國立交通大學

網路工程研究所

碩士論文

穩定流量重播至應用層代理伺服器

Stateful Traffic Replay on Application Proxies

研 究 生：廖鵬宇

指導教授：林盈達　教授

中 華 民 國 九 十 九 年 六 月

穩定流量重播至應用層代理伺服器

Stateful Traffic Replay on Application Proxies

研 究 生：廖鵬宇　　Student: Peng-Yu Liao

指導教授：林盈達　　Advisor: Dr. Ying-Dar Lin

國立交通大學

網路工程研究所

碩士論文

A Thesis

Submitted to Institutes of Network Engineering
College of Computer Science
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
In

Network Engineering

June 2010

HsinChu, Taiwan

中華民國九十九年六月

穩定流量重播至應用層代理伺服器

學生：廖鵬宇　　指導教授：林盈達

國立交通大學資訊科學與工程研究所

摘要

重播流量到網路產品並觀察它們的反應是一種測試網路產品的方法。目前已存在的重播工具大多提供傳輸層的流量重播，以測試交換器、路由器、閘道器等產品。然而，現有的流量重播工具無法針對應用層代理伺服器網路產品進行重播。有鑑於此，基於應用層代理伺服器具有建立連線與修改應用層訊息的特性，我們在本論文設計並實作一個適用於應用層通訊協定的 ProxyReplay 重播工具。代理伺服器常常具有一些網路流量處理功能，如內容快取與過濾。這些行為可能會改變重播應用層流量的基本流程，導致當內容快取與過濾行為發生時，重播工具送出不必要的訊息，或是訊息無法被正常處理。ProxyReplay 這個工具按照應用層協定規範的流程準確重播流量至應用層代理伺服器，它可以根據不同網路功能進行處理，如更新訊息標頭檔讓訊息正確快取，檢查要求訊息再重播對應的回應訊息，防止內容過濾發生時送出不必要的訊息。最後，為了避免重播時所帶來記憶體空間的限制，ProxyReplay 也提供同步重播的機制，以處理比實際記憶體空間還要大的流量資料，在實驗結果中，ProxyReplay 可達到 200Mbps 以上的輸出效能，並重播大於實際記憶體空間的真實流量。

關鍵字：流量重播、流量錄製、真實流量、應用層代理伺服器

Stateful Traffic Replay on Application Proxies

Student: Peng-Yu Liao **Advisor: Dr. Ying-Dar Lin**

Department of Computer Science

National Chiao Tung University

Abstract

It is common to test network devices by replaying network traffic to them and observe their reactions. Most existing replay tools support layer 4 stateful traffic replay and hence they can be used to test switch, router, and gateway devices. However, they do not support stateful traffic replay to application level proxies. Therefore, considering the characteristics of connection interception and content modification, we design and implement the ProxyReplay tool for application layer proxies in this thesis. Application proxies often provide features such as data-caching and content filtering. These features may affect replay procedures. As a result, it is possible that a replay tool replays invalid application messages that cannot be correctly processed by proxies. The proposed tool not only supports stateful replay to application proxy by following the protocol procedure, it also handles different proxy behaviors. For instance, it updates message headers so that web-caching works correctly and it also analysis messages received from a proxy to prevent from replaying of invalid responses when content filtering is adopted. Although replayed data is usually preprocessed and stored in memory during a replay process, the ProxyReplay tool provides a concurrent replay mode which allows it to replay network traces that are much greater than the physical memory. In the experiment, the throughput of ProxyReplay can achieve 200Mbps but also replays the real flow trace file which is larger than the physical memory.

Keywords: traffic replay, traffic capture, real flows, application proxy

Content

List of Figures

List of Tables

Chapter 1 Introduction

It is common to test network devices with manually generated network traces. There are two major approaches to generate network traces manually. One is the model-based approach and another is trace-based approach. The model-based approach generates simulated network traces based on mathematical properties of analyzed real network traces. On the contrast, the trace-based approach uses traffic replay technologies to replay previously captured real network traces. There are limitations on model-based approaches. First, the correctness of simulated network traces depends on the mathematical model used to generate the traces. However, it is difficult to prove that a model is 100% correct. Second, a mathematical model is often created based on normal network traces. Hence it is also difficult to simulate abnormal conditions, especially when an abnormal has not been identified. Third, model-based approaches often focus only on numerical properties; it is hard to simulate behaviors involving application content. Finally, it is undoubted that a simulated network trace will never act exactly the same as real network traces. Therefore, if there are choices, users would prefer to test their devices using real network traces and seeing how the device under test acts in the real world.

Replaying network traces can be either stateless or stateful. A stateless traffic player replays network traces based only on timestamps. The content of replayed network packets is exactly the same as that stored in the captured network traces. On the contrast, a stateful traffic player is much more complicated. The content of replayed network packets may need to be altered to fit the network environment. For example, the TCPReplay tool 0supports traffic replaying at the data-link layer and the network layer. It may have to modify MAC addresses and IP addresses to make sure the replay can be fluently done successfully. The NATReplay tool supports stateful traffic replay for NAT devices. Hence, it must at least maintain the mapping state

between private IP addresses and public IP addresses. The SocketReplay tool [2] supports stateful traffic replay for layer 4 firewall devices. It must maintain firewall response states to prevent from replaying blocked connections.

Most existing replay tools only maintain layer 2, layer 3, or up to layer4 states [1-5]. Although researches have shown the demands on stateful application-layer replay, these tools failed on replaying for application-layer devices [6-8]. In order to design a replay tool which can accurately replay network traffic to application-level devices, four problems should be solved.

(1) Protocol Dependency: As mentioned above, connection should be established before sending application traffic. Hence, replay tool should be able to establish connections to application-level devices.

(2) Functional Dependency: Different proxies could have different behaviors, for example, caching or content filtering [9-11]. Therefore, a replay tool must handle these diversities.

(3) Concurrent Replay: Since a traffic trace may include multiple concurrent connections, a replay tool must be able to replay these connections simultaneously.

(4) Error Resistance: Captured network traces may be incomplete due to packet loss [13]. This is the most important part to guarantee the fluency of traffic replay.

ProxyReplay is the replay tool which is designed based on the solutions above four problems.

In this thesis, we propose a framework for replaying traffic for application-aware proxies. Based on the framework, a traffic replay tool is also implemented to show its effectiveness and efficiency. The rest of this thesis is organized as follows. The background of traffic capture, traffic replay, and related works are introduced in Chapter 2. A formal problem statement is made in Chapter 3. The design and the implementation of the proposed method are introduced in Chapter

4 and Chapter 5, respectively. Evaluation of the proposed solution is done in Chapter 6. Finally, a concluding remark and possible future directions are given in Chapter 7.

Chapter 2 Background

In this chapter, we illustrate the background knowledge about application replay. First, Section 2-1 gives an example of HTTP proxy to explain the failure of existing tools. Section 2-2 makes taxonomy of application level devices and introduces their characteristics. Section 2-3 illustrates the factors that will affect application replay accuracy. Last, Section 2-4 gives the related work about replay approaches in detail.

2-1 Failure Replay of Existing Tolls

Figure 1 (a) shows a regular HTTP scenario without a proxy. The web client first establishes a connection to the web server. Then, the web client sends an HTTP request via the connection. Finally, the web server sends an HTTP response back to the web client. Figure 1 (b) shows an HTTP scenario with the existence of a proxy. First, the web client establishes a connection to the proxy server instead of the web server. The web client then sends an HTTP request to the proxy server via the connection. On receipt of the request, the web proxy establishes another connection to the web server according to the URL in the HTTP request and then forwards the HTTP request to the web server. To send the response back to the client, the web server sends the HTTP response to the web proxy in step 3. Finally, the web proxy forwards the HTTP response to the web client.

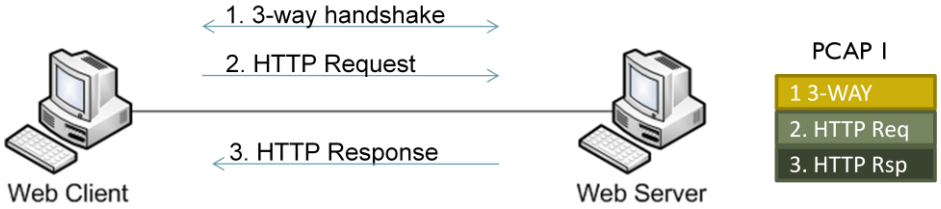


Figure 1(a): An HTTP scenario without a proxy

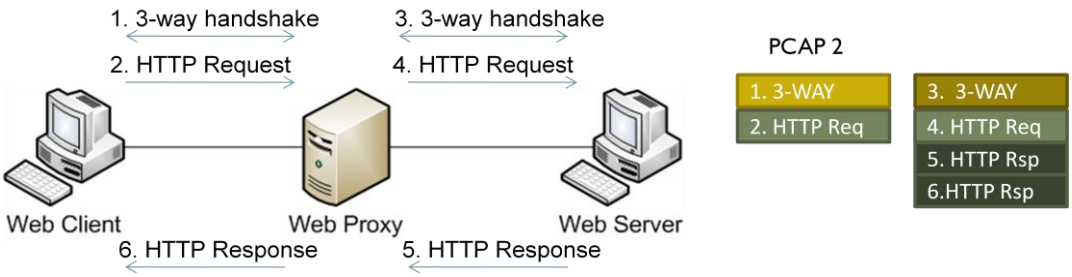


Figure 1(b): An HTTP scenario with the existence of a proxy

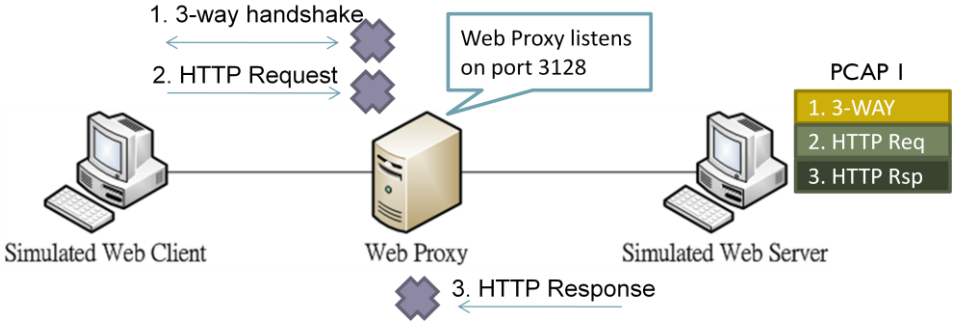


Figure 1(c): A failed replay scenario for a web proxy

As we can see in Figure 1 (a) and Figure 1 (b), the number of actions and the targets of actions are different. If we replay the network trace collected in Figure 1 (a) directly to a proxy, the result must not be what we expected. Figure 1 (c) shows a failed replay scenario for a web proxy. In Figure 1 (c), a web proxy is physically acted the man in the middle between the web client and the web server. The relayed web client and the replayed web server are the replay interfaces directly connected to the web proxy. Suppose the web proxy listens for incoming connections on port 3128. In the first step, the replayed web client traffic, which tries to establish a connection on port 80 would fail since the destination IP address and the destination port number are invalid to the proxy. Therefore, the connection cannot be established between the replayed web client and the web proxy. Since the connection is not established between the replay web client and the web proxy, web proxy will not receive the HTTP request. In step III, the replayed web server replays the HTTP response to the web proxy. Similarly, since there is no connection established between the web proxy and the replay web server, the HTTP response cannot be sent.

2-2 Taxonomy of Application Level Devices

There are various different types of application proxies. The differences can be caused by the protocols being focused, the targeted features, or the implemented network architecture. Table 1 lists network behaviors for each application level device. Understanding the network behavior of devices before designing the replay tool is important since each behavior will affect the accuracy of replay.

Table 1Taxonomy of Application Level Devices

Protocol	**HTTP**	**FTP**
Behavior	Web-Caching Content-Filtering	Content Filtering SOCKS authentication SOCKS transfer Mode Transfer
Protocol	**SIP**	**SMTP**
Behavior	SIP Registration SIP Signal Forwarding	Anti-Spam

Application level devices can also be classified into non-transparent devices and transparent devices. Non-transparent devices are servers which have network devices with public IP address. Transparent devices are firewall or gateway products which act as a router between LAN and WAN areas.

Figure 2(a) shows the difference between the implementation of two HTTP proxies. The proxy in Figure 2(a) is a non-transparent proxy while that in Figure 2(b) is a transparent proxy. The main difference between the two scenarios depends on whether the client knows the existence of the proxy. In Figure 2(a), since client knows existence of the proxy, it establishes a connection directly to the proxy and adds the HTTP “Proxy-Connection” header field in the request.

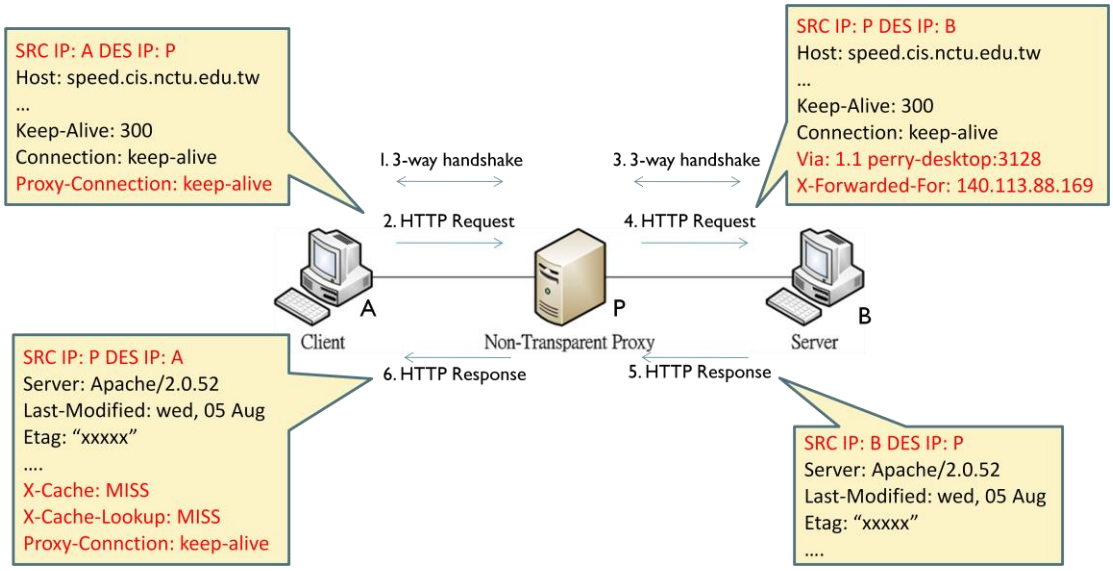


Figure 2(a): An HTTP scenario with a non-transparent proxy

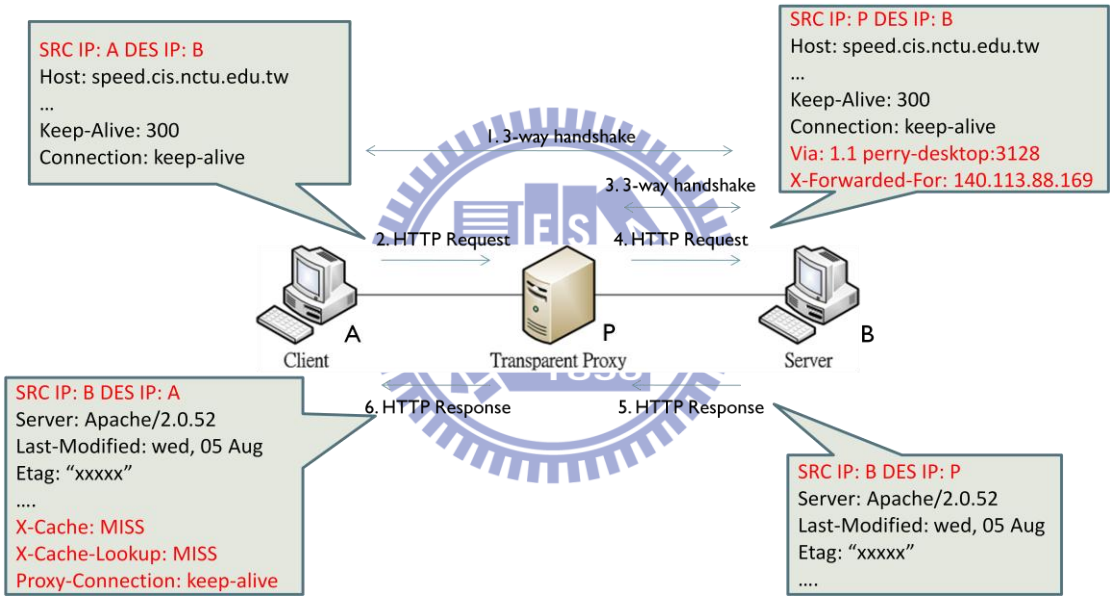


Figure 2(b): An HTTP scenario with a transparent proxy

On the contrary, the client in Figure 2(b) does not know the existence of a transparent proxy. Hence, the client establishes a connection directly to the server and then sends an HTTP request. When the client establishes a connection to the server, the transparent proxy intercepts the connection and then establishes another connection to the server. Table 2 compares the difference between non-transparent proxies and transparent proxies.

Table 2: A Comparison between non-transparent proxy and transparent proxy

	Non-transparent proxy	Transparent proxy
Client needs to configure for proxy	YES	NO
Product	Proxy server	Application-level Firewall Application-level Gateway

2-3 Issues Affecting Accuracy of Application Traffic Replay

The main factors affecting accuracy of application replay are the type of application device and completeness of network traces. For the type of application device problem, we can classify it into protocol dependency and functional dependency problem. Protocol dependency is to mimic different kinds of protocol procedure and functional dependency is to handle different network behaviors. Refer to Table 1; each type of device has different network behaviors. Replay tool should handle the behavior whenever it happens. Otherwise, replay maybe failed due to improper network behavior handling Mechanism. Figure 3 shows an HTTP scenario without handling the content filtering behavior. Replayed web client replays a HTTP request after establishing a connection with web proxy. Next, web proxy filters out the request and sends a 503 forbidden response message back to the replayed web client. If replay tool does not handle the content filtering behavior, replayed web server will replay HTTP response to web proxy. Finally, web proxy drops the unrealistic HTTP response message due to impractical network behavior.

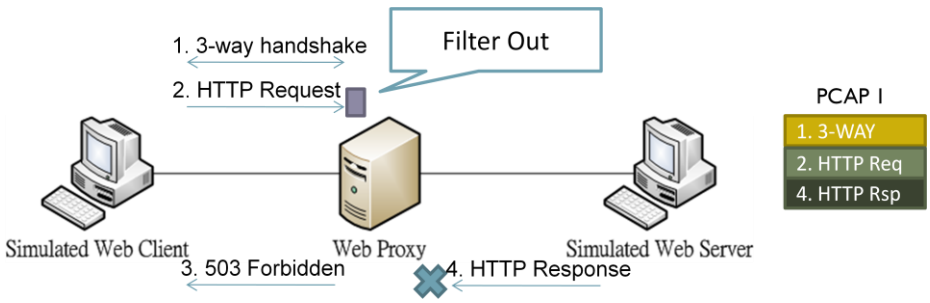


Figure 3: An HTTP Scenario without Handling Content Filtering

Another factor leading to failure in replay is incomplete network trace. We

classify this problem into error resistance problem. There are several reasons that cause the incompleteness. First, it is hard to capture traces in heavy-load environment due to the bounded I/O hardware performance. Second, it is possible that a connection is established before initiating a capture. It is also possible that a connection may still be alive after concluding a capture. Third, packet loss caused by network congestion also makes the trace incomplete. In conclusion, it is essential to design a replay mechanism for each type of application level device and make sure the completeness of network traces.

2-4 Related Works

As mentioned in chapter 1, there are tools developed for traffic replay. TCPReplay [1] replays packets based only on the timestamps. It does not interact with the DUT it interacts with. On the contrast, Tomahawk [3] replays consequent packets depending on the responses from DUTs. The above two tools replay traffic without maintaining any state of network protocols. Stateful traffic replayer, such as the NATReplay tool, maintains the mapping state between private IP addresses and public IP address so that traffic can be replayed through NAT devices. There are many studies focusing on network layer and transport layer stateful traffic replay. TCPopera [5] extends TCPReplay and follows TCP/IP stack by defining data dependencies between messages. SocketReplay [2] mimics TCP/IP stack and improves the selective replay. Furthermore, SocketReplay increases the traffic replay accuracy by recovering packet loss situation.

There are also three researches focusing on application layer stateful replay. Monkey [6] emulates web clients replaying web traffic to web servers and using Dummynet to emulate network environment. RolePlayer [7] proposed a machine learning method to identify state-specific protocol fields such as IP addresses, host names, etc. and then modify these fields accordingly to replay application traces.

Replayer [8] provides a binary analysis and program verification techniques to solve the application-level replay problem. In other words, the system can guarantee accurate response to a request received by a protocol state analysis formula. Saperlipopette [9] captures the web access pattern and replays the pattern to verify the functionalities of web caching system. However, RolePlayer and Replayer can only replay traffic to end point devices but not intermediate proxy servers. The reason is that different proxy servers have different behaviors as mentioned in Section 2.3 and the behavior of proxies are totally different from that of end point devices. Saperlipopette is a tool which can replay traffic through a web proxy. However, Saperlipopette benchmarks web-caching systems using one connection at a time and it is not able to replay connections concurrently. On the other hand, three of them cannot replay incomplete connections caused from capture loss. In this work, we develop a tool called ProxyReplay to solve these problems. Table 3 compares the existed network testing tools.

Table 3: Comparison of network testing tools including trace-based approach and model-based approach

Name	Tool type	Completed PCAP	Stateful	Proxy Replay
TCPReplay	Trace-based	Not required	No	No
NatReplay	Trace-based	Required	Partial	No
SocketReplay	Trace-based	Not required	Layer4 stateful	No
Monkey	Trace-based	Not required	Layer7 stateful	No
RoleReplay	Trace-based	Required	Layer7 stateful	No
Replayer	Trace-based	Required	Layer7 stateful	No
Saperlipopette	Trace-based	Required	Layer7 stateful	YES
ProxyReplay	Trace-based	Not required	Layer7 stateful	Yes

Chapter 3 the Design of the ProxyReplay Tool

3.1 Design Issues

The goal of the proposed solution is to replay network traffic through application level proxies. To replay smoothly and correctly, four issues, namely error resistance, protocol dependency, functional dependency, and concurrent replay, must be carefully considered. The four issues are introduced below.

Error Resistance: The quality of network traces always affects the accuracy of traffic replay. The quality can be affected by several different factors including packet losses, packet retransmissions, and packet reorderings. First, a traffic replay process may fail due to losses of critical packets. Second, a replay tool may send duplicated application messages if retransmitted packets are not handled properly. Finally, an application device may be not able to recognize a message if packets are replayed out of the order.

Protocol Dependency: Application proxy has ability to forward network traffic between hosts. However, before forwarding traffic hosts should follow the protocol policy to communicate with the proxy including setup connections and sending control messages. Since the protocol policy is different between original traces and hosts with proxy, proxy will not work correctly if we just replay the original trace.

Functional Dependency: As mentioned in Section 2-2, each application proxy has their own behaviors. For example, web proxy provides web caching and content filtering behavior to speed up the access rate and to block undemanding web contents. However, these proxy behaviors will alternate the data transfer procedure. Hence, replay tools should handle the situation when proxy behavior happened.

Concurrent Replay: It is common that a network trace contains a number of concurrent connections. If the replay tool can only replay one connection once, the replayed traffic will not real enough and the performance will be low. Therefore, a

replay tool should replay these connections concurrently to simulate real network conditions.

A good traffic replay tool for application level proxies must resolve these issues. In this chapter, we introduce the design of the proposed tool and the solutions to the above four issues. For the ease of explanation, the discussions focus on the replay of web traffic. Nevertheless, the proposed architecture can be easily extended for other application layer network protocols.

3.2 Architecture Overview

Figure 4 shows the architecture of a web traffic replay tool. There are two major components, namely the parser and the replayer. The parser is an engine to parse input data into valuable information and the replayer plays a role to simulate hosts. At begin, the parser loads a PCAP file as the input and converts the traces into internal data structure. After constructing the data structure, the parser stores the data into the queue of the replayer. Finally, the replayer replays data in the queue to network devices.

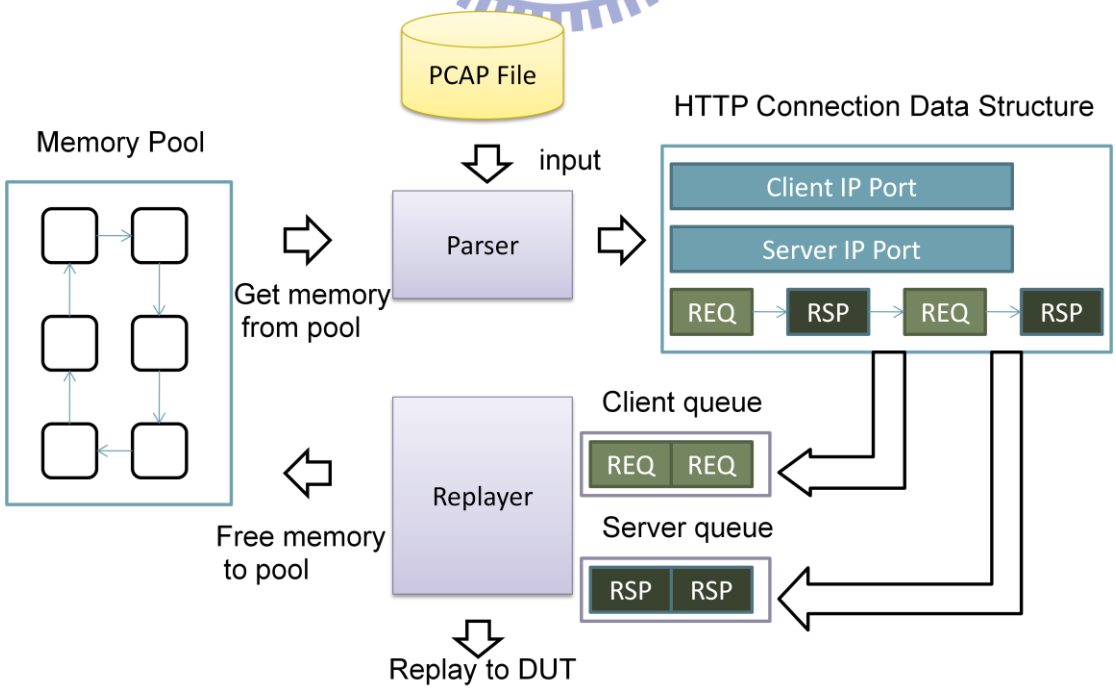


Figure 4: The system architecture for the ProxyReplay tool

3.3 Solutions to Application Level Proxy Replay Issues

Trace Recovery

Trace recovery is the solution to achieve error resistance. As shown in Figure 5, before application replay starts, ProxyReplay tool has to parse packets in a PCAP file into an internal application data structure. Since a flawed packet trace affects the correctness of the data structure, we have to identify flaws caused by packet losses, packet retransmissions and out-of-order packet deliveries.

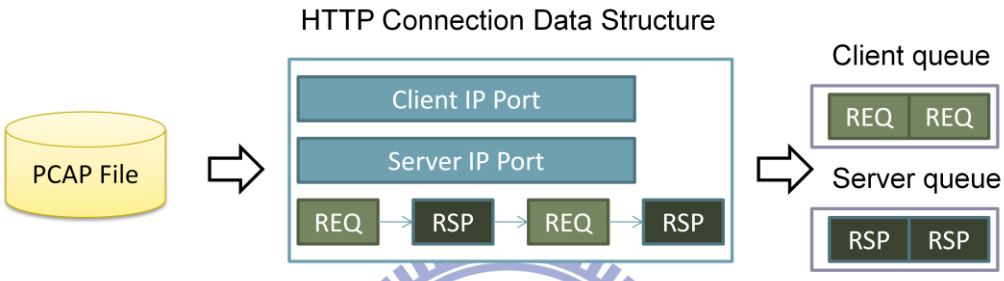


Figure 5: Parse a PCAP file into an internal data structure

(a) Packet Losses: Application data is the most important part for ProxyReplay tool to identify the protocol message. The data includes application headers and application payloads. If an application header loses, ProxyReplay tool should skip all the data in the connection since the corresponding application message cannot be identified. If application payload loses, we should add enough dummy data to let the payload length equal to the value of Content-Length header field. As shown in Figure 6, when RSP-P1 loses, it is possible to add 30 bytes dummy data at the end of the message to match the value store in the Content-Length header field.

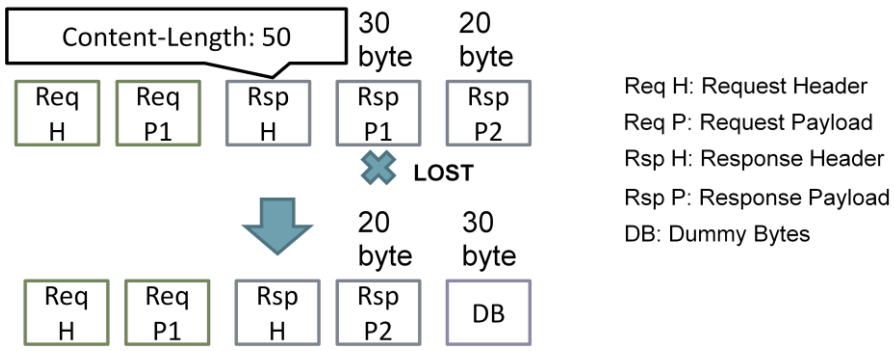


Figure 6: Application Payload lost recovery

(b) Packet Retransmissions: Packet retransmission can be detected by checking whether there are packets have the same sequence number. When a packet retransmission happens, we should ignore the duplicated data. As shown in Figure 7, we skip the second RSP-H since retransmission is detected.

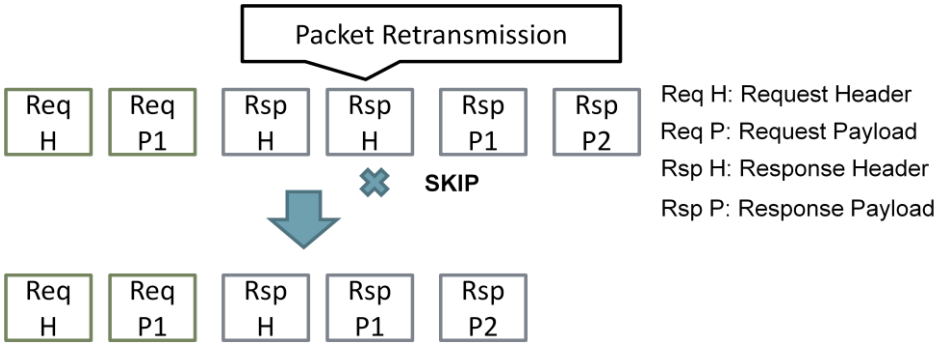


Figure 7: Ignore retransmitted packet

(c) Out-of-Order Packet Deliveries: Packets delivered out-of-order can be found by inspecting the sequence number of each packet. As shown in Figure 8, we list a sequence of packets which is used to send application data. We reorder them by compare their sequence number. For the purpose of solving out of order problem, we should reorder them in increasing order by their sequence number.

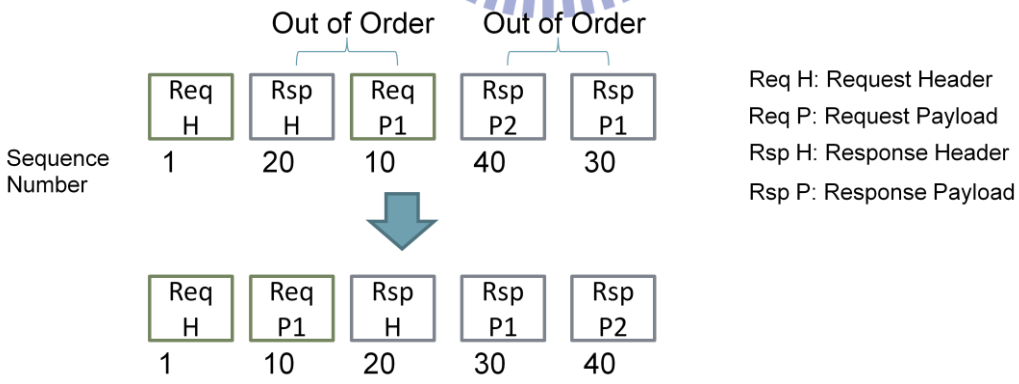


Figure 8: Packets out-of-order recovery

Stateful Proxy Replay

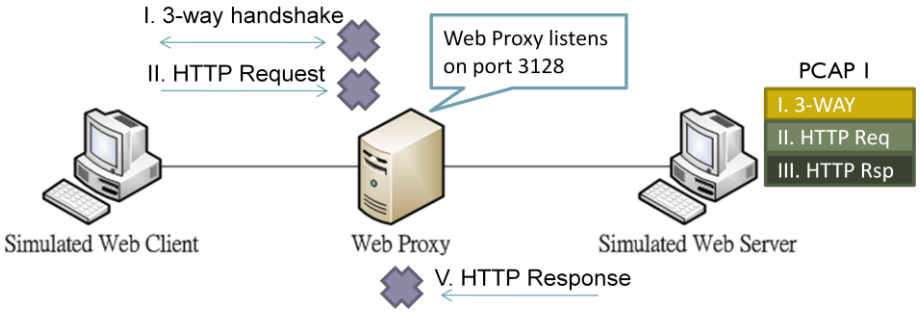


Figure 9: A failed replay scenario for a web proxy

We use stateful proxy replay to solve the protocol dependency issue. This issue can be solved by creating connections, accepting connections, ignoring messages, adding messages, or modifying messages. Moreover, network traces which were captured between hosts or host to proxy are all suitable for this method. Figure 9 gives an example of failed replay without manipulating packets. Figure 10 shows a successive replay scenario by manipulating packets. First, we ignore 3-way handshake packets in the PCAP file because the port number used by a web proxy is often different from that used by a web server. Second, in order to establish the connections with a web proxy, we need to create connections between simulated web client and web proxy. On the other hand, we also need to accept connection from web proxy to simulated web server. After a connection is established successfully, the replayer can start to replay application data.

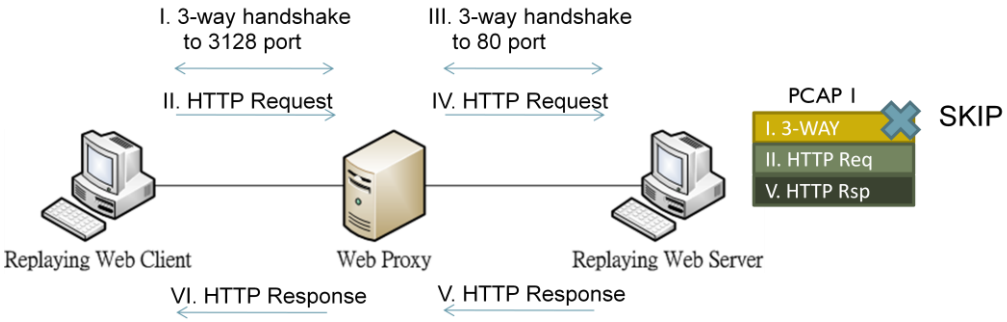


Figure 10: A successive replay scenario for a web proxy

It is common that requests send to a server are often different from requests send to proxies. Therefore, the replayer has to modify requests obtained from traces so that it can send correct requests to the proxy. Table 4 shows common HTTP headers that are required to be modified during a replay process. The Request-Line includes the HTTP request method and the request URI. Before replay the request, the host name or the IP address of the requested web server must be inserted before the URI. If the network trace is captured between proxy and host, we only modify the host IP in the Request-Line. Next, if the host header field is an IP address instead of a domain name, it needs to be modified to the IP address of simulated web server. Last, chunked encoding is not supported by the web proxy. Therefore, the transfer-encoding header field should be removed if the encoding method is chunked.

Table 4: HTTP Headers need to be modified before replay

HTTP Header Fields	Request or Response	Modification Example
Request-Line	Request	GET /index.html HTTP/1.1 => GET http://192.168.1.2/index.html HTTP/1.1 (change to full URL)
Host	Request	Host: 140.113.88.168 => Host: 192.168.1.2 (change to new IP)
Transfer-Encoding	Response	Transfer-Encoding: Chunked => (remove)

One important role during the replay process is the masqueraded DNS server. It is common that a proxy always need to look up the IP addresses of a target server. Therefore, we need a DNS server to handle DNS requests from proxies. Figure 11 shows the HTTP replay scenario with a masqueraded DNS server. At the beginning, a

masqueraded DNS server is run and creates domain names to IP addresses mappings based on network traces. Next, when simulated web client replays a HTTP request to a host “abc.com” via the proxy, the DNS request for “abc.com” sent from the proxy can be answered by the masqueraded DNS server

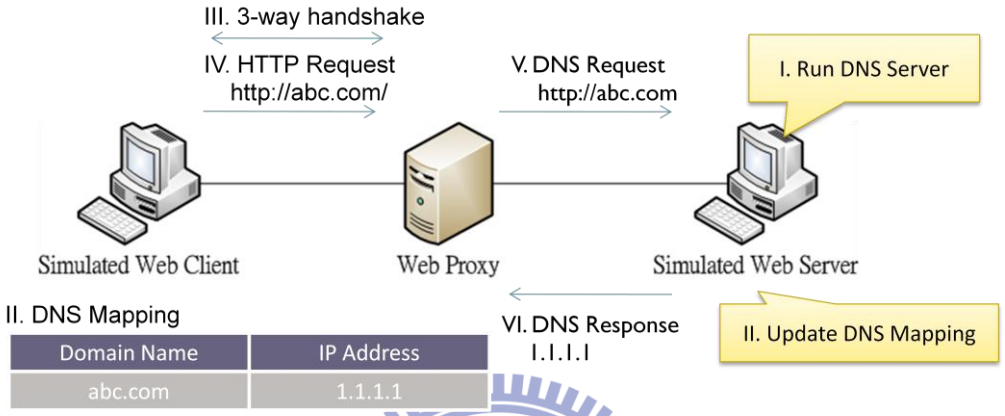


Figure 11: HTTP replay with DNS messages

Functional Dependency Resolver

As shown in Section 2-3, various different functions, such as web caching and content filtering, can be implemented on a proxy. The web-caching function caches the response data from servers to save the uplink bandwidth and shorten the response time. The content filtering function filters out malicious traffic based on predefined access control lists. If a replay tool is not aware of modifications made by the proxies, replays can be failed as shown in Figure 3. In this sub-section, we describe how to properly handle the above two cases.

As shown in Table 5, in order to make web-caching working correctly, the date header field should be updated to the current time to keep the freshness of the application data. Otherwise, some responses will not be cached due to the validation time is expired.

Table 5: Date header field modification

HTTP Header Fields	Request or Response	Modification Example
Date	Response	Date: Thu, 26 Nov 2009 07:34:04 GMT(Past) => Date: Thu, 26 Nov 2009 11:33:35 GMT(Cur) (change to current date)

After modify the header, Figure 12 shows our solution to replaying HTTP traffic through a web cache proxy. First we replay a cacheable HTTP request REQ-A and HTTP response RSP-A. Web proxy will cache the response in its cache swap space and send the cached response to simulated web client. Then, we replay the same request REQ-A to the web proxy again. Since REQ-A has been replayed before, the web proxy is able to find the corresponding response RSP-A in its cache and then sends a cache HIT response to the web client. Finally, we will not replay the last HTTP response because web caching behavior happened.

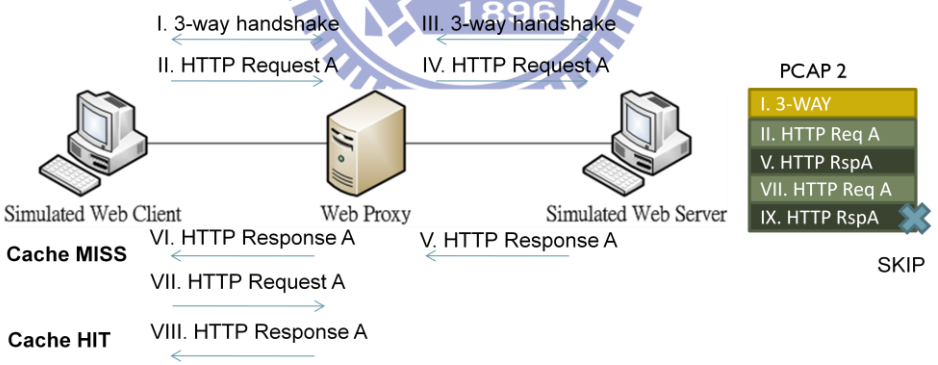


Figure 12: HTTP replay scenario with web caching

Figure 13 shows our solution of replaying HTTP traffic through a content filter proxy. At the beginning, we replay a HTTP request to the web proxy. If the web proxy filters out the request according to the ACL rule and sends a “503 forbidden” response back to the web client, we simply ignore the HTTP response since it is blocked by the content filtering behavior.

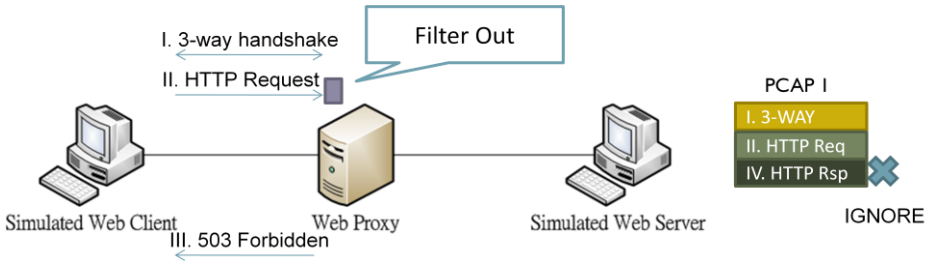


Figure 13: HTTP replay scenario with content caching

Concurrent Replay

To simulate the real network conditions, a trace replayer must be able to replay concurrent connections. In this sub-subsection, we describe the solution to replay multiple connections. As shown in Figure 5, before replaying multiple connections concurrently, we have to read the traces and parse each individual HTTP connections into a client queue and a server queue. The two queues are used to store HTTP requests and responses sent from the simulated web client and server, respectively. Therefore, the total number of active replay queues is identical to two times of the number of concurrent HTTP requests. Figure 14 shows a concurrent replay scenario. There are six pairs of parsed HTTP requests and responses store in the client queue and the server queue, respectively. In order to replay realistically, requests are sorted in the client queue according to the timestamp and then replay them sequentially. Since there are multiple connections we want to replay, we create the connection queue for each connection. Each connection queue handles one request at once. When the web proxy forwards the request to the simulated web server, the simulated web server will find the corresponding response and replay it to the web proxy. The advantage of this replay design is that we can handle web caching and content filtering easily since simulated web server will not replay the response when no request is received.

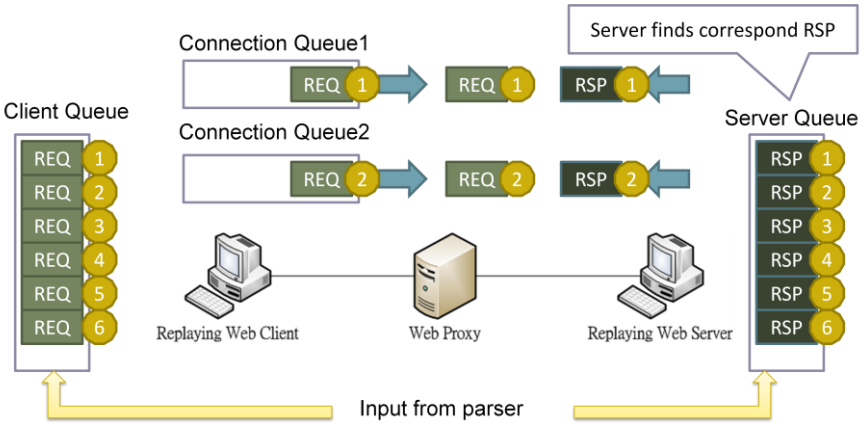


Figure 14: The replay strategy for multiple connections replay

One limit for the above replay design is the required memory space increases linearly according to the size of HTTP requests and responses. Since ProxyReplay tool stores connection data structure in the memory, the available memory space must be large enough to store all the data. In order to eliminate the memory limitation, we can run parser and replayer concurrently as the scenario in Figure 4. First, only a fixed-size memory pool is used in the scenario. When the parser creates the HTTP data structure, it gets a memory slot from the memory pool until the memory pool is out of use. Whenever parser generates a pair of request and response messages, parser will put the messages include client queue and server queue separately. Next, replayer replays the messages in client queue and server queue. After replayer replays the HTTP messages, replayer puts the memory slot back into the memory pool. In principle, concurrent replay will be done when parser finishes reading the input pcap file. When memory pool is out of use, we start to free the responses which are ignored due to web caching behavior happened. Finally, ProxyReplay tool can complete parsing and replaying network traces.

Chapter 4 Implementation

Software Architecture

Figure 15 shows the components of the ProxyReplay tool. Components of the tool are implemented in both user space and kernel space. The user space components include the replay interface, the application parser, and the replay engine. On the other hand, the kernel space components allow a single machine to emulate both replayed clients and replayed servers simultaneously. In this chapter, we describe each component of the ProxyReplay tool in detail.

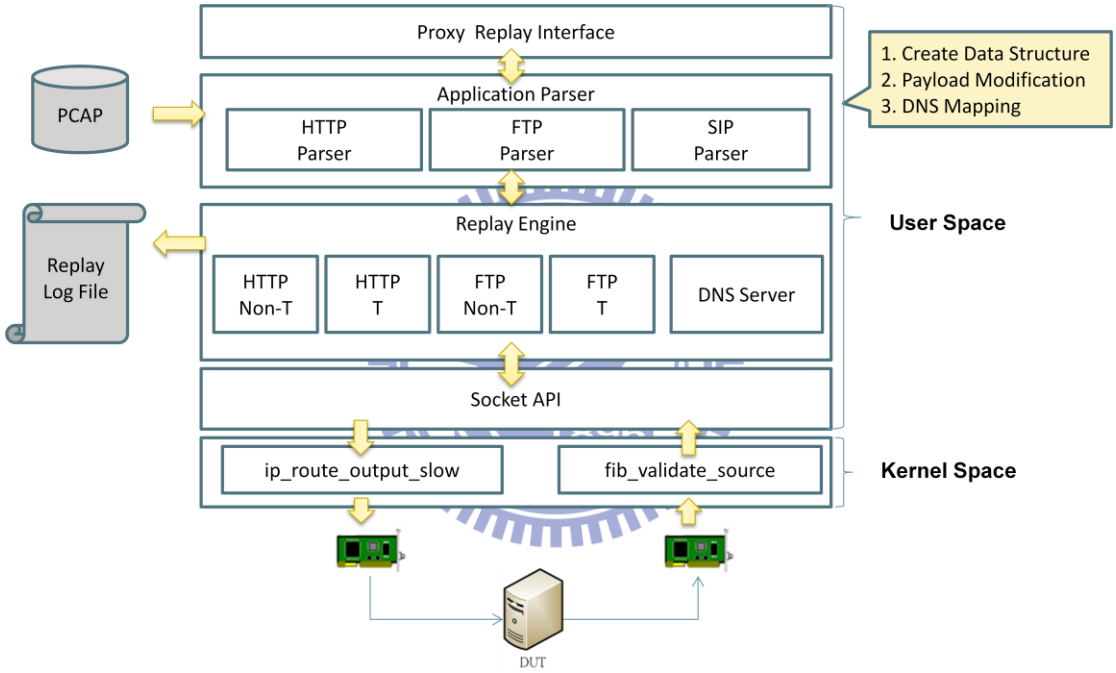


Figure 15: Software architecture of ProxyReplay

The Application Parser

The application parser is responsible to transfer a PCAP file into internal application data structures. In order to replay correctly, the error resistance problem must be solved by the component. Figure 16 gives an example of an HTTP parser. Based on the five-tuple information, the parser extracts application protocol message fragments from packets of the same HTTP connection and then put the messages into corresponding message queues. In the meanwhile, the parser also needs to modify the application payload according to the rules listed in Table 4 and Table 5. If necessary,

the parser has to add the hostname information to the masqueraded DNS server.

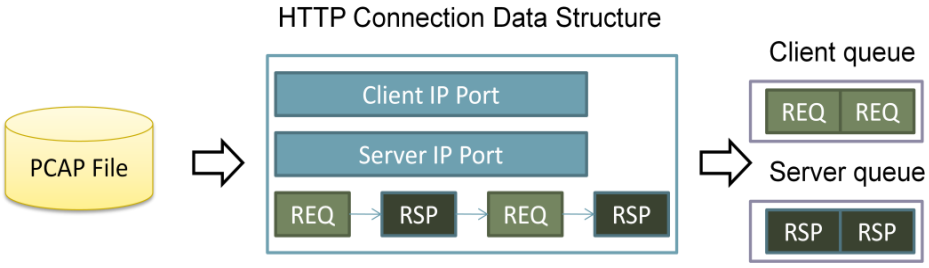


Figure 16: Parse a PCAP file into an internal data structure

As shown in Figure 11, a proxy server may send a DNS query when it does not know the IP address of a host. We use the BIND9 DNS server to handle DNS queries. Since BIND9 support dynamic updating of DNS mappings when DNS server is running, we can update DNS mappings without interrupting the DNS server. Figure 17 shows the example of updating a DNS mapping. We use the nsupdate command to update the mapping of a host named speed.cs.nctu.edu.tw to its new IP address 140.113.88.160.

```
root@bind:~#nsupdate
>server localhost
>update delete speed.cs.nctu.edu.tw
>update add speed.cs.nctu.edu.tw 600 A 140.113.88.160
>send
```

Figure 17: An example of updating DNS mapping

The Replay Engine

The replay engine replays protocol messages stored in message queues. Each protocol has its own replay module and the module also divided into replaying client and replaying server. . In order to increase the replay performance, we emulate the replayed client and the replayed server in two different threads. On the other hand, since the ProxyReplay tool has to handle multiple connections simultaneously, the ability to handle a large number of file descriptor must be taken into considerations. Table 6 shows three API which are used to handle multiple FDs (file descriptors). We use the epoll system call to implement the ProxyReplay tool because it has no limitation on the number of FDs. Besides, the performance is not dropped

significantly when the number of FD is large.

Table 6: Comparison between select, poll and epoll

	Select	Poll	epoll
Max number of file descriptors	2048	No limited	No limited
Performance	Depends on the number of FD	Depends on the number of FD	Independent to the number of FD

Kernel Enhancements

Since the ProxyReplay tool emulates replayed clients and servers on the same machine and it uses socket APIs to establish network connections, packets sent from a client to a server and vice versa are traversed via the loopback interface instead of real network interfaces. Therefore, we must modify the kernel so that the connections are really replayed through real networks. Based on the Linux kernel version 2.6.20, we have to modify two functions to fit the requirement. First, we modify the ip_route_output_slow function so that loopback packets can be sent to the real network interface conditionally. Second, we modify the fib_validate_source function to accept loopback packets received from real network interfaces.

The Proxy Replay User Interface

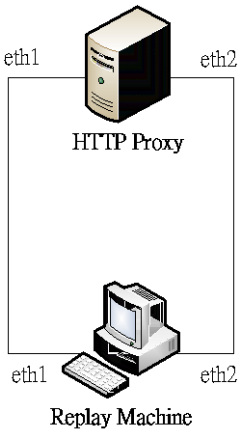
With the user interface of the ProxyReplay tool, a user can specify the proxy type and replay mode. The proxy type includes application protocol and the transparency including transparent and non-transparent mode, the ProxyReplay tool then loads the corresponding parser and the replay module according to the request of the user. The replay mode can be either (1) preprocessing and replay, or (2) parser and replay works concurrently. The former mode has a faster replay speed since it preprocesses PCAP file and replay network trace separately. However, the first mode is also limited by the amount of available memory space. On the contrast, the total execution time of the second mode can be shorter since the parser and the replayer works concurrently. In addition, there is no limitation on the size of replayed network trace files. Users can choose the appropriate mode depending on the input file size and replay speed.

Chapter 5 Evaluation

In this chapter, we will evaluate the functionality, performance and scalability of ProxyReplay with HTTP protocol. The evaluation of functionality focus on two proxy behaviors: web-caching and content filtering. The performance focus on the throughput of the two replay strategies: preprocessing and replay, and concurrent replay. Third, the evaluation of scalability replay focus on how large the input file can be supported by ProxyReplay. Last, we will compare different replay tools using different types of DUT.

5.1 Test Environment for ProxyReplay

As shown in Figure 18, we use two computers to do the experiment including a proxy server and a replay machine. Both of the computers have two interfaces which are connected to each other. We use squid3 as a proxy daemon running on the HTTP proxy. Next, we use Bind9 as a DNS server and run the ProxyReplay on the replay machine.



	HTTP Proxy	Replay Machine
CPU	AMD Athlon(tm) 64 Processor 3000+	Intel(R) Core(TM)2 Quad CPU Q8200 @ 2.33GHz
RAM	1GB	4GB
OS	Linux 64bit	Linux 64bit
Interfaces	Set eth1 and eth2 to bridge mode	eh1: http replaying client eth2: http replaying server
Software	Squid3	ProxyReplay tool Bind9:

Figure 18: The test environment for ProxyReplay

Before starting the experiment, it is important to make sure the completeness of the input PCAP file. For HTTP application replay, we need to find a corresponding response message for each request since an HTTP connection should include at least one pair of request and response. Therefore, we should remove the traces which

cannot be identified since its application header was lost. Table 7 gives a statistics of the input pcap file which is used in this experiment. The PCAP file records 2.6 giga bytes HTTP network flow in 20 minutes. The number of connections is 10964 including 363 HTTP clients and 916 HTTP servers. The number of requests and response is 26825 and 27420 accordingly. Last, the request and responds we can replay is 21166 pairs.

Table 7: Statistics of the input PCAP file

File type	HTTP PCAP File
File size	2.6 GB
Total time	20 minutes
Number of packets	3168672
Number of connections	10964
Number of clients	363
Number of servers	916
Number of requests (total)	26825
Number of response (total)	27420
Number of request and response pairs (can be replayed)	21166
Number of responses will be cached	6074
Number of responses will not be cached	15092
Number of requests will be filtered (.jpg)	7134
Number of requests will not be filtered	14032

5.2 Functional Evaluation

In this section, we evaluate the functionality of ProxyReplay. In order to make sure the correctness of replay, we make the comparison between predicted value and replay result for each proxy behavior. The predicted value is calculated by analyzing the packet trace.

5.2.1 Web-Caching

Before replay, we predict the number of cacheable responses according to the web caching rule which is shown in Table 8. The response will be cached if it satisfied the 5 factors including request method, status code, Cache-control header field, Expiration and Dynamic content.

Table 8: Web caching rules

	Cacheable	Non-cachable
Request method	GET, HEAD	POST, PUT, DELETE, OPTIONS, TRACE
Status code	200, 203, 206, 300, 301, 410	Other status
Cache-control header field	public, private, max-age>0	no-cache, max-age=0, no-store
Expiration and validation	Date < Expires	Date >= Expires
Dynamic Content	Others	Query terms ("?" mark in URL)

Figure 19 shows the results of replay with and without web caching. Without caching means that we replay to a web caching proxy which does not have any cache data. On the contrast, with caching means that the proxy we have replayed once and the proxy stores some cache data. In Figure 19, the number of messages between client and proxy is the same ether with caching or without caching since simulated client will replay all requests and all correspond responses should be received. Without web caching, the number of messages between proxy and server decrease a little bit because some requests with the same URL were cached. With web caching, we found that about 16263 requests did not be cached which is similar to the non-cacheable number of requests 15092. Due to each proxy daemon has their own caching rules, the number of non-cacheable requests will not the same with our estimation but they will be similar.

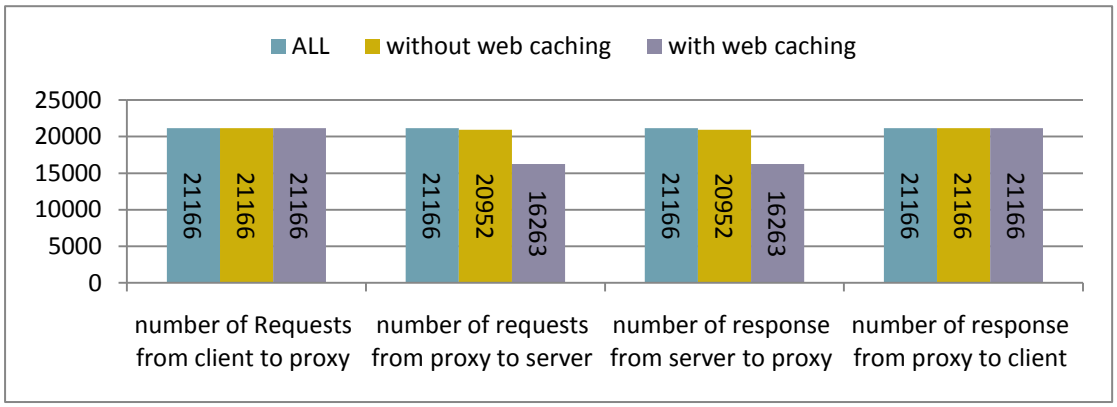


Figure 19: Results of ProxyReplay with and without web caching

5.2.2 Content Filtering

In order to evaluate the functionality of content filtering replay, we configure the

ACL (access control list) rule of the proxy server. The types we can filter out by ACL including Source/Destination IP address, Source/Destination Domain, Words in the requested URL (Ex: .jpg, .gif, .exe), Current day/time, Destination Port and HTTP Method.

In this experiment, we configure the proxy to filter out all .jpg requests and the total request will not be filtered out is 14032 as shown in Table 7. Figure 20 shows the result of with content filtering and without content filtering. We found that the sum of the number of requests from proxy to server 13845 and the number of cached response 187 which is the same with our predict number 14032.

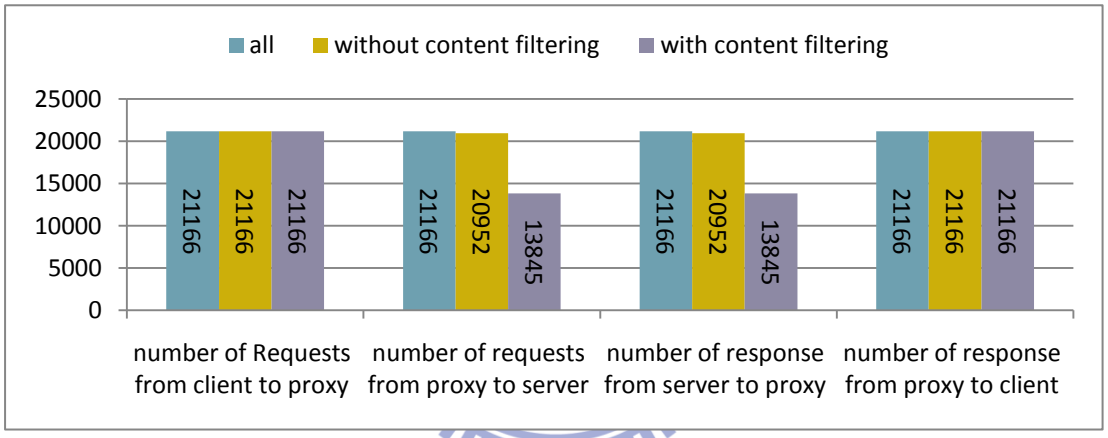


Figure 20: Results of ProxyReplay with and without content filtering

5.3 Performance Evaluation

In this section, we compare the performance between our two replay strategies including “preprocessing and replay” and “concurrent replay”. Figure 21 shows the result of the performance evaluation. We recorded the bandwidth of replay in each second. The result of two replay strategies is shown as preprocessing and concurrent. In Figure 21, we found that the throughput of preprocessing strategy is higher than concurrent strategy since parser will decrease the performance of replay. In order to achieve application stateful replay, parser needs to construct the application data structure, modify payloads and update DNS message. Therefore, parser would be the bottleneck since replayer should wait for the parser until the application data structure

is constructed. On the other hand, the parse time of concurrent strategy is shorter than preprocessing strategy since parser and replayer starts concurrently.

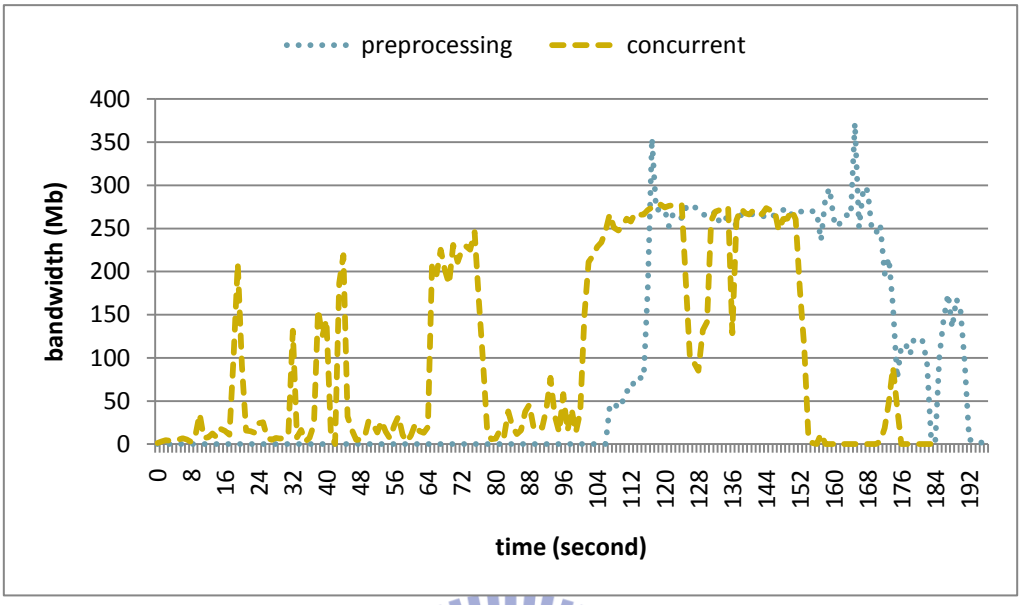


Figure 21: Performance of two replay strategies

Figure 22 shows the request rate of two replay strategies using the 2.6GB HTTP PCAP file as input. For preprocessing strategy, the average request rate can achieve 1200 requests per second. For concurrent strategy, the average request rate about 200 requests per second which is limited by the performance of parser. However, when parser finished parsing, the average request rate can achieve 1000 requests per second.

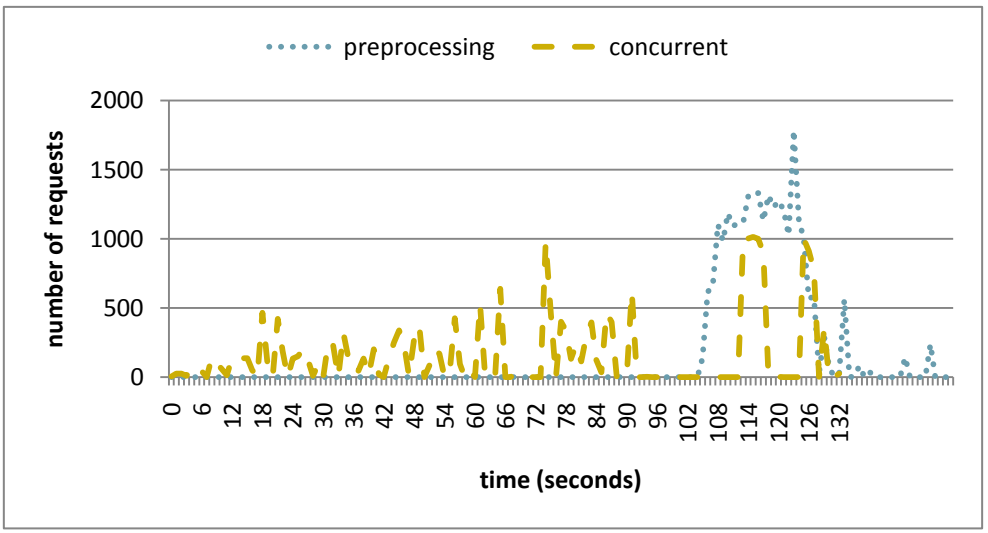


Figure 22: The request rate of two replay strategies

Table 9 shows the throughput of ProxyReplay tool in different size of responses.

We configure the size of each response to 1.5 KB, 15 KB and 50 KB. The average request rate of ProxyReplay tool is 1300 requests per second. We found that the throughput of ProxyReplay depends on the size of the response since the speed of data transfer is faster than establish connections. If the size of response is larger, ProxyReplay tool will spend more time on data transfer.

Table 9: Throughput of ProxyReplay tool in different size of responses

Size of response	1.5 KB	15 KB	50 KB
Throughput (Mbps)	28.18	106.74	244.60

5.4 Scalability Evaluation

In the section, we use two replay strategies to replay three input http pcap file with different sizes. The size of the three file is 800 MB, 2.6 GB and 5.2 GB separately. We found that “preprocessing and replay” can only replay the first two files due to the third file is too large to load to the memory. On the contrary, “concurrent replay” can replay all the three files. Table 10 gives a comparison between the two strategies. For “preprocessing and replay”, the average replay speed can achieve 200 Mb/s but the parse time is long and the input file size is limited by memory size. For “concurrent replay”, the average replay speed is only 103.1Mb/s but the parse time only needs 2 second for initial parsing and there is no limitation for the input file size.

Table 10: Comparison between two strategies

	Preprocessing and Replay	Concurrent Replay
Parse time	Depending on the file size	2 sec (initial parsing)
Average replay speed	200 Mb/s	103.1 Mb/s
Input file size	Depending on the memory size	No limited
Test Case (Using http flow)	800 MB (pass) 2.6 GB (pass) 5.2 GB (fail)	800 MB (pass) 2.6 GB (pass) 5.2 GB (pass)
Pros	1. High replay speed	1. Low parse time 2. No memory size limitation

Cons	1. Longer parse time 2. Limited be memory size	1. Low replay speed (parser is bottleneck)

5.5 Replay Tools Comparison Evaluation

In this section, we make a comparison between different existed replay tools. As shown in Table 11, the replay tools including TCPReplay, NATReplay, SocketReplay and ProxyReplay. We configure the computers which are listed in Figure 18 to NAT, transparent, router and proxy mode. As shown bellow, TCPReplay supports transparent mode and router mode device and the throughput is 363Mb/s and 288Mb/s accordingly. NATReplay can support NAT devises and the throughput is about 320Mb/s. SocketReplay supports TCP/IP stateful replay on transparent and router devices and the throughput is 40Mb/s and 12Mb/s separately. The last, ProxyReplay is the tool supports proxy mode devices and the throughput can achieve 200 Mb/s.

Table 11: Comparison between replay tools

	TCPReplay	NATReplay	SocketReplay	ProxyReplay
Transparent	(363Mbs)	X	(40Mbs)	X
Router	(288Mbs)	X	(12Mbs)	X
NAT	X	(320Mbs)	X	X
Proxy	X	X	X	(200Mbs)

Chapter 6 Conclusions and Future Works

In this thesis, we design and implement the ProxyReplay tool to statefully replay network traces for application level proxies. Four fundamental issues, namely error resistance, protocol dependency, functional dependency, and concurrent replay, are identified and solved. In addition, the ProxyReplay tool provides two replay modes, i.e., preprocessed replay mode and concurrent replay mode. The preprocessed replay mode can be used to launch stress test using real network traces. On the contrast, the concurrent replay mode can be used to replay network traces that are greater than physical memory.

In addition to the completeness of functionalities, the evaluations show that the captured network traces can be accurately replayed using the both replay modes. The overall throughput for the preprocessed mode exceeds 200Mbps by using an off-the-shelf desktop computer. When the concurrent replay mode is used, the size of the network trace file is not limited by the size of physical memory. It can be used to emulate a complete real application scenario.

Although we only discuss the case of replaying web traffic in this thesis, we believe that the proposed architecture can be easily extended for other application protocols. Take mail proxy replay for example, since mail proxy acts as a MTA (Mail Transfer Agent), we should follow the protocol procedure diagram to replay traffic for the mail proxy. On the other hand, it is also essential to convert incomplete network trace to mail application data structure by trace recovery. Then, we use stateful proxy replay to replay data to the mail proxy. In the future, we plan to add more application protocols to the ProxyReplay tool so that it can be a universal benchmark tool for various different types of application layer proxies. Moreover, it is also essential to enhance the request rate to test benchmark more high performance DUTs.

Reference

[1] A. Turner, Tcpreplay, http://tcpreplay.synfin.net/trac/.

[2] Tsung-Huan Cheng, Ying-Dar Lin, "Low-Storage Capture and Loss-Recovery Stateful Replay of Real Flows", Institutes of Network Engineering College of Computer Science National Chiao Tung University, June 2009.

[3] Tomahawk, http://www.tomahawktesttool.org/, 2005.

[4] A. Turner, "Flowreplay design notes," http://synfin.net/papers/flowreplay.pdf.

[5] Seung-Sun Hone and S. Felix Wu, "On Interactive Internet Traffic Replay," in 8th Symposium on Recent Advanced Intrusion Detection (RAID), LNCS, Seattle, September 2005.

[6] Yu-Chung Cheng, Urs Holzle, Neal Cardwell, Stefan Savage, and Geoffrey M. Voelker, "Monkey see, monkey do: A tool for tcp tracing and replaying," In Proceedings of the 2004 USENIX Annual Technical Conference, June 2004.

[7] Weidong Cui, Vern Paxson, Nick C. Weaver, and Randy H. Katz., "Protocol-Independent Adaptive Replay of Application Dialog," in Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS), February 2006.

[8] James Newsome, David Brumley, Jason Franklin, and Dawn Song, "Replayer: Automatic Protocol Replay by Binary Analysis," ACM Conference on Computer and Communications Security, October 2006.

[9] Guillaume Pierre and Mesaac Makpangou,"Saperlipopette!: a distributed Web caching systems evaluation tool," Proceedings of the Middleware conference, 2009.

[10] Ari Luotonen, "Web Proxy Servers", Netscape Communications Corporation, 1997.

[11] Duane Wessels, "Squid the Definitive Guide", O'Reilly & Associates, Inc. January 2004.

[12] Duane Wessels, "Web Caching", O'Reilly & Associates, Inc. 2001.

[13] Fabian Schneider, Jorg Wallerich, Anja Feldmann, "Packet Capture in 10-Gigabit Ethernet Environments Using Contemporary Commodity Hardware," Passive and Active Measurement Conference, April 2007.

[14] Pete Loshin, Peter H. Salus, "Big Book of Internet File Transfer RFCs", Morgan Kaufmann, 2000.

[15] Flavio E. Goncalves, "Building Telephony Systems with OpenSER", Packet Publishing, 2008.

[16] Web-polygraph, http://www.web-polygraph.org/.