C/SPAN: a Self-Adapting Web Proxy Cache

F. Ogel, S. Patarin
Regal group
INRIA Rocquencourt
Domaine de Voluceau
78153 Le Chesnay, France

I. Piumarta, B. Folliot
LIP6
Université Pierre et Marie Curie,
4, place Jussieu,
75252 Paris Cedex 05, France

Abstract

In response to the exponential growth of Internet traffic, web proxy caches are deployed everywhere. Nonetheless, their efficiency relies on a large number of intrinsically dynamic parameters, most of which can not be predicted statically. Furthermore, in order to react to changing execution conditions — such as network resources, user behavior or flash crowds, or to update the web proxy with new protocols, services or even algorithms — the entire system must be dynamically adapted.

Our response to this problem is a self-adapting Web proxy cache, C/SPAN,[1] *that applies administrative strategies to adapt itself and react to external events. Because it is completely flexible, even these adaptation policies can be dynamically adapted.*

1 Introduction

The World Wide Web has become a victim of its own success. The number of users and hosts, as well as the total traffic, exhibits exponential growth. To address this issue web proxy caches are being deployed almost everywhere, from organization boundaries to ISPs. Since the efficiency of the caching system relies heavily on a large number of parameters (such as size, storage policy, cooperation strategies, etc.), configuring them to maximize performance involves a great deal of expertise. Some of these parameters, such as user behavior, cannot be predicted statically (before deployment). Furthermore, because of the ever-changing nature of Web traffic, any configuration is unlikely to remain effective for long: benefits of inter-cache cooperation are subsequently reduced when cache contents tend to converge; effects like flash crowds[12] or evolution in users' interests and/or behavior require similar evolution in the web proxy's strategies to maintain good QoS.

Consequently the need for dynamic adaptability in web proxy caches seems obvious. Given the possibility of timely reconfiguration a cache can be adapted each time its configuration ceases to be effective, as soon as a new protocol appears or whenever the storage policy no longer matches user behavior, for example.

Even though dynamic flexibility allows better correspondence between cache configuration and the "real world" over time, administrators are still left with the responsibility of detecting misconfigurations and applying appropriate reconfiguration. Since adaptation is essentially determined by two issues, *when to adapt* and *how to react*, this implies watching for some set of events and modifying the cache in response to their occurrence (with a script, an agent, etc.). We argue that, as much as possible, dynamic adaptation should be handled by the cache itself:

- manual configuration tends to be error-prone;
- manual configuration does not scale;
- manual configuration drastically increases the system's reaction time and thus the relevance of the adaptation.

Indeed, not only might reconfiguration scripts require an in-depth knowledge of the cache's internals, thus becoming tricky to implement, but since a reconfiguration comes in response to an event it has to be applied quickly enough: between the occurrence of the event and the end of the reconfiguration the system may change, so that the reconfiguration is no longer suitable. Hence having an administrator monitor several caching systems in parallel and reconfigure them manually would not seem a reasonable option.

Whereas much work has been proposed around caching policies, most of which presents the "most efficient" policy to store documents for some given traffic traces, dynamic flexibility remains an open issue.

Some work has been proposed for flexible web caching, essentially to help writing specialized caching policies or

[1]This work is partially founded by RNRT PHENIX and IST COACH projects.

web caches through dedicated languages. The resulting systems are still static and rigid and cannot be dynamically adapted to anything.

Our response to this issue is a dynamically adaptive web cache, called C/SPAN (C/NN in Symbiosis with PANdora) that applies administration policies to drive its own adaptation based on a dynamically flexible set of monitored events. It is built on a completely open and flexible execution environment [11], called the YNVM (YNVM is Not a Virtual Machine) and consists of a dynamically adaptive web cache, C/NN (The Cache with No Name) and a dynamically flexible monitoring platform, Pandora.

The remainder of this paper starts with a presentation of related work on flexible web caching, in Section 2. Section 3 presents our architecture and its main components. Early results are presented in Section 4 followed by conclusions and perspectives in Section 5.

2 Related Work

Since Web cache complexity is growing nearly as fast as the Web is expanding, it presents extreme problems of configurability and exhibits complex and unpredictable behavior [6]. An increasing amount of work is therefore being done in the area of flexible Web caches.

Reconfiguration is a complex problem in existing systems due to the number of parameters that must be taken into account. (The configuration file for Squid [13] is a case in point.)

One popular solution is to capture "traces" of real activity and then use these traces to drive a simulation of cache behavior [10]. The performance of the simulated cache is analyzed while varying its parameters, in order to find an "optimal" configuration. This configuration is then applied to a live system, but, as stated before, the relevance of the configuration is unlikely to last for long: it will not necessarily correspond well to the continually changing conditions of real traffic. Moreover, the resulting system remains a completely static and closed system.

WebCal [7] and Cache-L [2] use a domain-specific-language (DSL) approach to construct specialized Web caches. Being dedicated to a particular domain, a DSL offers a powerful and concise medium in which to express the constraints associated with the behavior of the cache. Each propose a high-level language for expressing caching policy and the associated compiler. However, in spite of being well-adapted to the specification of new cache behavior and even to formal proofs of its correctness, a DSL-based approach does not support hot reconfiguration: it is still a completely static solution.

Other work [1] has proposed a dynamic cache architecture in which new policies are dynamically loaded in the form of components, using the "strategy" design pattern [3].

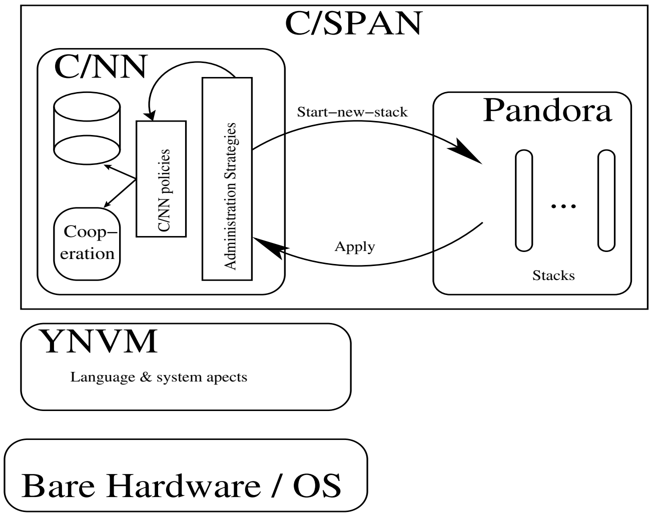

Figure 1. The architecture of C/SPAN.

While increasing extensibility, it is still limited: it is only possible to change those aspects of the cache's behavior that were designed to be adaptable in the original architecture. Moreover, since no administration interface is exported, reconfigurations must be pre-defined into the cache, preventing it from reacting to unforeseen events according to some unanticipated adaptation strategy.

3 Architecture

As illustrated in Figure 1, C/SPAN relies on a dynamic compiler, called the YNVM, that provides dynamic flexibility to applications. It is composed of a flexible Web proxy cache, called C/NN, and a monitoring platform called Pandora.

3.1 C/NN

C/NN is a reconfigurable Web proxy cache that relies on a high degree of reflexivity on its environment and the possibility for low-level intercession in its own execution. C/NN inherits these properties from an execution environment in which they are already intrinsic, the YNVM.

3.1.1 A Dynamically Reconfigurable Execution Environment

The YNVM is a flexible dynamic code generator that provides both a complete, reflexive language and a flexible execution environment. It has been developed in the context of the VVM project [5, 4] to provide dynamic generation of application-specific execution environments.

The YNVM is structured as a set of components and interfaces — such as lexer, parser, compiler, optimizer, code-generator and runtime assembler — implementing a flexible chain of dynamic compilation. By dynamically re-composing or modifying (at runtime) the layers within the YNVM, any dedicated chain of compilation can be constructed.

The default front-end "language" is a lisp-like, textual representation of a syntax tree. The execution model is identical to that of C and the dynamically-compiled native code has the same performance as statically compiled and optimized C programs.

Meta-data are retained during compilation to permit dynamic modification or serialization of application-level code by the application itself.
The main objectives of this environment are:

- to maximize the amount of reflexive access and intercession, at the lowest possible software level, while preserving simplicity and efficiency;
- to use a common language substrate to support multiple languages and programming paradigms.

3.1.2 Flexible Web Caching

Because it is built directly over the YNVM, C/NN inherits its high degree of reflexivity, dynamism and flexibility and so provides "hot" replacement of policies, on-line tuning of the cache and the ability to add arbitrary new caching functionality (performance evaluation, protocol tracing, debugging, and so on) or new proxy services (as in an active network environment) at any time, and to remove them when they are no longer needed.

Our approach supports both initial configuration based on simulation, and dynamic adaptation of that configuration in response to observed changes in real traffic as they happen.

C/NN is structured as a set of basic policies responsible for handling HTTP request/response, caching the documents or cooperating with other Web proxy caches.

The communication aspects are encapsulated in a component-based protocol stack, similar to the X-Kernel [8] framework, as illustrated in Figure 2. Any protocol exports a *low interface* to be used by a lower-level protocol and an *high interface* for upper-level protocols, applications or services. Incoming and outgoing packets are managed through a sessions stack, where a session represent, for a given protocol layer, a peer "local access point / remote access point"'.

Therefore, C/NN is considered as a service that is plugged into a generic proxy above the TCP protocol. This dynamically reconfigurable generic proxy supports dynamic construction of new services, from mobile code execution environment at the ethernet driver level up to TCP-based Web services.

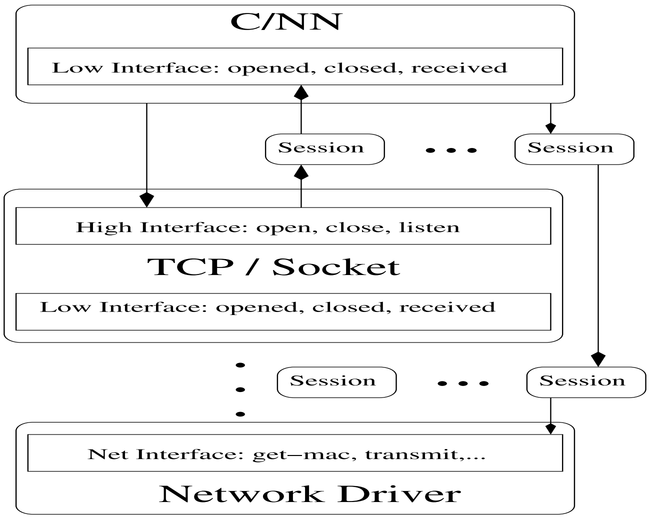

Figure 2. Architecture of the protocols stack.

As a service, C/NN exports three call-back functions to the underlying protocol: *opened* and *closed*, to be notified of connections and disconnections, and *received* to receive incoming packets. C/NN's is structured internally as a set of "well-known" symbols bound to functions implementing the various policies. The reflexivity of the YNVM allow dynamic introspection to retrieve those symbols, as well as modification and recompilation of associated native code.

The policy in charge of HTTP-requests has to classify them into categories (essentially HIT or MISS), possibly taking into account the request rate or the distance to the Web server to avoid caching nearby documents when heavily loaded. The policy dedicated to MISS requests will then have to decide whether or not to ask a neighbor for the document. The HTTP-response and HTTP-storage policies decide if the document should be stored and, if necessary, identifies documents to be evicted from the cache before storing the new documents. Other basic policies control the consistency of document and connection management, to enforce a certain quality of service among clients for example.

Since these policies define the cache behavior, reconfiguring the cache essentially involves replacing them with others or defining new ones. Because of the reflexivity provided by the execution environment, policies can be manipulated as data. Therefore, the administration/reconfiguration of the cache can be expressed with scripts or agents executed inside the YNVM.

3.2 Pandora

Pandora [9] is a general purpose monitoring platform. It offers a high level of flexibility while still achieving good performance. In this section we first present briefly the architecture of Pandora and its main characteristics. Then, we present in greater details the way Pandora interacts with C/NN. Finally, we describe how Pandora can monitor HTTP traffic.

3.2.1 Core Architecture

Each monitoring task executed by Pandora is split into fundamental, self-contained building blocks called *components*. These components are chained into *stacks* that constitute high-level tasks. Stack execution consists of components exchanging messages (data structures called "events") from the beginning of the stack, to the end.

Options are component-specific parameters that modify or specialize component behavior. The range of application options is quite vast: it goes from a simple numerical parameters (a timeout for example) to the address of a function to call in particular conditions.

Pandora provides a framework dealing with (amongst others) event demultiplexing, timers, threads and communication. This allows programmers to concentrate on the precise desired functionality and promotes code reuse. During stack execution, components are created as necessary and the resources they use are collected after some — user-specified — time of inactivity.

Pandora may be configured in two different (complementary) ways. First, at run time, Pandora reads static configuration files either from disk or from the network. Second, if told so, Pandora opens a control socket to which commands can be sent.[2] These commands support querying the current configuration of the platform and performing arbitrary modifications on it. These modifications also affect the stacks being executed. Configuration itself includes stack definitions and component library localization. A stack definition specifies the exact chaining of components while the localization of a component tells Pandora which library to load in order to create the component.

A single Pandora process is able to execute several stacks concurrently. Furthermore, a unique logical stack may be split into several substacks. These substacks may be run either within distinct threads inside the same process or within different processes, possibly on distinct hosts. Pandora provides the required communication components to support such stack connections.

[2] Pandora provides an API with C++, C (hence YNVM) and Guile bindings to ease the construction of clients.

3.2.2 Interactions with C/NN

There are two main techniques used by Pandora and C/NN to interact with each other. The first one involves public (re)configuration interfaces exported by Pandora. The second one makes use of the fact that Pandora and C/NN share a single address space.

Pandora's interfaces allow one to inspect and modify its current state while it is running via three different kinds of methods: *list*, *get* and *set*. Listing methods give the complete set of a given type of element (e.g., running stacks, components in a stack definition, options exported by a component, etc.). The *get* methods give a detailed description of one of those elements (e.g., the type of a component or the current value of an option). Finally, *set* methods may be used to modify any information obtained by one of the previous methods. All these functions apply equally to the static configuration of Pandora and to its dynamic state. Let's consider, for example, the value of an option. Using a *get* method on a (static) definition gives the *default* value of that option for a particular stack. Conversely, when applied to a running stack instance, the same method gives the *current* value of the option (which may have been modified). A last set of functions control the execution of the stacks. These methods allows one to start and stop stacks remotely.

Pandora's core engine is made of a single dynamic shared library. As such it may be loaded directly by the same YNVM instance as C/NN. Doing so enables C/NN and Pandora to share the same address space. Thus, it is possible for Pandora to call any function known to the YNVM, provided that it can find the address of the symbol it is interested in. Respectively, C/NN can use Pandora's API directly (without going through the above mentioned control socket). This enables a potentially unlimited range of direct interactions between the two applications. Synchronization primitives, when needed, are provided by the YNVM.

These two techniques may also be combined. In such hybrid interaction, the address of a symbol is passed as an option to a dedicated component. This allows for maximum flexibility since even the choice of shared addresses may be reconfigured at run-time.

3.2.3 HTTP Monitoring

The main monitoring task dedicated to Pandora in the context of C/SPAN is related to HTTP monitoring. In particular, we are interested in the following metrics:

- request rate: how many requests are made by units of time;
- request latency: how much time it takes to complete a request;

- request size: how many bytes are transferred during a complete transaction with the server (request and response).

Each of these may be considered either globally or clustered in some way (e.g., by server, by network, etc.).

The architecture of Pandora makes it possible to use various data sources and apply identical treatments on the information gathered from them. In order to do so, one only needs to provide appropriate components able to interpret the sources and reformat their content into Pandora's events. In our case, two data sources are available: raw network packets and C/NN itself. Indeed, Pandora is able to reconstruct a full HTTP trace from a passive network capture [9]. This use of passive network monitoring allows us to capture all the metrics we need without any harm to the network and the servers (popular sites are not overloaded with active probes). However, this approach could be considered a little too "heavy-weight" in our context. This is why we also provide a component able to obtain the same kind of information directly from C/NN. This information is then used to build identical HTTP events. The choice of the right approach to follow remains the proxy cache administrator privilege. Further treatments on the collected events are processed by other dedicated components. In particular event clustering is made easy by the generic demultiplexing facilities provided by Pandora.

3.3 C/SPAN

As we were experimenting with live reconfiguration of C/NN, we found that most reconfigurations could be expressed as strategies and thus could be integrated directly as part of the cache. By increasing the autonomy of the cache we also increase its robustness (by avoiding manual reconfigurations), its effectiveness (by decreasing its reaction time) and its ease of administration.

In order to reach this autonomy, with the resulting benefits in terms of reaction time or maintenance, administration policies can be dynamically loaded into the YNVM. Typically such a strategy defines an action (adaptation) to be performed in response to an event.[3] Then, Pandora is used to dynamically instantiate a monitoring stack responsible for notifying the strategy when the desired event occurs so that the corresponding adaptation can proceed.

This approach is highly reflexive because the dynamic management of the cache is expressed in the same language that is used to implement the cache. The cache can be modified at any time. New functionality and policies can be introduced into the running cache, and the "current bindings" modified to activate them. It is therefore possible to rewrite arbitrary parts (or even all) of the cache implementation during execution.

[3] We consider a conjunction of events to be an event.

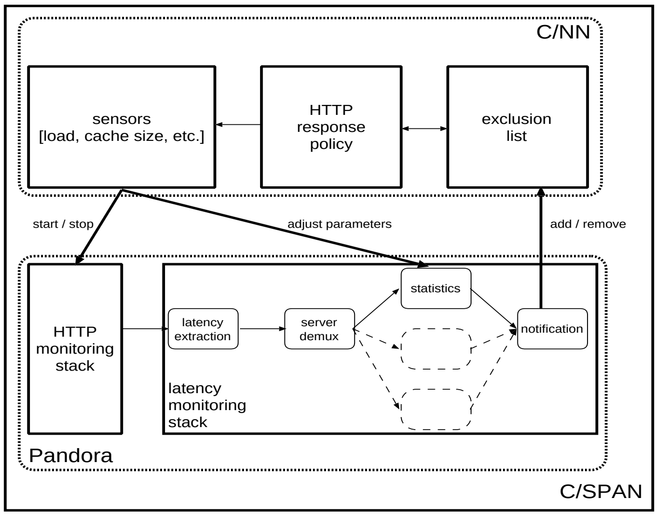

Figure 3. Interactions between C/NN and Pandora for the management of a site exclusion list.

3.3.1 Adaptive Site Exclusion List

Most proxy caches have some kind of a "site exclusion list". All documents from servers or networks in this list are not cached but rather fetched directly. The main purpose of such list is to avoid overloading the proxy cache and filling the disk with documents for which caching would have no benefits. Usually, only servers located in the local network are put in this list. We decided to generalize this concept and automate the management of the list. Indeed, under heavy load, documents located in nearby servers have few reasons to be cached.

The strategy that we follow is represented schematically in Figure 3. When the cache (C/NN in this case) detects that its load is getting high (high request rate or slow response time), it asks Pandora to start monitoring request latencies. Latency in our case is defined by the interval between the time at which the last byte of the request was sent and the time at which the first byte of the response was received. As such, it only includes network latency and server latency (the time it needs to handle the request and prepare the response). In contrast to round-trip time measurements, this does not take into account the time actually needed to transfer the document over the network. These measurements are clustered by server and a representative value for each of them is computed iteratively. In the current implementation, this representative latency (simply called "server latency" for the sake of brevity in the following) is computed as the mean of the 100 last measurements for the server, with the 10 highest and 10 lowest values removed.[4] At the

[4] All these numbers may be changed, even at run-time.

same time, C/NN HTTP-response policy is adapted to use the newly created exclusion list before deciding whether or not to store a document. When some server latency falls below some "low threshold", the corresponding server address is added to the exclusion list. Later on, if it rises again above an "high threshold", it is removed from the list. Initially, these high and low water marks take default values (respectively 100 ms and 200 ms). However, according to the actual load of the cache and the time interval between document evictions (documents are actually removed in batches, until substantial space is freed — typically 15 % of the total cache space — in order to avoid doing the same operations at each requests), these numbers are dynamically adapted by increments of 10 ms. For example, when the request rate goes up by 10 requests per second over a 5 minutes interval, both low and high thresholds are augmented by 10 ms. Similarly, if the average inter-eviction time decreases by 1 minute, the watermarks are decremented. This mechanism corresponds to a best-effort model: it allows the cache to keep response times low while providing an optimal quality of service for its clients. Finally, when the cache load comes back to normal, the whole monitoring process is stopped and standard response policies are restored. All measurements are passive (see Section 3.2). They are taken only as actual requests are made to servers. This ensures that no additional load is imposed on popular servers, and thus does not disturb the accuracy of measurements.

A single Pandora stack is responsible for the latency monitoring process. Its control (start and stop) and the modifications of the various parameters are made through a standard API that we mentioned before. For their part, additions to and removals from the exclusion are triggered by Pandora by direct calls of C/NN procedures.

3.3.2 Adaptive Quality of Service

Another example is the addition of QoS management. Given a metric to classify users, an administration policy can enforce QoS between them in terms of network resource consumption (bandwidth, response time, etc.), as well as local resources (available space in cache).

Dynamically introducing QoS management requires introducing resource allocation/consumption control mechanisms. This is done through the reconfiguration of several functions associated with the resources to be managed, such as the HTTP-storage policy for enabling per-user storage quotas or C/NN's *opened* and *closed* functions to start classifying user sessions according to a prioritized round-robin algorithm. Similarly, reconfiguring the *received* function of the underlying TCP component and the session's *send* function allows control of the bandwidth allocation among all TCP-based applications/services. Similar reconfigurations at a given service's level, such as C/NN, permits the control of bandwidth consumption among users, thus enforcing any given QoS policy.

Since reservation-based QoS protocols are well-known for wasting resources, QoS management should not be active permanently, but only when resource availability is falling and best-effort is not suitable any longer. Therefore we use an adaptation strategy consisting of a set of monitors (one for each "critical" resource) and some adaptation code. When a resource's availability falls below some "low threshold", the corresponding adaptation code replaces the original function managing the resource by a "QoS-aware" one, that will enforce a pre-defined QoS policy for the resource.

Thus dynamically adding new resources into the QoS management is handled either by dynamically reconfiguring the adaptation strategy responsible for QoS management (adding a new monitor and the associated code) or by simply defining a new adaptation strategy dedicated to the QoS management of this new resource.

Moreover, QoS policies can be dynamically adapted either by changing the adaptation code associated with a monitor in the adaptation strategy responsible for the QoS management or by directly reconfiguring/replacing the function controlling the resource.

4 Evaluation

In order to evaluate our prototype, we are interested in three kinds of measurement:

1. Raw performance, as a standard Web proxy cache. Using a flexible execution environment, such as the YNVM, should not prevent the system from having good performance.

2. Performance of the mechanisms supporting dynamic reconfiguration. The cost of dynamically reconfiguring the cache should not be prohibitive.

3. Performance of the mechanisms supporting autonomy.

First we evaluate the potential overhead of introducing dynamic flexibility and autonomy into a Web cache, by comparing C/SPAN to the widely used Squid cache (version 2.3) from an end-user point of view; that is, on the basis of their mean response time. We used several traces collected at INRIA (between 100,000 and 600,000 requests). Whereas Squid's mean response time was slightly more than 1 sec per request, C/SPAN's was about 0.83 sec. Using the YNVM as a specialized execution environment therefore does not introduce any performance penalty, compared to a statically-compiled C program.

Another important metric is the "cost" of dynamic reconfiguration: it must not be intrusive. Since this "cost" is tied to the semantics of the reconfiguration, we evaluate the basic mechanisms supporting the reconfiguration process. Switching from one policy to another pre-defined policy involves only changing a symbol's binding and thus requires only a few micro-seconds. As reconfiguration can imply defining new functionality or policies, its performance also relies on the dynamic compiler's performance. For example, defining (compiling from source) a new storage policy (Greedy-Dual-Size) takes slightly more than 300 μs. This should be compared to the handling of a request, which represents about 130 μs for a HIT and 300 μs for a MISS.

The mechanisms supporting dynamic adaptation are efficient enough that reconfiguration of the cache's policies does not interfere with its activity. Dynamic adaptation of the administration policies, which control the autonomous behavior of the Web proxy, does not rely just on these dynamic loading and compilation mechanisms but also on Pandora's dynamic monitoring stacks.

To achieve autonomy, C/SPAN relies on component-based monitoring stacks. Experiments show that the overhead related to component chaining is limited to 75 ns per component per event, on a 1 GHz Pentium III processor, hence allowing very fast reaction times.

Reconfiguration of the administration policies most often implies the activation of one or more new administration policies. This can be the result of either an external dynamic reconfiguration or an autonomous decision, in reaction to a monitored event. This operation involves two steps: (i) an optional dynamic compilation of the new strategy code (if it has not previously been defined); (ii) the dynamic construction of the required monitoring stacks.

The time to dynamically compile a new policy is typically less than a few hundred microseconds.

We evaluated the creation time for stacks with a variable number of components. Each experiment involved the instantiation of the stack, the creation of every component in it and the destruction of the stack. As expected, Figure 4 shows that this time is linearly proportional to the number of components in the stack. The dotted line in the graph shows the observed times (averaged over 1000 identical runs) and the solid line is the linear regression deduced from these observations. The squared correlation factor for the regression is $r^2 = 0.9992$. The exact equation is thus: $time = 3.55.10^{-3} + 9.40.10^{-5} \times n_{components}$ ($time$ is expressed in seconds). Both stacks (HTTP and ICP[5]) used by C/NN contain less than 10 components. Dynamic stack spawning overhead is thus less than 5 ms. This tells us that stack creation and destruction is a relatively inexpensive operation. Furthermore, this kind of operation is done only occasionally (the period is in the order of hours): the overhead becomes negligible when averaged over the life-cycle of C/SPAN.

[5] Internet Cache Protocol [14].

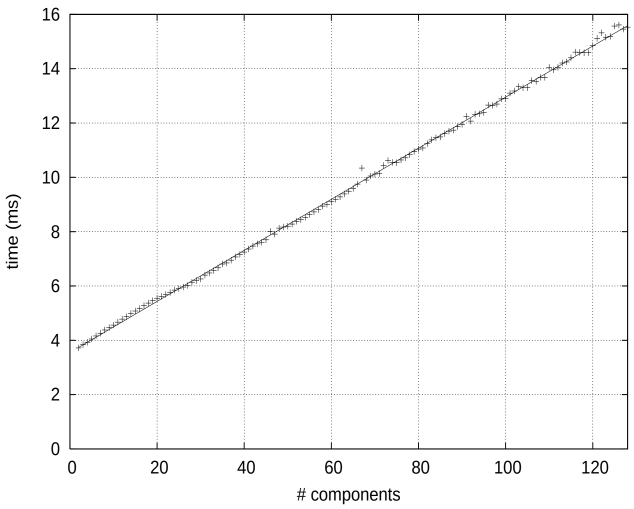

Figure 4. Time to spawn a Pandora stack as a function of the number of components in the stack.

5 Conclusions and Perspectives

This paper described the architecture of C/SPAN, a self-adapting Web proxy cache and some examples of self-adaptation strategies. It is based on a dynamically adaptable Web cache, C/NN, and a dynamically flexible monitoring platform, Pandora. Adaptation strategies are composed of some adaptation-code, in charge of modifying the configuration of the cache or to replace arbitrary functions, and a monitoring stack, responsible for triggering the adaptation-code when a given event occurs.

Early results show that dynamic flexibility can be both complete and efficient, while preserving satisfactory performance. The use of a dynamically adaptable execution environment (the YNVM) for developing a Web proxy cache (C/NN) does not introduce performance penalties compared to a statically compiled C Web proxy cache such as Squid. Furthermore, mechanisms supporting the dynamic reconfiguration of both the YNVM and C/NN are comparable with the handling of an HTTP request thus introducing no significant performance overhead. The autonomy, supported by the interactions between C/NN and Pandora, essentially relies on dynamic monitoring stack creation. Time required to spawn any of the stacks used in C/SPAN was less than a few milliseconds, which is negligible considering that this is still an infrequent operation.

Hence, moving from a dynamically adaptable architecture to a self-adapting one resulted in: (i) a simpler administration interface, with high-level adaptation policies; (ii) better reactivity to continually changing network conditions; (iii) a more robust system, since it avoids manual low-level (re)configuration.

We are currently defining more sophisticated administration strategies based on live analyses of traffic in order to try to anticipate user behavior in terms of request rate, type of documents or favored servers. We are especially interested in adaptive storage policies that try to reflect both user preferences (in terms of sites, type of documents, etc.) and user behavior (types of documents most often reloaded, periods, etc.).

The extraction of user and traffic characteristics may be partially automated in the future by the incorporation of learning algorithms into the administration policies.

References

[1] O. Aubert and A. Beugnard. Towards a fine-grained adaptivity in Web caches. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[2] J. F. Barnes and R. Pandey. CacheL: Language support for customizable caching policies. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[3] E. G. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[4] B. Folliot. The virtual virtual machine project. In *IFIP Symposium on Computer Architecture and High Performance Computing*, Sao Paulo, Brasil, October 2000.

[5] B. Folliot, I. Piumarta, and F. Ricardi. A dynamically configurable, multi-language execution platform. In *SIGOPS European Workshop*, 1998.

[6] S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd, and V. Jacobson. Adaptive web caching: towards a new global caching architecture. *Computer Networks and ISDN Systems*, 30(22-23):2169–2177, November 1998.

[7] G. Muller, L. P. Barreto, A. T. S. Gulwani, D. Gupta, and D. Sanghi. WebCal: A domain-specific language for web caching. In *Proceedings of the 4th International Web Caching Workshop*, 1999.

[8] S. O'Malley and L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, 1992.

[9] S. Patarin and M. Makpangou. Pandora : A flexible network monitoring platform. In *Proceedings of the USENIX 2000 Annual Technical Conference*, San Diego, June 2000. `http://www-sor.inria.fr/publi/PFNMP_usenix2000.html`.

[10] G. Pierre and M. Makpangou. Saperlipopette!: a distributed Web caching systems evaluation tool. In *Proceedings of the 1998 Middleware conference*, pages 389–405, Sept. 1998. `http://www-sor.inria.fr/publi/SDWCSET_middleware98.html`.

[11] I. Piumarta, F. Ogel, and B. Folliot. Ynvm: dynamic compilation in support of software evolution. In *Engineering Complex Object Oriented System for Evolution, OOPSLA*, Tampa Bay, Floride, October 2001.

[12] M. Seltzer. The world wide web: Issues and challenges. presented at IBM Almaden, July 1996. `http://www.eecs.harvard.edu/margo/slides/www.html`.

[13] D. Wessels. The Squid Internet object cache. National Laboratory for Applied Network Research/UCSD, software, 1997. `http://www.squid-cache.org/`.

[14] D. Wessels and K. Claffy. Internet Cache Protocol (ICP), version 2. National Laboratory for Applied Network Research/UCSD, Request for Comments 2186, September 1997. `ftp://ftp.isi.edu/in-notes/rfc2186.txt`.