Pandora : un système de collecte de traces du trafic Web de communautés d'utilisateurs réparties

Simon Patarin

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Pandora : un système de collecte de traces du trafic Web de communautés d'utilisateurs réparties

Simon Patarin

No 3743

Juillet 1999

THÈME 1

ISSN 0249-6399

INRIA
ROCQUENCOURT

Pandora : un système de collecte de traces du trafic Web de communautés d'utilisateurs réparties

Simon Patarin *

Thème 1 — Réseaux et systèmes
Projet SOR

Rapport de recherche n 3743 — Juillet 1999 — 25 pages

Résumé : *Pandora* permet de collecter les informations nécessaires pour caractériser le trafic Web d'une communauté d'utilisateurs répartie. Les informations sont obtenues en reconstituant le trafic HTTP directement à partir des paquets réseau. Sur le plan architectural, *Pandora* est constitué de trois composants logiciels coopérants : un collecteur, un observateur et un coordinateur, qui peuvent être déployés en différents points du réseau. En interne, chaque composant est implémenté par une série de filtres. Cette architecture autorise une grande souplesse d'utilisation et de déploiement. Les traces fournies par *Pandora* donnent des informations détaillées sur les profils des utilisateurs, les serveurs, les documents accédés, le réseau et les caches. Elles peuvent être utilisées pour déterminer la politique de cache ou de réplication qui offre la meilleure qualité de service possible aux utilisateurs.

Mots-clé : trace, capture de paquets, cache, Web, caractérisation

(Abstract: pto)

* Simon.Patarin@inria.fr

Pandora: a System for Collecting Web Traffic Traces of Distributed User Communities

Abstract: *Pandora* permits to gather information that can be used to characterize the Web traffic of a distributed user community. This information is obtained by reconstructing HTTP traffic thanks to network packet sniffing. Concerning its architecture, *Pandora* is made of three cooperating software components: a collector, an observer and a coordinator, that can be deployed at different points on the network. Internally, each component is implemented as a series of filters. This architecture permits flexible use and deployment of the system. Traces provided by *Pandora* give detailed information about users' profiles, servers, accessed documents, network and cache behaviors. These traces may be used to determine the caching or replication policy that offers the best quality of service to users.

Key-words: trace, packet capture, cache, Web, characterization

1 Introduction

La popularité du Web conduit à un engorgement croissant du réseau Internet et à une surcharge des serveurs les plus populaires. Ceci a pour conséquence de réduire les performances du Web.

Une solution à ce problème consiste à répliquer les documents Web à proximité des utilisateurs, par exemple dans des caches ou des serveurs miroirs. En effet, satisfaire une partie des requêtes localement permet à la fois d'améliorer les temps d'accès aux documents et de diminuer la charge sur les réseaux et les serveurs Web. Pour définir des politiques de réplication et de cache efficaces pour une communauté d'utilisateurs donnée, on doit prendre en compte le profil d'accès de cette communauté et son évolution, ainsi que les caractéristiques du réseau et des ressources accédées (serveurs Web et documents). Afin de disposer de ces informations, on peut surveiller l'activité réseau de ce groupe – pendant une période de temps suffisamment longue – et collecter les données relatives au trafic Web (flux de requêtes), aux serveurs (accessibilité, performances), au réseau (fiabilité, performances) et aux documents (nature, évolution).

La collecte de ces informations n'est toutefois pas sans difficultés. Du fait de la répartition à grande échelle des utilisateurs et des ressources, la surveillance simultanée de ces différents acteurs et du réseau pose le problème du choix du point d'observation et de la cohérence des informations. Ainsi, l'utilisation de fichiers journaux – soit de serveurs, soit de caches – est inadaptée car elle ne fournit qu'une vision partielle (vision impossible à compléter du fait des problèmes de cohérence). Par ailleurs, il faut à tout prix minimiser l'impact du processus de collecte sur le système surveillé. En particulier, il faut préserver la qualité de service à laquelle les utilisateurs sont habitués (s'ils disposent déjà d'infrastructures de cache, par exemple). Il faut aussi être indépendant des logiciels utilisés et donc ne pas avoir à les remplacer ou à les modifier. Le volume des données à collecter pose aussi problème et il faut gérer au mieux la charge que ceci engendre. La vie privée et la confidentialité, enfin, doivent impérativement être respectées.

De nombreux auteurs se sont déjà penchés – et se penchent encore – sur la caractérisation du Web. Le groupe WCA[1] (*Web Characterization Activity*) du *World Wide Web Consortium* tente actuellement de fédérer ces différents travaux. Cependant, on observe que quelle que soit la technique employée pour collecter les informations nécessaires à la caractérisation (analyse de fichiers journaux, instrumentation de logiciels, capture de paquets), aucune ne prend en compte la présence des caches sur le réseau. Il faudrait donc désactiver ceux-ci le temps de la collecte, mais c'est précisément ce que nous voulons éviter.

Pandora, le système que nous proposons, utilise une technique de capture et d'analyse des paquets circulant sur le réseau pour répondre aux exigences et aux difficultés que nous venons d'évoquer. En reconstituant le trafic Web à partir des paquets bruts, nous obtenons toutes les informations que nous souhaitons sans influer aucunement sur l'environnement. Afin de contourner, entre autres, les problèmes posés par les caches, *Pandora* coordonne et synthétise les observations et les mesures de différents modules élémentaires, placés en différents points du réseau. La flexibilité, la réactivité de *Pandora* font de ce système un outil original et efficace pour la caractérisation du Web, et plus généralement pour la surveillance des réseaux.

Nous commencerons par rappeler brièvement le fonctionnement du protocole HTTP (section 2) ; puis, après avoir mentionné les différents travaux qui se rapprochent du sujet de notre étude (section 3), nous présenterons notre solution (section 4). Nous nous pencherons ensuite sur les expérimentations que nous avons menées (section 6) pour valider *Pandora*. Nous discuterons enfin des limitations de notre outil (section 5) avant de nous intéresser aux perspectives suscitées par ce travail (section 7).

1. `http://www.w3.org/WCA/`

2 Rappels sur le protocole HTTP

2.1 Description générale

Le protocole HTTP est le standard pour l'échange de documents sur le Web. Il repose sur un protocole de transport fiable (habituellement TCP). Il est fondé sur le modèle client/serveur : pour accéder à un document, le client envoie une «requête» au serveur qui lui renvoie une «réponse» contenant le document désiré ou un message d'erreur en cas d'échec. Une requête HTTP est composée de différents éléments : une méthode (GET, POST, HEAD, ...) qui spécifie comment on accède à la ressource, une URL qui l'identifie et des en-têtes qui donnent des renseignements sur le client lui-même (ses capacités) et sur la requête (date d'émission ou vérification de cohérence par exemple). La réponse comporte, quant à elle, un code de retour qui explicite le statut de la réponse (succès ou type d'erreur), et des en-têtes qui fournissent des informations sur le serveur, la réponse et des méta-données qui concernent le document retourné (taille, type MIME, durée de vie, par exemple).

2.2 Transition HTTP/1.0 vers HTTP/1.1

Dans la version 1.0 du protocole [2], encore largement répandue dans le monde, une nouvelle connexion TCP est créée pour chaque nouvelle requête. Ceci permet d'identifier facilement quelles réponses satisfont quelles requêtes (l'URL ne figurant pas dans la réponse), mais est relativement coûteux en termes d'allers-retours sur le réseau (l'établissement et la fermeture d'une connexion TCP en nécessitent deux et demi chacun) ainsi qu'en ressources système. De plus, les mécanismes de contrôle de flux (*slow-start* en particulier) ne sont efficaces que pour des connexions de plus longues durée. C'est, principalement, pour corriger cette limitation de HTTP/1.0 que HTTP/1.1 a vu le jour [8]. Cette dernière version du protocole autorise les connexions dites «persistantes», à savoir que plusieurs requêtes peuvent être émises sur la même connexion TCP. Ceci a l'avantage de réduire le nombre moyen d'allers-retours et de limiter le surcoût lié à la gestion des ouvertures et fermetures de connexions pour le système. Afin d'éviter encore quelques allers-retours inutiles, le protocole permet de faire des requêtes en *pipe-line*. En effet, pour une connexion persistante «simple» (non *pipe-linée*), le client doit attendre d'avoir reçu la réponse du serveur avant d'envoyer une nouvelle requête ; ici, en revanche, le client peut envoyer toutes ses requêtes simultanément.

3 État de l'art

Différentes approches ont été utilisées pour collecter des traces Web: la première, et la plus simple, s'appuie sur les fichiers journaux générés par les serveurs ou les caches (section 3.1). Nous verrons ensuite (section 3.2) que certains auteurs ont préféré instrumenter les logiciels lorsque ceux-ci n'offraient pas de facilité de journalisation. L'analyse des paquets bruts, capturés sur le réseau a également inspiré de nombreux travaux (section 3.3). Enfin, nous insisterons (section 3.4) sur la nécessité de respecter la vie privée lorsqu'on collecte ce type d'informations, et sur les différents moyens mis en œuvre pour y parvenir.

3.1 Fichiers journaux

3.1.1 Journaux de serveurs

L'analyse des fichiers journaux des serveurs a été utilisée pour de nombreuses études [2]. En effet, tous les serveurs Web peuvent être configurés pour enregistrer les requêtes qui leur sont destinées, le format des fichiers du serveur *httpd* de NCSA s'étant imposé comme standard. Parmi ces fichiers, celui qui rend compte des accès au serveur (*access.log*) est particulièrement digne d'intérêt [3].

2. Nous ne discuterons cependant pas de celles qui se consacrent exclusivement à l'évaluation ou à l'optimisation spécifique de serveurs Web, dans la mesure où leurs intérêts divergent nécessairement des nôtres.

3. Il contient le nom de la machine émettrice de la requête, la date à laquelle celle-ci a été reçue, la requête elle-même (désignant l'URL accédée), le code de réponse du serveur et le nombre d'octets transférés. Apache a créé

Jeffrey Mogul [14] et Martin Arlitt et Carey Williamson [1] se proposent de caractériser le trafic Web dans son ensemble à partir de fichiers d'accès à des serveurs. Deux inconvénients majeurs sont inhérents à cette méthode : le premier est le peu d'informations disponibles dans ces fichiers. Tandis que Martin Arlitt se contente de s'en plaindre (il déplore notamment l'absence de données sur le temps nécessaire au traitement de la requête, la taille réelle des documents accédés, ou bien la nature – humaine ou automatique – des clients), Jeffrey Mogul a préféré instrumenter les serveurs dont il avait la charge afin d'enregistrer les renseignements qui lui manquaient (temps de traitement, entrées/sorties disques, charge du processeur). Le second inconvénient est lié à la position même du serveur dans le réseau. Comme le soulignent les deux auteurs, un serveur est incapable de détecter la présence de caches entre lui et certains clients : ces caches en satisfaisant des requêtes en amont des serveurs leur masquent une partie du trafic. Le développement des caches depuis la date à laquelle remontent ces études accentue encore ce biais dans les traces collectées de la sorte.

Cette approche semble donc inadaptée à nos objectifs : étant donné le nombre de serveurs Web existant à l'heure actuelle, il paraît illusoire d'envisager caractériser le trafic d'une communauté d'utilisateurs particulière, même si l'on disposait des fichiers des serveurs les plus populaires. Ceci est, en soi, une hypothèse aventureuse dans la mesure où ces fichiers constituent la plupart du temps, pour les serveurs commerciaux, une information confidentielle.

3.1.2 Journaux de caches

Les caches, en raison de la position qu'ils occupent vis-à-vis d'une communauté d'utilisateurs (fédérant leur trafic Web), sont particulièrement intéressants pour notre étude. Tout comme les serveurs Web, de tels logiciels ont été conçus dès l'origine pour journaliser leur activité à des fins d'administration. Par ailleurs, les traces qu'ils collectent ne sont évidemment pas biaisées par leur propre présence, contrairement aux serveurs ou aux clients Web.

Il est donc naturel que de tels fichiers journaux aient été utilisés dans de nombreux travaux ; parmi eux, ceux de Bradley Duska et al. [5] et de Jeffrey Mogul [15] sont les plus significatifs et les plus proches de notre sujet. Bien évidemment, l'analyse des fichiers de cache n'est pas encore la panacée. Comme le remarque Duska, de telles mesures ne sont possibles que si un nombre significatif d'utilisateurs passe par le cache ; Jeffrey Mogul ne se pose pas ce problème car, dans son cas, le cache est également un pare-feu (*firewall*). Cependant, les informations obtenues ne sont pas non plus parfaitement satisfaisantes, ce qui a (une fois de plus) poussé Mogul à instrumenter le logiciel (nous détaillerons dans la section suivante tous les problèmes que pose l'instrumentation des logiciels). Il est aussi à noter que les requêtes satisfaites par un cache situé en amont de celui étudié (en particulier ceux des clients) biaise quelque peu la collecte.

3.1.3 Remarques

L'analyse de fichiers journaux est assez tentante au premier abord car on pourrait croire que le travail de collecte est déjà fait en grande partie. Cependant, même s'il est vrai que serveurs et caches ont été conçus pour journaliser leur activité, les informations contenues dans ces journaux ne sont pas forcément ni tout à fait pertinentes (journaux initialement destinés à l'administration, pas à la caractérisation du trafic), ni tout à fait complètes (requêtes masquées par les caches en amont). Ceci a poussé certains auteurs à instrumenter ces programmes afin qu'ils fournissent les informations manquantes ; nous verrons que cette approche n'en demeure pas moins très limitée.

3.2 Instrumentation

Le lieu idéal pour observer le trafic Web d'un groupe d'utilisateurs semble être le logiciel de navigation lui-même. Malheureusement, contrairement aux deux types de logiciel précédents, les

un format étendu qui augmente le format précédent du nom du logiciel client et de l'URL de provenance lorsque ces informations sont disponibles.

clients ne sont pas prévus pour journaliser leur activité ; il faut donc les instrumenter. Ceci implique nécessairement de disposer du code source et explique pourquoi les études utilisant des traces collectées au niveau des clients remontent pour la plupart à des expérimentations menées en 1994/95, époque où le navigateur *XMosaic* (dont le code source est ouvert) était parmi les plus utilisés. Les temps ont changé, mais les politiques des éditeurs de logiciel aussi : les sources des deux navigateurs plus populaires actuellement *Netscape Communicator* et *Microsoft Explorer* sont désormais disponibles. On peut donc s'attendre à voir réapparaître de telles études prochainement. L'avantage incomparable que procure cette méthode est l'accès à des données autrement inaccessibles comme l'interaction de l'utilisateur avec l'interface graphique (boutons, signets), ou bien l'utilisation par le navigateur de son propre cache (lorsqu'il en dispose).

De telles expérimentations sont tout naturellement orientées vers la capture du comportement d'une communauté d'utilisateurs spécifique, ce qui tend à les rapprocher de notre problématique, aussi peut-on s'intéresser aux études de Carlos Cunha et al. [4] et à celles de Laura Catledge et James Pitkow [3]. L'instrumentation des logiciels elle-même est un obstacle, du fait du travail nécessaire et des précautions à prendre pour ne pas ajouter d'erreurs au code original. De plus, ce travail d'instrumentation est à recommencer à chaque évolution du logiciel ou lorsque les préférences des utilisateurs se tournent vers un autre navigateur. Ceci limite notablement l'utilisation d'une telle méthode sur des périodes de l'ordre d'une année, et interdit de ce fait de caractériser l'évolution d'une même communauté au cours du temps. Ce facteur humain exclut également la possibilité de conduire de telles mesures à grande échelle, auprès d'un public non nécessairement spécialiste, comme le sont, par exemple, les clients d'un fournisseur d'accès à Internet grand public.

3.3 Capture de paquets

La capture de paquets consiste à analyser tous les paquets de données circulant sur un réseau ou un sous-réseau donné. Ceci se fait au moyen d'une carte réseau placée en mode *promiscuous* sur un réseau à diffusion, ou en se situant au niveau d'un routeur. Cette technique permet de reconstituer l'intégralité du trafic d'un groupe d'utilisateurs, de manière totalement transparente (c'est-à-dire qu'elle ne nécessite aucune intervention de l'utilisateur et ne dépend d'aucun logiciel). Ceci est un avantage indéniable pour un déploiement à relativement grande échelle : les utilisateurs n'ont pas besoin d'être experts, l'environnement peut être hétérogène et les logiciels utilisés inconnus. Les renseignements obtenus par cette méthode sont très complets : ils concernent à la fois les informations contenues dans les protocoles réseaux ainsi que ceux extraits, dans notre cas, des entêtes HTTP, et on peut y ajouter les messages ICMP décrivant l'état du réseau lui-même; seules manquent les données concernant le client comme celles décrites à la section précédente (cache du client, actions de l'utilisateur).

Différents auteurs ont choisi cette méthode pour collecter des traces, les projets les plus aboutis sont *Httpfilt* et *Httpdump* de Roland Wooster et al., *WebWatcher* de Tommy Johnson et al. et *PacketScope* de Anja Feldmann. Après avoir discuté les principales difficultés de l'extraction HTTP à partir des paquets bruts, nous allons étudier dans le détail ces différentes solutions. Nous récapitulerons enfin les différentes fonctionnalités de celles-ci.

3.3.1 Difficultés de l'extraction HTTP

La reconstitution du trafic par l'analyse des paquets bruts soulève des difficultés qui conduisent bien souvent à développer une véritable implémentation du protocole cible. Cette attitude exclusivement passive – on se contente d'observer le trafic – complique la tâche puisqu'on ne peut que constater les pertes éventuelles de paquets. Par ailleurs, il faut jouer avec les différentes implémentations des protocoles, plus ou moins conformes aux normes.

Les problèmes soulevés peuvent être discriminés suivant la couche à laquelle ils appartiennent:

Couche IP Les paquets peuvent être fragmentés et les fragments arriver dans le désordre ; il faut donc pouvoir les réassembler lorsque cela se produit. Certains pouvant être perdus par l'application destinatrice ou rejetés par un routeur situé après le point d'observation, la phase

de réassemblage doit être capable de gérer des paquets dupliqués ou refragmentés de manière différente. Enfin, il faut être à même de détecter qu'un fragment a été perdu pour l'application et qu'il convient de traiter le paquet incomplet. Il est à noter que ce cas de figure est relativement peu fréquent dans la mesure où presque toutes les mises en œuvre modernes de la couche TCP ajustent la taille des paquets transmis à la couche IP de manière à éviter cette fragmentation coûteuse. Cela n'est cependant pas obligatoire, et la fragmentation peut intervenir au niveau d'un routeur intermédiaire.

Couche TCP On retrouve ici beaucoup des problèmes de la couche précédente : les paquets peuvent être désordonnés, dupliqués, redécoupés. De plus, contrairement à un paquet IP fragmenté dont la taille est limitée à $2^{16} = 65536$ octets, un flux TCP peut avoir une taille arbitrairement grande. Il est donc souhaitable de pouvoir traiter l'information aussitôt qu'elle est disponible (c'est-à-dire commencer à traiter les paquets qui arrivent sans attendre la fin de la connexion). Là encore, il faut pouvoir détecter qu'un paquet a été perdu, et passer outre.

Couche HTTP Les difficultés proviennent ici, pour la plupart, de la variété des interprétations du protocole HTTP par les différents logiciels et des ajouts récents de HTTP/1.1 qui, pour rester compatibles avec la version précédente encore majoritaire, sont parfois difficiles à gérer. En effet, la possibilité d'effectuer des requêtes sur une connexion persistante, voire de les *pipeliner*, rend délicate la délimitation des messages (il n'existe pas de marque de fin de message, on ne peut pas toujours se fier à la taille annoncée par le serveur dans les en-têtes car elle est parfois erronée ; on peut, de plus, perdre un paquet intermédiaire de taille inconnue). Il faut en conséquence analyser l'intégralité du contenu des paquets TCP car le découpage ne respecte pas nécessairement les requêtes. Par ailleurs, la présence de plusieurs requêtes HTTP dans la même connexion TCP complique la mise en correspondance des requêtes avec les réponses.

Il faut souligner le fait qu'extraire des informations à partir des paquets bruts réclame davantage de ressources (CPU, mémoire vive), que l'utilisation de fichiers journaux.

3.3.2 *Httpfilt*

Httpfilt [20] a le mérite d'être la première tentative de reconstruction du trafic Web à partir des paquets bruts. Cet outil ne fait que filtrer, au moyen de scripts *Perl*, la sortie du programme `tcpdump`[4] [10]. Ces filtres, comme indiqué sur la figure 1, convertissent dans un premier temps la sortie binaire de `tcpdump` puis, éliminent tous les paquets qui ne s'apparentent pas au début d'un en-tête HTTP (si le paquet commence par les caractères `HTTP`, `GET`, `POST` ou `PUT`, il est considéré comme tel et conservé). Le dernier filtre se charge de combiner les en-têtes des clients et des

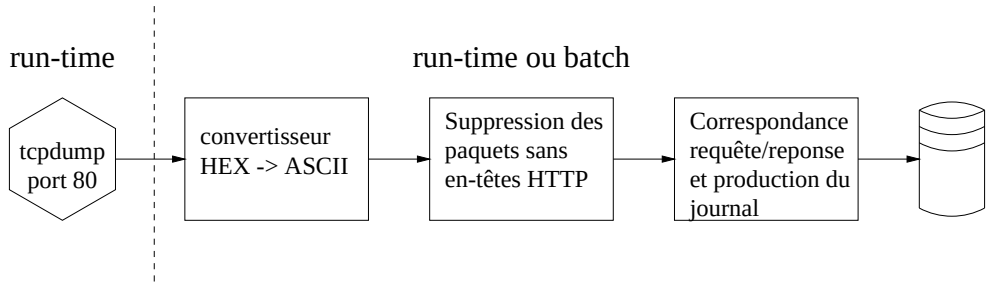

FIG. 1 – *Schéma de fonctionnement de Httpfilt.*

serveurs, et de produire un fichier journal au format des fichiers de serveurs (voir 3.1.1) auquel

4. Cet outil de surveillance permet d'obtenir à l'écran une description des paquets circulant sur le réseau ; il offre aussi la possibilité de sauvegarder dans un fichier binaire les premiers octets de tous ces paquets.

sont ajoutées les informations découlant de l'analyse HTTP (type du client, proxies traversés, type du serveur, type MIME du document, URL de provenance, date de dernière modification du document, date de la requête en millisecondes, temps de latence et temps de retrait du document).

L'inconvénient majeur de cette approche est qu'elle repose sur le protocole HTTP/1.0, en particulier sur le fait qu'une nouvelle connexion est initiée pour chaque nouvelle requête. Les auteurs soulignent eux-mêmes que sans cet aspect du protocole, ils seraient incapables de faire correspondre les requêtes avec les réponses. Ils oublient de mentionner également que dans HTTP/1.1 un en-tête ne commence pas nécessairement au début d'un paquet, auquel cas il faut analyser le contenu du paquet entier et le test choisi (trop souple) ne conviendrait pas. Par ailleurs, il se peut qu'un en-tête soit morcelé sur plusieurs paquets, ce qui limite davantage la généralité de leur outil parce qu'il risque de manquer des champs intéressants dans les paquets suivants.

Malgré ces quelques limitations, ce travail est à apprécier pour son originalité (ces auteurs furent en effet les premiers à utiliser la capture de paquets pour analyser le trafic Web) et pour les difficultés qu'ils mettent en avant, consciemment ou pas.

3.3.3 *Httpdump*

La même équipe que celle de *Httpfilt* (voir 3.3.2), trouvant que leur outil n'était pas assez performant, décida de développer le même programme mais en utilisant directement les facilités fournies par la bibliothèque de capture de paquets de Digital UNIX (sur laquelle s'appuie `tcpdump`). Elle choisit également d'employer les processus légers (*threads*) afin de gérer les communications simultanées. De ce fait, les deux premiers filtres sont remplacés par un unique programme en `C` qui capture les paquets et élimine ceux qui ne concernent pas le trafic Web ; ces paquets sont ensuite transmis à un second programme en `C` qui se charge du reste du travail : identification des sessions HTTP et calcul des informations destinées à former le fichier journal. Ici, chaque nouvelle connexion est attribuée à un processus léger parmi ceux disponibles dans un ensemble préalloué. Cette structuration de l'outil est présentée figure 2. Les auteurs n'ayant pu intégrer au sein d'un

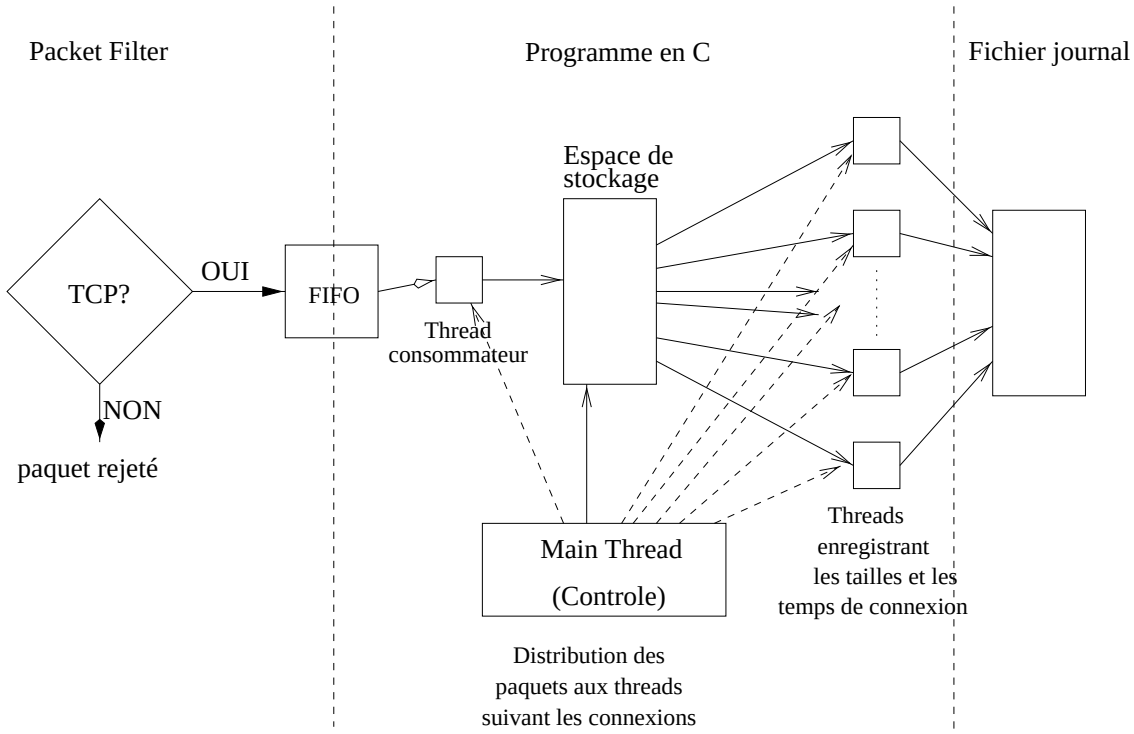

FIG. 2 – *Schéma de fonctionnement de Httpdump.*

même programme la bibliothèque de capture de paquets et la bibliothèque de processus légers, ils ont créé deux programmes disjoints et les ont fait communiquer au moyen d'une file d'attente FIFO.

Httpdump observe tous les ports (et non plus uniquement le port 80), malheureusement des mesures de performance ont montré que *Httpfilt* était bien meilleur (en termes de taux de pertes de paquets). Même si les problèmes que nous évoquions pour ce dernier restent valables (essentiellement l'exploitation d'une limitation de HTTP/1.0 – la création de nouvelles connections pour chaque transfert de document, corrigée dans HTTP/1.1), la démarche consistant à manipuler la bibliothèque de capture de paquets et les processus légers paraît louable : seules quelques contingences mineures ont limité les performances (machine peu puissante, incompatibilité de bibliothèques, volonté d'observer tous les ports). Le fait de vouloir traiter les paquets au fur et à mesure de leur arrivée est resté sans réelle comparaison par la suite, au profit de post-traitements des données.

3.3.4 *WebWatcher*

Développé au sein de la même institution (*Virginia Tech.*) que les deux outils précédents, *WebWatcher* [11] tire son originalité de son aspect distribué. En effet comme le montre la figure 3, l'outil utilise un modèle client/serveur pour synthétiser les collectes effectuées en différents points d'un réseau. L'outil de collecte n'a rien de particulier, il s'inspire de *Httpdump* et capture les

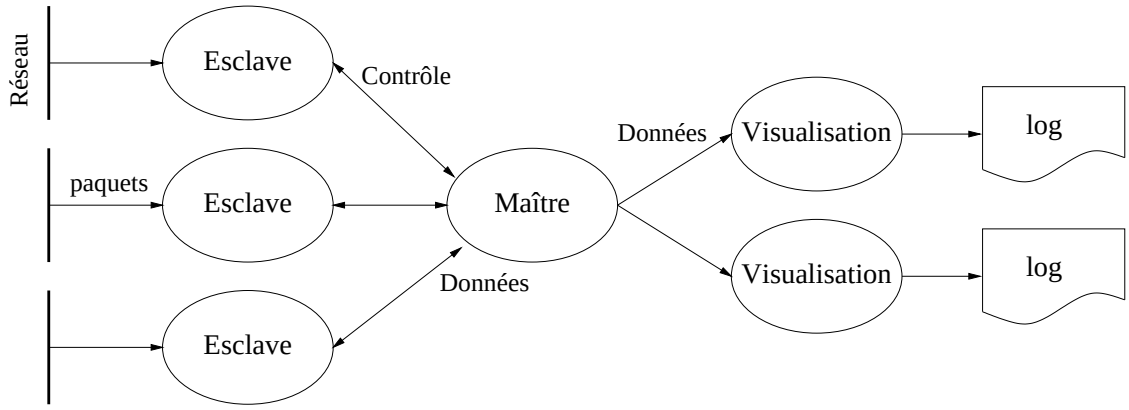

FIG. 3 – *Schéma de fonctionnement de WebWatcher.*

paquets sans l'intermédiaire de `tcpdump`. Le système `inetd` est utilisé pour lancer l'exécution des esclaves à distance. Le maître peut parfaitement contrôler ses esclaves, les configurant, testant s'ils sont encore en vie et les relançant au besoin. L'idée qu'un même ensemble de données peut produire des fichiers journaux différents (traduite par de multiples modules de «visualisation») ou être interprété graphiquement selon les cas est aussi séduisante, dans la mesure où cela permet de réutiliser des outils d'analyse préexistants, mais conçus pour d'autres logiciels (typiquement des outils d'audit de serveurs Web ou de proxies).

Ce logiciel souffre toujours de l'incompatibilité avec HTTP/1.1 qui rend son utilisation limitée. De plus, en développement depuis deux ans, ce projet n'a toujours pas été finalisé.

3.3.5 *PacketScope*

PacketScope [6] est le plus récent des outils de capture de paquets. Il a été conçu pour être particulièrement robuste à la charge, et la compatibilité avec le protocole HTTP/1.1 fait partie de ses objectifs. Il s'appuie également sur `tcpdump` pour capturer les paquets ; puis par une succession de filtres, contrôlés par un script *Perl*, il effectue le travail de reconstruction du trafic HTTP, pour finir par la mise en correspondance des requêtes avec les réponses (voir figure 4). La prise

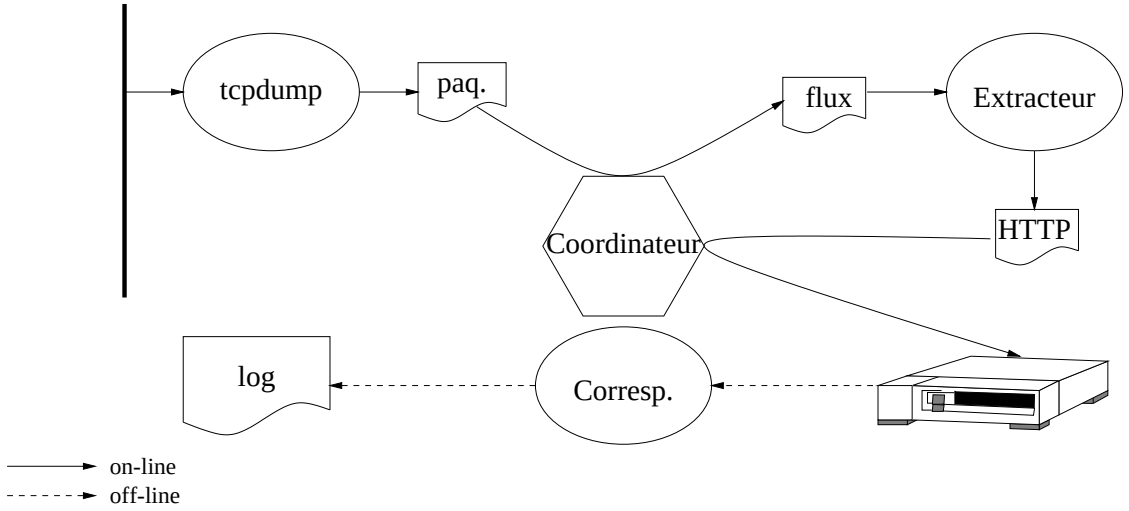

FIG. 4 – *Schéma de fonctionnement de PacketScope.*

en considération du protocole HTTP/1.1 est une avancée importante, même si elle complique la reconstruction du trafic à tous les niveaux. Par ailleurs, cet outil tient compte aussi des événements liés aux protocoles réseaux (comme le sont les débuts ou les fins de communication TCP/IP) qui fournissent souvent des informations utiles voire nécessaires pour expliquer certains phénomènes observables au niveau HTTP.

Cependant, le programme n'a pas été diffusé par ses auteurs ce qui rend sa réutilisation impossible.

3.3.6 Autres réalisations

Plusieurs autres travaux ont utilisé les paquets bruts pour reconstituer le trafic Web, mais leurs caractéristiques et les informations qu'ils collectent les rapprochent de ceux que nous avons déjà présentés dans les sections précédentes. Nous pouvons cependant citer Bruce Mah [12] et Steven Gribble [9]. Le premier a collecté le trafic d'un laboratoire de recherche afin d'en modéliser les propriétés statistiques, tandis que le second a analysé le trafic d'un accès à Internet offert aux étudiants et au personnel de l'Université de Berkeley pour émettre des considérations sur la conception de services sur Internet.

3.3.7 Synthèse

Nous proposons dans le tableau 1 un résumé des différentes caractéristiques des outils que nous avons présentés afin de permettre une comparaison plus aisée.

	Httpfilt	*Httpdump*	*Web-Watcher*	*Packet-Scope*	Mah	Gribble
Compatibilité HTTP/1.1	–	–	–	√	–	–
Extraction à la volée	√	√	√	–	–	√
Adaptation à la charge	∼	–	–	√	∼	–
Collecte distribuée	–	–	√	–	–	–
Directives de caches	∼	∼	–	√	–	√
Événements réseau	–	–	–	√	–	–
Disponibilité du logiciel	–	√	√	–	√	√

Légende : √ : oui – : non ∼ : partiellement.

TAB. 1 – *Récapitulatif des caractéristiques des différents outils de reconstitution de trafic Web à partir de capture de paquets.*

3.4 Confidentialité

Dans toutes les approches que nous venons d'évoquer, respecter la vie privée et la confidentialité des observations est une nécessité. Parmi les données collectées, les informations sensibles sont les adresses IP des clients (navigateurs) et des serveurs, ainsi que les URLs accédées. En effet, l'adresse IP d'un client peut permettre de remonter à l'identité de l'utilisateur, tandis que l'adresse IP du serveur et l'URL indiquent souvent la nature des ressources consultées.

Si les auteurs sont unanimes lorsqu'il s'agit de diffuser des traces vers l'extérieur, tous ne garantissent pas l'anonymat quand ces traces sont réservées à un usage interne. C'est le cas notamment de Roland Wooster et al. [20] et de Jeffrey Mogul [15] qui conservent les informations «en clair» pour leurs analyses. La méthode utilisée pour masquer les champs sensibles est de les remplacer par des nombres qui préservent l'unicité des valeurs (deux champs identiques auront le même code). Lorsqu'il est décidé que l'information doit être cryptée au cours de l'exécution du programme, les méthodes varient : *PacketScope* [6] utilise une fonction de hachage à sens unique, tandis que *WebWatcher* [11] utilise une fonction de cryptage à clé publique. On peut remarquer, par ailleurs, qu'aucune des techniques utilisées ici ne préserve d'information sur l'arborescence des ressources (une fois codées, il est impossible de déterminer si deux URLs appartiennent à la même sous-arborescence d'un serveur), information qui pourrait être utile dans le cadre d'études sur la réplication partielle de sites.

Se pose également le problème de l'information des utilisateurs concernés par les observations effectuées. Bien souvent seule une annonce par le biais de media internes est envisageable, cependant Laura Catledge et James Pitkow [3] proposent une démarche originale : ayant instrumenté le navigateur *XMosaic*, ils ont fait en sorte qu'une boîte de dialogue apparaisse, lors de la première utilisation du logiciel, demandant à l'utilisateur s'il acceptait que son trafic soit observé. Cette idée n'est cependant pas aisément généralisable aux autres approches.

4 Pandora

Nous avons choisi d'utiliser la méthode de collecte de traces à partir de la capture de paquets sur le réseau car c'est la seule qui permette de répondre à nos exigences. Avant d'aller plus loin, il

convient de préciser quelles sont exactement les informations que nous enregistrons : en plus des données sur les requêtes elles-mêmes (adresses IP du client et du serveur, URL accédée), nous recueillons les méta-données concernant les documents, comme leur taille, leur date de dernière modification ou leur type MIME. Les informations contenues dans le protocole HTTP qui concernent les caches (les en-têtes *If-Modified-Since*, *If-Unmodified-Since*, *Pragma* et *Cache-Control*) sont également collectées, ainsi que celles liées à l'environnement réseau (comme les estampilles qui marquent la réception des paquets par le système, les paquets ICMP qui décrivent l'état du réseau, les paquets TCP comportant les drapeaux SYN, FIN ou RST qui décrivent l'état des connexions et les quantités de données effectivement reçues ou envoyées sur ces connexions).

Après avoir détaillé l'architecture de notre système (section 4.1), et ses différents modes de déploiement suivant l'application cible (section 4.2), nous nous pencherons (section 4.3) sur le respect de la vie privée, avant de décrire plus précisément et techniquement (section 4.4) la mise en œuvre du système.

4.1 Architecture du système de collecte

4.1.1 Composants du système

Collecteur C'est l'élément du système qui se charge de la capture des paquets à proprement parler. Il fournit en sortie un fichier journal contenant la trace de toutes les requêtes HTTP qu'il a vues passer, mises en correspondance avec leur réponse. Le journal peut aussi bien être écrit directement sur le disque ou transmis par le réseau à un coordinateur (voir plus bas). On obtient également un fichier contenant les différents événements liés aux connexions TCP (début, fin, abandon), ainsi qu'un relevé des paquets ICMP. Les fichiers journaux sont produits par le collecteur sous forme binaire, pour des raisons d'encombrement. Un outil de lecture permet par la suite de produire des fichiers au format souhaité (reprenant, ou pas, des formats classiques, comme ceux du serveur Web NCSA).

Observateur Il apparaît que nous collectons parfois des informations inutiles : en effet, si seul nous intéresse le fait de savoir quelles requêtes ont été effectuées, il n'est pas besoin de renseigner complètement la trace collectée. Il suffit en réalité de disposer pour chaque requête, de l'URL accédée, de l'adresse IP source, et des estampilles. C'est un collecteur allégé, que nous appelons observateur, qui fournira de telles informations.

Coordinateur Le coordinateur, comme son nom l'indique, coordonne et synthétise les observations de plusieurs «mandataires» (coordinateurs, collecteurs, ou observateurs). Dans son rôle de coordination il est amené à diriger l'exécution des mandataires et à récupérer tous les enregistrements de requêtes qu'ils lui envoient ; dans son rôle de synthèse, il produit son propre fichier journal. Ce travail de synthèse dépend du contexte dans lequel il est utilisé, aussi y reviendrons-nous plus en détail sections 4.2 et 4.4.4.

4.1.2 Architecture interne des composants : la notion de filtre

La chaîne de traitement aboutissant à la création d'un enregistrement dans le fichier journal se décompose en un certain nombre de phases, qui correspondent à autant d'étapes dans l'outil lui-même. Nous avons choisi d'utiliser une architecture en couche, qui reprend à peu près la structuration de la pile des protocoles analysés. D'un point de vue général, une couche peut être considérée comme un opérateur qui transforme un flux de paquets d'un certain type en un flux de paquets d'un type différent. Dans cette description, il faut bien insister sur la notion de «flux». En effet, il est souvent impossible de traiter un paquet indépendamment des autres, ce qui peut conduire le filtre à accumuler plusieurs paquets avant de pouvoir les traiter. De plus la nature des protocoles réseaux est telle qu'il faut parfois effectuer de nombreux traitements avant de déterminer qu'un paquet est inacceptable pour la suite ; ainsi un filtre, quelle que soit sa position dans la pile, peut être amené à rejeter certains paquets.

Prenons l'exemple d'un filtre qui assurerait la transformation d'un flux IP en un flux TCP. L'en-tête IP contient un champ qui explicite le protocole de transport utilisé (ICMP, UDP, TCP, etc.). Le filtre rejettera donc tous les paquets non TCP. Cependant, il peut arriver qu'un paquet TCP soit fragmenté, auquel cas, il faut réassembler les différents fragments afin de former le paquet original. Cette dernière étape suppose donc que le filtre stocke temporairement les différents fragments jusqu'à ce qu'ils soient tous arrivés, et éventuellement soit prêt à les réordonner s'ils arrivent en ordre dispersé, ou bien à éliminer les duplicats le cas échéant.

Le seul effet de bord qu'on accepte d'un filtre est la production d'enregistrements pour la journalisation. Ainsi dans l'exemple précédent, le filtre peut être configuré de manière à ce qu'il écrive dans un fichier journal les paquets ICMP rencontrés, avant de les rejeter.

Cette conception rend l'outil flexible et adapté à d'autres applications que celles pour lesquelles il fut développé dans le domaine de la surveillance des réseaux. En effet, cette composition d'opérateurs permet d'envisager l'analyse de tout protocole pourvu qu'on développe les filtres nécessaires. Comme les protocoles réseaux restent peu nombreux, seule la partie supérieure de la pile aurait besoin d'être modifiée. Par ailleurs, cette flexibilité se traduit aussi en terme d'équilibrage de charge : on peut répartir les filtres sur plusieurs processeurs ou machines de manière aisée, ou bien affecter une priorité différente à chacun d'entre eux suivant la nature de leurs opérations. On peut enfin imaginer de leur imposer des contraintes temps réel lorsque le système d'exploitation le permet.

4.2 Configurations du système pour les applications cibles

4.2.1 Reconstitution du trafic Web en l'absence de cache

Lorsqu'il n'y a pas de cache dans le réseau, deux cas peuvent se présenter. S'il n'y a qu'une sortie du réseau vers l'extérieur, un unique collecteur placé à ce point suffit à reconstituer le trafic. Si, en revanche, il existe plusieurs sorties, il faudra placer un collecteur à chacune de celles-ci et synthétiser ces différentes observations au moyen d'un coordinateur (voir figure 5).

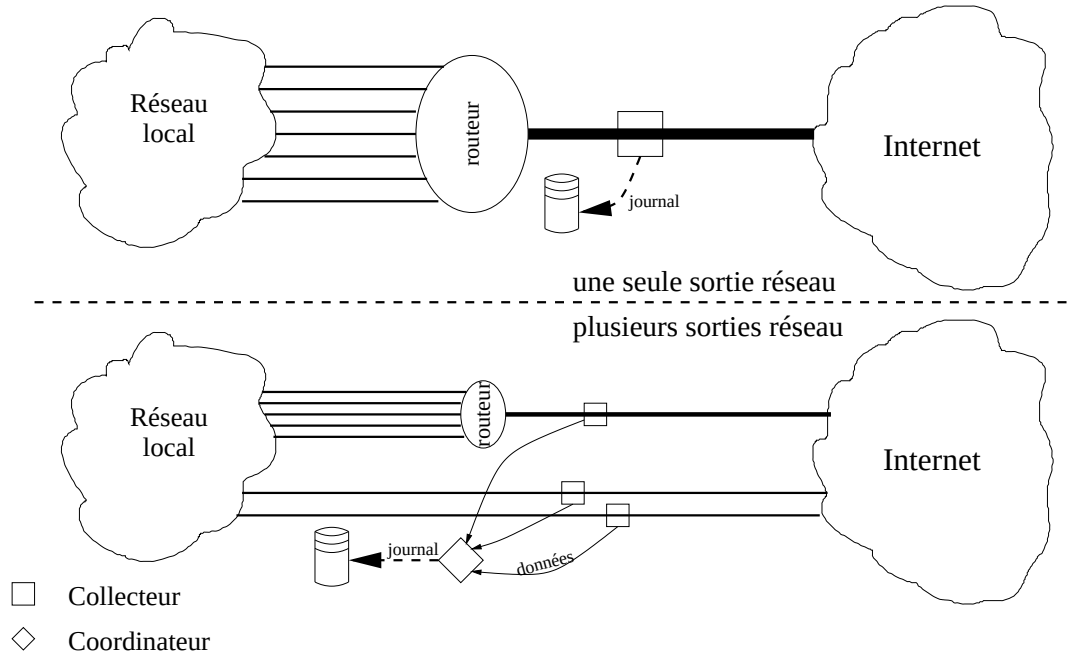

FIG. 5 – *Déploiement et configuration du système pour la reconstitution du trafic avec une seule sortie réseau (en haut), ou plusieurs (en bas).*

Il est à souligner que les points de mesure sont définis en termes de «connectivité», et ne présagent en rien des machines sur lesquelles les outils eux-mêmes sont exécutés. En effet, on peut imaginer une machine disposant de plusieurs interfaces réseaux et sur laquelle deux collecteurs seront à même de fonctionner simultanément.

Dans cette configuration, le travail de synthèse du coordinateur est minimal : il se contente d'unifier les observations des différents collecteurs afin de produire un fichier journal équivalent à celui qu'aurait fourni un collecteur seul si les différentes sorties du réseau avaient été réunies.

4.2.2 Reconstitution du trafic en présence de caches

Pour contourner le biais induit par les caches, il convient de disposer de points de mesure de part et d'autre de chacun des systèmes de cache, et ce afin d'isoler l'activité d'un cache. Il n'est cependant pas nécessaire que ces points (respectivement, en amont, et en aval) soient uniques (le système peut avoir plusieurs entrées et sorties).

On remarque néanmoins que, dans ce cas précis, les deux collecteurs situés autour du même cache observent des informations redondantes si le cache ne satisfait pas la requête. C'est donc une situation où l'observateur peut être utilisé : placé avant le cache, il ne fait que signaler les requêtes qu'il voit passer à un coordinateur. Le coordinateur reconstitue alors les requêtes en prenant dans le flux amont leurs estampilles, l'adresse du client et les en-têtes HTTP de la requête ; dans le flux aval, fourni par un collecteur, il récupère les informations liées à la réponse.

Un coordinateur différent doit être utilisé pour chaque système de cache, on peut ensuite unifier les résultats de ces coordinateurs grâce à un nouveau coordinateur qu'on pourrait qualifier de «second niveau». On aboutit ainsi dans le cas général à la configuration présentée figure 6.

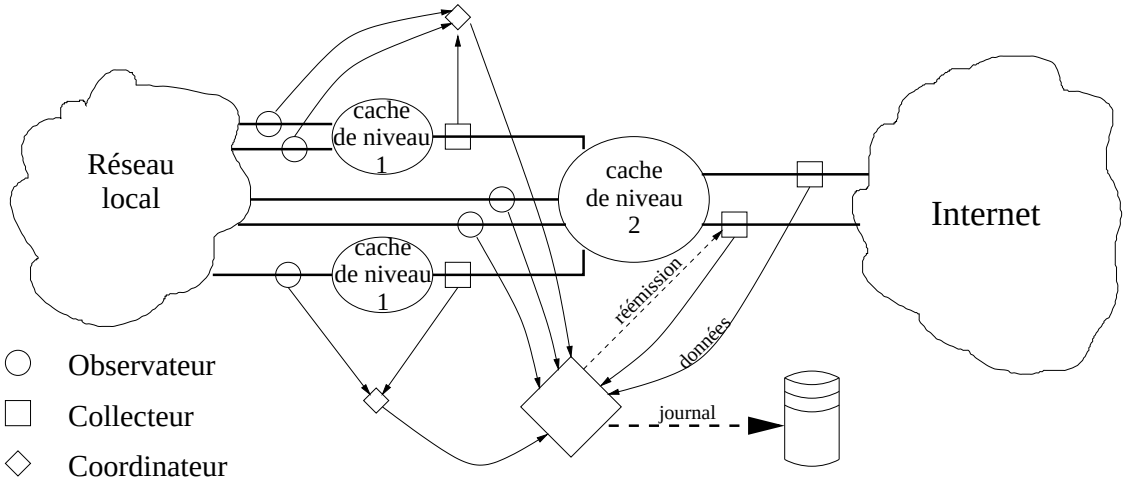

FIG. 6 – *Configuration des collecteurs et des observateurs pour la reconstruction du trafic HTTP en présence d'un système de cache.*

Le fichier journal fourni par un coordinateur de «premier niveau» dans ce cas de figure est équivalent à celui qu'on aurait eu s'il n'y avait pas eu de cache à cet endroit. Pour y parvenir, le coordinateur doit faire correspondre les requêtes vues avant le cache avec celles vues après lui. Il se fonde sur les URLs transmises par les observateurs et les collecteurs pour effectuer cette étape. Au cas où une requête est satisfaite par le cache (lorsque le coordinateur ne voit pas de requête en aval du cache correspondant à une requête en amont), il réémet lui-même auprès du serveur d'origine de l'URL une requête HEAD afin d'obtenir les informations manquantes, requête dans laquelle il remplace l'adresse IP source par celle du client réel.

4.2.3 Reconstitution du trafic et évaluation d'un système de cache

Lorsqu'on veut évaluer un système de cache, les informations données par les observateurs, en cas de succès de celui-ci, ne sont plus suffisantes (on cherche ici à mesurer les *modifications* du trafic induites par les caches). Ce sont donc bien des collecteurs qu'il faut, cette fois, placer autour du cache. Un coordinateur est toujours présent pour synthétiser les observations, et les recouper.

Ici, le comportement du coordinateur est identique au cas précédent (section 4.2.2) – mise en correspondance, réémission des requêtes satisfaites par le cache – mais le fichier journal produit est différent : chaque enregistrement est constitué d'un couple de traces : l'une observée en amont du cache et l'autre observée en aval. À partir de ce fichier on pourra à la fois reconstituer le trafic et analyser le comportement des caches (nous verrons comment section 7.2).

4.3 Confidentialité

4.3.1 Objectifs et contraintes

Au cours du processus de collecte, nous souhaitons masquer les données sensibles, qui touchent à la vie privée. En l'occurence, il s'agit des adresses IP des clients (navigateurs) et des serveurs, ainsi que des URLs accédées. En effet, l'adresse IP d'un client peut permettre de remonter à l'identité de l'utilisateur, tandis que l'adresse IP du serveur et l'URL indiquent souvent la nature des ressources consultées. Le contenu des documents n'est, quant à lui, jamais collecté, ni même analysé.

Ce brouillage conduit toutefois à une perte d'information. Afin de pouvoir utiliser ces données pour des études concernant la réplication de sites Web, il faudra conserver la hiérarchie des URLs (pouvoir dire si deux documents appartiennent à la même sous-arborescence d'un serveur). Pour caractériser l'évolution du trafic et des ressources, il serait également souhaitable de pouvoir comparer les résultats de différentes collectes : il convient donc de conserver l'unicité des données au cours du temps.

4.3.2 Mesures envisagées

Dans le but de garantir le maximum de confidentialité aux utilisateurs du réseau, *Pandora* met en œuvre les trois mesures suivantes:

1. Les informations considérées sensibles seront brouillées par le biais d'une fonction de hachage à sens unique, cryptographiquement sûre. Ceci garantit aux utilisateurs que personne – y compris nous-mêmes – ne pourra détourner ces données à des fins malveillantes, dans la mesure où il sera impossible d'inverser ce processus pour obtenir les informations originales (on parle alors d'anonymisation). Ce brouillage interviendra au plus tôt dans l'exécution du programme, en particulier avant tout enregistrement des données sur support de stockage stable (disque ou bande).
2. Au terme de la collecte, une correspondance entre les signatures (résultant du passage par la fonction de hachage) et les entiers sera établie[5]. Ceci afin de se prémunir contre une faiblesse cachée dans la fonction de hachage utilisée qui pourrait être découverte dans l'avenir.
3. Si malgré ces garanties une personne souhaite être exclue des fichiers journaux, il lui suffit de communiquer l'adresse IP de sa machine. Le système de collecte écartera des traces observées les paquets en provenance ou à destination de cette machine.

4.3.3 Détails de l'algorithme

Nous voulons obtenir une signature de longueur fixe pour chaque URL, tout en nous laissant la possibilité d'identifier les documents appartenant à une même sous-arborescence du serveur jusqu'à une profondeur fixée. Nous décrivons ici l'algorithme que nous avons utilisé. Dans la suite, H désigne la fonction de hachage à sens unique et K une clé secrète.

5. Pour une sécurité maximale, la table de correspondance peut être détruite aussitôt. Ceci empêcherait néanmoins de comparer entre eux les résultats de différentes campagnes de collecte.

Pour anonymiser une adresse IP, adr, nous calculons:

$$adr' = H(K \oplus adr), \text{ où } \oplus \text{ désigne l'opérateur de concaténation}$$

Pour anonymiser une URL, url, de la forme $http://host/p_1/\ldots/p_n$, nous calculons:

$$url' = H'(url) = \bigoplus_{0 \leq i \leq k} url'_i, \text{ où } \begin{cases} url'_0 = H(K \oplus http://host/) & \\ url'_i = H(K \oplus http://host/\ldots/p_i) & \text{si } i < \min(n,k) \\ url'_i = H(K \oplus url \oplus (\oplus^{i-n} K)) & \text{si } n \leq i < k \\ url'_k = H(K \oplus url) & \text{si } n \geq k \\ url'_k = H(K \oplus url \oplus (\oplus^{k-n} K)) & \text{si } n < k \end{cases}$$

On peut noter ici que url'_0 identifie systématiquement le serveur, tandis que url'_k identifie l'URL dans sa totalité.

Nous avons choisi d'utiliser MD5 (RSA Data Security, Inc. MD5 Message Digest Algorithm) [18] comme fonction de hachage. La clé K aura une longueur de 128 bits, et nous prendrons k égal à 7.

4.4 Implémentation

4.4.1 Mise en œuvre des filtres

Comme nous l'avons présenté section 4.1.2, un filtre peut être considéré comme un opérateur binaire sur des flux de paquets.

La principale structure de données d'un filtre est un arbre binaire équilibré (un arbre rouge et noir) de «listes» de paquets. Chacune de ces listes est constituée de paquets appartenant à la même entité logique (un paquet d'une couche protocolaire supérieure, par exemple). Chaque paquet traité par le filtre est inséré dans la liste à laquelle il appartient logiquement (si elle n'existe pas encore, elle est créée). Lorsqu'un paquet vient compléter une liste, les différents composants de la liste sont assemblés en un nouveau paquet et la liste est détruite.

Chaque filtre est exécuté au sein d'un processus léger distinct (et ce, afin d'amortir au mieux les pics de charge), et possède une file d'attente de type *FIFO* (premier entré, premier sorti) en entrée. Il suffit alors d'y insérer un paquet pour réveiller le processus léger, le cas échéant, et commencer le traitement.

La résistance aux pertes de paquets a de fortes implications. Étant donné les débits des réseaux actuels, il est utopique de penser pouvoir capturer l'intégralité des paquets, les pics de charge étant relativement fréquents. Nous nous sommes donc appuyés sur la notion de «flux TCP/IP» pour déterminer le moment où l'on décide de traiter un groupe de paquets incomplet. Un flux TCP/IP est défini comme étant un ensemble de paquets appartenant à une même connexion TCP/IP (mêmes adresses IP et mêmes ports source et destination), «proches dans le temps» : c'est-à-dire qu'on mesure le temps qui sépare la réception des différents paquets, et dès que celui-ci dépasse un seuil (choisi au préalable), le flux est considéré comme terminé. Nous étendons cette notion aux différents flux qui nous intéressent ici, lors de la reconstruction des paquets TCP fragmentés, ou de la mise en correspondance des requêtes et des réponses HTTP.

Suivant la notion de «flux», une liste est vidée au bout d'un certain temps d'inactivité. Plus précisément, lorsqu'une liste est créée, une minuterie est positionnée (pour un temps défini pour chaque filtre). Chaque fois qu'un paquet est inséré dans une liste, la minuterie de celle-ci est remise à zéro. Si cette minuterie vient à expirer, la liste est vidée et les ressources sont libérées. Ceci peut conduire, dans le meilleur des cas, à délivrer des paquets incomplets au filtre suivant, ou sinon à rejeter les paquets accumulés.

La plupart des mécanismes que nous venons de décrire n'ont pas à être manipulés directement pour chaque filtre : seule la gestion des listes de paquets est laissée sous la responsabilité du développeur du filtre. C'est en effet dans cette partie que se situe l'essentiel des spécificités du traitement. On dispose de deux classes de base génériques qu'il convient d'instancier (par le biais de patrons) pour obtenir le squelette d'un filtre. Le processus léger, l'arbre, la file d'attente en entrée et les minuteries y sont gérés.

4.4.2 Structure du collecteur

Pour la partie «capture de paquets» de notre collecteur, nous avons adapté le code de `tcpdump` [10]. Cet outil intègre en effet un module de compilation de filtres pour le Berkeley Packet Filter [13]qui est très appréciable. De plus, un mécanisme de *callbacks* permet d'effectuer la capture sur des types d'interface *a priori* quelconques, dans la mesure où seule une partie minime de code (une dizaine de lignes) suffit à extraire les paquets IP des paquets réseau[6]. Ceci constitue ce que nous pourrions appeler la «première couche» de notre logiciel : elle transforme les paquets de la couche liaison en paquets IP.

Intervient ensuite notre premier filtre qui effectue la transformation IP vers TCP. Cette couche se charge de réassembler les paquets fragmentés et de ne retenir que les paquets TCP. C'est dans cette couche que sont hachées les adresses IP et rejetés les paquets correspondants aux adresses IP à exclure (voir section 4.3). C'est encore ici que les paquets ICMP sont analysés pour produire les enregistrements correspondants.

La couche suivante est un peu atypique, et nous allons la décrire plus en détails. Elle transforme les paquets TCP en messages (ou paquets, par abus de langage) HTTP. Dans le protocole TCP, chaque octet d'un flux est numéroté afin de permettre le réordonnancement des paquets à l'arrivée et la détection des pertes [16]. Nous avons utilisé cette numérotation pour analyser les paquets aussitôt qu'ils forment une série cohérente.

La difficulté qui survient alors est que les paquets provenant d'une même connexion TCP peuvent être séparés par des paquets d'autres connexions, il faut donc pouvoir suspendre l'analyse d'un flux et la reprendre dans l'état où nous l'avons suspendue. La solution que nous avons adoptée consiste à créer un nouveau processus léger pour chaque nouvelle connexion, et à y effectuer l'analyse en son sein (voir la figure 7 pour un exemple).

En tout état de cause, les paquets HTTP extraits dans chaque processus léger par cette analyse lexicale et syntaxique (nous utilisons pour cela les outils `flex` et `bison`, qui seuls supportent les processus légers) sont placés dans l'unique file d'attente du filtre suivant. Un processus léger est donc détruit dès que nous détectons la fermeture de la connexion, ou lorsque la minuterie expire. Par ailleurs c'est dans ce filtre que sont produits les enregistrements liés aux événements réseau (début, fin ou réinitialisation d'une connexion).

Le troisième et dernier filtre a pour charge de mettre en correspondance les requêtes Web avec leur réponse. Cette tâche a ceci de délicat que le serveur ne mentionne pas dans sa réponse l'URL de la ressource. Il faut donc commencer par isoler les messages échangés sur une même connexion, différencier les requêtes et les réponses, puis associer deux à deux les messages dans leur ordre d'arrivée. C'est en effet la seule garantie du protocole HTTP/1.1 : les réponses doivent être délivrées en respectant l'ordre de réception des requêtes. Si nous nous apercevons que nous avons perdu au moins un paquet sur une connexion, cette technique ne fonctionne plus car nous pouvons avoir manqué un message complet (contenu à l'intérieur du ou des paquets manquants), nous nous appuyons alors sur des heuristiques faisant intervenir les numéros de séquence et d'accusé de réception TCP. Ce sont alors ces couples de messages qui constituent les traces Web des fichiers journaux.

4.4.3 Structure de l'observateur

L'observateur a pour objectif d'être plus léger que le collecteur, son travail étant beaucoup plus aisé. Nous reprenons intégralement la partie capture de paquets du collecteur (provenant elle-même de `tcpdump`) mais n'utilisons pas cette fois les filtres. Seule l'extraction des paquets IP est effectuée (afin d'obtenir les adresses IP source et destination), puis un analyseur lexical recherche les lignes de statut des requêtes HTTP[7]. L'URL accédée et la méthode en sont donc extraites pour former l'enregistrement réduit, destiné usuellement au coordinateur.

6. Pour l'instant seules les interfaces *Ethernet* et *loopback* sont implémentées.

7. Elles sont de la forme : `METHODE url HTTP/x.y CRLF`, où METHODE est l'un des mots `GET`, `HEAD`, etc., `HTTP/x.y` est le numéro de version de HTTP utilisé, et `CRLF` correspond aux caractères de retour chariot (habituellement, les caractères `\r\n`)

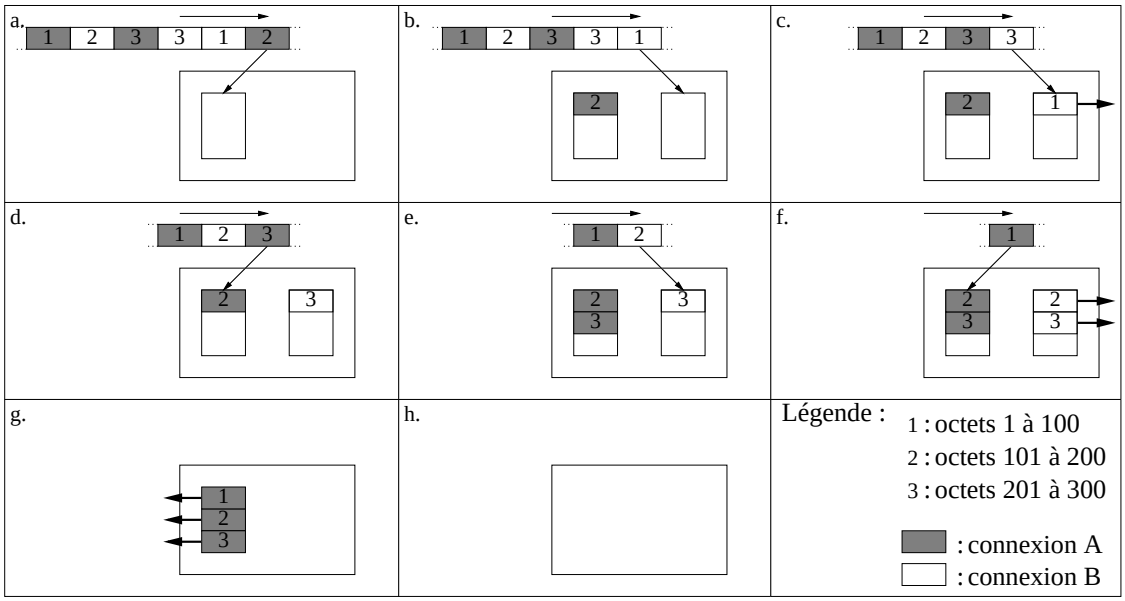

FIG. 7 – *Exemple de fonctionnement de l'analyse d'un flux TCP. Dans cet exemple nous considérons deux connexions A et B qui comportent chacune trois paquets qui arrivent dans l'ordre indiqué en (a). (a) une nouvelle connexion est détectée et une liste de paquets ainsi qu'un processus léger sont créés pour la traiter. (b) idem pour la connexion B ; le paquet 2 de la connexion A a été inséré. (c) le paquet 1 de la connexion B est analysé et sort donc de la liste. (d) le paquet 3 est inséré à sa place dans la liste correspondant à la connexion A. (e) le paquet 2 de la connexion B est inséré et l'analyse de la connexion B peut se poursuivre avec les paquets 2 et 3. (f) le paquet 1 de la connexion A est inséré en tête de la liste ; les paquets 2 et 3 de la connexion B sortent. (g) les trois paquets de la connexion A peuvent désormais être analysés ; le traitement de la connexion B est terminé, le processus et la liste ont été détruits. (h) idem pour la connexion A.*

Ce programme s'exécute ainsi au sein d'un unique processus léger et requiert bien moins de ressources système.

4.4.4 Structure du coordinateur

La première fonction du coordinateur est de lancer l'exécution des programmes qu'il doit coordonner. Celui-ci attribue à chacun un numéro d'identification. Lorsqu'un tel programme est lancé, il se connecte immédiatement au coordinateur en spécifiant ses caractéristiques : nature (collecteur/observateur ou coordinateur), position (logiquement avant ou après un cache), numéro d'identification. Si un des programmes venait à mourir, le coordinateur pourrait ainsi le relancer aussitôt.

Dans son rôle de synthèse (unifier les fichiers journaux de plusieurs collecteurs, par exemple), le coordinateur se contente d'écrire les messages qu'il reçoit au fur et à mesure de leur réception. Lorsqu'il s'agit de contourner ou d'évaluer un cache, un nouveau filtre est mis en place. Dans ce filtre, les listes de paquets correspondent aux requêtes ayant la même URL, discriminées suivant leur position par rapport au cache.

Pour établir les correspondances entre les requêtes vues en amont et celles vues en aval, le filtre ne se fonde que sur les URLs (et pas les adresses IP). En effet le cache réémettant lui-même la requête, aucune information supplémentaire n'est donnée par les adresses. Ceci peut conduire à certaines confusions dans le cas où des clients différents accèdent au même document, mais fonctionnellement il est équivalent d'attribuer un succès du cache à tel ou tel client en particulier.

Par ailleurs, afin de récupérer les renseignements qui manquent pour caractériser le comportement du cache et l'état du réseau, lorsque le coordinateur détecte un succès du cache il réémet une requête vers le serveur d'origine. Cette nouvelle requête doit être vue par un collecteur afin qu'il l'analyse et la retransmette au coordinateur, un champ personnel ajouté aux en-têtes HTTP lui permettra de distinguer ce type de requêtes. Le coordinateur maintient par ailleurs son propre cache de données afin de limiter le nombre d'émissions de requêtes identiques dans un court intervalle de temps. Ce cache est paramétré de sorte que les données ne soient pas conservées suffisamment longtemps pour éviter que des questions d'incohérence interviennent (les entrées du cache sont invalidées au bout de quelques dizaines de secondes, au pire quelques minutes).

4.4.5 Portabilité du code

L'environnement matériel et logiciel de déploiement est volontairement souple, afin de profiter pleinement des libertés accordées par la méthode de capture des paquets. Ainsi la topologie du réseau peut être arbitraire pourvu que l'on sache identifier, et que l'on ait accès, à un ensemble de «points de mesure» qui regroupent l'ensemble du trafic à analyser.

Dans un souci de déploiement aisé, *Pandora*, écrit en C++, a été porté pour les systèmes Digital UNIX, Solaris et Linux. Toutefois, seul le noyau de Digital UNIX de DEC intègre un module, appelé *packet filter*, qui permet de réduire considérablement la charge du système en filtrant au sein du noyau les paquets intéressants : l'économie de ressources (tampons, CPU) ainsi réalisée minimise le nombre de paquets perdus. Les paramètres restants n'influent pas sur notre étude : que ce soient les types de logiciels utilisés comme clients, caches ou serveurs, ou bien le niveau de qualification des utilisateurs (chercheurs ou grand public).

5 Limitations

Qu'elles soient liées à la méthode de capture de paquets (section 5.1) ou bien à notre mise en œuvre (section 5.2), plusieurs limites sont à souligner.

5.1 Limites inhérentes à la méthode

Malgré les avantages incomparables en termes de souplesse et d'adaptabilité que la technique de surveillance du réseau procure, celle-ci n'est pas sans inconvénient.

5.1.1 Pertes de paquets

La limite majeure de cette méthode est le risque permanent de perte de paquets qui peut conduire à manquer complètement des requêtes. Deux questions se posent alors : la première est de savoir comment cet état de fait va évoluer avec la technique, et la seconde est de se demander si cela peut être négligé. Il semble que la comparaison entre l'évolution des vitesses des réseaux et celles des ordinateurs tourne à l'avantage de ces derniers, et donc qu'avec le temps nous devrions subir de moins en moins cet inconvénient. La seconde question est plus problématique. En effet si un taux de perte de moins de 0,1% dans des conditions normales d'utilisation, peut paraître bien peu de chose (en termes de statistiques), le fait que ces pertes n'interviendront qu'au moment des pics de charge peut avoir un effet non négligeable pour l'analyse qui sera faite de ces traces. Il convient donc de regarder d'un œil sévère les pertes de paquets, et tenter à tout prix de les minimiser.

5.1.2 Informations manquantes

Toutes les informations dont nous pourrions avoir besoin ne sont pas collectées : les requêtes satisfaites par le cache du logiciel client nous sont inaccessibles. On peut néanmoins considérer ceci comme faisant partie intégrante des caractéristiques du trafic observé en aval des navigateurs. Étant donné que les applications pour lesquelles nous souhaitons utiliser les traces se situent toutes à ce niveau, ce problème a relativement peu de conséquences ; notre position serait tout autre si l'on étudiait, par exemple, les habitudes de navigation des utilisateurs, d'un point de vue plus sociologique.

En revanche, se pose la question de savoir quels ports observer. Pour des raisons d'efficacité, on évite d'observer la totalité du trafic circulant sur le réseau[8]. Ceci nous conduit inévitablement à manquer un certain nombre de requêtes dont il faudrait apprécier le poids statistique. Encore une fois, il s'agit de trouver le bon compromis entre précision et performance.

5.2 Limites inhérentes à notre implémentation

5.2.1 Processus légers

L'utilisation même des processus légers pourrait être contestée, au profit d'un modèle événementiel, mais cette dernière solution doit être rejetée car le noyau gère la capture des paquets par interruption et il ne convient pas de le concurrencer à ce niveau.

Cependant, comme nous l'avons décrit section 4.4, un processus léger est créé chaque fois qu'une nouvelle connexion commence. Même si relativement peu d'entre eux sont actifs simultanément, l'ordonnancement de ces processus légers, et le coût lié à leur création et à leur destruction peuvent être problématiques pendant les périodes critiques que sont les périodes de charge. La solution consistant à créer un *pool* de processus légers pourrait être envisagée, mais il paraît délicat de se priver de la possibilité d'en créer ponctuellement un grand nombre lorsque la charge le nécessite. Par contre, une technique permettant d'augmenter par blocs un tel *pool* semble meilleure. Il n'est pas évident de savoir s'il convient de libérer les blocs supplémentaires après une période d'inactivité, dans un souci d'économie, ou s'il vaut mieux les conserver, dans un souci de précaution.

5.2.2 Recopies

La gestion de la charge et donc des performances de l'outil est un point clé comme nous venons de le souligner, car ce sont d'elles que dépendent les pertes éventuelles de paquets. En effet, ces pertes interviennent lorsque le noyau a épuisé les tampons dont il dispose. Pour éviter ceci, il faut traiter au plus vite les paquets pour libérer les ressources du système.

Le nombre de recopies des tampons est un paramètre important dans l'optimisation d'un programme et nous comptons malheureusement deux recopies inutiles du contenu des paquets. En

8. On limite ainsi considérablement la charge de travail en mettant à profit les filtres de bas niveau de la bibliothèque de capture de paquets.

effet, la bibliothèque de capture de paquets utilisant un grand tampon circulaire, il faut recopier le contenu de ce tampon dans un des nôtres si l'on veut pouvoir rendre la main à la bibliothèque avant la fin du traitement d'un paquet – ce pourquoi nous nous servons des processus légers. Pour résoudre ce problème, il faudrait modifier la bibliothèque de manière à ce qu'elle nous fournisse un nouveau tampon à chaque appel, cependant ces modifications ne seraient pas portables et nous avons préféré les remettre à plus tard. D'autre part, nous recopions une fois de plus le contenu des paquets lorsque nous les donnons à l'analyseur lexical. Ceci est dû au fait que l'implémentation de l'analyseur n'est pas encore tout à fait finalisée ; il n'est pas possible en l'état actuel d'éviter cette recopie.

6 Évaluation des performances

Nous avons mené une série de tests afin de mesurer le comportement de notre outil en termes de résistance à la charge. Après avoir présenté l'environnement matériel et logiciel que nous avons installé pour conduire ceux-ci (section 6.1), nous détaillerons les résultats que nous avons obtenus (section 6.2).

6.1 Environnement de test

Notre objectif est de mesurer le taux de perte de requêtes subi par le collecteur. Le collecteur n'est certes pas le seul composant logiciel de *Pandora*, néanmoins, il apparaît que c'est lui le point critique de tout le système. En effet, le coordinateur ne fait qu'interpréter et analyser les données fournies par des éléments en amont : il n'y a pas de risque de perte d'information. Quant à l'observateur, il doit être, par essence, plus performant que le collecteur ; il ne saurait en aucun cas l'être moins, puisqu'un collecteur est aussi un observateur! Ceci explique que nous focalisons nos efforts à l'évaluation des performances du collecteur.

Pour nos tests, nous avons installé un serveur Web Apache et un client sur deux machines partageant un bus *Ethernet* 10BaseT. Le collecteur est placé sur une troisième machine connectée au même bus.

La machine exécutant le serveur Web est une *AlphaWorkstation 500*, munie d'un processeur 21164 cadencé à 333MHz. Le client et le collecteur sont exécutés sur des *Personnal WorkStations* disposant du processeur 21164A, cadencé à 500 MHz.

Le client émet des requêtes, aussi vite que le réseau le permet, pour demander un même document (de 1622 octets) au serveur ; le nombre de connexions simultanées, ainsi que leur mode (persistantes ou non) est paramétrable. Chaque test dure 3 minutes et est séparé du suivant par 30 secondes d'inactivité pour permettre au collecteur de terminer les opérations en attente. Le collecteur est démarré une fois pour toutes au début de la série de tests ; il est configuré pour écrire ses fichiers journaux sur un disque dur local.

6.2 Résultats et discussion

Le tableau 2 présente les résultats de nos expérimentations[9]. Nous avons mené entre 20 et

	Connexions non persistantes			Connexions persistantes			
Connexions	8	16	32	8	16	32	64
Nb. de tests	20	25	25	25	25	25	24
Requêtes	899 896	1 167 934	1 245 701	1 678 030	1 629 477	1 708 376	1 806 712
Débit (req/s)	249,9	259,5	276,8	388,4	362,1	379,6	418,2
Tx. de perte	0,11%	0,12%	0,12%	0,03%	0,01%	0,01%	0,06%

TAB. 2 – *Résultats des expérimentations en termes de pertes de paquets.*

9. Nous n'avons pas pu faire de test avec 64 connexions simultanées non persistantes, le serveur ne le supportant pas.

25 tests pour chaque configuration. Nous observons ici que les débits moyens (en requêtes par seconde) s'échelonnent pour les connexions non persistantes entre 250 et 280, tandis que pour les connexions persistantes ils sont compris entre 360 et 420.

Les taux de pertes moyens que nous avons rencontrés sont, pour les connexions non persistantes, de l'ordre de 0,12%, et pour les connexions non persistantes de 0,01% pour 16 et 32 connexions simultanées, 0,03% pour 8 connexions simultanées et 0,06% pour 64 connexions simultanées [10]. Il apparaît donc, au premier abord, que nous supportons moins bien les connexions non persistantes. En effet, malgré des débits plus faibles, le nombre de requêtes que rate le collecteur est plus important. Pour expliquer ce phénomène, nous avons mesuré le nombre moyen de paquets par requête dans chacun des cas : celui-ci est d'environ 11 dans le cas des connexions non persistantes [11], contre 4 dans le cas des connexions persistantes. Ces dernières échangent donc en réalité deux fois moins de paquets. Ceci, combiné au fait qu'une nouvelle connexion réclame un traitement plus coûteux que la simple insertion d'un paquet dans une liste existante, contribue à expliquer les moins bonnes performances du logiciel dans le cas des connexions non persistantes.

En comparant les taux de pertes pour différents nombres de connexions simultanées, nous nous apercevons que ce facteur a une influence limitée, ce qui tend à confirmer l'hypothèse que le nombre de processus légers à ordonnancer concurrentiellement ne croît pas avec leur nombre total. En effet, les paquets étant délivrés séquentiellement par un unique filtre, le nombre de processus légers en train d'analyser des connexions concurrentiellement est borné par le rapport entre le temps maximum d'analyse d'un paquet et le temps minimum d'insertion d'un paquet dans une liste.

Malgré un réseau peu performant comparé aux technologies actuelles, les résultats que nous avons obtenus sont prometteurs. En effet, les débits auxquels nous avons soumis *Pandora* s'inscrivent parmi les débits maximum que peuvent supporter la plupart des caches commerciaux (voir à ce sujet l'étude de Alex Rousskov et al. [19]) et correspondent aussi à ceux rencontrés lors d'études d'outils comparables au nôtre [7].

7 Perspectives

Ce travail est encore loin d'être achevé ; nous nous intéressons à présent aux perspectives qui s'offrent à nous : le prototype, tout d'abord, qui s'il est fonctionnel mérite néanmoins quelques améliorations (section 7.1). Nous présenterons également les applications que nous envisageons (section 7.2). Nous évoquerons enfin la généralisation souhaitée de cet outil dans un avenir proche (section 7.3).

7.1 Amélioration du prototype

Comme nous l'avons vu section 5.2, *Pandora* reste limité par certains aspects et beaucoup d'entre eux sont aisément corrigibles. Nous ne reviendrons pas sur ces points détaillés précédemment.

Par ailleurs, plusieurs améliorations doivent être apportées au prototype lui-même : parmi elles, les plus importantes seraient de faciliter la configuration du collecteur et de permettre une répartition des couches entre différents processus (pour l'équilibrage de charge).

Il faut encore modifier un peu l'outil afin d'en permettre un usage plus général, par exemple développer une interface de programmation claire et établir une hiérarchie de classe plus générale : la solution actuelle utilisant des patrons (*templates*) a l'unique avantage de la performance. Il

10. On peut également noter qu'environ 60% des tests effectuées se sont achevés sans aucune perte.

11. Dans un schéma de communication TCP classique, on s'attendrait plutôt à une valeur de l'ordre de 7 ou 8 – selon la position des accusés de réception. Une étude plus détaillée a montré que ceci s'expliquait par une proportion importante de réémissions – du fait de collisions *Ethernet*, et surtout par le fait que le client provoquait l'envoi de plusieurs paquets RESET lors des fermetures de connexions – nous n'avons pas pu déterminer s'il s'agissait d'un bogue au niveau du client ou à celui de TCP.

faudra également écrire plusieurs autres filtres (pour différents protocoles comme FDDI ou UDP) et sans doute développer une application permettant une exploitation plus aisée des fichiers journaux.

7.2 Exploitations des traces

Les traces collectées par *Pandora* sont précises et détaillées : les informations fournies concernent aussi bien les profils d'accès que les serveurs, le réseau, les documents ou les caches. De plus la comparaison de collectes espacées dans le temps permet d'appréhender l'évolution de ces différentes métriques.

Évaluation de politiques de cache et de réplication La principale tâche, ici, est en réalité d'isoler les métriques pertinentes. En ce qui concerne, par exemple, la conception et le développement de sites miroirs, il serait intéressant de voir si l'on peut isoler des seuils à partir desquels une réplication s'avère utile, en fonction de critères propres à une communauté d'utilisateurs. Ou bien, on peut vouloir vérifier pour une communauté particulière l'hypothèse qui fonde l'utilisation de caches : à savoir qu'un nombre conséquent des accès se fait vers des serveurs qui ont une latence importante ou, lorsque la connexion réseau est coûteuse, que de nombreux documents sont accédés plusieurs fois à intervalles de temps rapprochés. À partir de ces analyses, de nouvelles politiques pourront être élaborées, puis testées au moyen de simulateurs (tel que *Saperlipopette!* [17]). On peut aussi imaginer concevoir des logiciels auto-adaptatifs qui seraient à même de modifier leur configuration en fonction des caractéristiques du trafic rencontré.

Mesure des performances d'un système de cache Pour évaluer le comportement externe d'un système de cache, on peut mesurer les gains en nombre d'octets transférés et en temps de retrait des documents induits par le cache, ainsi que le taux de succès de ce dernier ; il convient également de vérifier la cohérence des documents délivrés par le cache. Toutes ces informations sont fournies par *Pandora* : il suffit de les recueillir.

7.3 Déploiement à grande échelle

Une fois le prototype dans un état stable, l'outil a vocation à être largement déployé. Nous envisageons deux axes : le premier est de rechercher activement des partenaires qui accepteraient de nous laisser mesurer leur trafic en contrepartie des améliorations que nous serions alors à même de leur apporter [12], le second est de diffuser publiquement l'outil avec l'espoir que les utilisateurs éventuels nous feront part de leurs résultats.

Ce déploiement est une étape indispensable dans la réalisation de nos objectifs à moyen terme. On ne saurait en effet prétendre analyser le profil des utilisateurs du Web sans en avoir recueilli au préalable le comportement d'un échantillon statistiquement intéressant. De plus la généralisation de telles mesures nous apparaît comme un critère clé dans l'évolution d'Internet, faute de quoi, nous serons incapables de prévoir l'évolution des usages sur le réseau et donc d'anticiper les maux à venir.

8 Conclusion

Nous avons présenté *Pandora*, un logiciel de collecte de trafic Web, par capture de paquets sur le réseau. Son architecture fondée sur la composition de filtres et la définition de composants logiciels coopérants, déployables en différents points du réseau, et la prise en considération des caches en font l'originalité. *Pandora* se distingue aussi des systèmes comparables par sa capacité à concilier le respect de la vie privée des utilisateurs et la nécessité de conserver les informations pertinentes pour les exploitations envisagées.

Les mesures de performances du prototype sont encourageantes : à des débits soutenus, les taux de pertes de requêtes que nous avons rencontrés sont relativement faibles. Concernant les

12. Des négociations avec différents établissements publics ou privés sont d'ores et déjà prévues ou en cours.

connexions persistantes, ces taux sont même tout à fait comparables à ceux expérimentés par les outils existants qui n'effectuaient cependant pas leurs traitements à la volée. Ceci nous laisse donc penser qu'il sera aisément utilisable dans le contexte que nous nous sommes fixé pour cible. Dans un futur proche, nous envisageons de raffiner encore le prototype pour continuer d'améliorer ses performances. Par ailleurs, des mesures plus agressives sont prévues afin de déterminer précisément les limites de *Pandora*.

Les perspectives qu'ouvre cette étude sont nombreuses et répondent à des préoccupations actuelles : la croissance de l'Internet réclame une caractérisation afin de prévoir son évolution et de maintenir une qualité de service acceptable. Les caches Web par ailleurs font maintenant partie intégrante de la plupart des réseaux locaux mais leurs réglages exigent souvent des informations plus fines que celles qu'ils fournissent par eux-mêmes. Notre outil fournit la matière première de ces futurs travaux mais il faut encore l'exploiter. Par ailleurs, la souplesse d'utilisation et de configuration de *Pandora* permet d'envisager des applications plus larges pour cet outil dans le domaine de la surveillance des réseaux, notamment dans l'évaluation de piles de protocoles.

Références

[1] Martin F. Arlitt and Carey L. Williamson. Trace-driven simulation of document caching strategies for Internet Web servers, September 1996. `http://www.cs.usask.ca/faculty/carey/papers/webcaching.ps`.

[2] Tim Berners-Lee, Robert Fielding, and Henrik Frystyk. Hypertext Transfer Protocol – HTTP/1.0. Request for Comments 1945, May 1996. `ftp://ftp.isi.edu/in-notes/rfc1945.txt`.

[3] Laura Catledge and James Pitkow. Characterizing browsing strategies in the World-Wide Web. In *Proceedings of the 3rd International World Wide Web Conference*, Darmstadt, Germany, 1995.

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

[5] Bradley M. Duska, David Marwood, and Michael J. Freeley. The measured access characteristics of World-Wide-Web client proxy caches. In *Proceedings of the 1997 Usenix Symposium on Internet Technologies and Systems (USITS-97)*, Monterey, CA, December 1997. `http://www.cs.ubc.ca/spider/marwood/Projects/SPA/wwwap.ps.gz`.

[6] 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`.

[7] 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`.

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

[9] 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`.

[10] Van Jacobson, Craig Leres, and Steven McCane. `tcpdump` software, (latest release: version 3.4), July 1998. `ftp://ftp.ee.lbl.gov/tcpdump.tar.Z`.

[11] Tommy Johnson, Jinzhao Lu, and Jinhuo Shan. WebWatcher: A distributed HTTP network monitoring system. Technical report, Network Research Group, Virginia Polytechnic Institute and State University, July 1997.

[12] Bruce A. Mah. An empirical model of HTTP network traffic. In *Proceedings of INFOCOMM '97*, Kobe, Japan, April 1997. `http://www.ca.sandia.gov/~bmah/Papers/Http-Infocom.ps`.

[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] Jeffrey C. Mogul. The case for persistent-connection HTTP. In *Proceedings of the SIGCOMM'95 conference*, Cambridge, MA, August 1995. `http://www.acm.org/sigcomm/sigcomm95/papers/mogul.html`.

[15] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM '97 Conference*, September 1997. `http://ftp.digital.com/~mogul/sigcomm97.ps`.

[16] Information Sciences Institute University of Southern California. Transmission Control Protocol. Request for Comments 793, September 1981. `ftp://ftp.isi.edu/in-notes/rfc793.txt`.

[17] Guillaume Pierre. *Architecture et dimensionnement d'infrastructures de caches Web pour Intranets décentralisés*. PhD thesis, Université d'Evry-Val d'Essonne, Evry (France), June 1999. `http://www-sor.inria.fr/publi/pierre_thesis99.html`.

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

[19] Alex Rousskov, Duane Wessels, and Glenn Chisholm. The first IRCache Web caching bakeoff. In *Proceedings of the fourth Web Caching Workshop*, San Diego, California, April 1999. `http://bakeoff.ircache.net/bakeoff-01/`.

[20] 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