Pandora: A Flexible Network Monitoring Platform

Simon Patarin, Mesaac Makpangou

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Pandora: A Flexible Network Monitoring Platform

Simon Patarin and Mesaac Makpangou

No 3834

Décembre 1999

THÈME 1

ISSN 0249-6399 ISRN INRIA/RR--3834--FR+ENG

INRIA
ROCQUENCOURT

Pandora: A Flexible Network Monitoring Platform

Simon Patarin* and Mesaac Makpangou†

Thème 1 — Réseaux et systèmes
Projet SOR

Rapport de recherche n ° 3834 — Décembre 1999 — 27 pages

Abstract: This report presents Pandora, a network monitoring platform that sniffs packets using purely passive techniques. Pandora addresses current needs for improving Internet middle-ware and infrastructure by providing both in-depth understanding of network usage and metrics to compare existing protocols. Pandora is flexible and easy to use and deploy. The elementary monitoring tasks are encapsulated in independent components we call filters. The actual packet analysis is performed by stacking the appropriate filters. Pandora also preserves user privacy by letting them control the "anonymization" policy to be enforced. Finally, the evaluation we conducted shows that overheads due to Pandora's flexibility do not significantly affect performance. Pandora is fully functionnal and has already been used to collect Web traffic traces at INRIA Rocquencourt.

Key-words: network monitoring, passive capture, flexibility, components

(Résumé : tsvp)

* 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 : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Pandora : une plateforme de surveillance réseau flexible

Résumé : Ce rapport présente Pandora, une plateforme de surveillance réseau qui utilise des techniques exclusivement passive de capture de paquets. Pandora répond aux besoins actuels d'amélioration des middlewares et de l'infrastructure de l'Internet en fournissant à la fois une compréhension poussée de l'usage du réseau, ainsi que des métriques pour comparer les protocoles existants. Pandora est flexible, facile à utiliser et à déployer. Les tâches de surveillance élémentaires sont encapsulées au sein de composants indépendants que nous appelons filtres. L'analyse des paquets à proprement parler s'effectue en empilant les filtres adéquats. Pandora préserve également la vie privée des utilisateurs en les laissant contrôler la politique d'«anonymisation» à assurer. Enfin, l'évaluation que nous avons menée montre que les surcoûts liés à la flexibilité de Pandora n'affectent pas significativement ses performances. Pandora est d'ores et déjà opérationnelle et a été utilisée pour collecter des traces du trafic Web à l'INRIA Rocquencourt.

Mots-clé : surveillance réseau, capture passive, flexibilité, composants

1 Introduction

Network monitoring is essential for the improvement of the Internet performance. It could serve to capture usage profile, to evaluate the impact of Internet services (e.g. caching, replication and cooperative caching), to compare the overheads of different implementations, or to help debug complex distributed applications.

In the recent past years, several network monitoring tools have been proposed. These include naturally `tcpdump` [11], and specialized softwares like BLT [3] or HttpFilt [19] for HTTP extraction, and `mmdump` [2] for multimedia monitoring. One must also mention more generic platforms like IPSE [7] or Windmill [12].

As the Internet grows, the demand for a flexible monitoring system increases. Such a system could permit to ensure good performance while keeping pace with the rapid evolution of Internet protocols and services by enabling rapid implementation of dedicated monitoring tools.

Unfortunately, existing network monitoring tools are too specific. They are designed to collect information required for some particular analysis. They often depend on a particular version of a protocol, or a particular configuration of the underlying infrastructure. For instance, HttpFilt only works for HTTP/1.0. IPSE assumes the existence of a gateway where one could observe the entire traffic within the monitored organization. Such dependences make most existing tools out-of-date if the targeted protocol evolves or the assumed configuration changes. Also, it makes them difficult to adapt to cope with new problems.

This report presents Pandora,[1] a flexible and extensible network monitoring platform that could be easily adapted to monitor new Internet protocols or application-specific ones, while still offering good performance. Pandora uses passive network monitoring to reconstruct high-level protocols while keeping track of lower levels events. It provides basic building blocks implementing the commonly required analysis.

The rest of the report is organized as follows: Section 2 presents design goals Pandora should meet. Section 3 describes the architecture of the system,

[1] Our platform is called Pandora to recall the potential dangers of overly intrusive curiosity.

while Section 4 describes how it is realized. Then, Section 5 focuses on the implementation. Next, we consider two examples of use of Pandora in Section 6; then we discuss the performance of the system in Section 7. Finally, we compare Pandora to related work in Section 8 and present some concluding remarks in Section 9.

2 Design Goals

Our design has four main goals: monitoring a system should not affect the system's behavior; the tool should be fast enough to monitor high-bandwidth links; user privacy must be preserved; and the tool must be flexible and simple enough to allow reuse in diverse applications.

2.1 Preserving the Quality of Service

A monitoring tool should perform its work without being noticed by the system or its users. In particular, it should not introduce artificial perturbations in the environment. It should also not degrade the quality of service provided to the users of the system.

Although easier to realize, we decided not to implement the tool as an active element of the system: a proxy, for example, could be used to intercept the traffic and to log information. However, such an active tool necessarily has an impact on the system's performance; in the case of a failure, for example, the monitored service could be interrupted.

Therefore, we decided to use passive network monitoring. Such a tool captures the packets on the network and treats them without interfering with the actual traffic. This choice implies in particular that we must be able to deal with packet losses, without the opportunity to request the sender for packet re-transmission.

2.2 Performance Issues

A monitoring tool can be used to trigger a fast reaction to certain events. For example, one could decide to modify a Web cache's configuration because of a decrease in the observed quality of service. Therefore, the monitoring tool

must treat all information on-line, preventing it from storing intermediate data destinated for off-line analysis.

Our system must be able to monitor high-speed links, such as a 100 Mb/s Ethernet and a T3 wide-area link. Therefore, the design of the tool must be light enough to accommodate such a traffic. Moreover, it should be able to deal with load peaks (which can frequent in networking systems).

2.3 Preserving User Privacy

A serious concern when monitoring a system is to preserve user privacy. Therefore, we should not output personal information such as which Web pages a particular user has accessed. Yet, to have enough insight into the system's behavior to provide interesting results, we often need precise information about user behavior.

We consider that there must be a tradeoff between user privacy and the level of details that are provided by the monitoring tool. Depending on the planned use of the collected information, different levels of trace "anonymization" must be used.

Therefore we consider that privacy should be treated as a policy (hence flexible) and that the system should only enforce a level of privacy compatible with the study to be done.

2.4 Ease of Use and Deployment

We anticipate that the monitoring system will be used for various purposes: gathering traffic traces for several Internet protocols (e.g. HTTP, FTP or ICP), comparing network overheads for different protocols stacks, or debugging distributed applications. This list of possible uses of the system is of course not exhaustive.

An administrator should be able to easily use the same monitoring tool for numerous different purposes. Therefore, the system must provide useful default options while being easily customizable. Moreover, it should not require specific hardware, and it should be portable across several, widely used, platforms. Finally, new protocols arise every day that people might want to

analyze. It should therefore be easy to add new protocol-specific elements to the system.

3 Architecture

Pandora is designed as a filter stack. Each filter encapsulates an elementary monitoring task (e.g. IP layer reassembly, TCP layer reordering, etc.). We first illustrate the problem and its solution with the example of HTTP monitoring; then, we describe the general design of the platform.

3.1 Example of HTTP Extraction

This example consists of gathering the Web traffic generated by a given user community. Such traces can lead to a better understanding of traffic patterns, hence to the design of better protocols and tools.

The protocol we need to monitor here is HTTP [6]. HTTP allows a client to send a request to a server, which in turn sends back the corresponding document. Concretely, we want here to keep track of all meta-data that are present in either the requests and the responses.

The `libpcap` library is used to capture any packet which passes through the network. It provides raw network packets, i.e. arrays of bytes. To extract the useful information from such packets, we need to reconstruct the HTTP sessions. First, we remove the link layer headers (the Ethernet headers, for example). This provides an IP packet. IP packets can be fragmented when traveling through the networks. Therefore, we need to reassemble the fragments (based on the IP headers). Once the reassembling is done, we can extract a TCP packet. Based on the TCP headers, we can determine which TCP stream it is part of, its proper place in the stream, etc. After demultiplexing and ordering TCP packets, we obtain the HTTP stream. Then, we have to parse the HTTP header fields in order to obtain the HTTP meta-data. Finally, the request meta-data must be matched with the corresponding response meta-data before being output.

3.2 Basic Components

As one may notice from the above example, a few elementary tasks appear, namely: IP reassembly, TCP reordering and HTTP request/response matching. The only dependence between those tasks is the order in which they are performed. These tasks operate on packet *flows* rather than on individual packets. A packet flow is a sequence of packets related to the same higher level entity (an IP packet for IP fragments, or a connection for TCP packets for example) exhibiting *temporal locality*.

The notion of temporal proximity depends on the nature of the flow itself, and can be seen as a threshold beyond which the flow is considered to be closed (a similar but more restrictive definition is given in [5]). In our example, a packet flow could be the set of all fragments — including duplicated ones — forming a single IP packet or the set of all TCP packets sent from a browser to a Web server, containing HTTP requests.

To capture the separated tasks that must be performed to produce the trace, and the order in which they must be proceeded, we define two basic notions: filters and filter stacks. A filter is responsible for a specific elementary task, while the stack is the structure where these filters are chained together.

A filter may be considered as an operator on packet flows: it takes some flow as input, performs its work on it and then produces a flow of a different nature as output. Thus a filter designed to perform IP reassembly transforms a flow of IP fragments into a flow of IP packets.

A filter relies only on the properties of its input. In particular, it does not have to know about the other filters in its stack. This permits to replace a filter by an equivalent one, or to introduce new filters in a stack with no effect on the rest of the stack. For example, imagine we want to encipher the data collected (perfectly legitimate issue); we have only to add a filter that will encrypt IP addresses and URLs, after the HTTP protocol has been parsed.

Many different monitoring experiments can be conducted without effort, if all the required filters exist, or at minimal cost concerning only the development of the missing ones. Using filters makes it easier to debug the monitoring path; effort can be concentrated on the optimization of filters that constitute bottlenecks.

The stack determines how filters are connected. It represents an ordered set of filters through which packets flow from one end to the other. One may notice that this structure implies that each filter has exactly one input and one output.

3.3 Generalized Stacks

In our example we stated that IP packets could be fragmented, yet not all IP packets are. Why, then, should we make them all pass through the reassembly filter?

Moreover, we have to demultiplex packet flows at several levels (namely IP, TCP and HTTP). This demultiplexing stage will have to be performed in *each* of these filters, since having only a single output is equivalent to re-multiplexing everything before passing packets to the next filter. This seems like a waste of CPU time (and complicates the design of a single filter) since the TCP reordering and the TCP to HTTP filters — for example — need the same level of demultiplexing.

In order to address these two issues while preserving the stack structure, we extend the previous simple model by introducing two kinds of pseudo-filters:

Switch filter This filter permits packets to follow along different processing paths: dynamically configured with a fixed number of alternative paths, it can forward a packet to any of them. In our case-study, it may be used with the IP reassembly component: routing IP fragments to the reassembly sub-stack but routing complete packets directly to TCP packet extraction filter.

Demux filter Such filter identifies the flow that each packet belongs to. Then, for each flow, it dynamically instantiates a dedicated sub-stack. It then forwards the packets to their relevant stack. Once a session is finished (e.g. a TCP connection is closed), the sub-stack is removed. This allows a clear separation between mechanism from policy, and simplifies the design of other filters.

Figure 1 shows one solution for the Web trace collection example that uses switch and demux filters. The "connection demux" receives TCP packets

from the IP to TCP extraction filter. It identifies which flow each packet belongs to. For each flow, it dynamically creates a "direction demux" which in turn identifies in which direction of the flow each packet is going (from the client to the server, or the other way round). TCP packets are then passed to filters performing TCP reordering and HTTP reconstruction. Finally, the two opposite HTTP streams (i.e. a stream of requests and the corresponding stream of responses) are given to the HTTP matching filter, which determines which request corresponds to which response.

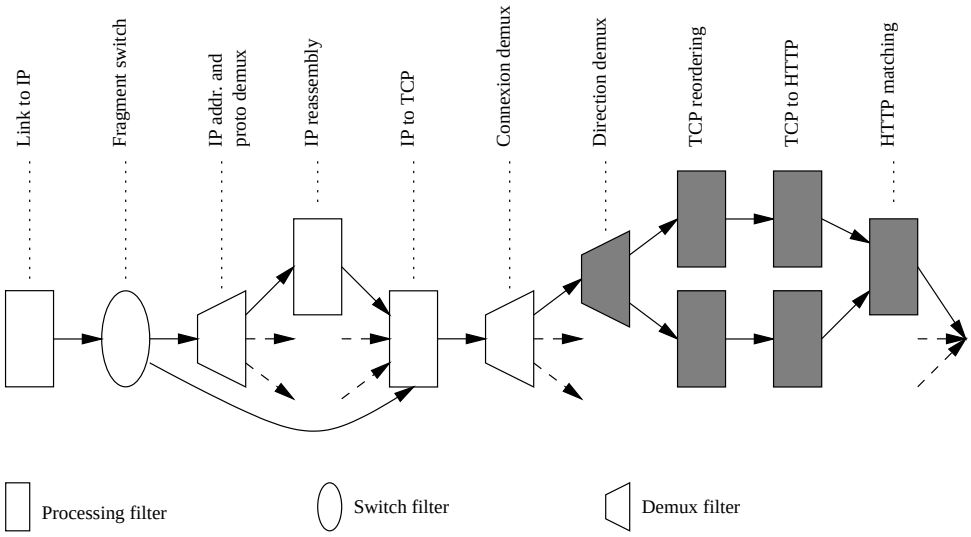

Figure 1: The filter stack used in the Web traffic collection example. The shaded filters are those dynamically created by the "connection demux" when a new TCP flow is identified.

3.4 Stack Configuration

We designed a small language which allows specifying a particular stack. Its grammar is presented in Figure 2.

```
stack  ::= filter+

filter ::= simple | demux | switch

simple ::= fname '-'?

demux  ::= fname '<' stack '>'

switch ::= fname '(' stack ('|' stack)* ')'

fname   ::= [a-zA-Z]+
```

Figure 2: Stack definition grammar.

A stack is composed of a number of filters. The simple filters are identified only with their name.[2] The switch and demux filters are described by their name as well as the definition of their upper stacks. The demux is given the definition of the sub-stack to instantiate when identifying a new session; the switch is given the list of sub-stacks it can forward packets to.

Other parts of the configuration files allow to setup parameters of each filter in the definition. For example, Figure 3 presents an example of a configuration file specifying two output files.

4 Stack Instantiation

Once the stack and filter options have been parsed, one has to instantiate the stack. This involves dynamically constructing the sequence of filters, linking them appropriately. The difficulty stems from the fact that none of the filters knows the topology of the processing graph. To address this problem we need an independent entity which holds this graph and the configuration parameters to be passed to different instances of filters. We call this entity the *dispatcher*.

[2]The "-" connector is optional and is used only for clarity.

```
[stack]
fragswitch (
  ipfragdemux <
    ipreass
  > output
) output

[ipfrag]
timeout = 60

[output(0)]
static file = ip_frags.dat

[output(1)]
static file = ip_pkts.dat
```

Figure 3: Configuration file for a stack that reassembles fragmented packets and logs fragmented and unfragmented packets in separate files.

4.1 Filter Creation

The dispatcher is responsible for creating filters — which includes properly setting any dynamic options it might need, and constructing the ordered sequence of filters specified in the stack description.

Each filter holds a pointer to its successor in the processing path. This reference is set to *nil* when the filter is created. When a filter wants to pass on its resulting packet, it uses this reference, unless it is uninitialized. In the latter case, the filter asks the dispatcher to instantiate the appropriate successor.[3]

The dispatcher does not keep track of previously created filters. However, by always creating new instances of filters, it is impossible to ever multiplex outputs as is necessary for the demux and switch filters.

To instantiate a generalized stack, we note that there is a one-to-one mapping between the demultiplexing and the multiplexing, and that demultiplexing always occurs before multiplexing in the graph. It is therefore sufficient to create the multiplexing filter along with the demultiplexing one. This works

[3]It is still a question to decide whether it is best suitable to initialize this pointer at the time the filter is created or at the time of its first use.

as follows. A demux or a switch filter holds a pointer to its corresponding *mux* filter, initialized at its creation time. Then, each filter in a demultiplexed branch carries a reference back to its demultiplexing filter; this reference is passed dynamically when the filter is created.

When the last filter in a branch (before multiplexing occurs) asks the dispatcher to instantiate its successor, the dispatcher retrieves the reference to the multiplexing filter from the demultiplexing one, by following a back-pointer to the branching point that is held by each element of a branch.

4.2 Memory Management

Filters that are no longer needed should be destroyed. As with their creation, destruction of filters is performed incrementally. The process starts with a filter notifying the dispatcher of its intention to destroy its successor (or one of them, if this is a demultiplexing filter). This notification is then propagated by the dispatcher. It first asks the target filter to prepare for destruction. Upon completion of this request, the dispatcher effectively destroys the filter (and resets the caller's reference). The process iterates up to the filter at which the initiator's branch is multiplexed, where it stops.

For a multiplexing filter, destruction preparation involves recursively initiating a new destruction process for each of its branches. Other filters simply flush all their unprocessed packets.

We have two ways to trigger filter collection: active and passive. Active collection occurs when a filter decides itself that its processing is finished. It then notifies its predecessor[4] that it can start the destruction process. This is the case when an IP reassembly filter receives the last fragment of an IP packet. Passive collection involves timers. Pandora has indeed an independent thread, sleeping for fixed intervals of time. When it wakes up, it calls some specific method in any filters that previously requested it. For example, we have implemented a filter that triggers the destruction of its successors if it does not receive packets during a certain period. This is also the typical behavior of a demux filter.

It must be noted that the choice of timeout value is the result of a delicate tradeoff: too short a timeout may interrupt ongoing connections, while large

[4]By way of the return value of the function used to pass a packet to a successor filter.

timeouts might dramatically increase the memory footprint of the application. There is no canonical value: it depends on the nature of the observed traffic and of the type of link being monitored.

4.3 Concurrent Processing

Pandora allows filters to be executed concurrently insofar as the logical flow of packets is respected. More precisely, it offers the possibility of running some parts of a stack in different threads on the same machine or even on distinct machines, forwarding packets over the network.

We could, for example, allocate an independent thread to fetch packets, to limit the risk of kernel queue overflow. It is also possible to perform some kind of load balancing or to correlate several observations made from different vantage points in cases where information is not fully available at a single location. For example we may think of HTTP requests and responses flowing through two distinct links. Two distinct kinds of filters may be used to achieve this: thread and I/O filters.

Each thread filter creates a new thread of execution in the filter stack. All packets added to such a filter are forwarded to the newly created thread. In other words, they split the stack into two parts, each of them being executed in an independent thread. A thread filter manages synchronization — packets are added in a shared, mutex-protected, FIFO — but relies on other filters being fully reentrant.

I/O filters are used to exchange packets across the network. A TCP connection is established between an output filter (acting as a client) and an input filter (acting as a server, thus allowing packets coming from different machines to be merged). Input filters dispatch incoming packets to the remainder of the stack. An administration facility allows easy use of these filters: it sets up the TCP server, performs the necessary configuration on the client side of the connection (setting the name and incoming port of the remote host) and monitors clients for crashes, restarting them if necessary.

5 Implementation

The software consists currently of about 10,000 lines of C++ code. Beyond the core system, described Section 4, we have implemented various filters — grouped into a library.

This library includes filters that can reassemble IP packets and reorder TCP packets. In more specific domains, we also provide filters for parsing and matching HTTP traffic (compatible with version 1.1 of the protocol). The same functionalities exist for ICP traffic.

A generic demultiplexing filter is also provided. It is template class which is parameterized by the packet type on which it operates, along with the demultiplexing function. It also contains a timer which selectively times out inactive branches.

Only two switch filters are currently implemented. The first is used for IP reassembly, allowing unfragmented packets to skip the reassembly sub-stack. The second one is used to demultiplex IP packets according to their protocol (TCP, UDP, ICMP, ...).

6 Examples of Use

Pandora has been used in two different, but related, fields: Squid cache and Web traffic collection. These two examples exercise the different features of Pandora presented so far, and its ability to handle transactional protocols. The cache monitoring example was set up in very short amount of time (it took only a few hours, starting from scratch). To further illustrate Pandora's ability to capture complex protocol interactions, we combined these two analysis into a single experiment.

6.1 Cache Monitoring

The Internet Cache Protocol [17] permits caching proxies to cooperate (in particular Squid [16, 18] caches).

Basically, a cache emits an ICP query for each miss, asking its siblings whether they possess the requested document. Each peer responds in turn with a hit, a miss, or an error message. In a second phase, the original cache

forwards the request either to a sibling that responded with a hit, or to the original server. Each message is distinguished by an unique 32-bit identifier and uses UDP as the transport protocol.

Figure 4 shows the stack we use to evaluate the overheads caused by this protocol in terms of additional delay and bandwidth usage.

```
[stack]
fragmented? (
  ipfragdemux <
    ipreass
  >
) ipcnxdemux <
  scanudp - scanicp - icpdemux <
    matchicp
  >
> output
```

Figure 4: Cache monitoring stack definition.

The first part of the stack (up to `ipcnxdemux`) is dedicated to IP reassembly and has the same structure we discussed in Section 3. The `ipcnxdemux` filter demultiplexes packet according to their source and destination IP addresses, but no discrimination is made on the direction of the connection (i.e. we make no distinction between packets coming from the requesting host or from the responding host). The `scanudp` filter extracts UDP packets from IP ones and passes them to `scanicp` filter. The latter in turn produces ICP packets. These are demultiplexed according to their request identifier in the `icpdemux` filter. Then, given our demultiplexing steps, only the two matching ICP messages are finally passed to the `matchicp` filter, whose only work is to associate them in a single ICP transaction packet.

6.2 Web Traffic Collection

We already introduced HTTP extraction in Section 3. This application[5] is meant to help to characterize the Web activity of a community of users. This

[5] A Web traffic collection experiment has actually been led at INRIA for one month using this software.

information can then be used to dimension cache infrastructure for example (as in Saperlipopette! [14]), or to give hints about which sites are worth mirroring.

Figure 5 shows the configuration file of the Web traffic collection system.

```
[stack]
iptotcp - tcpcnxdemux <
  tcpdirdemux <
    tcporder - httpscan
  > anonymize - httpmatch
> output
```

Figure 5: Web traffic collection stack definition.

The `tcpcnxdemux` filter demultiplexes packets according to their connection identifier (the 4-tuple IP address and port for the source and the destination) independent of the direction of the flow. The separation of both flows is made by the `tcpdirdemux` filter. This way, requests and corresponding responses are directly multiplexed together.

6.3 Combined Experiment

We may combine the previous two analyses to further investigate the interaction between ICP and HTTP in Squid hierarchies. We set up on our local network the experiment shown on Figure 6. Before the experiment begins, we filled the cache on `hostB` — following the annotations in the figure — with some of the documents of the Web server on `hostC`. The experiment itself consists of sending requests through an *empty* proxy cache on `hostA` and monitoring the HTTP and ICP traffic.

To achieve this, we placed two Pandoras running an HTTP extraction stack on both sides of the cache. Their traces are sent to another instance of Pandora which combines these in order to deduce the requests served by the cache and those sent to the server. Simultaneously ICP traffic is logged. ICP and HTTP flows are finally sent to still another instance of Pandora responsible for producing a single log file.

This log file allows us to infer time overheads due to ICP exchanges, considering not only the network delays but also the time needed by the cache to

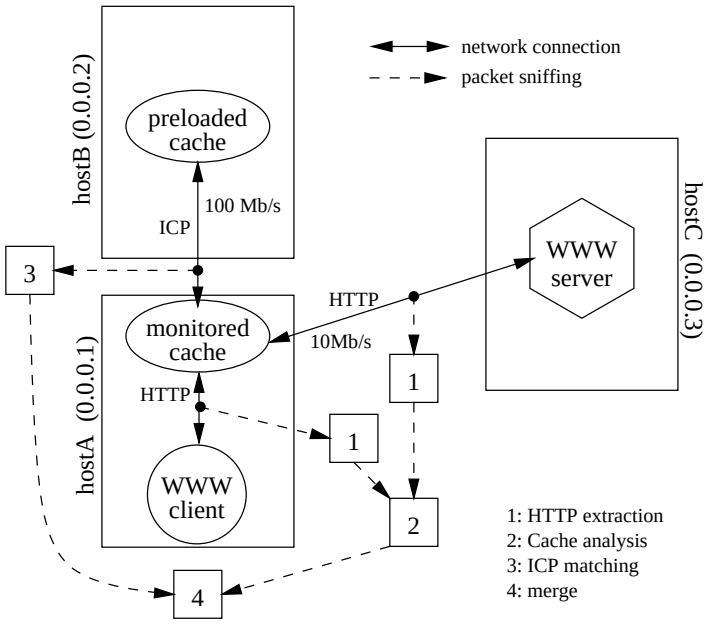

Figure 6: Experiment for ICP/HTTP relationship analysis.

send and process these datagrams (ranging from 2 ms to more than 100 ms). One might also analyze a cache's internal choices and policies: sending an *If-Modified-Since* request instead of a plain one, bypassing the sibling when it is considered too slow. Some further post-processing of the file might also help to understand which documents are replaced, and how.

7 Evaluation

Performance is a major issue that we set out to address. Indeed, packets are buffered by the system in a kernel queue which is emptied by the monitoring process. When the queue is filled up,[6] packets are discarded by the system. It is a common belief that flexibility has a price. This could jeopardize our performance goal. After having described our experimental environment, we will show the results of several tests meant to measure respectively the overheads due to our flexible design, those related to TCP reordering and Web traffic collection. Finally we will quantify the effective throughput of Pandora when used to collect Web traffic in more realistic conditions.

7.1 Experimental Environment

In order to stress our system we used several traces collected by `tcpdump` on the link connecting INRIA with the outside world. Their characteristics are presented in Table 1. The traces were read from a file on a local disk, and the results presented in the following sections correspond to the arithmetic mean of 50 runs of the same test. The workstation used to run these tests was a DEC *Personal Workstation* using a 21164A processor at 500 MHz, with 684 MB of RAM.

All times measured are wall-clock execution time (in milliseconds) on this workstation, otherwise idle (yet sill not exclusively dedicated to this task). Runs are divided into two stages: in the first stage, the stack is initialized in an independent, blocked thread, then all packets are read into memory and

[6] With the BSD Packet Filter [13], this queue has a maximum size of 255 packets. However, this limit dates back to 1989. With modern workstations, it would certainly be appropriate to increase this limit up to 1023, for example, where possible.

Name	IP	TCP	HTTP1	HTTP2	HTTP3
Filter used	ip	tcp	port 80	port 80	port 80
Packets	500,000	500,000	400,000	747,907	799,455
Size (MB)	196.6	217.5	222.5	397.6	400.0
Web requests	N/A	N/A	17,308	34,414	38,694
Average packet size (bytes)	412.2	456.1	583.2	557.4	524.6
Duration (s)	1,053	1,138	4,341	9,324	9,374

Table 1: Characteristics of traces used for performance evaluation.

stored in a FIFO queue with no processing other than extracting IP packets from link-level protocol. No timing is done in this phase. Then, the timer starts and the stack thread is signalled to starting processing. The timer is stopped when all packets have been processed and the stack cleaned up (a destruction notification is sent to the first filter in the stack).

7.2 Flexibility Support Overhead

In this section we will quantify the overheads directly linked to our design: transforming layering and demultiplexing into "first-class" entities.

7.2.1 Layering

Number of identity filters	0	1	2	4	8
Execution time (ms)	1,935	2,014	2,084	2,199	2,439
Standard error (ms)	7	9	9	7	7
Average per filter overhead (ms)	N/A	79	74.5	66	63

Table 2: Evaluation of layering overhead, using the IP trace.

To measure the cost of layering we run Pandora against the IP trace with a variable number of identical filters that are only pushing packets as soon as they receive them. After traversing these *identity* filters, packets are silently

discarded. This gives us the minimal processing time per packet per filter. We used the IP trace for these runs. The test with no identity filter acts as a reference; its execution time is mainly due to synchronization (due to the initial thread) and the destruction of packets.

The results are presented in Table 2. They show approximatively 70 ms per filter overhead for 500,000 packets (i.e. 140 ns per packet, per filter). This overhead is relatively small, compared to the cost of effective computation time. We can safely say that the layering structure of Pandora has little impact on performance.

7.2.2 Demultiplexing

We want to quantify the cost of demultiplexing packets according to their connection identifier. The test consists of extracting TCP packets from the TCP trace, demultiplexing them, passing them through the identity filter and finally discarding them.

Run	Construc.	Demux.
Exec. time (ms)	2,912	6,634
Std. err. (ms)	14	15
Avg. overhd. (ms)	N/A	3,722

Table 3: Evaluation of demultiplexing overhead, using the TCP trace.

The results are shown in Table 3, which includes the cost of constructing TCP packets (and immediately discarding them) as a reference. It appears that the demultiplexing overhead is rather high (about 7.4 μs per packet) and confirms that the "layering only" overhead is small (representing less than 2 % of the former). This highlights the relative importance of the algorithm used for demultiplexing. We used an STL map, and we think that using dedicated structure could greatly improve the efficiency of demultiplexing, at the cost of flexibility.

7.3 TCP Reordering Overhead

We consider a more realistic example of use: the reordering of TCP packets, which reconstructs the original byte-stream of TCP. To achieve this, we have to demultiplex packets according to their connection identifier before any attempt at reordering. Each reordering filter keeps track of the state of the connection in order to deliver packets as soon as possible (in particular we pay attention to SYN packets to keep an up-to-date expected next sequence number). Yet, since we may miss SYN packets — or even not see them at all if the connection started before our monitoring — or intermediate packets, we must timeout idle connections to release the resources they use. Given the speedup ($\times$ 147) we artificially introduce by reading the trace from a file, we choose a timeout value of 2 seconds (equivalent to 5 minutes). We use the TCP trace as input.

Run	Demux	Reorder.
Exec. time (ms)	6,634	7,752
Std. err. (ms)	15	16
Avg. overhd. (ms)	N/A	1,118

Table 4: Evaluation of reordering overhead, using the TCP trace.

Table 4 shows the overhead in terms of CPU time due to the TCP reordering filter. This is obtained by comparing the reordering filter execution time against the demux filter. This measurement shows that more than 80% of TCP reordering cost is due to the demultiplexing feature. Reordering in itself does not cost too much.

7.4 Web Traffic Collection Overhead

As a final evaluation we use our HTTP reconstruction example. The tasks performed are a subset of those we previously described: we did not perform IP reassembly (we filtered out fragmented packets), nor the on-line anonymization. As for the previous test, we have to timeout idle connections. Moreover, to get deeper insight on layered demultiplexing we used two different stacks: one making intensive and "intelligent" use of layered demultiplexing (such as

the one shown on Figure 1), the other remultiplexing packets after each processing step (namely reordering, HTTP parsing, and HTTP matching). We used here the HTTP1 trace.

Run	Reordering	Layered extraction	Std. extraction
Execution Time (ms)	7,001	12,618	13,598
Standard Error (ms)	19	15	11
Overhead (ms)	N/A	5,617	6,597

Table 5: Evaluation of HTTP extraction overhead.

Results are shown in Table 5. They highlight the benefit of using layered demultiplexing. Indeed, despite the overhead occurring at the first demultiplexing step (one has to swap source and destination parameters in the key used whenever necessary, and many more filters are created), the layered extraction performs 17% better than the stack re-multiplexing packets after each processing step.

It also shows the real cost of some realistic processing. On average, we require about 25 μs per packet. We see then that the layering cost (i.e. splitting into two steps what could be done in one) now represents only 0.56 % of the time to process a packet.

7.5 Web Traffic Collection Throughput

Given that HTTP reconstruction is in fact a real application, we also measure its global throughput. This involves reading each packet from a file, processing them through the filter stack, and writing records to a log file. These runs, more than per-packet costs, give us an idea of the maximum bandwidth such an application can monitor without dropping packets. The packets are given to the filters at an almost constant rate, which differs greatly from real network conditions. Conservatively, we may say then that the application will be able to monitor links on which the peek packet rate (or requests in our HTTP example) are below the one we achieved.

Table 6 shows the results of these tests. With the three different traces we used, we never dropped below 600 requests/s. This leads us to think that

Run	HTTP Extraction		
Trace used	HTTP1	HTTP2	HTTP3
Execution Time (ms)	28,274	57,216	60,386
Standard Error (ms)	56	196	207
Throughput (reqs/s)	612.2	601.5	640.8

Table 6: Evaluation of HTTP extraction throughput.

Pandora can to cope with most high bandwidth links, without suffering from too many packet losses.

8 Related Work

We borrow the stacking approach from several platforms, in various domains. Well known examples include Ficus [9], *x*-Kernel [10], Horus [15] and Ensemble [8]. Yet, to the best of our knowledge, it has never been used for network monitoring tools.

Among these monitoring tools, `tcpdump` [11] was the first to be widely used. Its main use today is for off-line analysis. This may not be acceptable in many situations where the volume of data is too big — on busy, high-speed links, it is not uncommon to collect more than 1 Gigabyte of data within 15 minutes. Moreover, it is impossible to have active behavior in this situation.

Several other tools have been developed since `tcpdump`, but for some obscure reason very few of them have been publicly released. This is partly why we decided to develop Pandora.

CoralReef [1] is a complete (software and hardware) solution to perform passive network monitoring. Designed for high performance, it is primarily meant only to log packet headers, which forbids any application-level analysis.

Ian Goldberg's IPSE [7] is another recent approach to network monitoring: built as Linux kernel modules, it promises good performance. However, an undetectable leak of memory forced the authors to periodically stop and restart their experiments. Besides, flexibility did not seem to be a primary concern when designing this tool.

Windmill [12] is the closest tool to Pandora. Yet, it was not originally designed to perform the same tasks we intend to do with Pandora: Windmill is a platform designed to evaluate protocol performance — via a set of experiments, while uses of our tool are of broader scope and focus on a single application. This leads to very different architectural choices. Windmill's implementation of protocols is rather rigid (recursive calls to lower protocol layers are hardwired into the code). It has the advantage of better performance but lacks flexibility and a clear distinction between mechanism and policy. With respect to Web traffic monitoring, which motivated the development of Pandora in the first place, several tools have been proposed in the recent past.

HttpFilt and HttpDump [19] were one of the first attempts to construct such specialized tools. The first, based on simple Perl scripts, was nevertheless rather restrictive in the use (it could only handle "not-pathological" HTTP/1.0 transactions). The second, meant to improve the performance of the previous tool, happened to behave worse — even not coping with 10 Mb/s links. Yet, their complete on-line processing of packets remained without comparison. In contrast, Pandora can monitor versions 1.0 and 1.1 of the HTTP protocol. As shown by the performance evaluation in Section 7.5, Pandora can achieve throughput more than an order of magnitude better than HttpDump claims.

BLT [3] is meant to provide on-line HTTP traces at a high performance level. With a machine comparable to ours (a 500-MHz Alpha workstation), BLT was able to capture a 12 day trace on an FDDI ring, handling more than 150 millions packets a day (an average of about 1750 packets per second) with a loss rate of less than 0.3% [4]. Compared to Pandora, BLT claims better performance. However, this has to be slightly tempered since HTTP request/response matching (which is an essential stage in Web logging) has to be performed off-line. Also, we consider the overhead of our demultiplexing filter too high (it represents more than 33% of the total overhead) and we are planning to optimize it.

9 Conclusion and Future Work

We presented Pandora, a passive network monitoring platform. We discussed its basic components and how they could be used to set up different monitor-

ing systems at a very low cost. We presented a performance evaluation that supports the claim that Pandora is an efficient flexible continuous monitoring tool, that is: it can run 24 hours a day, 7 days a week. It allows *real* on-line analysis of network protocols, up to the higher levels — including analysis beyond strict application-level protocol, such as HTTP matching. The use of filters makes it easy to extend the platform.

Nevertheless, Pandora still needs improvements at various levels. First the stack configuration process should benefit from a more user-friendly interface. We also plan to implement a tool that can check stack description validity, in order to avoid run-time errors when feeding a filter with packets it does not expect. Last but not least, we must continue developing filters and improving the existing ones.

Pandora has already been used to retrieve HTTP traces during a one month period (representing a total amount of more than six million requests). These traces were then used in Saperlipopette! [14], a tool designed to help dimension proxy cache infrastructure.

More generally, Pandora is meant to facilitate the development of access infrastructure in the ever-growing Internet. It can also be used as a way to compare the various solutions offered to address specific problems. The number and the diversity of tools developed by the research community in the recent past that are related to these issues prove that it corresponds to a real need, and highlight as well the lack of a flexible monitoring tool that could satisfy these.

Availability

Pandora will be publicly available from January 2000 in source-code form. See `http://www-sor.inria.fr/relais/` for more information.

References

[1] Joel Apisdorf, K. Claffy, and Kevin Thomson. OC3MON: Flexible, affordable, high-performance statistics collection. In *Proceedings of the INET '97 Conference*, Kuala Lumpur, Malaysia, June 1997. `http://www.isoc.org/isoc/whatis/conferences/inet/97/proceedings/F1/F1_2.HTM`.

[2] Ramon Caceres, Cormac J. Sreenan, and J. E. van der Merwe. mmdump - a tool for monitoring multimedia usage on the internet. submitted for publication, July 1999. `http://www.research.att.com/~ramon/papers/mmdump.ps.gz`.

[3] Anja Feldmann. Continuous online extraction of HTTP traces from packet traces. Position paper for the W3C Web Characterization Group Workshop, November 1998. `http://www.research.att.com/~anja/feldmann/w3c98_httptrace.abs.html`.

[4] Anja Feldmann, Ramon Caceres, Fred Douglis, Gideon Glass, and Michael Rabinovich. Performance of Web proxy caching in heterogeneous bandwidth environments. In *Proceedings of the INFOCOM '99 conference*, March 1999. `http://www.research.att.com/~anja/feldmann/papers/infocom99_proxim.ps.gz`.

[5] Anja Feldmann, Jennifer Rexford, and Ramon Caceres. Efficient policies for carrying web traffic over flow-switched networks. *IEEE/ACM Transactions*, December 1998. `http://www.research.att.com/~anja/feldmann/papers/ton98_flow.ps.gz`.

[6] Robert Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, and Tim Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. Request for Comments 2068, January 1997. `ftp://ftp.isi.edu/in-notes/rfc2068.txt`.

[7] Steven D. Gribble and Eric A. Brewer. System design issues for Internet middleware services: Deductions from a large client trace. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, December 1997. `http://www.cs.berkeley.edu/~gribble/papers/sys_trace.ps.gz`.

[8] Mark Hayden. The ensemble system. Technical Report TR98-1662, Cornell University, January 1998. `http://cs-tr.cs.cornell.edu/Dienst/UI/1.0/Display/ncstrl.cornell/TR98-1662`.

[9] John S. Heidemann. Stackable layers: An architecture for file system development. Technical Report UCLA-CSD 910056, University of California, Los Angeles, CA (USA), July 1991. `http://www.isi.edu/~johnh/PAPERS/Heidemann91c.ps.gz`.

[10] Norman C. Hutchinson and Larry L. Peterson. The *x*-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991. `ftp://ftp.cs.arizona.edu/xkernel/Papers/architecture.ps`.

[11] Van Jacobson, Craig Leres, and Steven McCane. `tcpdump`. software (latest release: version 3.4), June 1989. `ftp://ftp.ee.lbl.gov/tcpdump.tar.Z`.

[12] 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`.

[13] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of the 1993 Winter USENIX conference*, San Diego, California, January 1993. `http://www.ntua.gr/rin/docs/bpf-usenix93.ps`.

[14] 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`.

[15] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in horus. In *Proceedings of Principles of Distributed Computing*, August 1995. `http://www.cs.cornell.edu/Info/People/rvr/papers/podc/podc.html`.

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

[17] 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`.

[18] Duane Wessels and K. Claffy. ICP and the Squid web cache. *IEEE Journal on Selected Areas in Communication*, 16(3):345–357, April 1998. `http://www.ircache.net/~wessels/Papers/icp-squid.ps.gz`.

[19] Roland Wooster, Stephen Williams, and Patrick Brooks. Httpdump: Network HTTP packet snooper. working paper, April 1996. `http://ei.cs.vt.edu/~succeed/96httpdump/final_paper/paper.ps.gz`.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

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