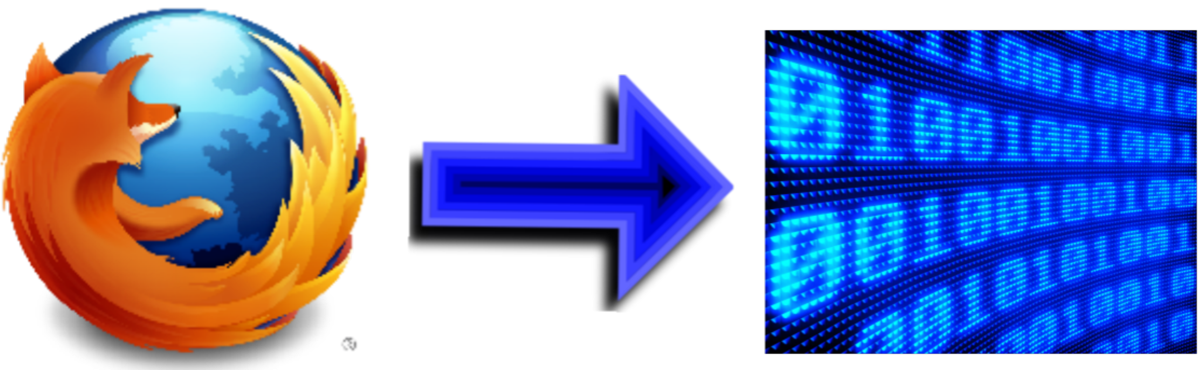
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Computer Engineering and
Networks Laboratory

Manuel Widmer

Wirefox – A Measurement Plug-In for Firefox

Semester Thesis
Fall 2010



Tutors:
Dominik Schatzmann
Wolfgang Mühlbauer

Professor:
Bernhard Plattner

Abstract

Network failure troubleshooting is a difficult task mostly involving human interaction. While fault identification can be done with active measures such as traceroutes or network tomography, the detection of a fault may not be trivial. Looking for ways to automate the detection process passive end-host measurements seem to be a good starting point. In contrary to existing capturing tools such as Wireshark, Tcpdump or similar, we suggest an implementation in form of a JavaScript based browser extension, which allows to monitor network traffic directly from within the browser.

Die Behandlung von Netzwerkfehlern ist eine schwierige Aufgabe, die meist menschliche Interaktion involviert. Die Fehleridentifikation kann mittels aktiver Methoden wie z.B. Traceroutes oder Netzwerktomographie geschehen, während die Feststellung, dass überhaupt ein Fehler vorhanden ist nicht immer trivial ist. Will man diesen Prozess automatisieren, scheinen Messungen direkt am End-Host ein guter Startpunkt zu sein. Im Gegensatz zu existierenden capturing Lösungen wie Wireshark, Tcpdump etc., schlagen wir eine Implementierung in Form einer JavaScript basierten Browsererweiterung vor, die es ermöglicht Netzwerkverkehr direkt im Browser zu überwachen.

Author:	Manuel Widmer	widmerma@ee.ethz.ch
Tutors:	Dominik Schatzmann	dominik.schatzmann@tik.ee.ethz.ch
	Wolfgang Mühlbauer	wolfgang.muehlbauer@tik.ee.ethz.ch
Professor:	Bernhard Plattner	plattner@tik.ee.ethz.ch

Acknowledgements

I want to thank very much my two tutors Dominik Schatzmann and Wolfgang Mühlbauer for their helpful suggestions and valuable input at weekly meetings and through email correspondence. I profited well from their encouraging guidance and technical proficiency which has simplified many aspects of the work.
Also the constructive advice of Prof. Plattner during the intermediate review is very well appreciated.

Contents

1 Introduction **1**
1.1 End-host Measurements 1
1.2 Why A Firefox Extension 1
1.3 Development Steps 2

2 Implementation **3**
2.1 High Level Architecture 3
2.2 Connection Matching 6
2.3 Port Matching 7
2.4 Data Storage & Output 7

3 Evaluation **9**
3.1 Browsing Performance 9
3.2 AJAX Monitoring 10
3.3 Packet/Flow Capture Performance 11

4 Conclusions **13**

A Task Description **15**

Chapter 1

Introduction

1.1 End-host Measurements

As explained in [2] fault detection is mostly a manual process. Automatic detection combined with fault identification would be the optimal solution. However, for the detection active measures are inappropriate as they produce a large traffic overhead by additional pinging or probing. In addition some protocols (e.g. icmp) may be blocked by intermediate devices. Hence passive end-host measurements are a good starting point as they don't produce network traffic overhead. In addition end-host measurements can be useful in many other scenarios such as network and application performance profiling, network management or energy management. A disadvantage of end-host measurements is that they use some computational resources of the corresponding host.

Also [1] and [2] mention that there are many pitfalls concerning deployment and user perception of end-host measurement tools. The most important of them being uptime of the host, privacy issues with data collection and impacts on computing/browsing performance.

1.2 Why A Firefox Extension

In the last few years the browser has become one of the most important applications on the typical consumer PC. Current trends are to provide entire office environments as online services accessible through the browser and even first browser based operating systems are showing up, e.g. "eye os" or google's "chrome OS". So we have to ask why not also move monitoring infrastructure into the browser?

There already exist standalone packet capturing tools. Most of them involve the installation of a systemdriver and/or require administrative privileges on the corresponding system. In addition, if one is interested only in browsing traffic analysis, those tools require the setup of rules and filters. Furthermore, it is not possible without any difficulty to inspect packages with encrypted payload. We found that there are currently no browser-based approaches to end-host monitoring around. Regarding all those aspects, a browser extension has much potential. It is installable by every user without special permissions, all browser related traffic is visible and because the extension can essentially see what the user sees, there is also access to all information during an encrypted session. So comparing to conventional layer 3 capturing we can get exactly the same information for http as well as for https sessions. We chose the Mozilla Firefox browser because it provides a very rich and reasonably well documented JavaScript interface and allows for quick add-on development. In addition it is available on the prevalent platforms Windows, Linux and Mac OS which simplifies making the extension also cross-platform available.

1.3 Development Steps

A main concern of this thesis is to explore the possibilities of network traffic monitoring through the browser. For Firefox there already exists an advanced add-on for debugging purposes called Firebug but it focuses heavily on timing analysis of a single pageload and lacks long term monitoring and IP resolution facilities. Our aim is to develop a library to extract relevant data possibly in real-time from the browser and write an add-on that exports similar information as Netflow or packet capture data. Especially we also want to have IPs and ports of the connections available. The name Wirefox implies the idea of “porting” Wireshark to Firefox. Finally the add-on will be tested with real user data where the impact on browsing performance and the packet capturing performance are measured.

Chapter 2

Implementation

2.1 High Level Architecture

At first we have to define which data we are interested in. To get something similar to Netflow, for each connection at least the tuple of {srcIP, srcPort, dstIP, dstPort} is needed. In addition it is useful to know how many packets or bytes were transmitted and starting- as well as finishing-time of the connections. Furthermore we could be interested in the URL, MIME type or other browsing related information. Firefox already offers an interface where one can access the so called httpChannel of the connection. This is done by registering an observer (or event listener) to intercept corresponding events. This gives us access to following information:

- Starttime of Connection
- End of transmission (may not be end of connection due to pipelining)
- URI (see also figure 2.1)
- Transmitted / Received Bytes
- MIME Type
- HTTP headers

Unfortunately the IP and port information of a connection is not accessible through JavaScript. Taking a closer look at the URI (fig. 2.1) we can see that we can get the destination port from there. Although the port may not be specified all the time, we could deduce it from the protocol used e.g. http uses port 80, https uses 443 etc.

Firefox offers also a DNSService accessible for add-ons. Hence in most cases we can get the destination IP address by querying the DNS System with the host taken from the URI. In many cases multiple IP addresses are returned

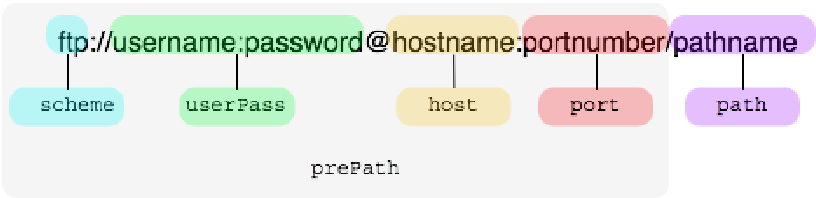


Figure 2.1: Components of a URI[1]

for a single hostname, a measure taken by providers for load balancing reasons. This means that a DNS query on its own is insufficient to determine the correct destination IP of the connection.

There is the tool Netstat which displays all currently open connections of a computer. As Firefox allows to start arbitrary executable programs from within an add-on, we can exploit this and run Netstat in parallel to our add-on. We just redirect all the output of Netstat to files and then search those files for the IP addresses returned by the DNS query described before. This procedure solves most remaining problems. The destination IP address is now uniquely determined by the Netstat output an we can find the source IP address and the source port on the same line.

Putting all those things together the design of the library looks as depicted in fig. 2.2. The communication of the library and the "‘outside world"’ happens again via the observer and notification mechanisms of Firefox.

To do the whole processing in real-time we started with a fully event driven architecture. The idea was to execute Netstat on the fly each time a new connection is established and then parse the output to directly obtain IP addresses and ports. But we soon had to give up this approach as a normal hard drive just was too slow. When connecting to an average[2] homepage, several connections to the webserver are established in a few milliseconds. Tests yielded that we could execute Netstat and parse its output at most every 200ms which is far too little. In addition there where some multithreading problems such as deadlocks or variable overwrites when processing intensive tasks were performed during monitoring. Most problems originated from the fileIO and process API that was used to run Netstat and parse the output file. Sometimes just empty strings were read from files, or Netstat could not be started. At first this was very strange because JavaScript actually is a strictly single threaded language. It turned out that there were multiple instances of the same observer-functions running in parallel on their own threads if many registered events occurred in a short period of time.

[2]consisting of a css, favicon, some text and a few pictures and maybe some JavaScript

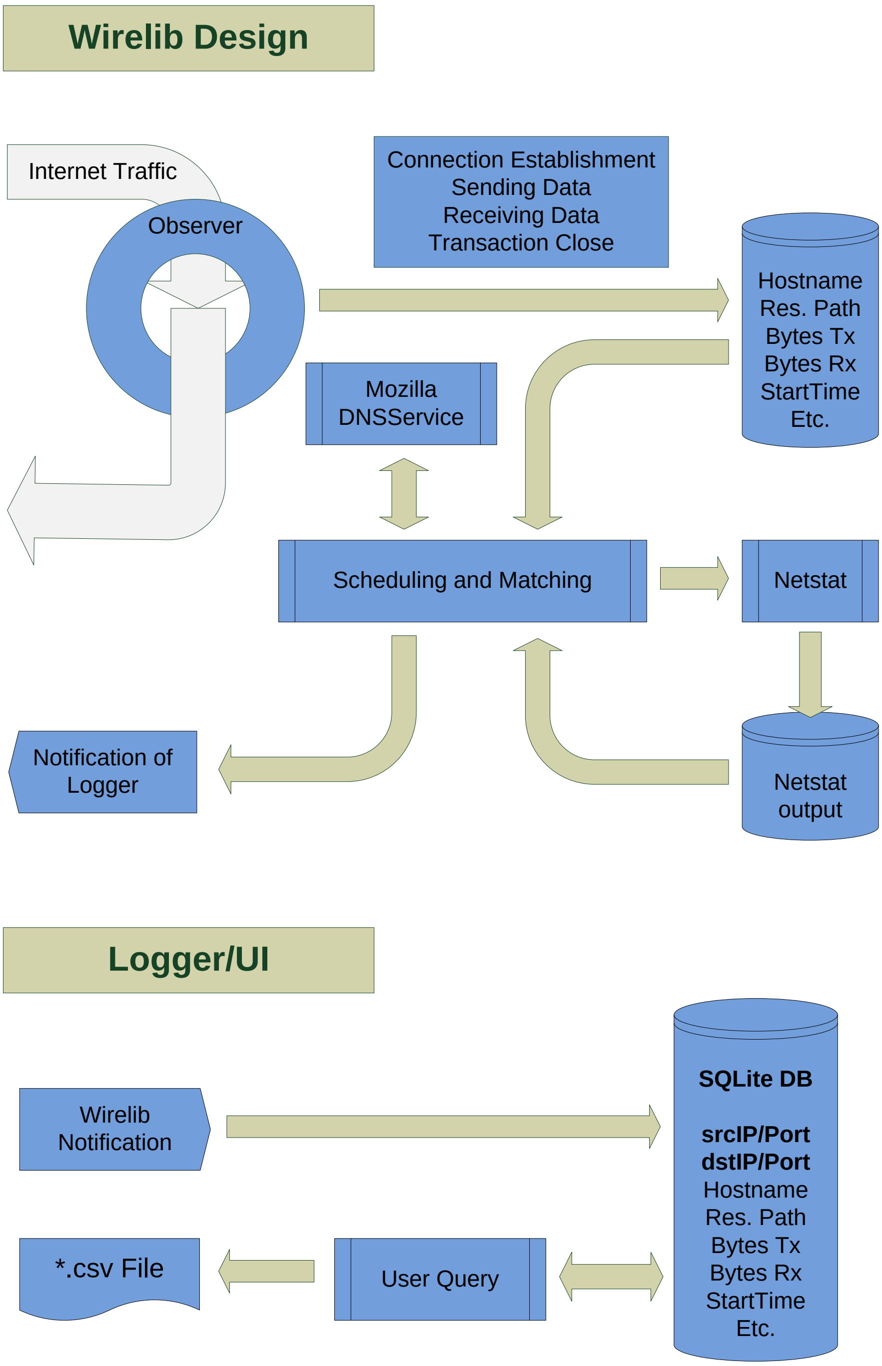


Figure 2.2: High level view of the library

Finally we decided for a hybrid approach. That means the completion of the connection records with IP and ports runs no longer immediately but periodically. Netstat is run in interval mode for X seconds (so far X has been 10). In interval mode Netstat writes an output every second. Then after those X seconds have passed Netstat gets terminated and restarted to write to a new file. The old file then can be parsed without worrying about hard drive access times.

2.2 Connection Matching

When counting the transmitted bytes, an important issue is to keep track of the connections. Here are some important events used later to explain the matching process:

Eventname	Available Data
CE (Connection Established)	URI, Time
SND (Sending To)	URI, #Bytes, #Bytes global, Time
RCV (Receiving From)	URI, #Bytes, #Bytes global, Time
CLS (Transmission Closed)	URI, Time

Table 2.1: Important events for connection matching

#Bytes global means the total amount of bytes sent on a specific connection, whereas #Bytes simply is the number of bytes transmitted in the current request or response.

At CE a record is created containing URI and starting time. In Http 1.0 SND or RCV are no big deal. A new connection is opened for every element we can just check the URI and compare host and path to find the corresponding record and add the bytes. In Http 1.1 where pipelining is allowed, things get a bit more sophisticated. We first have to distinguish whether it is a new connection or an old one getting reused. We can conclude it is a old connection if SND occurs without CE occurring before. To find out which of the previous connection was reused we have to update #Bytes global in the record at every SND. Now if SND occurs we check all previous records with the same host. We found the matching connection if following equation holds:

$$BytesGlobal_{oldConn} + Bytes_{newConn} = BytesGlobal_{newConn}$$

In addition we can estimate the duration of the tcp connection if we add a second timestamp to the record and update it at CLS.

2.3 Port Matching

In the following the process of finding the correct source ports is explained in more detail. As already stated at the end of section 2.1 this task is done periodically every 10 seconds. During this time all connection records are cached in memory. Every record has an additional entry identifying the corresponding Netstat output file. When the matching process starts all the records with corresponding Netstat file identifier get processed sequentially. With the starting time and response from the DNS system we can locate the respective rows in the Netstat file[3]. The parser can detect IPv4 and IPv6 addresses. The biggest problem now is that we have only a resolution of one second in the Netstat file which means from one output to the next there may be several new connections and our timing information in the records is of little help. A trivial approach is just to pick the first source port that matches our destination IP. This obviously leads to many falsely assigned ports. The best we could come up with was to assign source ports in the order presented by Netstat. This way we managed to get most of the ports right. As all of this post-processing should be done during browsing and recording new connections one has to pay attention that processing load does not become too heavy or it will cause the browser to lag or even become unresponsive.

2.4 Data Storage & Output

When the logger is notified, it stores all records in an SQLite database (see also fig. 2.2). There is a very simplistic UI to query the database which can be seen in fig. 2.4.
The database contains two tables[4]:

- netflow{ID, srcIP, srcPort, dstIP, dstPort, StartTime, EndTime, txBytes, rxBytes}
- addinfo{ID, Host, Path, PathHistory, Mime, dnsRecord}

There are to options to query the database. The first is to export a format very similar to Netflow where every record is split in two tuples {StartTime, EndTime, srcIP, srcPort, dstIP, dstPort, txBytes} and {StartTime, EndTime, dstIP, dstPort , srcIP, srcPort, rxBytes} with src and dst swapped respectively. The second option is to write your own SQL query and export the result to a ';'-separated .csv file.

[3]Remember that there is an output of Netstat written to the file every second an we have to pick the correct one

[4]This was just an arbitrary choice to keep things a bit structured

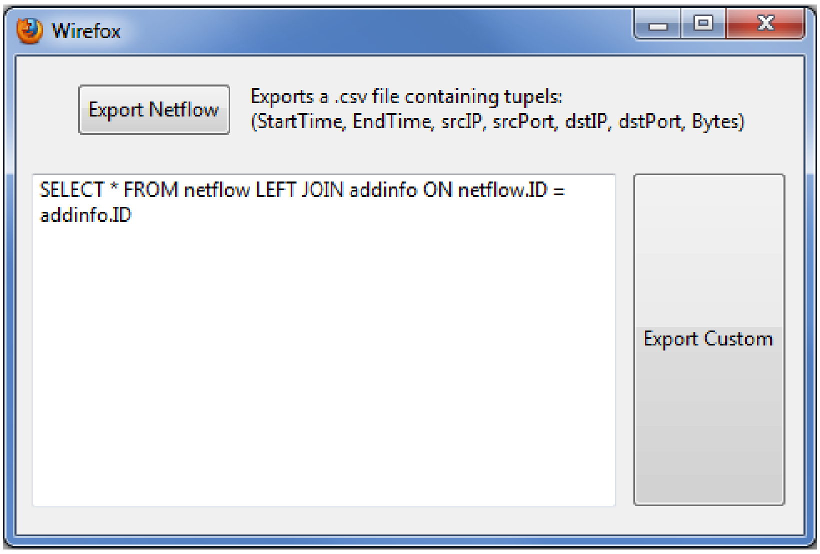


Figure 2.3: Database query interface

Chapter 3

Evaluation

3.1 Browsing Performance

As it is intended to run the add-on during browsing, it is important that browsing performance is not reduced to heavy. In other words, the browser should stay responsive and page load times should not increase to level making the user feel uncomfortable. Because those are highly subjective measures we are only going to restrict analysis in this section to purely numerical information. The impact of the add-on on load times was measured on two different systems:

- System A: 3.3GHz quadcore CPU, Ethernet
- System B: 2.0GHz dualcore CPU, 802.11g

Ten different homepages were visited 30 times in a cycle with caching disabled. The whole process was scripted which means, as soon as a page load is complete the next page gets loaded. This means that the add-on is constantly under stress, as permanently new connections are made and content is downloaded. The table 3.1 shows the worst-case results with the add-on disabled and enabled respectively.

System, Addonstatus	Total Time [s]	Increase
A, disabled	24.69	N.A.
A, enabled	72.93	295%
B, disabled	30.76	N.A.
B, enabled	159.12	517%

Table 3.1: Load time performance

The load time with add-on enabled is strongly correlated with CPU Speed. As JavaScript is a singlethreaded language it makes sense that the number

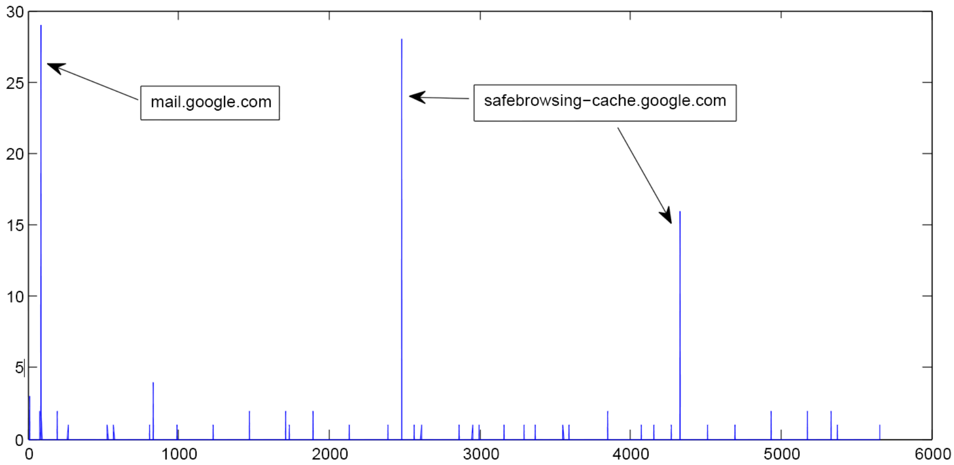


Figure 3.1: X-Axis: Time in Seconds, Y-Axis: Number of transactions

of cores does not influence the load time. The transition form wired to wireless connection decreases the performance in either case not depending whether the add-on is running or not.

3.2 AJAX Monitoring

This section discusses a sample application of Wirefox. Todays rich web application heavily rely on Ajax based technology. We used Wirefox to monitor the network traffic caused by google's email service Gmail. Figure 3.1 shows the measured activity over time.

The monitoring ran for about 1h 35min. The first big peak is the initial login to the webmail client. After the login there was no further interaction with the client or browser. Most of the small peaks are connections to gmail, hence there some activity every two to five minutes. There were also very few connections to standard news feeds of Firefox and two major peaks where connections to the safebrowsing service of google were made. It is very likely that the connections to the safebrowsing service are caused by the browser and not by gmail, because there were also connections to safebrowsing in various other tests where no google related page was visited[1]. Hence not only

[1]Indeed some digging in the web turned out the following statment: "'Phishing and Malware Protection works by checking the sites that you visit against lists of reported phishing and malware sites. These lists are automatically downloaded and updated every 30 minutes or so when the Phishing and Malware Protection features are enabled"'(which they are by default). This holds for Firefox version 3 and later. Source: http://en.community.dell.com/support-forums/virus-spyware/f/3522/p/19282391/19511756.aspx

Ajax applications cause traffic without the user being aware of something happening but also the browser itself does periodically some communication.

3.3 Packet/Flow Capture Performance

The packet capture performance of Wirefox was evaluated and compared against tcpdump. Wirefox as well as tcpdump were recording while a selection of homepages was visited. The comparison is restricted to the tuple {scrIp, srcPort, dstIP, dstPort, Bytes}. Nfdump was used to aggregate the output of Tcpdump and export the needed data.

A total of 6149 different flows were recorded by Tcpdump. For the time being the number of bytes is not considered in comparison of flows, as it is dependent of the level where the recording is done[2]. Wireshark recorded a total of 10331 transactions but only 4188 (65.24%) tuples {scrIP, srcPort, dstIP, dstPort} occured in those transactions. As discussed in section 2.3 the uncertainty lies mostly in the source port[3]. This means that the probability of assigning a wrong source port is roughly 34% (2231 tuples which were not matched).

Figure 3.2 shows what happens when the number of bytes transmitted on the flow is taken into account. The peak at distance 40 bytes indicates that there were at least some very accurate matches, because 40 bytes is exactly the sum of the tcp and IP header bytes. Actually there are some smaller peaks visible at 80, 120, 160 and 200 bytes which correspond to payloads being fragmented to two, three, four or five tcp packets respectively. Although, summed up over all multiples of 40, there are only 601 (9.77%) flows out of 6149 wich completetly matched tcpdump, it shows that it is at least partially possible to record netflows from within the browser.

[2]tcpdump records at ethernet level wich means that tcp and ip header bytes are counted as well, whereas Wireshark only sees the "‘payload"’

[3]Sometimes there are also connections which get nothing at all assigned if there was a parsing error

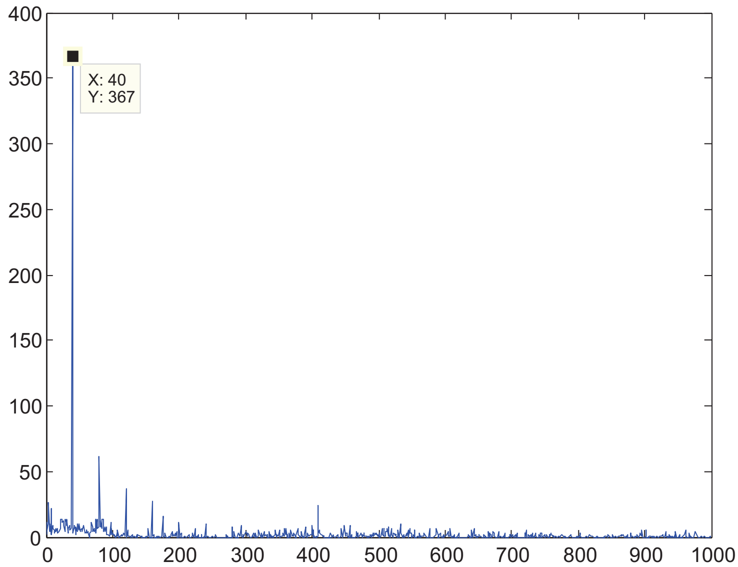


Figure 3.2: Bytedistance between Wirefox flows and tcpdump flows
X-Axis: $|Bytes_{Tcpdump} - Bytes_{Wirefox}|$ Y-Axis: Number of Wirefox flows

Chapter 4

Conclusions

One thing that may greatly improve both, performance and reliability would be to drop the resolution of the source IP address and the source port and focus on other useful information directly available from the browser. With this approach not very much information would be lost. It would still be possible to tell if a new or an existing connection was used for data transmission, which for example cannot be seen in Firebug. The only thing missing is the exact source port number. If really needed, the mapping to source IP and ports could then be done in a post-processing step with additional data from a packet capturing tool which would provide much better time resolution than Netstat. So far the Netstat-related part has been the biggest source of performance decrease and portability problems but there is also room for improvements in the matching algorithms.

During development we also found that certain resources and interfaces provided by Firefox behave quite different on different operating systems. For example some blocking methods do not block on one OS while they do on the other. This made cross-platform development very difficult and the add-on ended up running only on Windows and Linux but not on Mac OS.

In its current state the extension is not very reliable if one is interested in the exact endpoints (port numbers). Nevertheless as we have seen in chapter 3 traffic monitoring in principle is very well possible and we can already get useful information about the behaviour of Ajax based Internet applications. Taking possible improvements, e.g. those mentioned above, into consideration there is definitely great potential in this approach, also keeping in mind that version 4 of Firefox is coming out soon with an improved JavaScript engine.

Appendix A

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für
Technische Informatik und
Kommunikationsnetze

Semester Thesis

for

Manuel Widmer

Tutors: Dominik Schatzmann, Wolfgang Mühlbauer

Issue Date: 20.09.2010
Submission Date: X.12.2010

WireFox – A Measurement Plug-In for Firefox

---

1 Introduction

Today, web browsers such as Firefox, IE, or Chrome are used in the combination with Ajax based technology to provide Rich Internet Applications (RIA). These new and often very popular RIAs produce new traffic patterns. For example the network traffic created by Gmail's Javascript shows a strong time correlation that is even measurable from traffic statistics collected at the Internet backbone.

2 Requirements

The task of this thesis is to design and implement a plug-in to capture network measurement traces directly inside the browser. The captured traces should contain similar information as exported by Wireshark (pcap) or Internet Routers (netflow).

This task is split into four major subtasks: (i) literature study on current work about endhost measurements with the focus on browser-based systems, (ii) design and implementation of a library to extract network related information from the browser. (iii) implementation of a plug-in for Firefox that exports similar information as Netflow, and (iv) testing and evaluating this measurement plug-in using real user data.

2.1 Literature study

The student should actively search and study literature and write a short survey on comparable approaches. The focus are endhost-based measurement platforms.

2.2 Design of the library

The library should allow the user to extract network related information such as communication end-points (IP addresses), duration of the communication, or number of exchanged bytes or packets. Preferably, the complexity of the library is increased step by step. The library should be designed to work in real-time, should be implemented in JavaScript, and should run within normal Firefox.

2.3 The measurement plug-in

The measurement plug-in should export similar information as Netflow. Furthermore, the plug-in should export tags to describe the current activity.

2.4 Evaluation of the application

The efficiency and accuracy of the application has to be evaluated with real data.

3 Deliverables

The following results are expected:
- Short survey on literature research.
- Design of a library to collect network-related information from a browser session
- Implementation of a measurement plug-in that exports similar information as netflow
- Evaluation of the application with real data
- A final report, i.e. a concise description of the work conducted in this project (motivation, related work, own approach, implementation, results and outlook). The abstract of the documentation has to be written in both English and German. The original task description is to be put in the appendix of the documentation. The documentation needs to be submitted electronically. The whole documentation, as well as the source code, slides of the talk etc., need to be archived in a printable, respectively executable version on a CDROM.

4 Assessment Criteria

The work will be assessed along the following lines:

1. Knowledge and skills
2. Methodology and approach
3. Dedication
4. Quality of results
5. Presentations
6. Report

5 Organisational Aspects

5.1 Documentation and presentation

A documentation that states the steps conducted, lessons learned, major results, and an outlook on future work and unsolved problems has to be written. The code should be documented well enough such that it can be extended by another developer within reasonable time. At the end of the project, a presentation will have to be given at TIK that states the core tasks and results of this project. If important new research results are found, the results can be published in a research paper.

5.2 Dates

This project starts on September 20, 2010 and is finished on December 31, 2010. At the end of the second week the student has to provide a schedule for the thesis, that will be discussed with the supervisors.

An intermediate presentations for Prof. Plattner and all supervisors will be scheduled after one month.

A final presentation at TIK will be scheduled close to the completion date of the project. The presentation consists of a 20 minutes talk and reserves 5 minutes for questions. Informal meetings with the supervisors will be announced and organised on demand.

5.3 Supervisors

Dominik Schatzmann, schatzmann@tik.ee.ethz.ch, +41 44 632 54 47, ETZ G 95
Wolfgang Mühlbauer, muehlbauer@tik.ee.ethz.ch, +41 44 632 70 17, ETZ G 90

References

17th February 2011

Bibliography

[1] D. Joumblatt, R. Teixeira, J. Chandrashekar, N. Taft, "Perspectives on Tracing End-Hosts: A Survey Summary", CCR online, the Computer Communication Review, April 2010

[2] R. Teixeira, "Network Troubleshooting from End-Hosts", HDR Dissertation, Universite Pierre et Marie Currie, May 2010

[3] Mozilla Developer Network (Javascript and XUL reference, Codesnippets, etc) `https://developer.mozilla.org/en/`

[4] SQLite Syntax reference `http://www.sqlite.org/lang.html`

[5] Article on Firefox's SafeBrowsing Feature `http://en.community.dell.com/support-forums/virus-spyware/f/3522/p/19282391/19511756.aspx`