Monitoring Web Proxy Caches

Simon Patarin, Mesaac Makpangou

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Monitoring Web Proxy Caches

Simon Patarin and Mesaac Makpangou

N° 4023

Octobre 2000

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--4023--FR+ENG

Monitoring Web Proxy Caches

Simon Patarin* and Mesaac Makpangou†

Thème 1 — Réseaux et systèmes
Projet SOR

Rapport de recherche n° 4023 — Octobre 2000 — 21 pages

Abstract: This report presents how Pandora, our flexible monitoring platform, can be used to monitor a system of cooperating proxy caches. It circumvents many of the drawbacks of currently used techniques: Pandora operates on-line, it does not depend on specific cache software, and it can be adapted to any system configuration. We also present two applications that exercise the tool. The first one extracts a Web access trace in presence of cooperating proxy caches purified from the biases they introduce. The second one is an evaluation tool that measures the instantaneous quality of service provided by a system of caches according to customizable compromises. Finally we show effective utilization of these applications, in a real environment for the pure trace extraction and in an artificial experiment for the second.

Key-words: network monitoring, proxy cache, evaluation

* Simon.Patarin@inria.fr
† Mesaac.Makpangou@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Surveillance de caches Web

Résumé : Ce rapport présente comment Pandora, notre plateforme de surveillance réseau flexible, peut être utilisée pour surveiller un système de caches Web coopérants. Ceci contourne grand nombre d'inconvénients présents dans les techniques actuellement utilisées : Pandora travaille en continu, ne dépend de logciel de cache spécifique et peut être adaptée à toute configuration système. Nous présentons également deux applications qui mettent en œuvre cet outil. La première extrait une trace d'accès Web en présence de caches, épurée des biais que ces derniers introduisent. La seconde est un outil d'évaluation qui mesure la qualité de service instantannée fournie par un système de caches selon des compromis personnalisables. Enfin, nous montrons l'utilisation effective de ces applications, dans un environnement réel pour l'extraction de traces épurées, et dans une expérience artificielle pour la seconde.

Mots-clés : surveillance réseau, cache, évaluation

1 Introduction

Web proxy caches are deployed almost everywhere, in organization boundaries as well as at ISPs. However, most existing caching softwares require complex configuration,[1] and the efficiency of a given configuration depends heavily on the actual characteristics of the HTTP traffic going through the cache. Since these characteristics are likely to change as time goes by, it is necessary to continuously monitor the actual performance of Web proxy caches in order to alert the system administrator whenever the current configuration is no longer satisfying.

In order to do so, we need to be able to capture continuously the characteristics of HTTP traffic in organizations possibly hosting several proxy caches. Furthermore, this has to be associated with a system that can evaluate in real-time the performance of a caching system according to user-defined compromises. This raises two issues: first, how to compensate for the effect of caches on the traffic that could be observed depending on their position with respect to the point of observation; second, how to adapt the monitoring and the evaluation process to help system administrators meet their objectives.

Several traffic monitoring tools have been proposed in the past. The most interesting ones include traffic capture software like BLT [5] and Windmill [8]. However, though these tools are able to collect all the needed information, they do not take into account the biases introduced by the caches. With respect to the evaluation issue, trace-driven simulation tools [13, 6, 16] — which are the most widely used — are often too restricted; they are usually designed to evaluate some specific metric, which makes these tools difficult to extend or adapt to other studies or to different trace formats.

In this report, we present how Pandora [10], our flexible monitoring platform, can be used to circumvent the limitations of the existing tools. In particular, we show that it can obtain a pure trace of the Web traffic going through an arbitrary complex system of caches, by combining observations made at well-chosen vantage points in the network. Also, we describe how Pandora can extract pertinent metrics for simulation purposes. Finally, we used Pandora

[1] The latest version of the Squid [14] proxy cache provides 185 parameters, 59 of which are numerical values.

to monitor in real-time the quality of service offered by a system of caches, as specified by a system administrator.

The rest of this report is organized as follows: in Section 2 we present some related work concerning the various techniques and tools developed to monitor and evaluate proxy caches. Then, in Section 3 we describe the architecture of Pandora and in Section 4 how it can be used to monitor proxy caches. Next, we describe in Section 5 several real experiments that use them. Finally we give some concluding remarks in Section 6.

2 Related Work

Our work is related both to monitoring and evaluation techniques. Many other studies have been conducted in one of these two close fields. In this section we will consider them separately; as they were rarely envisioned as a whole.

2.1 Monitoring Techniques

The usual technique used by system administrators to monitor a system of proxy caches is to analyze the log files they produce. Several applications exist that summarize the information they contain [7, 9]. Calamaris [2] is one of the most popular of them. It gives various statistics about the characteristics of the traffic seen by the cache, the behavior of the cache (in terms of hit ratio) and its performance (throughput, load). Yet, even if the results given by these tools are very valuable for statistical purposes, they are usually not suitable for evaluating a system of proxy caches. As noted by Brian Davison [4], logs may be inaccurate, mostly because of stale documents returned by the cache (that are indistinguishable from the others). More problematic is the information missing in logs: much information present in HTTP headers is not recorded, the missing `no-cache` directives and cookies are particularly embarrassing when needing to determine whether a document were cachable, or not. Useful temporal data (like latency, round trip time) are also often absent from logs.

Another approach is to use raw packet capture and traffic reconstruction. Bi-Layer Trace [5] specializes in extracting an Web trace together with lower-level network events (like TCP control packets). More flexible, Windmill [8]

analyzes a wide range of network protocols including HTTP. With such tools, we can retrieve potentially any necessary information (at the cost of minor modifications in the program if something was missing). If the problem with cache logs was mainly the lack of information, here it is its inaccuracy. Indeed, wherever the traffic is observed, it is biased by the cache: if the vantage point is between the client and the cache, temporal data are biased when the request hits a document in the cache, and it is even worse when the document returned is stale. Similarly, if we observe the traffic between the cache and the server, it is impossible to differentiate clients, and every requests satisfied by the cache cannot be observed. In such cases, the only solution proposed so far to get a representative trace would be to put the proxy cache off-line during the traffic monitoring. This is not acceptable either. First, because users' usage profiles may change depending on whether there is a proxy cache or not; but mostly because the volatility of the Web requires traces to be collected frequently, which would prevent entirely the use of the proxy cache.

2.2 Evaluation Techniques

Brian Davison has published a survey of Web cache evaluation techniques [4], which covers most of the studies published about single cache performance (i.e., not considering consistency issues or cache cooperation). Its classification is made along two axes: the trace used, and the simulation environment. The trace can be either artificial, or coming from a captured log, or the actual traffic. In the other direction, the simulation environment can be composed of simulated systems and network, real systems on an isolated network, or real systems and network. There are examples of evaluations conducted with artificial and captured traces in every kind of environment, whereas current traffic can be only used on real systems and networks. What interests us in this study is the wide diversity of approaches: the methodology used and the metrics considered are highly dependent on the intended use of the cache. Therefore it appears that only a flexible evaluation system can satisfy the majority of each individual needs.

Saperlipopette [11] is a lot more concerned with inter-cache communication and consistency. It is an entirely simulated system and network operating on a captured Web trace. After the simulation the different metrics observed (cache

size, bandwidth gain, consistency and perceived latency) are combined into a cost function representing the compromises desired by the administrator. This approach permits the optimization of the various cache parameters for a specific user community (given the trace) and a particular choice of compromise.

3 Pandora

Pandora has been described thoroughly in [10]. It is a flexible and extensible network monitoring platform that can be easily adapted to monitor new Internet protocols or application-specific ones, while still offering good performance. First, we briefly present the architecture of Pandora. Then we focus on the specific features that make it highly flexible.

3.1 Architecture

Pandora is based on passive packet capture. Basic building blocks (that we call *components*) implementing the commonly required analyses[2] are chained inside stacks and Pandora makes packets flow from one end to another.

We also have "control" components. These allow several data flows to coexist in a single stack. This approach reduces resource consumption and avoids unnecessary component traversals. Furthermore, taking flow control mechanisms out of processing components lets component developers concentrate their efforts on the precise functionality they want to implement.

The stacking approach provides several benefits:

Flexibility: One can easily replace one component by an alternative implementation to adapt to a specific situation, or to test new algorithms.

Evolutivity: Protocols are evolving rapidly and independently from each other. The use of components facilitates easy adaptation to these changes.

Extensibility: It is straightforward to add new protocol extraction capability, by simply adding a new component to an existing (lower-level) stack.

[2]For example, there is a component performing IP fragment reassembly, another extracting HTTP requests from a TCP flow, and so on.

Modularity: Each task is encapsulated in its own component which enforces a clear cut between mechanism and policy and eases readability and has proven very valuable for maintaining existing code and debugging. There are no obscure side effects, nor hidden dependencies.

In order to manage packet losses, which can impede the reclamation of unused resources, a general mechanism of timeouts is provided. Indeed, resources are usually freed when a component detects itself that its job is finished (for example, when an IP reassembly component has received all fragments). Yet, we need to handle cases when a packet is lost, or unseen (because it was sent before the collection began) or simply when a component job is finished but it did not notice it (because it was impossible to know or too expensive to test for it). This is achieved using timeouts: when no packets flow through a component during a certain timeout (specified on a per-component basis and dynamically reconfigurable by the component itself), the component is collected and the resources it used released.

User privacy is a serious concern when using passive monitoring techniques. In Pandora, we address using components, which provides flexibility and enforces a level of privacy compatible with the study to be done. For instance, we have implemented an "anonymizer" component that hashes IP addresses using keyed MD5 (RSA Data Security, Inc. MD5 Message Digest Algorithm) [12].

Concerning performance, we have shown in [10] that the overhead related to component chaining is limited to 260 ns per component and per packet, on a 500 MHz DEC Alpha processor. We also achieved complete HTTP extraction at a sustained 80 Mb/s rate, when reading a packet trace from a file.

3.2 Flexibility Support

Pandora provides flexibility at three distinct levels: the configuration of the stacks used, the implementation of the components and the deployment and cooperation of various instances of Pandora. We detail each of them in the following paragraphs.

Stack Configuration
Stack descriptions are given to Pandora at run time, via a configuration file.

This configuration file contains several sections which describe the stacks[3] (components to be used and how to chain them) and the values of run-time options for these components. This allows customization of the behavior of the stack to specific environmental conditions which cannot be known in advance. For example, the value of the timeouts used to collect unused components depends on the type of traffic it treats and the kind of network being used.

Dynamic Component Loading

The configuration technique described above allows the building and customizing of monitoring tasks in a flexible way. By flexibility we mean also that one should be able to do things that were never imagined before. This often implies the design and use of at least one entirely new component. To allow this, Pandora is able to dynamically load shared libraries containing the C++ classes implementing a component. This way, Pandora does not need to know anything about the set of available components at compile time. It scans at run-time a file describing the locations of the components (i.e. which file contains which component). In this file, it is also possible to define dependencies between libraries. This enables another level of flexibility: the component can call functions located in a separate library which can be replaced by a customized implementation if necessary.

Distributed Monitoring

It is not always possible to perform a complete monitoring task on a single host, usually because all the information needed cannot be captured from a single vantage point (for example, when routing paths are asymmetric) or to balance the monitoring load across multiple hosts. In order to be able to perform on-line monitoring in such cases, one needs to combine the observations made by different stacks into a single one that will gather the results and perform further analysis. Pandora's design allows this kind of cooperation. More precisely, it allows linking the last component of a stack to the first component of one another. These stacks may be executed either concurrently by the same instance of Pandora, or by different Pandoras possibly located on distinct hosts

[3] One can specify several stacks to be run by Pandora, in such cases (and unless specified otherwise) all stacks are executed concurrently in independent threads.

(in this case, packets are passed via the network), or both. The only configuration needed to achieve this is to add the right components at the end and the beginning of the stacks. Delegating this at the component level provides much flexibility: the relations between stacks are determined by options which configure them at run-time.

4 Monitoring Web Proxy Caches

Extracting the Web access trace of an organization in the presence of proxy caches, without disconnecting the caches, is essential to enable various evaluation studies. The traces obtained this way are very useful for off-line simulations, or to characterize the traffic observed.

We consider in Section 4.1 the issues concerning the capture of the HTTP traffic trace in the presence of proxy caches. Then, we discuss in Section 4.2 support that could permit a system administrator to customize or to adapt the cache configuration in order to obtain optimal benefits from the caching system, despite the dynamic of the Web or the evolutions of the objectives.

4.1 HTTP Traffic Trace in Presence of Proxy Caches

The Case for Isolated Proxy Caches

To monitor the HTTP traffic flowing through a system of isolated Web proxy caches, one needs to observe traffic both before and after each cache in the system. The comparison of the traffic observed around the caches lets us determine the biases they introduce in the traffic.

Figure 1 shows the typical configuration of Pandora to achieve this for a single cache. Stacks located before the cache (tagged 1 on the figure) collect a trace of the traffic between the clients and the cache (we will call it the *client trace*) and those located after the cache (tagged 2) collect the traffic between the caches and the Web servers (*server trace*). These two traces are sent to the analysis stack (tagged 3) that processes them (and possibly queries the originating servers for any information missing from the traces) and then produces the final trace.

HTTP extraction has been described in [10], and we focus on the analysis stage. It consists of matching HTTP transactions stemming from both traces.

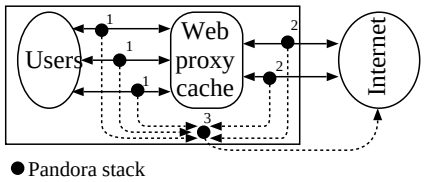


Figure 1: Pandora's stacks deployment for a single cache monitoring.

If a match is found, it means that the document was not present in the cache (since a request has been sent to the Internet) and one just needs to generate a record merging the information collected from both the client and the server trace. If no match is found (after a specified timeout), we assume that a cache made a hit. Then, according to the intended use of the trace, different actions may be taken: (1) do nothing; (2) send an HTTP GET request to the original server for the same document; (3) send an HTTP HEAD request instead. The first solution has the advantage of remaining entirely passive. It is appropriate when all the needed information is already available in the client trace (for example, if we are only interested in capturing the profile of a community in terms of document popularity). The others provide more insight about the cache behavior: by measuring the characteristics of the document transfer, we can determine precisely what influence the proxy cache had in terms of bandwidth and latency gain or loss, and we are able to know whether the delivered document was consistent with the original copy. Sending HEAD requests instead of plain GETs is a compromise: we generate much less additional traffic, but the information obtained is less precise, particularly concerning temporal metrics. When such requests are sent, they are extracted eventually by the HTTP monitors and passed back to the analysis stack where they are matched with the original requests in the client trace. After this step the stack produces a record containing all the information pertinent to the request.

If there are several isolated proxy caches, we deploy the stacks described above around every caches. The output of each analysis stack is passed to an unique "merging" stack, whose only role is to produce an unified trace.

Monitoring Cache Cooperation

When several proxy caches are used within a single organization, they are often configured to cooperate with each other. This means that when a cache misses a document it will first try to fetch it from one of the other caches before sending a request to the origin server on the Internet. Caches are organized hierarchically: all caches at the same level are named *siblings* and may be connected to a higher level *parent* cache. Top level caches are connected directly to the Internet. Cooperation involves an inter-cache protocol to let caches know which documents are available from their siblings. We consider here the Internet Cache Protocol [15], which is one of the most widely used. ICP is a simple transactionnal protocol over UDP.

Monitoring cooperating proxy caches introduces some extra monitoring stages. We need to parse ICP messages and match them (according to their unique identifier) in a single stack to get the metrics we are interested in: the latency of the request, the number of bytes transferred, and of course the status of the request (sibling's hit or miss). Extra care should also be taken in the HTTP analysis stack. Indeed, when retrieving a document from one of its sibling, a proxy cache behaves just like any other standard client. Thus, such sibling hits produce two distinct records: one on the first accessed cache denoting a sibling hit (we see that the request did not go to the server) and one on the sibling cache for a standard hit. In a second phase we merge these two records into one record describing completely this event.

4.2 Customizing QoS Evaluation

Another application for our cache monitoring system is to let system administrators know about the quality of service of their Web proxy cache system in real-time. Since this notion is rather subjective, we choose to let administrators express it via a *compromise function* (much related to the cost function presented in Saperlipopette [11]). This function maps the parameter space into the interval $[-1, 1]$, where -1 indicates the most harmful configuration (which should never be reached), 1 denotes the best achievable quality of service and 0 would correspond to a situation where caches are not present (which should be the actual minimum of the function in real conditions). The parameters available for the function are all those computed by the analysis stack. They

include (but are not limited to) the latency, the number of document bytes transfered, the number of cooperation bytes transfered and the consistency of the document.

There are different ways to specify this compromise function. By default, it is a simple linear combination of all the parameters mentioned above whose weights are configured at run-time via component options. Yet, using the dynamic library loading capabilities of Pandora, an administrator may just write their own implementation of the function and have it used at run-time. This function is defined inside a component in charge of its computation which is given the packets stemming from the analysis stack (containing the parameters). In order to smooth the function variations, instantaneous evaluations (performed each time a packet is received) are averaged on a time basis; the time aggregation factor is chosen at run-time via a component option. Then, these values are averaged again over a parameterizable-width window. These two steps permit values covering large time scales (the window's width) to be updated more frequently (the time aggregation factor).

The compromise function we use by default is of the following form:

$$cost = \frac{w_l \left(\frac{l_s}{l_t}\right) + w_r \left(\frac{r_s}{r_t}\right) + w_n \left(\frac{n_s}{n_t}\right) + w_b \left(\frac{b_s}{w_c b_c + b_t}\right)}{w_l + w_r + w_n + w_b}$$

where:

w_l = latency weight	n_t = #documents
l_s = saved latency	n_s = #consistent documents
l_t = total latency	w_b = bytes weight
w_r = RTT weight	b_s = saved bytes
r_s = saved RTT	b_t = total bytes
r_t = total RTT	w_c = cooperation bytes weight
w_n = consistency weight	b_c = bytes used for cooperation

We call *saved* the values representing gains related to cache activity. These values may be negative if the cache behavior actually degrades performance.

The values produced by the compromise evaluation component may be displayed as they are computed (e.g. for plotting) or be fed into a mechanism of alarms which are triggered each time these values, either absolutely or relatively, exceed specified thresholds. Once again, any other exploitation of the results may be considered: it only requires to implement the right component.

5 Experiments

In this section, we show real examples of Pandora's uses as presented above. The first experiment was conducted at INRIA[4] Rocquencourt during the month of July 1999, to extract a pure Web access trace in presence of proxy caches. The second one is a synthetic test run on a LAN to demonstrate the capabilities of our tool to monitor the quality of service of a system of cooperating proxy caches.

5.1 Pure Trace Extraction

INRIA Rocquencourt is connected to the Internet through an unique border router which sees all outcoming and incoming traffic. For Web access, users have the choice to use a proxy cache or not. To reconstruct a pure trace, we deployed several cooperative Pandoras as shown in Figure 2.

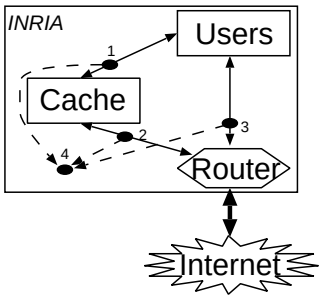


Figure 2: INRIA Rocquencourt network configuration.

The format of the trace records allowed us to extract traces captured at each vantage point, of which we obtained four: one with requests captured *before* the cache (position 1 on the figure), one with those seen *after* the cache (position 2), one with those not going through the cache (position 3) and finally our reconstructed *pure* trace (pseudo-position 4). For the sake of representativity, we choose to use only a one day long trace. This day corresponds to a usual working day at INRIA. The numbers of requests with HTTP 200 responses

[4]Institut National de Recherche en Informatique et en Automatique.

(that we filtered out) and unique servers in these traces are presented Table 1.

Trace	Requests	Servers
before	107074	3531
after	96573	3203
avoided	216317	4990
pure	323391	5301

Table 1: Number of requests with HTTP 200 responses and number of unique servers accessed in the four traces used.

The first thing to notice is that the cache is not very efficient, with a hit rate inferior to 10%, and underutilized: almost $^2/_3$ of the total amount of requests bypass it. We also see that a single observation on any vantage point would have missed a large proportion of the traffic.

To further investigate the intrinsic characteristics of the traces, we computed the cumulative distribution of request latencies[5] for each of the four traces. Results are shown in Figure 3. The distribution is almost identical for the two traces (*avoided* and *after*) of requests directly sent to the Internet, and this distribution is well reproduced in the purified trace. The trace of requests sent to the proxy cache is quite different: observed latencies are much longer. This is due to the cache processing overhead whose effect is no longer noticeable when latencies exceed 2 seconds. Another overhead source is related to name resolution: when the first packet is seen by Pandora name resolution has already occurred and a simple HTTP monitoring cannot tell actually how long ago the user typed the new URL.[6] In the case of requests made through a cache, the resolution of cache's name is immediate and it is up to the cache to make the actual resolution of the server's name, and this delay is fully included in the observed latency. Finally, the low hit rate (the cache needs to contact origin servers rather frequently) and the good network connectivity of INRIA (that makes average latency low itself) amplify all of these phenomena.

[5]By latency, we mean the time elapsed between the last packet from the request (which is usually also the first) and the first packet from the response.

[6]Pandora is now able to monitor DNS traffic and, from there, determine this additional latency, but this was not implemented when we collected this trace.

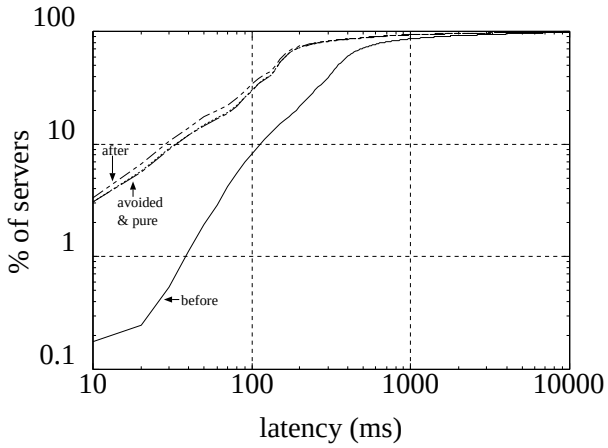


Figure 3: Cumulative distribution of average access latencies to servers (logarithmic scale on x and y axis).

Beyond these facts, Pandora proved that it was able to collect useful metrics usually not available in log files and that it could efficiently reconstruct a Web access trace in the presence of a proxy cache. This example also shows that the only log file available in this environment (proxy cache) would have given biased information.

5.2 Customizing QoS Evaluation

In a second experiment, we would have liked to evaluate the quality of service provided by a system of proxy caches. Yet, we did not have the opportunity to perform this test in real conditions. Thus, we set up on our LAN the system shown on Figure 4. The four cooperating caches run Squid [14] on distinct machines. The clients are simulated by processes with multiple threads; each process is dedicated to a single cache and its threads tend to simulate the behavior of normal users: they alternate requests and inactivity in variable time periods, making the generated traffic rather "bursty". The server holds 20,000 files of variable size (whose distribution mimics the one observed on the Web [3, 1]). Given the small number of distinct files, default cache settings rapidly lead to almost 100% hit rate. Thus, we drastically reduced storage

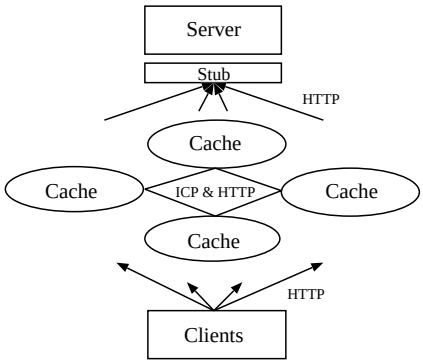


Figure 4: Experiment set-up for QoS evaluation.

size so that hit rate increased more slowly. Besides, caches are cleaned up before each run, which ensures that they start with the same known state. Also, we added a stub on the server that allowed us to artificially increase network latency (so that retrieving a document for a cooperating cache rather than from the server is somehow interesting). The stub used is a simple port redirector that uses a `select` loop and non-blocking sockets: at each execution of the loop, all pending data are treated, then the process sleeps for a variable amount of time.

An instance of Pandora runs on each cache and is configured as shown in Figure 5. The three stacks tagged 1 on the figure perform HTTP extraction[7]

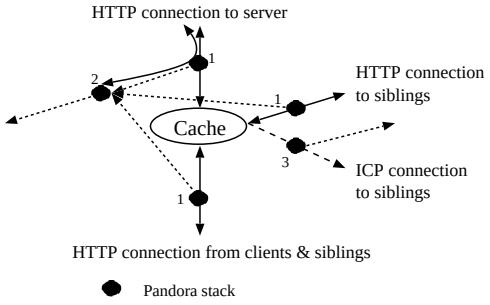


Figure 5: Pandora configuration around a cache.

and forward their results to the analysis stack, tagged 2. Another stack (tagged 3) monitors outgoing ICP packets. These last two stacks pass their records to a global common stack (not represented on the figure) which will match ICP requests and analyze sibling hits. Instead of directly computing the compromise function in this stack (which should be done for on-line monitoring), we choose to log these packets into a file so that we could compare functions expressing different compromises on the same data.

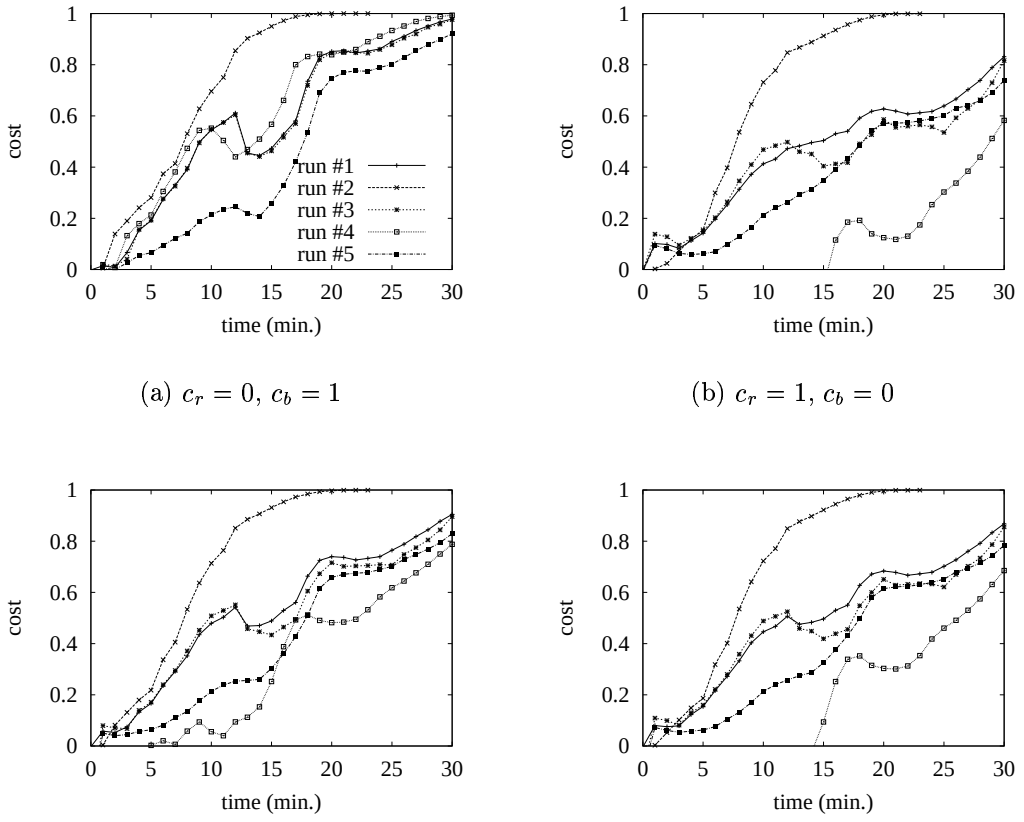
Before presenting further details of our experiments, we would like to emphasize the fact that they do not aim to simulate a real environment. These are "fake" clients producing synthetic workloads in a network with artificial latencies. Our unique goal is to illustrate Pandora's capabilities. In the same way, it appears that, despite our efforts to reduce cache size, hit rates remain still very high in every configuration (it tends even towards 100% before the end of runs). This is due to the necessarily limited number of files that can be requested from a single Web server. However, we don't think it is a real drawback, since it allows us to explore a wide range of cache behaviors in a short time. On the other hand, it was impracticable to deal with document consistency. Run duration is rather short, and Squid's consistency policies relied on well-known average age constants.

We present in Table 2 the characteristics of the different tests we ran. The different parameters we took into consideration are: the number of clients, the artificial latency value, the additional load we put on the Web server and whether the caches were cooperating or not.

Run	#clients	load	latency	cooperation
1	4×10	$\surd$	$\surd$	$\surd$
2	4×20	$\surd$	$\surd$	$\surd$
3	4×10	$-$	$\surd$	$\surd$
4	4×10	$\surd$	$-$	$\surd$
5	4×10	$\surd$	$\surd$	$-$

Table 2: Characteristics of the test runs.

[7] One must notice that incoming siblings' HTTP connections are mixed with those stemming from standard clients.



(a) $c_r = 0$, $c_b = 1$

(b) $c_r = 1$, $c_b = 0$

(c) $c_r = 0.5$, $c_b = 0.5$

(d) $c_r = 0.75$, $c_b = 0.25$

Figure 6: Evaluation of the five configurations with different compromise functions (the weights used are indicated below each graph).

Figure 6 shows the results of these evaluations for different compromises. We used an aggregation factor of 1 minute and a window of 5 minutes. For the sake of simplicity we used only two dimensions to evaluate the system: round-trip time and the amount of bytes transfered. In the top two figures (6(a) and 6(b)) each of these dimensions is used alone, and in the bottom two figures, they are combined with different weights. We notice a performance drop in most runs between 10 and 15 minutes. Closer examination of the logs shows that it is due to cache misses of several large files occuring at this time which are not balanced by enough hits on other smaller files. The short time aggregation factor we used makes this much more prominent than it would be in realistic conditions.

Analysis of the graphs allows to characterize the influence of the various parameters we used and leads to expected conclusions. Indeed, increasing the number of clients achieves much better performance: caches are filled up faster and thus exhibit higher hit rates sooner. Increasing latency between caches and the server does not influence the achieved hit rate. Naturally, suppressing it has a dramatic effect concerning round-trip time savings. We can also see that cooperation is more effective at reducing the number of bytes transferred than round-trip times. Of course we notice that the impact of cooperation decreases as local hit rate increases. Besides, server load seems to have very little impact, meaning that the server was never really stressed. This was therefore not a limiting factor.

What we would like to retain from these tests, more than the results in themselves that are quite straightforward, is that Pandora's evaluation lets us analyze precisely the different configurations we tried. The accuracy it shows leads us to think that it could be used efficiently to evaluate real systems.

6 Conclusion and Future Work

We showed how Pandora can be used to monitor a system of cooperating Web proxy caches. In particular, we showed how it could extract a pure Web access trace in presence of caches and evaluate in real-time a specific configuration according to user-defined compromises. We illustrated these issues with two examples that assess Pandora's usability in a real environment.

Nevertheless, we would like to continue to improve Pandora at various levels. Seeking still more flexibility, and especially adaptability, we plan to develop Pandora's reflexiveness. That is, being able to introspect the stacks, components and options used by our tool and to change them while Pandora is running. This way we could add new monitoring tasks as they become necessary, insert a component in a running stack to refine analysis or adapt component behavior to reflect changes in the environment. We also need to facilitate the deployment and the coordination of different instances of Pandora used in distributed monitoring. This means being able to control them from a single location and set up fault tolerance and error recovery mechanisms.

In a longer term, we are interested in building a system where applications will be able to contract guaranteed "deals" in spite of the dynamics of the Internet. These "deals" can be seen as constraints on the quality of the service that an application requires for its users. In this context, Pandora will help evaluating in real-time the achieved quality of service and will require system reconfiguration when "deals" cannot be guaranteed any longer.

Availability

The source code of Pandora is publicly available under the GPL license in a beta, undocumented version. Documentation is planned to be written in the near future. See `http://www-sor.inria.fr/projects/relais/pandora/` for release information.

References

[1] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the SIGMETRICS '98 conference*, June 1998. `http://cs-www.bu.edu/faculty/crovella/paper-archive/sigm98-surge.ps`.

[2] Cord Beermann. Calamaris. software. `http://Calamaris.Cord.de/`.

[3] Carlos R. Cunha, Azer Bestavros, and Mark E. Crovella. Characteristics of WWW client-based traces. Technical Report BU-CS-95-010, Computer Science Department, Boston University, 111 Cummington St, Boston, MA 02215, July 1995. `http://www.cs.bu.edu/techreports/95-010-www-client-traces.ps.Z`.

[4] Brian D. Davison. Web traffic logs: An imperfect resource for evaluation. In *Proceedings of th INET'99 Conference*, June 1999. http://www.isoc.org/inet99/proceedings/4n/4n_1.htm.

[5] Anja Feldmann. BLT: Bi-Layer Tracing of HTTP and TCP/IP. In 9^{th} *International World Wide Web Conference*, Amsterdam, The Netherlands, May 2000. http://www.www9.org/w9cdrom/367/367.html.

[6] Syam Gadde, Jeff Chase, and Michael Rabinovich. A taste of crispy Squid. In *Proceedings of the Workshop on Internet Server Performance (WISP'98)*, June 1998. http://www.cs.duke.edu/ari/cisi/crisp/crisp-wisp.ps.gz.

[7] Maciej Koziński. Squeezer. software. http://www.geocities.com/maciej_kozinski/w3cache/squeezer.html.

[8] G. Robert Malan and Farnam Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings of ACM SIGCOMM '98*, Vancouver, British Columbia, September 1998. http://www.eecs.umich.edu/~rmalan/publications/mjSigcomm98.ps.gz.

[9] Mark Nottingham. Weblog. software. http://www.mnot.net/scripting/python/WebLog/.

[10] Simon Patarin and Mesaac 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.

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

[12] Ron Rivest. The MD5 message-digest algorithm. Request for Comments 1321, April 1992. ftp://ftp.isi.edu/in-notes/rfc1321.txt.

[13] Junho Shim, Peter Scheuermann, and Radek Vingralek. Proxy cache design: Algorithms, implementation and performance. *IEEE Transactions on Knowledge and Data Engineering*, 1999. http://www.ece.nwu.edu/~shimjh/publication/tkde98.ps.

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

[15] Duane 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.

[16] Roland P. Wooster and Marc Abrams. Proxy caching that estimate page load delays. In *Proceedings of the 6rd International WWW Conference*, April 1997. http://www.scope.gmd.de/info/www6/technical/paper250/paper250.html.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38330 Montbonnot-St-Martin (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

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