Pandora: A Flexible Network Monitoring Platform

Simon Patarin and Mesaac Makpangou
INRIA SOR Group
Rocquencourt, France
Simon.Patarin@inria.fr, Mesaac.Makpangou@inria.fr

Abstract

This paper presents Pandora, a network monitoring platform that captures packets using purely passive techniques. Pandora addresses current needs for improving Internet middleware 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 as independent entities we call monitoring components. The actual packet analysis is performed by stacking the appropriate components. Pandora also preserves user privacy by allowing control of the "anonymization" policy. Finally, the evaluation we conducted shows that overheads due to Pandora's flexibility do not significantly affect performance. Pandora is fully functional and has already been used to collect Web traffic traces at INRIA Rocquencourt.

1 Introduction

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

In recent years, several network monitoring tools have been proposed. These include naturally `tcpdump` [11], and specialized software like BLT [2] and HttpFilt [21] for HTTP extraction, and `mmdump` [1] for multimedia monitoring. One must also mention more generic platforms like IPSE [6] and Windmill [12].

As the Internet grows, the demand for a flexible monitoring system increases. Such a system can be used 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 remain 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 observes the entire traffic within the monitored organization. Such dependencies make most existing tools redundant if the targeted protocol evolves or the assumed configuration changes. Also, it makes them difficult to adapt and cope with new problems.

This paper presents Pandora,[1] 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. Pandora uses passive network monitoring to reconstruct high level protocols while keeping track of lower level events. It provides basic building blocks implementing the commonly used analysis.

The rest of the paper is organized as follows: Section 2 presents our design goals for Pandora. Section 3 describes the architecture of the system, 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 re-

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

marks 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 into 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 be able to process all packets in real-time; this means that it cannot be designed only to store packets and process them off-line.

Our system must be able to monitor medium-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 occur frequently 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, DNS 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 use the same monitoring tool for numerous different purposes easily. 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. Also, it should be easy to add new protocol-specific elements to the system.

3 Architecture

Pandora is designed as a stack of monitoring components. Each component encapsulates an elementary monitoring task (e.g. IP layer reassembly, TCP layer resequencing, 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, and hence to the design of better protocols and tools.

The protocol we need to monitor here is HTTP/1.1 [5]. HTTP follows a request/response model. Meta-data carried by headers are line-oriented and use textual attributes and values. Version 1.1 of the protocol introduces the possibility of making persistent connections between clients and servers and of pipelining requests and responses (not waiting for the other endpoint to reply before sending additional data on the same connection).

The `libpcap` [10] 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. We finally 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 resequencing 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 locality 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 [4]). 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 separate tasks that must be performed to produce the trace, and the order in which they must be proceeded, we define two basic notions: monitoring components and component stacks. A monitoring component (or simply, component) is responsible for a specific elementary task, while the stack is the structure where these components are chained together.

A component 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 component designed to perform IP reassembly transforms a flow of IP fragments into a flow of IP packets.

A component relies only on the properties of its input. In particular, it does not have to know about the other components in its stack. This lets us replace a component by an equivalent one, or to introduce new components into the stack with no effect on the rest of the stack. For example, imagine that we want to encrypt the data collected (a perfectly legitimate issue); we have only to add a component 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 components exist, or at minimal cost concerning only the de-

velopment of the missing ones. Using components makes it easier to debug the monitoring path; effort can be concentrated on the optimization of components that constitute bottlenecks.

The stack determines how components are chained. It represents an ordered set of components through which packets flow from one end to the other. It is important to notice that, in such a structure, each component has exactly one input and one output; in other words, there is a single, linear data flow for the entire stack. Such a simple stack model has two drawbacks. First, packets that belong to distinct connections are merged into an unique flow. This implies that every components that operates on separate packet flows will have to perform the demultiplexing on its own, since having only a single output is equivalent to re-multiplexing everything before passing packets to the next component. With respect to our example, packets coming from all HTTP connections are captured and delivered to the system interleaved, and demultiplexing must occur at IP, TCP and HTTP level. Second, this simple stack model (with a unique data flow) makes it impossible to shortcut some of the processing components, even if we know *a priori* that this will not be necessary: all packets must pass through each component. For example, we stated that IP packets could be fragmented, yet rather few are. Given that this property can be determined early (by simply looking at IP flags), why should we make them all pass through the reassembly components (demultiplexing and effective reassembly)?

These may look like straight-forward, standard problems, but it requires some careful thought to fit them easily into the component framework.

3.3 Generalized Stacks

The two limitations of the simple stack model we mentioned above have led us to extend this model by introducing *control* components. They allow several data flows to coexist in a single stack. This approach permits us to save resources and avoids unnecessary component traversals. Furthermore, taking flow control mechanisms out of processing components permits component developers to concentrate their efforts on the precise functionality they want to implement.

The control components are the following:

Switch component: It permits packets to follow along different processing paths: configured with a fixed number of alternative paths, it can forward a packet to any of them, according to some hard-coded internal logic. 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 component.

Demux component: Such a component dynamically instantiates a dedicated sub-stack for each new packet flow it detects. 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.

Figure 1 shows one solution for the Web trace collection example that uses switch and demux components. The "connection demux" receives TCP packets from the IP to TCP extraction component. 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 components performing TCP resequencing 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 component, which determines which request corresponds to which response.

3.4 Benefits of the Stacking Approach

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

Evolvability: Protocols are evolving rapidly and independently from each others. The use of components permits easy adaptation to these changes.

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

Modularity: Each task is encapsulated in its own component which enforces a clear division between mechanism and policy, eases readability

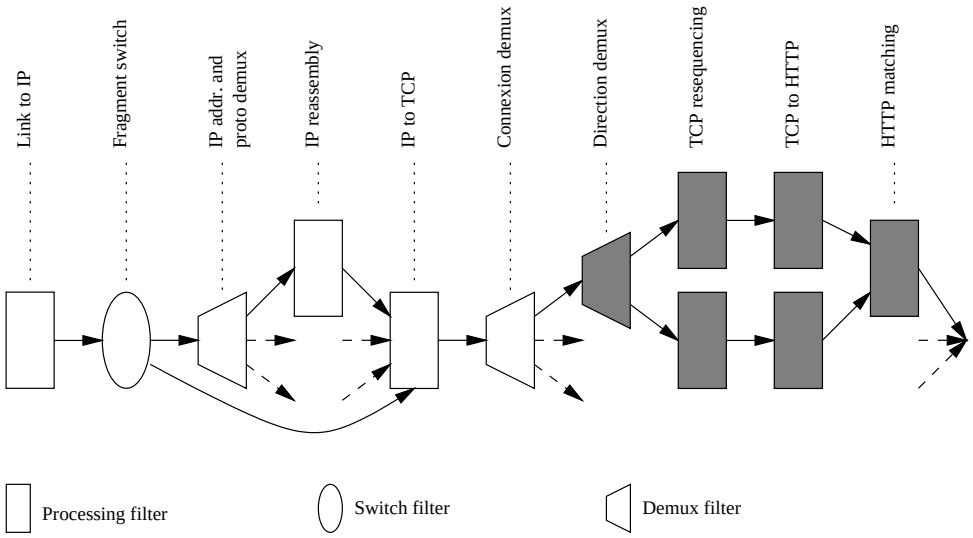

Figure 1: The component stack used in the Web traffic collection example. The shaded components are those dynamically created by the "connection demux" when a new TCP flow is identified.

and has proven very valuable for maintaining existing code and debugging. There are no obscure side effects, nor hidden dependencies.

3.5 Configuration

Figure 2 presents a sample configuration file for an application that logs packets in distinct log files according to their transport layer protocol (TCP, UDP or ICMP). In details: packets first flow through the `ipfragswitch` switch component. This component makes IP fragments go into the IP reassembly sub-stack and let the other not-fragmented packets skip this part. Inside the IP reassembly sub-stack, packets are demultiplexed according to their IP addresses, protocol number and identifier in the `ipfragdemux` demux component. Then the `ipreass` component reassembles demultiplexed IP fragments. The timeout specified in the corresponding option section tells the filter to flush any incomplete packet after a 60 seconds inactivity period. At this point, all IP packets reaching the `ipproto` component are complete IP packets. This switch component forwards them to distinct sub-stacks according to their transport protocol. The options it is given tells it what are the actual branch indexes used. TCP packets go to the first one where they are parsed and written to a log file. As are UDP and ICMP packets to the second and the third sub-stack, respectively. Finally, all packets with a different protocol skip this and are discarded.

The configuration file is made of several sections. The stack section (beginning with a `[stack]` tag) describes the components used and how they are chained. We designed a small language to specify this. Its grammar is presented in Figure 3. A stack is composed of a number of components. The simple components are identified only with their name.[2] The switch and demux components are described by their name as well as the definition of their sub-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 to which it can forward packets.

Other sections of the configuration file allow us to set up options for any component mentioned in the stack section. Such option sections start with the name of the component enclosed in square brackets: for example, `[ipfragdemux]`. In addition, components of the same type are numbered from left to right, starting at 0, in order of appearance. This is needed in cases where several instances of the same component are used, and one wants to set different options for each instance. That is, one creates an option section identified by this component name followed by its rank (e.g. `[output(0)]`). Unnum-

[2]The "`-`" connector is optional and is used only for clarity (e.g. one can alternatively use `component1 - component2` or `component1 component2`).

```
# beginning of stack definition
[stack]
ipfragswitch (
    ipfragdemux <
      ipreass
    >
) ipprotoswitch (
    tcpscan output |
    udpscan output |
    icmpscan output
) discard
# end of stack definition

# beginning of options sections
[ipreass]
timeout = 60

[ipprotoswitch]
tcpbranch = 0
udpbranch = 1
icmpbranch = 2

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

[output(1)]
static file = udp_pkts.dat

[output(2)]
static file = icmp_pkts.dat
# end of option sections
```

Figure 2: Configuration file for a stack that reassembles fragmented packets and logs TCP, UDP and ICMP packet into different files. Packets of other transport level protocols are discarded.

bered option sections apply to all components of the specified type. Inside an option section, each line describes the setting of an option according to the following syntax:
`option_name = option_value`
Values can be of numeric or string type. String type values are processed by a function defined by the component developer. By default, this function is evaluated each time the component is created. Yet, this may not be always suitable; prefixing option name with the keyword `static` guarantees that the function will only be evaluated once.

4 Stack Instantiation

Once the stack and component options have been parsed, one has to instantiate the stack. This involves dynamically building the sequence of components and linking them appropriately. The difficulty

```
stack  ::= component+

component ::= simple | demux | switch

simple ::= fname '-'?

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

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

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

Figure 3: Stack definition grammar.

stems from the fact that none of the components know 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 components. We call this entity the *dispatcher*.

4.1 Component Creation

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

When created, a component does not know anything about other components in the stack. If it wants to pass a packet to its successor, it must first ask the dispatcher for a pointer to its successor (which it stores for future accesses). If the requested component does not exist, the dispatcher instantiates it, then returns its reference to the calling component. This is the case for simple components: successors are never created until they are accessed. This way, the dispatcher does not need to keep track of previously created components.

However, this technique poses problems for demux and switch components: the ending components of every created branches must share the same successor (the *multiplexing* component). To address this issue, 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 component along with the demultiplexing one (be it demux or switch). Then, each component in a demultiplexed branch carries a reference back to its demultiplexing component (this

reference is passed dynamically when the component is created). When an ending component asks the dispatcher for its successor, the dispatcher retrieves the reference to the multiplexing component from the demultiplexing one, by following a back-pointer to the branch point.

4.2 Memory Management

As with their creation, destruction of components is performed incrementally. The process starts with a component notifying the dispatcher of its intention to destroy its successor (or one of them, if this is a demultiplexing component). This notification is then propagated by the dispatcher. It first asks the target component to prepare for destruction. Upon completion of this request, the dispatcher effectively destroys the component (and resets the caller's pointer to a nil value). The process iterates up to the component at which the initiator's branch is multiplexed, where it stops. For a multiplexing component, destruction preparation involves recursively initiating a new destruction process for each of its branches. Other components simply flush all their unprocessed packets.

We have two ways to trigger component collection: active and passive. Active collection occurs when a component determines autonomously that its processing is finished. It then notifies its predecessor[3] that it can start the destruction process. This is the case when an IP reassembly component receives the last fragment of an IP packet. Passive collection occurs for inactive components whose processing remains unfinished either because the component has no way to know it, or because of a packet drop. This process is based on predetermined timeouts.

Pandora provides a generic framework to timeout inactive components. A global clock is maintained according to the time-stamps of incoming packets. Then, each component has the possibility to register itself for a specific timeout. When no packets are passed to the component during the specified period, the destruction process for that component (and all its successors) is triggered.

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 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.

[3]By way of the return value of the function used to pass a packet to a successor component, or via the dispatcher.

4.3 Concurrent Processing

Pandora allows components 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 a thread to fetch packets, to limit the risk of kernel queue overflow (for kernel structure has a fixed size, whereas our buffers are only limited by the available memory). 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 components may be used to achieve this: thread and I/O components.

Each thread component creates a new thread of execution in the component stack. All packets added to such a component 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 component manages synchronization — packets are added in a shared, mutex-protected, FIFO — but relies on other components being fully reentrant.

I/O components are used to exchange packets across the network. A TCP connection is established between an output component (acting as a client) and an input component (acting as a server, thus allowing packets coming from different machines to be merged). Input components dispatch incoming packets to the remainder of the stack. An administration facility allows easy use of these components: 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 (although no state recovery is performed).

5 Implementation

The software consists currently of about 15,000 lines of C++ code. Beyond the core system, described Section 4, we have implemented various monitoring components — grouped into a library.

The list of components included in this library is presented in Table 1. The library also provides all data structures by these components (packet types, hash functions, etc.).

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).

6.1 Cache Monitoring

The Internet Cache Protocol [19] permits caching proxies to cooperate (in particular Squid [18, 20] caches).

Basically, for each miss, a cache emits ICP queries to know if one of its siblings 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.

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` component demultiplexes packet according to their source and destination IP addresses, but no discrimination is made on the direction of the connection

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

Figure 4: Cache monitoring stack definition.

(i.e. we make no distinction between packets coming from the requesting host or from the responding host). The `scanudp` component extracts UDP packets from IP ones and passes them to `scanicp` component. The latter in turn produces ICP packets. These are demultiplexed according to their request identifier in the `icpdemux` component. Then, given our demultiplexing steps, only the two matching ICP messages are finally passed to the `matchicp` component, 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[4] is meant to help to characterize the Web activity of a community of users. This information can then be used to dimension cache infrastructure for example (as in Saperlipopette! [15]), or to give hints about which sites are worth mirroring.

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

```
[stack]
tcpscan - tcpcnxdemux <
    tcpdirdemux <
      tcpreseq - httpscan
    > anonymize - httpmatch
> output
```

Figure 5: Web traffic collection stack definition.

The `tcpcnxdemux` component 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

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

Monitoring Components	
PcapHandler	Encapsulation of the `libpcap` library
ICMPScan	Extracts ICMP packets from IP packets
TCPScan	Extracts TCP packet from IP packet
UDPScan	Extracts UDP packet from IP packet
ICPScan	Extracts ICP packet from UDP packet
RPCScan	Extracts RPC packet from UDP packet
DNSScan	Extracts DNS packet from UDP packet
HTTPScan	Extracts HTTP message from TCP packet flow
IPReassembly	Reassemble IP fragments into IP packet
TCPResequence	Resequence TCP packets in TCP packet flow
HTTPMatch	Matches HTTP request and response into HTTP transaction
GenMatch (template)	Generic component allowing to produce transaction packets for "simple" protocols (currently used for ICP, RPC, DNS).
Input	Passes to the first component of the stack the packets transmitted over the network between two instances of Pandora
Output	Outputs packets to log file (dealing with file rotation), console or to another instance of Pandora through the network
Discard	Discards any packet it receives
Thread	Inserts a thread into the processing path
Demux Components	
GenDemux (template)	Generic demux component (given the input packet type and the hash function to apply)
TCPPortDemux	Demultiplexes TCP packets according to their port number
Switch Components	
IPFragSwitch	Switches between fragmented and non-fragmented IP packets
IPProtoSwitch	Switches between transport level protocols of IP packets (currently TCP, UDP, ICMP)

Table 1: Existing components and packet types in Pandora.

separation of both flows is made by the `tcpdirdemux` component. Once TCP packets are resequenced, the `httpscan` components extract HTTP messages (requests and responses) from each flow. Then, these messages are multiplexed, so that requests and their corresponding responses belong to the same flow. After being anonymized, requests and responses are matched to each other (without any interference from messages of other connections).

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,[5] 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 demultiplexing and Web traffic collection. Finally we will quantify the effective throughput of Pandora when used to collect Web traffic in more realistic conditions.

[5]With the BSD Packet Filter [13] on a DEC OSF1 operating system, this queue has, by default, a maximum size of 256 packets.

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. 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 (we always present the standard error of these measurements).

The workstation used to run these tests was a DEC *Personal Workstation* using a 21164A processor at 500 MHz, with 684 MB of RAM. We chose relatively small trace files (500000 packets) so that their entire content can remain in main memory between successive runs.

All times measured are the sum of system and user execution time (in seconds) as given by `getrusage` on this workstation (still used for light office tasks at the same time).

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

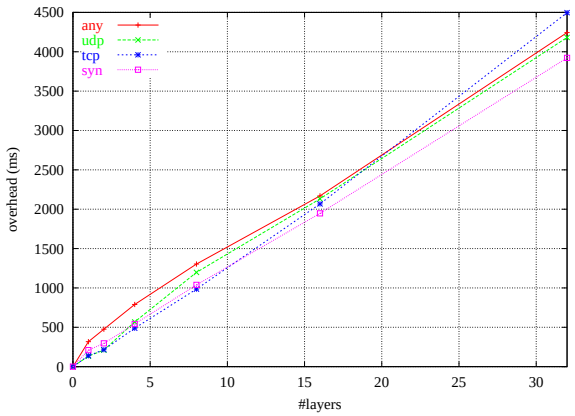

Figure 6: Evaluation of layering overhead, using the ANY, SYN, TCP and UDP traces.

To measure the cost of layering we ran Pandora against four traces (whose characteristics are presented in Table 2) with a variable number of identical components that are only passing packets to their successor component as soon as they receive them. After traversing these *identity* components, packets are silently discarded. This gives us the minimal processing time per packet per component. We also ran a test with no identity component; this acts as a reference: its execution time represents the amount of time spent in reading the trace file, and in the creation and the destruction of packets.

Figure 6 plots the measured overhead (compared to our reference) for each of the traces. This shows approximatively 130 ms[6] per component overhead for 500,000 packets (i.e. 260 ns per packet, per component). As one might expect, this value does not depend on the size of packets. 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.

[6] We obtained this value by computing the mean overhead by number of layers over each trace, and then performing a linear regression analysis. The squared correlation factor (r^2) is equal to 0.9996.

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 traces with various characteristics (presented in Table 3), demultiplexing them, passing them through the identity component and finally discarding them, while timing out (with different timeout values) inactive demultiplexed branches. Our reference is the time taken to extract TCP packets from the traces.

The results are shown in Table 4. It appears that, though the demultiplexing problem is very simple, its cost is rather high. It ranges from 2.1 μs to more than 20 μs per packet, and has an average around 8 μs for the two "realistic" traces: TCP and WWW. First, this confirms that the "layering only" overhead is small (only 0.26 μs per packet) and highlights the importance of the algorithm and the data structures used for demultiplexing. Second, it shows that it is very difficult to predict what the cost of demultiplexing will be: it depends on the size of the demultiplexing table and the ratio of packets inserting new entries in the map, and the quantities are very hard to estimate *a priori*.

7.3 Web Traffic Collection Overhead

As a final evaluation we use our HTTP reconstruction example. The tasks performed are only 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 timeout idle connections (with a timeout value of 255 seconds). To compute the overhead, we take for reference the time needed to reorder TCP packets.

Results are shown in Table 6. It presents the cost of some realistic processing. On average, we require about 50 μs per packet. We see then that the layering cost (i.e. splitting into two steps what could be

Name	ANY	TCP	UDP	SYN
Packets	500000	500000	500000	500000
Size (MB)	288.9	268.0	181.2	29.6

Table 2: Characteristics of traces used for layering overhead evaluation.

Trace	SYN			TCP			WWW			WWW-PP		
Total Packets	500000			500000			500000			500000		
Processed Pkts	500000			334771			317572			363107		
Unique Connect. (%)	unknown			11.9			9.8			0.7		
Timeout Value (s)	2	30	255	2	30	255	2	30	255	2	30	255
Map Size (avg)	31	312	2486	780	5788	19736	47	307	1905	19	84	393
Insert/Total (%)	98.7	93.9	93.7	16.9	12.0	11.8	18.5	11.1	10.0	0.4	0.7	0.7

Table 3: Characteristics of traces used for demultiplexing evaluation. The SYN trace is a collection of connection establishment TCP packets. WWW-PP is a trace collected when repeatedly requesting the same document from a Web server with an unique client host. TCP and WWW are "real" traces of TCP and WWW traffic, respectively. The number of processed packets varies since we are discarding acknowledgment-only packets in this experiment. The *Map Size* line is the mean map size in which a new entry was to be inserted. The *Insert/Total* line indicates the ratio between the number of created entries in the map over the total number of processed packets.

Tim.	SYN	TCP	WWW	WWW-PP
2	13.22 (0.14%)	8.21 (0.15%)	7.98 (0.83%)	2.76 (0.65%)
30	18.40 (0.10%)	8.11 (0.20%)	7.54 (0.26%)	2.08 (0.40%)
255	20.3 (0.07%)	7.1 (0.26%)	9.1 (0.32%)	3.8 (0.83%)

Table 4: Evaluation of demultiplexing overhead. Numbers show the average overhead in μs per packet demultiplexed, for the trace and timeout value specified.

Name	WWW1	WWW2	WWW3	WWW-PP
Total Packets	500000	500000	500000	500000
Processed Packets	317572	315992	315826	363107
Unique Connections	31238	24443	21067	2422
Size (MB)	256.6	302.0	272.6	307.5
Requests	21155	21997	18636	119168

Table 5: Characteristics of traces used for HTTP extraction experiments.

Trace	WWW1	WWW2	WWW3	WWW-PP
Overhead	27.8 (0.17%)	63.0 (0.09%)	48.6 (0.11%)	51.1 (0.12%)

Table 6: Average overhead (in μs per packet) for HTTP extraction.

done in one) now represents only 0.52 % of the average time to process a packet. Differences between execution time mainly comes from the different average length of requests.

7.4 Web Traffic Collection Throughput

Given that HTTP reconstruction is in fact a real application, we also measure its global throughput. Compared to other tests, this involves an additional step: 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 components at an almost constant rate, which differs greatly from real network conditions.

Table 7 shows the results of these tests. This leads us to think that Pandora can cope with most medium bandwidth links (100 Mb/s), without suffering from too many packet losses. Indeed, except for the WWW-PP trace (where we spent nearly 35 % of our time in writing records), the throughput achieved represents three quarters of the maximum bandwidth. Yet, these results show that Pandora is not fast enough to monitor high bandwidth network, dedicated to HTTP (like ISP backbones).

8 Related Work

We borrow the stacking approach from several platforms, in various domains. Well known examples include Ficus [8], *x*-Kernel [9], Horus [17] and Ensemble [7]. To the best of our knowledge, it has never been used for network monitoring tools.

Some intrusion detection systems (IDS) have addressed similar issues. In particular, Network Flight Recorder [16] and Bro [14]. Such tools are used to detect network attacks in real-time, and allow fast response. They both split their processing of incoming packets into distinct stages, each being the responsibility of a specific component. This approach gives such tools the flexibility and the extensibility they require (one cannot know what kind of attack will have to be monitored in the future). They also provide an advanced configuration language to specialize the behavior of the engine. Compared to them, Pandora, uses finer-grained components, and its configuration relies more on the chaining of specialized components. IDS have a fixed-size stack because packet processing to be performed is well-known and is not likely to change. This allows better performance (we saw Section 7.2 that layering has still a cost), but prevents from using the tool for a different goal.

Among existing monitoring tools, `tcpdump` [11] was the first to be widely used. Its output (a line per packet received) is invaluable for simple network monitoring purpose. For complex cases, the software offers the possibility of dumping a trace of complete packet contents for off-line analysis. Yet, this approach may not be acceptable in 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.

Ian Goldberg's IPSE [6] is another recent approach to network monitoring: built as Linux kernel modules, it promises good performance. Yet, such construction does not allow portability and complicates the development of the tool itself. Unfortunately, this software has not been sufficiently documented, preventing us from further investigation.

Windmill [12] is the closest tool to Pandora and looks very similar in spirit to it. Windmill is a platform designed to evaluate protocol performance, via a set of dynamically loaded experiments. Its authors focused on large set of such experiments, each using possibly overlapping components to match their input packets. This lead them to develop their own packet filter (WPF) and to design their protocol modules to avoid redundancy among experiments (e.g. not reassembling twice IP fragments needed by two distinct experiments). Windmill implements protocol extraction through a set of modules statically chained to each other: the inner-most module — i.e. the higher protocol level — recursively calls lower layers, first to let them update their internal state, and then to ask for information they need. For example, the BGP module first lets the TCP module process the packet, and only then asks TCP whether the packet is carrying a FIN flag. In contrast to Pandora, Windmill does not use a "pure" stacking approach and thus perfectly illustrates benefits and drawbacks of both techniques. For example, with Pandora, adding or removing data encryption support, or changing of encryption algorithm is a simple matter of configuration that can be done at run time. Also, implementing an IPv6 module in Windmill can be tricky. On one hand, if we

Trace	WWW1	WWW2	WWW3	WWW-PP
Mb/s	88.3 (0.18%)	75.2 (0.19%)	75.1 (0.12%)	44.8 (0.12%)
reqs./s	868	653	612	2600

Table 7: Throughput of HTTP extraction.

use different interfaces from the IPv4 module, all transport-layer modules have to be updated. On the other hand, if the interface is the same, this prevents from using both IPv4 and IPv6 layers in the same program. However, Windmill should outperform Pandora when executing similar tasks.

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 [21] were one of the first attempts to construct such specialized tools. No other tool have achieved such complete on-line processing of packets. Unfortunately, their limited performance have lead the authors to give up their development, restricting them to only HTTP/1.0 transactions.

BLT [2, 3] is the more promising one. It is meant to provide on-line HTTP traces, related with lower-layers events (TCP aborted connections, packet loss rate or duplicated, etc.) 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 packet loss rate of less than 0.3%. Compared to Pandora, BLT claims better performance. However, unlike Pandora, BLT performs HTTP request/response matching (which is an essential stage in Web logging) off-line.

We regret that none of these recent monitoring tools (namely IPSE, Windmill and BLT) have been publicly released so far, which prevented us from further comparing them with our tool and design choices. This is also partly why we decided to develop Pandora from scratch.

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 monitoring 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 *true* on-line analysis of network protocols, up to the higher levels — including analysis beyond strict application level protocols, such as HTTP matching. The use of components 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 component with packets it does not expect. Last but not least, we must continue developing monitoring components 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! [15], 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.

Acknowledgments

We owe to thank the anonymous reviewers and our shepherds Vern Paxson and Greg Minshall for their helpful comments. Also thanks to Ian Piumarta and

Thomas Colcombet for their useful suggestions from the early beginning of this work.

Availability

The source code 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] Ramon Caceres, Cormac J. Sreenan, and J. E. van der Merwe. mmdump - a tool for monitoring multimedia usage on the internet. Technical Report TR 00.2.1, AT&T Labs-Research, February 2000. `http://www.research.att.com/~ramon/papers/mmdump.ps.gz`.

[2] 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`.

[3] 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`.

[4] 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`.

[5] 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`.

[6] 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`.

[7] 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`.

[8] 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`.

[9] 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`.

[10] Van Jacobson, Craig Leres, and Steven McCane. `libpcap`. software (latest release: version 0.4), June 1989. `ftp://ftp.ee.lbl.gov/libpcap.tar.Z`.

[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] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998. `ftp://ftp.ee.lbl.gov/papers/bro-usenix98-revised.ps.Z`.

[15] 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`.

[16] Marcus J. Ranum, Kent Landfield, Mike Stolarchuk, Mark Sienkiewicz, Andrew Lambeth, and Eric Wall. Implementing a generalized tool for network monitoring. In *Proceedings of the Eleventh Systems Administration Conference (LISA '97)*, San Diego, California, October 1997. `http://www.usenix.org/publications/library/proceedings/lisa97/full_papers/01.ranum/01.pdf`.

[17] 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`.

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

[19] 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`.

[20] 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`.

[21] 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`.