is that the alterations to tailor the cache to the information system are generally crosscutting the generic cache modularization. Section 3 proposes to use aspect-orientation to solve this problem. Section 4

¹ Common Gateway Interface

analyses how this technique can support dynamic adaptation. Section 5 presents future work and discusses research tools that might be suitable. Finally, section 6 concludes.

2 Motivations: modifying a cache

This section summarizes the problems we encountered while implementing a Web caching system. First, an overview of the different subsystems is given. It allows us to describe the issues arising while augmenting an existing cache; Squid, with a new cooperation protocol. Finally, the last subsection provides an abstract view of the described problems.

2.1 Web caches

Basically, a cache is a software that stores commonly accessed data elements on disk. The cache management policy drives data replacement to keep in the cache only the elements that are likely to be reaccessed. Considering network topology, the idea of making network caches cooperate has emerged.

The hierarchical approach to network cache cooperation is a pioneering idea. The idea was suggested by a study on Internet traffic in the United-States [5]. It concludes that Internet traffic can be a priori reduced by 30% by introducing a cache on every network node. The cache system's hierarchical structure is mapped on the national network hierarchical organization. In this type of system, a missing object is located by propagating the request to the upper level in the cache hierarchy. This process is iterated until either the object is found or the root cache is reached; it might ultimately involve contacting the object's server. While returned to the client, the object is then copied in all the contacted caches. A transversal system refines the hierarchy concept by grouping, at each level, in sets of caches roughly having the same latency time. In these systems, on a cache miss, a cache not only contacts its ancestral cache but also the other caches in its sets, its siblings.

For different reasons, detailed in [6], a transversal approach is ineffective: it introduces a significant augmentation of bandwidth consumption. Based on the aforementioned evaluation of transversal systems, we designed a new protocol for transversal cooperation combining their benefits while having negligible network and machine overhead, hence providing a scalable solution. The principle is to fairly distribute knowledge among the caches composing the transversal system. We described the integrated protocol in previous publications [6, 7].

2.2 Technical introspection

We integrated our work with the most successful free cache software : Squid. This cache integrates directly the transversal cooperative protocol previously described, named I.C.P. [8]. Squid is the basis for our prototype: it provides the base cache management and a first hierarchical cooperative protocol. The purpose here is not to describe technically and precisely how the Squid cache version 1.2.22 is designed but only to give an intuition on how and what was modified to build our prototype.

Squid is a Web cache developed by the National Laboratory for Applied Network Research and members of the Web community. Squid implements only one unblocking process based on a main loop. This process handles all requests. For portability reasons, Squid do not use the thread programming paradigm. To manage different simultaneous requests Squid use a sequence of “functions handlers” invoked on I/O operations. Thus the code is difficult to understand. Moreover, the source is not commented and variables names are not explicit.

Moreover, the Squid main features are split in different modules. We classified these different modules in four essentials functionalities: client treatment (authorization, request parsing etc), server treatment (contact, fetching etc), store manager (hash table etc), and communication (protocol cooperation). In fact, in version 1.2.22, 17 modules are clearly separated. These modules can be roughly grouped in five groups as follow:

Client treatments `client_side`, `icp`, `acl` and `url`
Server treatments `http`, `gopher`, `wais`, `ssl`, `pass` and `proto`
Storage management `store`, `store_clean`, `disk`, `stmem` and `hash`
Communication management `comm` and `neighbors`
Misc. (mainly for statistic) `debug`, `objcache` and `stat`

To integrate our cooperation protocol we believed naively that only a couple of modules should be modified : `comm` and perhaps `neighbors`. Unfortunately, the final implementation altered seven modules: `icp`, `http`, `store`, `comm`, `neighbors`, `url`, and `client_side`. Acknowledging only the main alterations that we have committed, the replacement of the Squid cooperation protocol - protocol already localized in two modules (roughly 10% of the code) - required in practice the alteration of seven modules representing roughly 40% of the code.

Moreover, we did not simply add functions and mask others: we added variables, inserted hooks into some functions, modified some structures, and spread enumerate variables. We did not succeed to modularize the modifications on the original software. We experienced the same difficulties when inserting a new cache policy management in our prototype.

2.3 Discussion

In the previous experiment, the modification of a particular strategy (cooperative protocol or cache policy), required an in-depth analysis and the modification of more than 40 % of the source code, while the cache was designed in a modular way. Actually the Squid modularization was not exactly reflecting what we needed to change. Unfortunately, this kind of problem; variability mismatching the original program modularization, is not an isolated case.

For instance, in the Active Cache research prototype [9], information servers supply applets attached with documents. On cache hits, the cache invokes these applets. They provide the necessary processing without contacting the information server they came from. We understand Active Cache as a relatively generic cache specialized by applets techniques. More precisely, variability is modularized and parameterized through applets. In Active Cache, the interface through which applets and the cache are interacting is focused on file management. This design decision is motivated by security and performance concerns. Nevertheless, this interface constrains the huge variability anticipated by this kind of modular architecture. For example, the introduction of a new cooperative protocol requires a complete rewriting of all the applets: only the file manager can be reused. In Active Cache, the adaptation interface does not anticipate variability of the file management system.

These examples highlight the fact that the alterations to tailor the cache to the information system are generally crosscutting the generic cache modularization. As described before, this was the case with Squid, it is the case with Active Cache. It is even likely that it is always the case with traditional modularization techniques.

3 Towards a solution

The previous section shows that a good modularization is not sufficient to support the adaptation needed in section 1. Thus, we wonder if existing adaptation techniques could not allow us to tailor a generic cache for a specific information system. The first subsection 3.1 will show that aspect-orientation offers this potential. This technique will be described in subsection 3.2.

3.1 Adaptation techniques

As seen previously, the required adaptation is dynamic and motivated by a variability that can not be anticipated. Unexpected variability, or in other words, unanticipated changes in how the computation has to be done, puts a strong requirement on the adaptation technology. Typical applications anticipates variability through configuration files: most applications can now reload them at

runtime. An advanced technique like reflection [10] supposes that variability occurs at the language level. For example, reflection has been used in OpenCorba [11] to build an adaptable object broker. Typically in these approaches, variability is modelled through meta-classes. Open Implementation [12, 13] requires the application developer to anticipate the changes to offer an adaptation interface. Using these techniques to support dynamic adaptation seems difficult because they anticipate variability prior to runtime.

At runtime, to support dynamic adaptation, the program has to change “how” it is doing its computation. As shown before, there is a potential mismatch between this change and the activities promoted by the original decomposition of the program: the required changes potentially crosscut the compositional structure of the original program. In this context, because variability can not be predicted in advance, adaptation appears as a dynamic cross cutting concern. We propose to use Aspect Oriented Programming (A.O.P.) [14, 15] to support dynamic adaptation. We characterize dynamic adaptation as an alteration of how the program performs its computation motivated by the occurrence of a variability, unanticipated by the program. According to this definition, supporting adaptation means re-composing how the computation is done by the original program so that the resulting program anticipates the encountered variability.

3.2 An overview of Aspect Orientation

Because aspect orientation is still a young fertile research domain, it is relatively difficult to define. We propose a description based on the operational model described in [16] using AspectJ [17] terminology. Typically *advices* are used to incrementally modify “how” the computation is done by the program. More precisely, an advice specifies an action to be taken whenever some condition arises during the execution of the program. The events by which advices may be triggered are called *join points*. The process of executing the relevant advice at each join point is named *weaving*. The condition is specified using a predicate language over the join points. These predicates are known as *point cuts*.

Thus, compared to other adaptation techniques, aspect-orientation appears as having the potential to support dynamic adaptation. The choice of an appropriate aspect-language should be based on: the join point model, the point cut language, and the advice model. The first element designs which events the weaver can observed. The second element covers how the information from the events are communicated to the advice. Finally, the advice model describes how advices are composed. Composition is twofold: first the advice has to be composed with the program but the composition of advices together is an important issue too.

4 From dynamic adaptation to A.O.P.

Considering the A.O.P. key elements described previously and the characterization of dynamic adaptation given in section 3, we can propose a mapping between the two. Point cuts describe the occurrence of variability that advices are addressing. The advice model defines how the computation performed by the program will be re-composed. Join points defines where variability could be observed in the original program. Dynamic adaptation puts specific constraints on each of these elements. To move from tailor-made to ready-to-wear, the reusability of adaptation code is a key issue.

4.1 Dynamic adaptation and the join point model

The join point model actually limits the scope of the variability that the point cuts can address. The generic cache cannot anticipate all the information system it may encounter. Consequently, because “what has to be changed” can not be anticipated previously, point cuts need to be able to describe an occurrence of variability at any point during the execution of the program. Therefore, to fully support dynamic adaptation, the join point model should cover the entire execution of the program. It should not only be dynamic: each action performed at runtime should be a potential join points. Furthermore, runtime monitors could also be used as join point, allowing, for example, an aspect to be woven when the bandwidth drops under a certain value. However this could lead to poor runtime performance. But, we believe that observing only the relevant part² of the program through dynamically inserted probes, combined with the use of partial evaluation can lead to an acceptable overhead.

4.2 Dynamic adaptation and the advice model

The advice model merging aspects with the original program corresponds to the program recomposition in the characterization of dynamic adaptation. In most A.O.P. systems, composing advices together is not easy. It means that composing different adaptations on the same program using A.O.P. can be potentially difficult. However it is very desirable to be able to compose different dynamic adaptation together: a Web cache is intended to meet several different information systems. Most problems arising in composing advices together occur because of a static weaving process³ and because it is unclear if once weaved, the advice is observed by the join points. Thus an advice model to support dynamic adaptation should allow dynamic weaving and ensure that woven advices are observed by join points.

² That is the one referenced by the point cut.

³ I.e. performed before load time

4.3 Dynamic adaptation and the point cut language

To describe the variability, a declarative language seems an appropriate choice: information systems are not interested into “programming” they would rather “declare” the changes. This task however can be complex, the point cut language must be expressive enough.

Because the point cut language intends to describe something: a change in the original program, it raises the problem of the desirable coupling degree between point cuts and the program. What level of coupling between the advice and the program should the point cut language allow to express? Coupling entails that the advice (or the program, or both the program and the advice) knows of the program (of the advice or of each other respectively). To support dynamic adaptation, the program simply can not make assumptions about the woven advices: from the program viewpoint, weaving must occur transparently. Therefore, an appropriate point cut model should only allow advices to assume, through the point cuts triggering the weaving, where they are woven. To support the adaptation required by web caching, the program should never know about advices.

4.4 Dynamic adaptation code reusability

In the mapping defined between dynamic adaptation and A.O.P. in section 4, the reusability of the code achieving dynamic adaptation corresponds to the aspect reusability. The less an aspect assumes where it will be woven, the more it becomes reusable. This is another side of the question expressed previously: what level of coupling between the advice and the program should the point cut language allow to express? This section rephrases this question in: should the point cut language forbid coupling between the advices and program to ease aspects reusability? Aspect-oriented development case studies, like for example Kersten and Murphy [18], stress that complete decoupling occurs rarely. It is likely that this is the case in dynamic adaptation as well. An aspect adapting the storage management strategy in an a Web cache must know about the program it is adapting: it needs to know at least that the program described by the join points is a Web cache.

Thus, assuming that few adaptation concerns are truly orthogonal to the adapted program appears as a reasonable hypothesis. Consequently the advice must be aware of the program. Therefore, the declarativeness of the point cut language should grant that a programmer can start by writing advices assuming completely their weaving location. And its expressiveness should grant that ad hoc advices can be incrementally refined to reach the other side of the spectrum: the full decoupling of an aspect from the program it is adapting. More precisely, it should be possible to write an ad hoc aspect for a specific program encountering a given variability and to refine this advice incrementally so it can

adapt almost any program encountering this variability. Technically, the point cuts of the ad hoc aspect version refer to class and method names while the generic advice version analyzes the context, through the state of the runtime, and the actions taken by the program. While fully decoupled aspects open exciting perspective like aspect-on-the-shelf, it still makes sense to write aspects assuming their weaving positions. This kind of aspects are not reusable but they still localize the adaptation code.

5 Future work

We are currently writing the cache sketched in the introduction to test our ideas. This requires to dispose of an adequate A.O.P. system. It is likely that we shall write one from scratch. From an implementation viewpoint, join points are often reduced to positions of syntactic elements appearing in the source code. This restriction allows the weaving to be performed entirely before the program execution. While this approach avoids the introduction of overhead at runtime, it fails to handle dynamic concerns. But we are still reviewing the recent work of Kris Gybels [19]. This work supports dynamic cross cutting concerns and follows Kris de Volder [20] idea of using a logic programming language as point cut language. Kris Gybels tool is an aspect system for Smalltalk with a Prolog system named SOUL on top. SOUL is used as a point cut language. Another alternative PROSE [21, 22] allows dynamic aspect weaving and unweaving. It stands for PROgrammable extenSion of sEervices. PROSE uses a Java API as predicate language over the join points. Technically PROSE is based on the Java Virtual Machine Debugger Interface (JVMDI).

6 Conclusion and future works

Caching techniques are a software answer to the growing demand of bandwidth on the Web. But the Web is becoming increasingly dynamic. New dynamic objects like Web Services are emerging. This identification is done both in terms of server localization and semantic description of the offered functionality. It is therefore possible to cache them. Still, Web services are strongly coupled to their information systems. Typically they need databases connections, access to other application servers, etc This paper advocates that information systems should tailor a cache to their needs. Our practical experiments shows that the required adaption is dynamic, motivated by a variability that could not be anticipated before. We suggested to understand dynamic adaptation as a dynamic cross cutting concern. Therefore, we believe that aspect orientation has the potential to support dynamic adaptation. However, as explained previously, dynamic adaptation puts some specific constraints on the aspect system that can be used. First, the join point model should cover every action performed at runtime by

the program. Secondly, the point cut language should be declarative, expressive, and minimizing the coupling between advices. Finally, the weaving process should be dynamic and advices once woven should be observed by join points. Our on-going work is focusing on validating our ideas through the implementation of a Web cache achieving the scenario sketched in the introduction.

Moving to a conceptual point of view, Aspect Oriented Programming tries to provide tools allowing “*the modularity of a system to reflect the way “we think about it” rather the way the language or others tools force us to think about it*” [17]. Because the program authors and the program users are different, the code achieving a particular adaptation should be a concern separated from the program. From a software engineering point of view, because different users have different and diverging adaptation needs, it is desirable to separate the adaptation code from the original program.

Another way to understand our proposition is to say that we would like to use A.O.P. re-modularization capabilities to solve the potential mismatch between the original program modularization and what dynamic adaptation needs to alter in it. We use aspects to dynamically separate the concern that have to be adapted from the original program.

As a proof of concept of our ideas, we currently prototype a Web cache supporting dynamic adaptation. We are paying attention to security (code verification, code validation, ...) and to performance.

References

- [1] Lawrence, S., Giles, C.L.: Accessibility of information on the web. *Nature* 400 (1999)
- [2] Cho, J., Garcia-Molina, H.: The evolution of the web and implications for an incremental crawler. In: *The VLDB Journal*. (2000) 200–209
- [3] Lim, L., Wang, M., Padmanabhan, S., Vitter, J.S., Agarwal, R.C.: Characterizing web document change. In: *Web-Age Information Management*. (2001) 133–144
- [4] Menaud, J.M.: Cooperative Caches System for Large Scale Distributed Information System. PhD thesis, IRISA/INRIA Rennes France (2000)
- [5] P. Danzig, R.S.H., Schwartz, M.F.: A case for caching file object inside internet-networks. In: *Proceedings of ACM Sigcomm’93*. (1993) 239–248
- [6] Menaud, J.M., Issarny, V., Banâtre, M.: A new protocol for efficient cooperative transversal web caching. In: *International Symposium on Distributed Computing*. (1998) 288–302
- [7] Jean-Marc Menaud, Valrie Issarny, M.B.: A scalable and efficient cooperative system for web caches. In: *IEEE Concurrency*. (2000)
- [8] Wessels, D.: Configuring hierarchical squid caches. *Squid/Hierarchy-Tutorial* (1997)
- [9] Cao, P., Zhang, J., Beach, K.: Active cache: Caching dynamic contents on the web. In: *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware ’98)*. (1998) 373–388

- [10] Smith, B.: Reflection and semantics in Lisp. In: Proceedings of the Symposium on Principles of Programming Languages, ACM Press (1984) 23–35
- [11] Ledoux, T.: OpenCorba: A reflective open broker. Lecture Notes in Computer Science **1616** (1999) 197–215
- [12] Kiczales, G., Lamping, J., Lopes, C., Maeda, C., Mendhekar, A., Murphy, G.: Open implementation design guidelines. In: International Conference on Software Engineering. (1997) 481–490
- [13] Rao, R.: Implementational reflection in Silica. In: ECOOP. (1991) 251–267
- [14] Lopes, C., Hirsch, W.: Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University (1995)
- [15] Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In Akşit, M., Matsuoka, S., eds.: ECOOP. Volume 1241. Springer-Verlag, New York, NY (1997) 220–242
- [16] Douence, R., Motelet, O., Südholt, M.: A formal definition of crosscuts. In: Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns. Volume 2192 of LNCS., Springer Verlag (2001)
- [17] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP. (2001) 327–353
- [18] Kersten, M., Murphy, G.C.: Atlas: a case study in building a Web-based learning environment using aspect-oriented programming. ACM SIGPLAN Notices **34** (1999) 340–352
- [19] Gybels, K.: Using a logic language to express cross-cutting through dynamic joinpoints (2002)
- [20] Volder, K.D., D’Hondt, T.: Aspect-oriented logic meta programming. Volume Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection’99. (1999) 250–272
- [21] A. Popovici, T. Gross, W.B.: Dynamic homogenous AOP with PROSE. Technical report, ETH Zürich Department of Computer Science Institute of Information Systems (2001)
- [22] A. Popovici, G. Alonso, T.: AOP support for mobile systems. OOPSLA 2001 Workshop: Advanced Separation of Concerns in Object-Oriented Systems (2001)