Mobile Multi-protocol HTTP Proxy

Yang Wang[1,2]

[1] Nicta, Sydney. Eveleigh, NSW, Australia, first.last@nicta.com.au
[2] Université de Technologie de Compiègne, France
*Nicta supervisor and corresponding contact: Olivier Mehani[1]

Published: 16 October 2012

Acknowledgements

I would like to express my gratitude to all those who helped me during this internship, especially my supervisor Olivier Mehani, Prashanthi Jayawardhane, Gillian Adidi, and Ben Mason for their help in all the administration stuff, the directors of NICTA who gave me this precious occasion, Valérie Texier and Véronique Cherfaoui for their support from the University of Technology of Compiègne, and all colleagues of NICTA with whom I passed a very nice period.

Abstract

Multimedia content is becoming the most prominent traffic over the Internet. However the transport of multimedia objects between mobile devises and web servers is slowed down due to the use of the TCP protocol because its congestion control, reliability, and in-order delivery take time when the loss of a few packets is not important. The development of an HTTP multi-protocol proxy is one of the reasonable solutions to mitigate this problem.

This internship is focused on developing a light multi-protocol HTTP proxy which supports different transport protocols such as Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Stream Control Transmission Protocol (SCTP) or Datagram Congestion Control Protocol (DCCP). At first, we present how to make a simple transparent proxy using the TCP protocol. Then we explain and extend the Polipo proxy, an Open-source software supporting IPv4 and IPv6, to detect the type of data to transfer from the web server and to select the most appropriate transport protocol. Last, we finish by assembling the previous works of the development of the transparent proxy and the extension of Polipo to make our multi-protocol HTTP proxy.

Environment tools:

- Programming language: C
- Operation system: Ubuntu 11.10 (Oneiric), Kernel Linux 3.0.0-12-generic
- Development tools: Gedit, Terminator
- Version control system: Git
- Compilation, debug & test tools: GCC, GDB, DDD, Mozilla Firefox
- Documentation tools: Kile, Dia, Meld, LibreOffice Draw, Microsoft Power Point

Key words: HTTP, socket, proxy, transport protocol, HTML5, C, mobile

Contents

Chapter 1

Context and State of the Art

Introduction

With the growing popularity of HTML5, multimedia content is more and more used on mobile devices. However, the use TCP to transport all content is not neccessarily appropriate. In case where the loss of few packets might not be so important, TCP retransmissions might introduce unwanted delay. On the other hand, many transport protocols such as UDP, SCTP or DCCP are available for the transmission of data. The objective of this internship is to develop a multi-protocol proxy able to select the most appropriate transport protocol depending on the content type of the object transferred over HTTP.

We begin by presenting the context of this internship at NICTA, the SAIL project, and background information about the Internet communication protocols. Then, we explain how we made a simple transparent proxy able to receive requests from the client application, send them to the web server, receive the answers from the web server, and transfer them to the client application. This step allows us to understand the basic technologies behind an HTTP proxy. We then present an open source proxy named Polipo and extend its functionality. We finish by taking the simple proxy and the completed Polipo code to create a full multi-protocol proxy.

1.1 National ICT Australia

National ICT Australia Ltd (NICTA) [1] is Australia's Information and Communications Technology research centre of Excellence. Established in 2002 by the Federal Government as part of the Backing Australia's Ability initiative, and funded by the Australian Government through the Department of Broadband, Communications and the Digital Economy and the Australian Research Council under the ICT Centre of Excellence Program, it is also supported by the New South Wales, Queensland and Victoria Governments, and many Australian universities.

There are more than 600 researchers and PhD students in the 5 laboratories around the country. Aiming to pursue high-impact research excellence and to create national benefit and wealth for Australia, their research is divided in 6 domains:

- Computer Vision

[1] http://www.nicta.com.au/

- Machine Learning
- Networks
- Optimization
- Software Systems
- Control and Signal Processing

This internship was done in the Networking Research Group. It was part of its involvement with the SAIL project.

1.2 SAIL Project

Today's Internet rests on a foundation of technologies, needs and visions that emerged 40 years ago. SAIL [11] is a project leading a consortium of 25 operators, vendors and research institutions which aims to design technologies for the networks of the future and develop techniques to transition from today's networks to such future concepts.

The research of SAIL is organized in four principal areas:

- Network of Information (NetInf)
- Cloud Networking (CloNe)
- Open Connectivity Services (OCons) [1, 9]
- Migration, Standardization, Business and Socio-Economics

As one of the 25 participants, NICTA is doing prototyping work for the Open Connectivity Services and integration of research outcomes [4, 10].

1.3 State of the art

1.3.1 HTTP Protocol

The HTTP protocol [5] is an application layer protocol of the Open Systems Interconnection (OSI) model. It is the foundation for web transmission. HTTP works following a client/server model. The client application submits a request message to the server, and the server transfers the objects to the client. The most common HTTP application is a web browser which prepares and sends HTTP requests when an user visits a web site. The web server application, such as Apache,[2] receives the HTTP requests, and sends the required data back to the client application. Figure 1.1 shows the model of the HTTP communication.

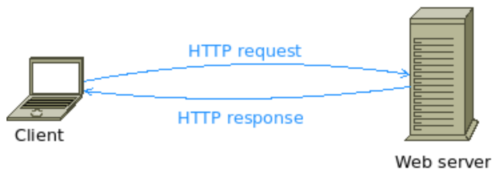

Figure 1.1: HTTP communication. HTTP works following a client/server model. The client sends requests, such as a `GET` request, to ask for a ressource from server, and the server returns ressources with HTTP headers..

```
GET http://www.google.com.au/ HTTP/1.1
Host: www.google.com.au
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux i686; rv
   :11.0) Gecko/20100101 Firefox/11.0
Accept: text/html,application/xhtml+xml,application/xml;q
   =0.9
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Proxy-Connection: keep-alive
```

Figure 1.2: HTTP request. The first line contains the method, the URL, and the protocole version. The following lines provide information about the client, such as the browser or the supported language..

HTTP Request

As figure 1.2 shows, the first line of an HTTP request is composed of three elements: the HTTP method, the Uniform Resource Locator (URL), and the protocol version.

The usual HTTP methods are `GET`, `HEAD`, `POST`, `PUT`, and `DELETE`. The `GET` method requests a complete HTTP response which is the combination of the HTTP header, containing the information about the requested content, and the body of the content, separated from the header by an empty line. The `HEAD` method requests the server to only send the HTTP header. This method is useful for caching proxies which have to check whether the local data has expired by parsing the `Last-Modified` information. If the last-modified date is earlier than local data, the proxy can send this data directly when a client requests, otherwise the proxy forwards the `GET` request to the web server to update the data. When a client application needs to send some information, it can use the `PUT` or `POST` methods to the server. These two methods are very similar, the difference is that if the object already exists on the server, `PUT` replaces the original data and `POST` lets the server determine what to do with this data. The `DELETE` method allows the client to delete data from the server, but this command is denied by almost all server configurations for security reasons.

HTTP Response

As presented in figure 1.3, an HTTP response is composed of an HTTP header and the body of content after an empty line. The first line of the HTTP response header contains the version of the protocol and the status code. HTTP has different versions, each version has its specification. The status code advises the state of HTTP response. The 2xx class of status codes indicates that the client's request was successfully received, understood, and accepted. The 3xx class indicates that the user-agent can select a preferred representation or redirect its request to that location. The 4xx class indicates the client appears to have erred. The 5xx class signifies there is an error of the web server's side.

The other lines of HTTP response header respect the `name: value` format. These lines comprise the information about the content in the body of the HTTP response, such as the `Cookie`, the `Content-Type` or the `Content-Length`.

HTTP/1.1

The first documented version of HTTP protocol, HTTP/0.9, was published in 1991.[3] Since then, the HTTP protocol has developed. The latest version is HTTP/1.1 [5]. Compared with HTTP/1.0 which has been most used in the last years, the new version has a number of new features. They include:

Persistent connections The HTTP/1.0 protocol closes the connection after receiving an object, and creates a new connection for the next request to the same server. By adding connection negotiation mechanisms, the connection can `Keep-Alive` after the transmission of one document. This mechanism

[2] http://www.apache.org/

[3] http://www.w3.org/Protocols/HTTP/AsImplemented.html

```
HTTP/1.1 200 OK
Date: Sun, 26 Aug 2012 07:13:20 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Set-Cookie: PREF=ID=b49a8a136c8......
Server: gws
Content-Length: 26997
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

<!doctype html><html itemscope="itemscope"
itemtype="http://schema.org/WebPage"><head><meta itemprop
   ="image"
content="/images/google_favicon_128.png"><title>Google</
   title>
......
```

Figure 1.3: HTTP reply. An HTTP reply contains an HTTP header and the content after an empty line. The HTTP header provides the protocol version, status code and the status string in the first line, and information about the server in the following lines, such as the time of server, the type of content or the length of content..

avoids connecting many times to the same server and retains the rate of the Internet transmission.

Caching The caching mechanism exists in the HTTP/1.0 version. It was controlled by time. The `Expires` value and the `If-Modified-Since` date are used for the validity of a web page. In case of clock skew, the web client cannot correctly verify the update from the web server. The new HTTP version added an `Etag` header to identify objects. If the local `Etag` value equals the `Etag` value of the HTTP header from server, the local data is assumed to be up-to-date and is sent to the client.

Pipelining If both the client and web server support pipelining, the client can make several HTTP requests without waiting for their responses. This technique decreases the number of Internet connections as well as benefit from the transfer rate of previous transmission.

`OPTIONS` **method** The `OPTIONS` method allows to obtain information about the capabilities of a server without actually requesting a resource. This method is not currently supported by most web servers.

As HTTP uses TCP as transport protocol, we now take a look at this protocol. It has some limitations in this context, which other protocols could alleviate, we present them too.

1.3.2 TCP protocol

Transmission Control Protocol (TCP) [2] is a connection-oriented transport protocol. It is used by major web applications requiring reliable connection such as the World Wide Web (WWW), file transfers or email.

The TCP communication is divided by 3 phases, the establishment of connection, the transmission of data, and the termination of connection. Before sending data, the system creates a TCP connection which stays alive until both sender and receiver receives a command. During the TCP communication, the receiver sends regular acknowledgment messages in order to advise the state of received data. When a packet is lost, the sender receives a failed acknowledgment message and resends this data.

The use of TCP makes transmission reliable, avoids a lot of transport problems such as losses, duplication or error in packets. On the other hand, the establishment of connection and the acknowledgement mechanism introduce too much delay for several types of data which do not require relability.

1.3.3 UDP protocol

UDP [8] is a connection-less transport protocol. It is used for transmissions for which error detection and correction are not nessesary. As a connection-less protocol, there is no end-to-end semantics for UDP transmission, packets are sent individually. It is possible that the first message will not reach the receiving application first. Contrary to TCP, UDP has no acknowledgment mechanism, a message can be lost in transmission without detection, is unreliable. Another difference comparing with

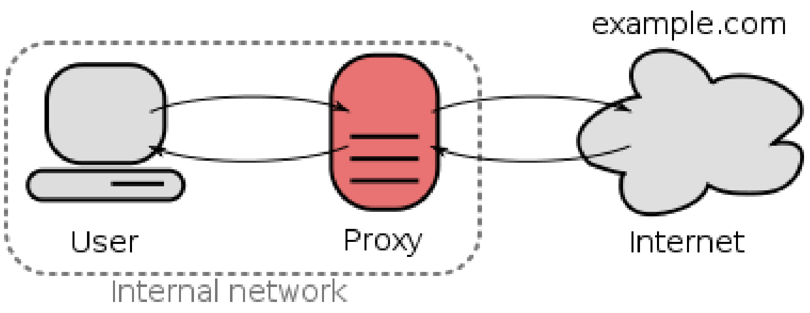

Figure 1.4: A proxy sits between the client et the server, it transfers requests from the client to the server, and relays responses to the client from the server.. *http://en.wikipedia.org/wiki/File:Forward_proxy_h2g2bob.svg*

TCP is that UDP has no congestion control. This makes UDP useful for multiple multimedia communications such as voice over IP. Therefore, UDP is usually used for multimedia transmission because it does not introduce delays.

1.3.4 Other transport protocols

Sometimes, the characteristics of TCP or UDP are not suitable for certain types of Internet transmission, there are other transport protocols which can provide some of the advantages of both TCP and UDP.

SCTP [12] SCTP is a message-oriented protocol. Like TCP, this protocol maintains a relationship between endpoints until all data has been successfully transmitted. With SCTP transmission, data can be sent without error and in sequence, with a clear delimitation of packets.

DCCP [6] DCCP is a transport protocol that provides bidirectional unicast connections of congestion-controlled unreliable datagrams.

1.3.5 Proxy

A proxy is an application acting as the intermediary between a client application and a server application. The client application, which seeks resources such as web pages in the case of HTTP, multimedia content, or services from servers, can send requests to the proxy. The proxy receives the requests, manipulates data, and relays it to the server. When the proxy receives data from the server, it tansfers it back to the client. Figure 1.4 shows the position of proxy in a usual communication.

The use of proxy has many advantages:

Security As the proxy can be the only point to connect from the external network, external machines have no way to communicate directly with the internal network without going through the proxy.

Management Proxies can control access to external networks by filtering addresses.

Performance When a client seeks a resource existing in the cache memory of the proxy, the proxy can provide this copy, rather than making a connection with the server, to speed up the communication and to save the upstream capacity.

1.4 Objectives

The objective of this internship is to implement a multi-protocol proxy for mobiles. This proxy can be executed on the client and the server side. The client-side proxy should be able to receive requests from client applications, send `HEAD` requests to the server to detect the type of content, select the appropriate transport protocol to send the `GET` request to communicate with the server, and transfer the server's responses to the client application. The server-side proxy is installed on the web server, it listens on different ports, for all supported transport protocols, to communicate with the client-side proxy and uses TCP to communicate with the web server.

Because the Linux system is reliable and provides a lot of development tools for networking programing, we decided to use is as the operating system, and C as the programing language.

To make this proxy, two solutions are possible:

- Write a proxy from zero
- Use an existing off the shelf proxy and add multi-protocol features

It is a good idea to use an existing off the shelf proxy, because it will have more features and will be more stable. However, the code of the proxy is likely to be complex to read and no open source proxy supports multi protocol operation on the server side. Therefore the best solution is to start with a simple proxy in order to investigate basic proxy operation. Then, we select and study an open source proxy to modify the code for the client side. Last, we assemble the client side and the server side.

In the rest of this document is structured as follow. In chapter 2, we present the implementation of a simple proxy which can only transfer data between the client and the server. In chapter 3, we explain the extension of an open source proxy. In chapter 4, we descreibe the development of the multi-protocols proxy. Finally, in chapter 5, we summerase the work presented here as well as what is left for future work.

Chapter 2

Development of a simple HTTP proxy

In this chapter, we develop a simple transparent HTTP Proxy which only transfers data between the client application and the web server using the TCP protocol. The implementation of this transparent proxy is divided in 3 steps: the development of proxy on the client side, on the server side, and the use of the synchronous I/O multiplexing techniques.

2.1 Proxy on the client side

The client-side proxy should be able to listen on a TCP port (*e.g.*, 3128) to receive requests from client applications, then connect to the server and exchange data before closing the connection.

2.1.1 Socket creation

```
int socket(int domain, int type, int protocol);
```

The `socket()` function allows to create a socket. The `domain` indicates the communication domain, `AF_INET` for IPv4 addressing, `AF_INET6` for IPv6 or `AF_LOCAL` for local communication. The type of socket is specified by the `type` option. The value of this option can be `SOCK_STREAM` for connection-based byte streams, or `SOCK_DGRAM` for connection-less communication. The `protocol` option allows to select a transport protocol. We can include `arpa/inet.h` file to use the name of protocol such as `IPPROTO_TCP` or `IPPROTO_UDP`. This function returns the descriptor of the created socket, or -1 on error.

2.1.2 Address preparation

Figure 2.1 shows the preparation of the network address. We create a `struct sockaddr_in` variable named `servaddr` which contains the IP address, the port, the address family, and 8 additional bits in order to be compatible with the `struct sockaddr` structure which is the generic format of network address.

The `htons()` and `inet_pton()` functions convert the integer port number and the IP address string to the network format.

```c
    struct sockaddr_in servaddr;
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(8123);
    if( inet_pton(AF_INET, "192.168.1.10", &servaddr.
       sin_addr) <= 0){
      printf("inet_pton error for %s\n",argv[1]);
      exit(0);
    }
```

Figure 2.1: Preparation of a network address for socket.

2.1.3 Connection to the server

```c
    int connect(int sockfd, const struct sockaddr *addr,
       socklen_t addrlen);
```

Once the socket and the network address are prepared, we can connect to the server with the `connect()` function. The parameters are the file descriptor of the socket, the network address, and the length of this address, normally calculated by the `sizeof()` function. If the connection is created successfuly, the function returns 0, otherwise, the function returns -1.

2.1.4 Sending/receiving data

```c
    ssize_t read(int fd, void *buf, size_t count);
    ssize_t write(int fd, const void *buf, size_t count);

    ssize_t recv(int sockfd, void *buf, size_t len, int
       flags);
    ssize_t send(int sockfd, const void *buf, size_t len,
        int flags);
```

`read()/write()` and `recv()/send()` are two groups of function for receiving/sending data. The `read()/write()` functions allow to receive/send `count` bits from/to the buffer pointed by `buf` to the number of socket `fd`. The `recv()/send()` functions are very similar, they have `flags` that we don't use at the moment.

2.1.5 Close socket

```c
    int close(int fd);
```

After sending/receiving data, it is important to close the socket connection to free the system resources. The function is `close()`, and the only parameter is the descriptor of socket.

2.2 Proxy on the server side

The proxy on the server side is an application which listens on a TCP port to receive client requests and sends them to the web server listening on another port, then receives responses from the web server and transfers them to the client.

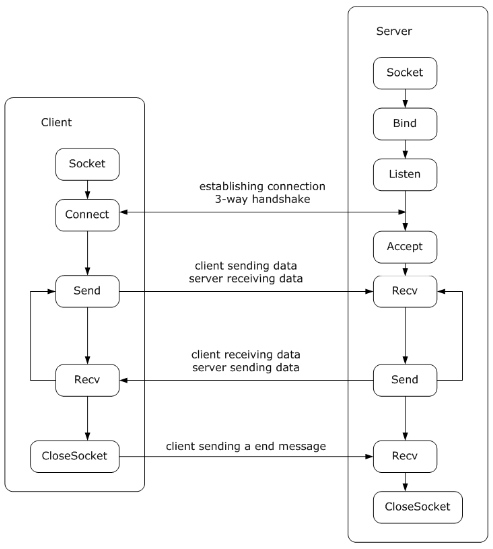

Figure 2.2: Diagram of Socket usage for TCP connection..
https://upload.wikimedia.org/wikipedia/commons/a/a1/InternetSocketBasicDiagram_zhtw.png

As figure 2.2 shows, the process of creating a TCP connection by using a socket on the server side is different from the client side. On the server side, the proxy has to use the `bind()`, `listen()`, and `accept()` functions.

2.2.1 bind() function

```
int bind(int sockfd, const struct sockaddr *addr,
   socklen_t addrlen);
```

After creating the socket `sockfd`, the server-side proxy has to listen on address `addr`. The `addrlen` value is the length of `struct sockaddr`. This operation makes the server socket accessible by other network devices.

2.2.2 listen() function

```
int listen(int sockfd, int backlog);
```

The `listen()` function marks the socket as a passive receiving socket. This function makes the socket able to accept new client connection. The `backlog` option is the maximum number of clients that the proxy can have waiting.

2.2.3 accept() function

```
int accept(int sockfd, struct sockaddr *addr,
   socklen_t *addrlen);
```

When the proxy receives a new connection request, a client socket is created by the `accept()` function. This function stores the network address of the client application in addr, and returns the descriptor of the newly created socket. The proxy can then use this socket to communicate with the client application.

2.3 Infinite loop

In order to allow the proxy to stay active after exchanges of information and to receive new client requests, we put the accept(), read(), and write() functions in a permanent while(1) loop. However, the socket system is set in blocking mode, when the proxy is sending/receiving data, all sockets are blocked, therefore the new client connections would not be accepted.

To resolve this problem, we first created new processes with the fork() system call when new clients connected. Thereby, the proxy could receive new client connections and treat clients in different processes. However, the creation of new processes requires more system resources and slowed down the data transmission. In our experience, it took about 15 minutes to open the Facebook home page which contains the HTML text, CSS/JavaScript files, and images.

Another solution is to implement synchronous Input/Output Multiplexing mode. In this mode, the proxy is blocked by the select() or poll() function which monitors all sockets. When there is data to read/write, the read()/write() function is activated and the proxy is blocked again by select() or poll() until there is data to exchange on any monitored socket. As the select() and poll() functions are similar, we present the select() function in this section. The poll() will be described in a later section (section 3.2.1 page 19).

```
    int select(int nfds, fd_set *readfds, fd_set
      *writefds, fd_set *exceptfds,
struct timeval *timeout);
```

The select() function allows to monitor multiple file descriptors, it blocks the proxy until one of the sockets changes state. The usual parameters of the select() function are nfds, which indicates the number of sockets to monitor, and readfds, which is a list of all sockets to monitor.

Figure 2.3 is an example of the use of the select() function for sockets. We create a variable of type fd_set named fdset which is a list of sockets to monitor. Then, we initialize the list by adding clients and server sockets in the list. The program is blocked by the select() function until a socket changes state. If there is data to read/write in the buffer for the client socket, the function FD_ISSET(sock_client) returns a non-NULL value which triggers processing of the requests from client applications in the if statement. If there are objects to read/write in the buffer for the server socket, the FD_ISSET(sock_server) function returns a NON_NULL value which triggers processing of the responses from server. After that, the program is blocked again by the select() function.

By using a synchronous Input/Output Multiplexing technique, the proxy is much faster than before, and its performance impact is no longer noticeable.

After implementing a simple proxy which repeats data between the client application and the web server that we will reuse as the base for the server-side proxy in chapter 4, we can use an existing open source proxy and add the required functionalities by modifying its source code.

```c
fd_set fdset;
while(1){
    /* Initialization */
    FD_ZERO(&fdset);
    /* Add client socket to monitor */
    FD_SET(sock_client, &fdset);
    /* Add server socket to monitor */
    FD_SET(sock_server, &fdset);

    /* The select function monitors the sockets in fdset */
    if(select(FD_SETSIZE, &fdset, NULL, NULL, NULL)<0){
        mppx_log(conf, -1, "Error: select\n");
        exit(-1);
    }
    /* In case that a client socket's state has changed */
    if(FD_ISSET(sock_client, &fdset)){
        /* Read request from client */
        if((iolen = receive(sock_client, buff_in, conf)) <= 0)
            break;
        /* Connect to server */
        sock_server = connect_server(&send_addr_out, host,conf);
        /* Send request to server */
        send_message(sock_server, buff_in, iolen, conf);
    }
    /* In case that the server socket is set */
    if(FD_ISSET(sock_server, &fdset)){
            /* Read response from server */
        if((iolen = receive(sock_server, buff_out, conf)) <= 0)
            break;
        /* Send response to client */
        send_message(sock_client, buff_out, iolen, conf);
    }
}
```

Figure 2.3: Use of the `select()` function.

Chapter 3

Extension of an open source proxy

Polipo is an open source proxy with multiple features [3]. In this chapter, we introduce its background, usage, and architecture. After presenting the main functions of Polipo, we modify its code to send HTTP `HEAD` requests, receive replies from the server, and determine the type of transferred content before sending the HTTP `GET` request.

3.1 Presentation of Polipo

3.1.1 Context

Polipo is a caching web proxy, it is designed for use by one person or a small group of people. First published by Juliusz Chroboczek in 2003, it is a free software released under the MIT License.[1] Since 2009, the Polipo code is maintained by Chrisd who was a 2009 Google Summer of Code for Tor/EFF. As Polipo is meant for small networks, Chrisd worked on integrating support for Libevent which is a library for managing I/O events portably. [2]

3.1.2 Particularity

Compared with the simple proxy of chapter 2, Polipo has several advantages.

Multiple HTTP versions support Polipo supports new features of HTTP/1.1 [5], such as the connection negotiation mechanism, and new HTTP header fields like `ETag`. It can use HTTP/1.1 to communicate with a server even when the client only supports HTTP/1.0.

Caching Polipo can store transferred objects in its cache in order to resend to other client requesting them. This mechanism avoids creating new connections with the server for the same request and saves bandwidth.

String control Polipo has strong capacities to control incoming strings. Functions such as `strcasecmp_n()` or `httpParseHeaders()` can compare strings

[1]MIT (Massachusetts Institute of Technology): It is free software license which allows users to *use, copy, modify, merge, publish, distribute, sublicense, and/or sell* copies of the Software

[2]https://blog.torproject.org/blog/polipo-portability-enhancements-summary

or parse HTTP headers. This allows Polipo to extract information even when in erroneous format.

IP address filtering By default, Polipo accepts requests only from `localhost`. Administrators have to register the allowed client IP addresses in the `allowedClients` variable.

Easy access to configuration With the `polipo -v` command, all configuration variables via the Polipo are documented. These variables can be set in the configuration file of Polipo, or simply command line. For example, `polipo logFile=log.txt` indicates that Polipo should write its log in file `log.txt` rather than on the terminal.

Web interface Polipo also has a web interface accessible at address `http://localhost:xxx` (xxx is the listening port of Polipo, 8123 by default). This interface shows all configuration variables as well as the manual of Polipo.

Compared other proxies on the market, we chose Polipo because it has advantages listed below.

Light Mobile devices have limited capacity, a proxy for these cannot consume a lot of resources. Polipo was designed as personal proxy and is a light application.

Pipelining Polipo is able to send several requests back to back, and parse responses from the server to check whether it supports HTTP pipelining. The use of pipelining saves bandwidth and speeds up network transmissions.

C language Polipo is written in C. This language makes it easier to port to other Unix-based systems.

Open source Polipo is an open source software. Any developer is permitted to use, modify, and republish the code source of Polipo.

Community Polipo also has a large community. Whoever registered in the community[3] and needs help to use or modify Polipo can send emails to the mailing list [4] and discuss with other people.

As Polipo has many advantages, we decided to extend it with multiprotocol support for multiple input protocols.

3.1.3 Installation

Polipo is available from the Ubuntu repositories. We can use `apt-get install polipo` to install Polipo or `apt-get source polipo` to get its source code.

On the other hand, the best way to get the source code of Polipo is to clone its git repository, because the last version is always on this platform, to do so we can use the `git` version control system.

```
git clone git://git.wifi.pps.jussieu.fr/polipo
```

[3]https://lists.sourceforge.net/lists/listinfo/polipo-users

[4]polipo-users@lists.sourceforge.net

3.1.4 Configuration

The `polipo -v` command line shows the configuration variables and their values. To modify these, two ways are possible: by the configuration file or by command line. The default configuration file, is `/etc/polipo/config`. One can also use `polipo -c filename` to specify the file to use. When the modification is just for one run, we can use `polipo --name=value` to specify a value to the name of configuration variable.

3.2 Structure of Polipo

Polipo uses `poll()` to manage the Input/Output streams, therefore we first present the `poll()` function. We also present the main architecture of Polipo. As Polipo has data structures, we then present them and their relationship. We finish by presenting how pipelining is implemented in Polipo.

3.2.1 `poll()` function

```

        struct pollfd {
            int   fd;           /* file descriptor */
            short events;       /* requested events */
            short revents;      /* returned events */
        };

        int poll(struct pollfd *fds, nfds_t nfds, int
            timeout);
```

As presented in section 2.3 page 14, the `poll()` function blocks the program until there is data to send/receive. The first parameter of `poll()` is a list of type `struct pollfd` which contains file descriptor, `events` to mornitor, and `revents` which are events triggered by the function indicates which event happened. The events are represented by integer numbers definined in `poll.h`, for example, `POLLIN` for data to read or `POLLOUT` for data to write. The combination of `POLLIN & POLLOUT` in `events` specifies that both `POLLIN` and `POLLOUT` can be triggered for this file descriptor. When there is data to read/write, `poll()` fills the events in `revents`, checks the value of `revents` to be informed which event has happened. The third parameter, `timeout`, specifies an upper limit on the time for which `poll()` will block, in milliseconds. This function returns the number of file descriptors which have data to exchange. If a timeout accored, it returns the value 0. On error, it returns -1 and sets `errno`.

3.2.2 Architecture of Polipo

In order to understand the operation of Polipo, we added functions to trace function calls by using GCC[5] macros `__FUNCTION__`, `__FILE__`, `__LINE__`, and we used GDB[6] to check the behaviors of the main functions step by step.

[5]The GNU Compiler Collection is a open source compiler supporting multiple programming languages such as C, C++ or objective C.

[6]The GNU Debugger is the standard debugger for the GNU operating system and derivatives.

The `poll()` function is in a `while(1)` infinite loop of the `eventLoop()` function in `event.c`. In parallel, there is a list of type `struct FdEvents` which contains file descriptors events to monitor, and the behavior when there is data to treat for a given file descriptor. When a client request or a server response comes, the `findEvent()` function searches `FdEvent` from the list by checking whether its `revents` value is not `NULL`, and calls the function informed by `data`. For example, after creating a listener socket, the `create_listener()` function calls `schedule_accept()` which stores the listener socket in the `pollfds` and the `FdEvents` lists with the function `do_scheduled_accept()` as `data`. When a new client connection comes, `poll()` function sets `revents` to `POLLIN`, the `findEvent()` function finds the file descriptor and calls the function `do_scheduled_accept()` informed by `data` of the the event corresponding to this file descriptor and calls the `accept()` function to accept this client.

Figure 3.1 is the flowchart of Polipo. When a new client request comes, Polipo reads the buffer to parse HTTP headers, and tries to find the server connection in memory. If a connection to this server was used, Polipo uses this server to send the client request, otherwise, Polipo can find the server via the Domain Name Server (DNS) protocol. Once a connection with the web server is established, Polipo sends client requests to the server, receives server responses, and parses HTTP headers. If the `Content-Length` value is bigger than the length of the received chunk, the received content is not complete. Polipo transfers this packet to the client and waits to receive subsequent packets. This functionality is realized by the `httpParseHeaders()` function called by `HTTPServerReplyHandler()=>httpServerHandlerHeaders()` which compares the number of received bytes with the `Content-Length` to determine whether the content is complete. It calls `HTTPServerFinish()` to free the connection with the server or `httpServerReadData(int immediate=1)` to immediately read data from the connection.

To manage this much information for servers, clients, connections, and requests, Polipo has its own data structures.

3.2.3 Data structures

Figure 3.2 shows the data structures of Polipo. After receiving a client request, Polipo creates a variable of type `HTTPRequest` named `request`, parses HTTP headers and stores them in an variable of type `Object` which contains fields such as `key`, `via`, `date`, and creates a variable `connection` of type `HTTPConnection` which contains information about the Internet connection, such as `fd` for file descriptor, `pipelined` for number of requests sent back to back or `buf` to store data to send or receive. The `object` and `connection` are pointed by `request` pointers for the client. On the other side, `request->request` points to the request prepared to communicate with the web server. The `request` of the server side has similar pointers for `HTTPConnection` and `Object` which contain respectively information about Internet connection with the web server and the HTTP headers parsed from server responses.

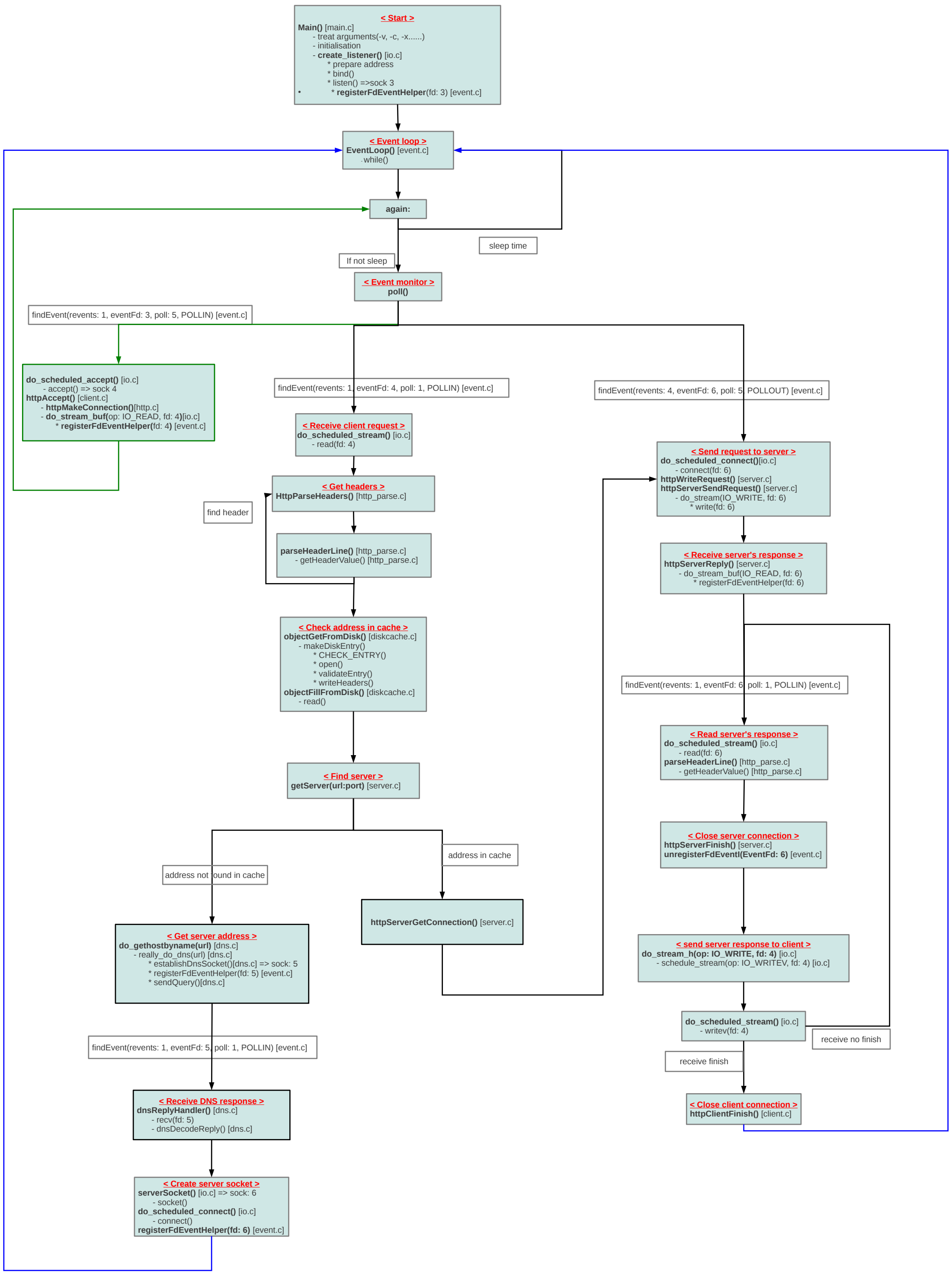
< Start >
Main() [main.c]
- treat arguments(-v, -c, -x......)
- initialisation
- create_listener() [io.c]
* prepare address
* bind()
* listen() =>sock 3
* registerFdEventHelper(fd: 3) [event.c]
< Event loop >
EventLoop() [event.c]
while()
again:
sleep time
If not sleep
< Event monitor >
poll()
findEvent(revents: 1, eventFd: 3, poll: 5, POLLIN) [event.c]
do_scheduled_accept() [io.c]
- accept() => sock 4
httpAccept() [client.c]
- httpMakeConnection()[http.c]
- do_stream_buf(op: IO_READ, fd: 4)[io.c]
* registerFdEventHelper(fd: 4) [event.c]
findEvent(revents: 1, eventFd: 4, poll: 1, POLLIN) [event.c]
< Receive client request >
do_scheduled_stream() [io.c]
- read(fd: 4)
< Get headers >
HttpParseHeaders() [http_parse.c]
find header
parseHeaderLine() [http_parse.c]
- getHeaderValue() [http_parse.c]
< Check address in cache >
objectGetFromDisk() [diskcache.c]
- makeDiskEntry()
* CHECK_ENTRY()
* open()
* validateEntry()
* writeHeaders()
objectFillFromDisk() [diskcache.c]
- read()
< Find server >
getServer(url:port) [server.c]
address not found in cache
address in cache
< Get server address >
do_gethostbyname(url) [dns.c]
- really_do_dns(url) [dns.c]
* establishDnsSocket()[dns.c] => sock: 5
* registerFdEventHelper(fd: 5) [event.c]
* sendQuery()[dns.c]
httpServerGetConnection() [server.c]
findEvent(revents: 1, eventFd: 5, poll: 1, POLLIN) [event.c]
< Receive DNS response >
dnsReplyHandler() [dns.c]
- recv(fd: 5)
- dnsDecodeReply() [dns.c]
< Create server socket >
serverSocket() [io.c] => sock: 6
- socket()
do_scheduled_connect() [io.c]
- connect()
registerFdEventHelper(fd: 6) [event.c]
findEvent(revents: 4, eventFd: 6, poll: 5, POLLOUT) [event.c]
< Send request to server >
do_scheduled_connect()[io.c]
- connect(fd: 6)
httpWriteRequest() [server.c]
httpServerSendRequest() [server.c]
- do_stream(IO_WRITE, fd: 6)
* write(fd: 6)
< Receive server's response >
httpServerReply() [server.c]
- do_stream_buf(IO_READ, fd: 6)
* registerFdEventHelper(fd: 6)
findEvent(revents: 1, eventFd: 6, poll: 1, POLLIN) [event.c]
< Read server's response >
do_scheduled_stream() [io.c]
- read(fd: 6)
parseHeaderLine() [http_parse.c]
- getHeaderValue() [http_parse.c]
< Close server connection >
httpServerFinish() [server.c]
unregisterFdEventI(EventFd: 6) [event.c]
< send server response to client >
do_stream_h(op: IO_WRITE, fd: 4) [io.c]
- schedule_stream(op: IO_WRITEV, fd: 4) [io.c]
do_scheduled_stream() [io.c]
- writev(fd: 4)
receive no finish
receive finish
< Close client connection >
httpClientFinish() [client.c]

Figure 3.1: Flowchart of Polipo. Polipo uses the `poll()` function to supervise events and to react. The reactions can be accepting new client connection, reading from the buffer, or writing to the buffer..

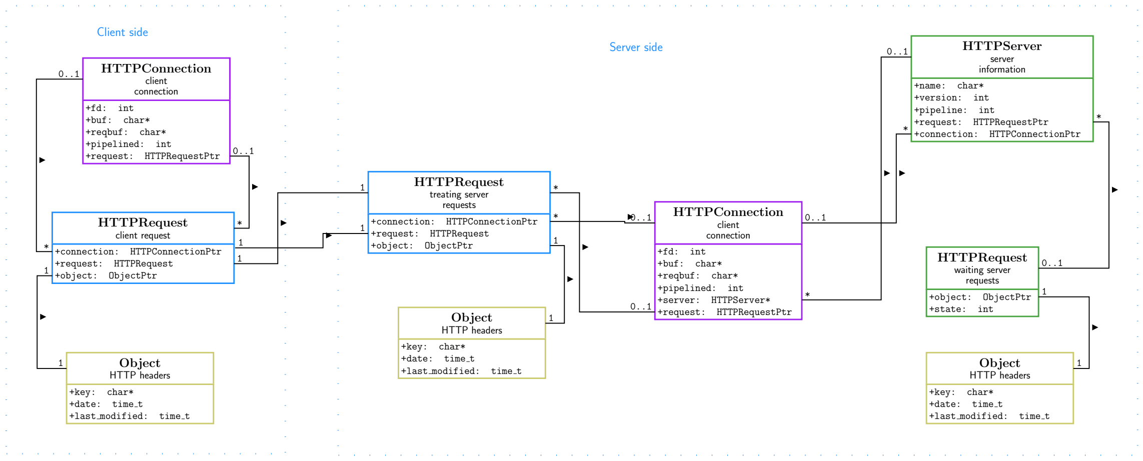

Figure 3.2: UML diagram of the data structures of Polipo. On the client side, a request list is pointed by `connection`. On the server side, there is a list of requests pointed by `connection`, to be treated. Another list of requests are pointed by `server`, these requests are waiting for a free connection. Each `server` has a list of connections.

3.2.4 Pipelining

The support of HTTP pipelining is one of Polipo's specialties. In function `HTTPServerTrigger()`, when there is a list of requests for the web server, it sends two requests back to back. If it receives a response for each, it concludes that the server supports pipelining, and continues to send multiple requests in a row to the web server, otherwise, it sends one request at a time. Requests in an HTTP stream are separated by an empty line, as Figure 3.3 shows. The number of requests sent to the server in one stream is indicated by `connection->pipelined` value.

If the pipelining mode is activated, after receiving data from the server and parsing the HTTP headers, Polipo sets the `connection->request` pointer to the second request to treat, and moves the response for the second request to the head position of buffer, then parses this response. This mechanism is realized in the `HTTPServerFinish()` function.

3.3 Extension of Polipo

After presenting the operation of Polipo, we now modify the code to add the desired functionalities. Firstly, we add a `state` indicator to track the state of requests at each step. Then, we add functions to send HTTP HEAD requests. Finally, we implement functions to process the received HTTP HEAD responses.

3.3.1 State of request

As one request for the web server corresponds to a `request` variable in the data structure of Polipo, it is important to keep track of the state of the request in order to know at which step the request is. We add an `int state` attribute in the `struct HTTPRequest`. When a request is made by `HTTPMakeRequest()`, we initialize the state to `REQ_INIT`. When the request is prepared by the `prepareRequests()`, it is

```
GET /rsrc.php/v2/yY/r/u8iA3kXb8Y1.css HTTP/1.1
Host: static.ak.fbcdn.net
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:7.0.1) Gecko
   /20100101
Firefox/7.0.1
Accept: text/css;q=0.1
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Referer: http://fr-fr.facebook.com/
Connection: keep-alive

GET /rsrc.php/v2/y-/r/507fwwUzcWs.js HTTP/1.1
Host: static.ak.fbcdn.net
User-Agent: Mozilla/5.0 (X11; Linux i686; rv:7.0.1) Gecko
   /20100101
Firefox/7.0.1
Accept: *
Accept-Language: en-us,en;q=0.5
Accept-Encoding: gzip, deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Referer: http://fr-fr.facebook.com/
Connection: keep-alive
```

Figure 3.3: Polipo sends two requests back to back. If the server supports pipelining, it returns a response for each request, otherwise, it only returns a response for the first one..

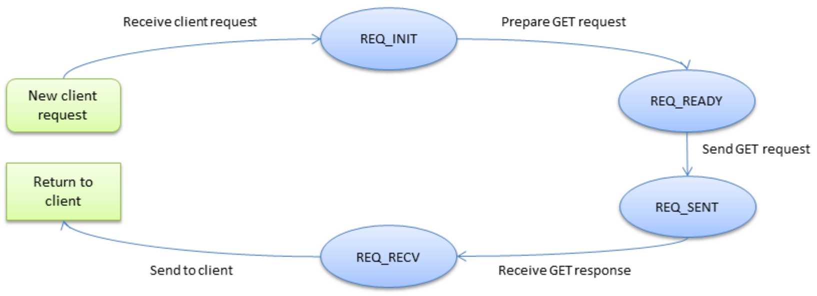

Figure 3.4: Different states of a request variable through a full process.

ready to be sent to the server and we change its state to `REQ_READY`. If the request is sent correctly, the `httpServerSideHandlerCommon()` function is called, it changes the state to `REQ_SENT`. When response to this request is successfuly received, we change the state to `REQ_RECV`. Figure 3.4 presents the different states of a request of our extension to Polipo.

3.3.2 Sending HEAD requests

The `prepareRequests()` function prepares requests in the `connection->reqbuf` buffer. This is the step where we prepare a HEAD request. The strategy is that we change the HTTP method to use in terms of the state of request. If request is in state `REQ_INIT`, we modify `request->method` to `METHOD_HEAD` to send an HTTP HEAD request. If the state of the request is `HEAD_RECV`, this means that the HTTP HEAD response is already received, we then restore the `request->method` back to the original method to send the full request initially received from the client. Once the request prepared, it is sent by the `HTTPServerSendRequest()` function.

3.3.3 Reception of HEAD replies

When Polipo receives data from web server, the `HTTPServerReplyHandler()` function is called. In this function, we check the state of request. If the state was `REQ_SENT`, we change the state of request to `REQ_RECV`, and call the `receiveHandler()` which is the normal behavior of Polipo. If the state was `HEAD_SENT`, we change the state to `HEAD_RECV`, and the `receiveHeaderHandler()` to parse the `Content-Type` value of the HTTP headers by calling the `parse_header()` function. After that, the `HTTPServerTrigger()` function is called to prepare a `GET` request message and to send this message to the server.

Pipelining is a hard point to manage after receiving HEAD responses. Inspired by the `HTTPServerFinish()` function, the idea is to check the `connection->pipelined` value of the server connection structure, and to compare number of bytes received with the `Content-Length` value. If the `connection->pipelined` value is larger than 1, and Polipo received more bytes than the length of header returned by `findEndOfHeaders()`, these additional bytes are certainly a response to a second request. We then move these bytes to the top of the connection buffer, and call the `httpServerReply(int immediate=1)` function to immediately receive the second reply. In the other case, if the `connection->pipelined` value is bigger than 1, but Polipo received the same number of bytes as the length of header, this means Polipo sent more requests but received only one response, so this server does not support the pipelining, we then deactivate the pipelining mode for this server by setting the `server->pipeline` value to a negative number.

3.4 Results and conclusion

In this chapter, we studied the functionalities of Polipo, its structure, and we modified the relevant functions to get HTTP header information before sending the full GET request.

After modifying the code of Polipo, the proxy can send an HTTP HEAD request before sending the original request of the client to the web server to obtain the HTTP header information about the required object. As figure 3.5 describes, the new Polipo requests have `HEAD_SENT` and `HEAD_RECV` states, and are able to get information of content to transfer without receive this content.

This modified Polipo proxy is useful for the development of our multi-protocol proxy on the client side. In the next chapter, we present the implementation of a multi-protocol proxy on the server side, and its integration with this Polipo proxy.

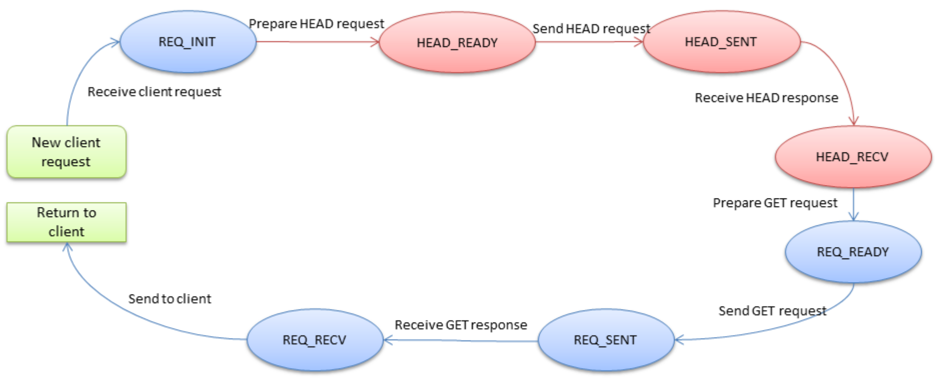

Figure 3.5: Compared with the original states of a request, we added functions and states to treat HEAD requests. These are colored in pink.

Chapter 4

Development of a multi-protocol proxy

After developing a simple proxy and extending an open source one, we now implement our full multi-protocol proxy, MPPX. This proxy has two sides. The client-side proxy is based on the extended Polipo proxy. It should be able to select the transport protocols to use to relay data to the server-side proxy. The server-side proxy, which is running on the web server, should be able to receive requests from several transport protocols, forward the requests to web server over TCP, and return responses from the server to the client as figure 4.1 shows.

In this chapter, we firstly present the implementation of the multi-protocol proxy on the server-side. Then, we explain the implementation on the client-side, as well as the result of this implementation. We finally present the usage of this proxy.

4.1 MPPX on the server side

MPPX on the server side is installed on the web server, it is designed to use several transport protocols, such as TCP or UDP, to communicate with the MPPX of the client-side, and TCP for communication with the web server.

4.1.1 Data structures

In MPPX, there are three types of sockets, the `listening_socket` to receive new client connections, the `client_sockets` to communicate with client applications, and the `server_socket` to communicate with the web server. Each type of socket is managed by a list of its type.

As MPPX uses different protocols with different client applications, a client is identified by the `file_descriptor`, `addr`, and the `protocol`. Also, a `protocol` used by a client is identified by protocol number and protocol type. The protocol numbers are defined in the `proto.h` header of Linux. The type of protocol is the type of connection such as `SOCK_STREAM` or `SOCK_DGRAM`.

The list of listening sockets is of type `MPPXSocket` because the sockets of this list use different transport protocols. In order to distinguish listening sockets and client sockets, we use a `mode` attribute which allows to assign `LISTEN_SOCK`, `CLIENT_SOCK`, or `SERVER_SOCK`.

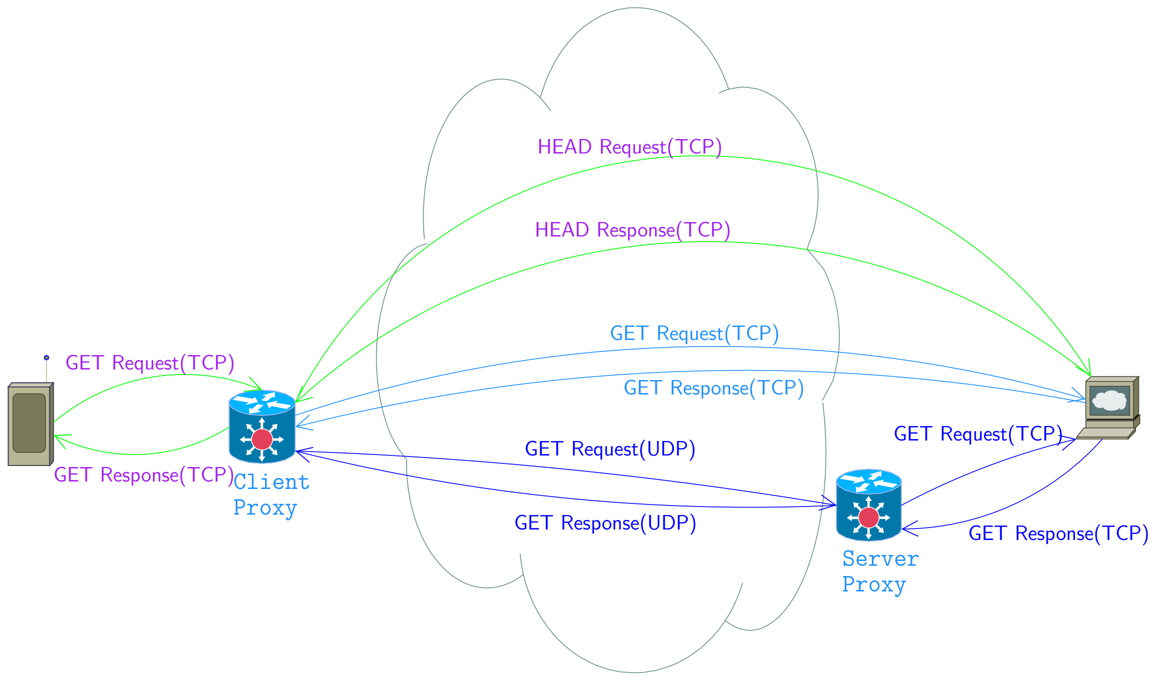

Figure 4.1: General MPPX architecture. The client-side proxy sends `HEAD` requests to the server, receives replies, and selects the appropriate transport protocol to transfer the content..

The data structure to manage the server seems to be simple because the proxy uses only `TCP` as the transport protocol. However, the challenge is to identify which client socket is linked to this server socket, and to advise which server socket is free to transfer data. This is why we use a `client` pointer. When a server response comes, the proxy can use this pointer to identify the corresponding client to forward this response to. After finishing communication with a client, the proxy sets a `NULL` value in the `client` attribute of the server structure.

There is a list of type `pollfd` including all file descriptors from which MPPX receives data. The number of file descriptors of this list is the sum of number of `fd` in `listering_socket`, `client_socket`, and `server_socket`. These structures are described by figure 4.2.

4.1.2 Process of MPPX

For performance reasons, MPPX uses the `poll()` function to monitor different sockets and to control input/output streams because this function can differentiate input/output events.

When the server mode is activated, The proxy creates one listening socket per supported protocol to receive new client connections. It then puts every listening socket in a list of type `pollfds`, named `poll_sockets`, which is supervised by the `poll()` function in an infinite loop.

When data is available, the `revents` of the corresponding socket is set to `POLLIN`. MPPX finds this socket and calls the `mppx_read()` function to read data. There are three types of socket.

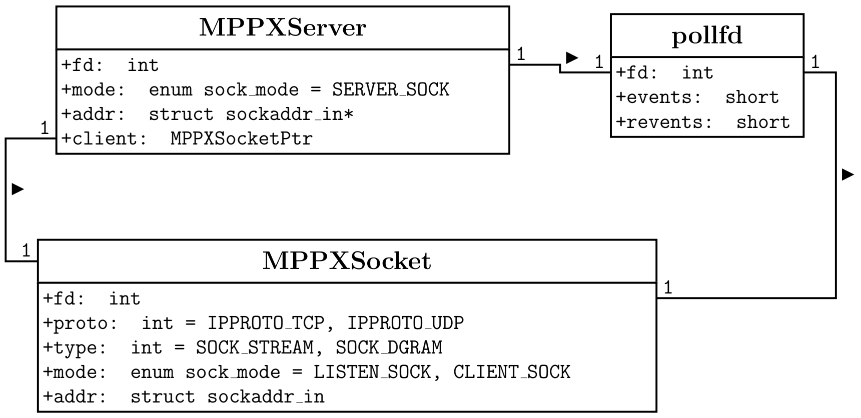
MPPXServer
+fd: int
+mode: enum sock_mode = SERVER_SOCK
+addr: struct sockaddr_in*
+client: MPPXSocketPtr
pollfd
+fd: int
+events: short
+revents: short
MPPXSocket
+fd: int
+proto: int = IPPROTO_TCP, IPPROTO_UDP
+type: int = SOCK_STREAM, SOCK_DGRAM
+mode: enum sock_mode = LISTEN_SOCK, CLIENT_SOCK
+addr: struct sockaddr_in
1
1
1
1
1
1

Figure 4.2: Main data structures of MPPX. MPPX has a list of type `MPPXServer` to manage servers, a list of type `MPPXSocket` to manage clients, and a list of type `pollfd` to control the input/output events.

Listening sockets receive new client connections, MPPX adds this new client to the `client_sockets` list, reads data, and adds the new client socket in the `poll()` function to receive potential data. Then, MPPX finds a free server connection or connects to the web server to send the data.

Client sockets receive messages from identified clients. Without creating a new client, MPPX just reads the messages and forwards them to the web server over a free connection or a new connection.

Server sockets receive client requests and return responses to the proxy. After receiving server responses, MPPX finds the client from which it received the request and sends the response to this client.

Additionally, when the `mppx_read()` function receives 0 bytes from the `read()` or `recvfrom()` functions, it is assumed the transmission is finished. MPPX then removes the socket if it is from the client application, or frees the server connection if it is from the web server. As we don't control the output stream, the `POLLOUT` event is deactivated and the `mppx_write()` function is not used at the moment. This process is described by figure 4.3.

Having described and implemented the server-side multi-protocol proxy, the client-side needs to be adapted. The extended Polipo proxy can only send `HEAD` requests to web servers and detect the type of content to transfer, it does not support transport protocols other than TCP yet, we now implement this on the client side.

4.2 MPPX on the client side

In this section, we reuse the extended source code of Polipo from chapter 3 in order to add support for multiple transport protocols.

After receiving `HTTP` headers and parsing the type of content in the `receiveHeaderHandler()` function, we first add the `selectProto()` function which

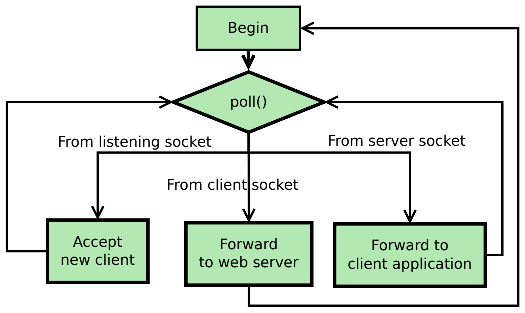

Figure 4.3: MPPX's `poll()` loop. MPPX looks for the type of incoming socket to decide the next action..

checks the `Content-Type` in its parameter and returns the number of an appropriate protocol to transport this content. By default, the function returns TCP for unknown types of content.

Once the transport protocol is chosen, the program runs a `switch()` statement which defines the followup behavior. For TCP, the program runs its normal functions to communicate directly with the web server. For other protocols, the program creates new connections to the server-side proxy, prepares and sends request then registers the new file descriptor in the `pollfds` list. When reply on this file descriptor is received, the `read_udp_stream()` is called. As Polipo already manages HTTP headers, we use its functions for UDP as well: `read_udp_stream()` reads the reply message into a local buffer, copies the message into the buffer of the TCP connection, then calls `HTTPServerReplyHandler()` to let Polipo manage server replies and to return to the client application.

4.3 Result

After adding a server-side and client-side proxy, the MPPX can work correctly over TCP. However, as the time of this internship was limited, the MPPX UDP part on the client side is not quite stable yet and requires some debugging. As no web server accept UDP connections, the client MPPX can only send UDP requests to `localhost` on which the server MPPX is running. This issue can be fixed by putting an IP address string to `serverName` which was `localhost` in the function `receiveHeaderHandler()`.

4.4 Usage of MPPX

MPPX is based on the source code of Polipo, it can be executed by Polipo command options.

polipo launches the proxy on client mode.

polipo -s launches the proxy on server mode.

polipo -d allows to check logs of transmission in detail.

polipo -v shows configuration variables of Polipo.

polipo -h shows the list of options of polipo command.

Almost options can be combined, for example `polipo -s -d` to show logs on server mode.

Chapter 5

Conclusion

Content on the web is increasingly multimedia. However, the use of TCP as the transport protocol is not always the most appropriate for some types of data. The implementation of a multi-protocol proxy is one possible solution to address this.

A multi-protocol proxy is an application able to transfer data using different transport protocols. It is divided in two sides. The client side proxy receives HTTP `GET` requests over TCP, modifies the HTTP method to `HEAD`, and sends the `HEAD` request to the web server in order to detect the type of content. Then, the proxy selects an appropriate transport protocol for this type to transfer this content. The server-side proxy receives HTTP requests from the proxy of the client side over multiple protocols, transfers the requests to the web server over TCP, and returns responses of the web server to the proxy of client side.

The implementation of the multi-protocol proxy was done in 3 phases. Firstly, we did a simple proxy which forwards requests and responses between the client application and the web server. This simple proxy allowed us to understand the HTTP and the socket functions. As there are several open source proxies on the market which are more stable and have more functionalities, we decided to extend the source code of one of them, Polipo, as the multi-protocol proxy on the client-side. This extension realized the function of sending/receiving `HEAD` requests/responses to the web server. Once this extension finished, we implemented the multi-protocol proxy on the server side and modified the client side to support multiple transport protocols.

This 24 weeks internship took place at National ICT Australia Ltd. During this period, we practiced a lot of network knowledge such as the HTTP protocol, the `socket` interface, and the OSI model, improved programming skills in the C language, and familiarized ourselves with a lot of open source tools such as `gdb`, `git`, Dia or LaTeX.

Future work

The multi-protocol proxy is still in development, future work includes the following:

Support of multiple servers This functionality can be added by using the IP address of `serverName` rather than `localhost` for the `gethostbyname()` function in the `server.c` file.

Support of others transport protocols The actual multi-protocol proxy supports only TCP and UDP. The support of other transport protocols such as DCCP or SCTP would improve the flexibility of the proxy.

Support of OML OML [7] is a C library which would allow to measure the time of transmission of the proxy, and visualize the improvement of using the multi-protocol proxy.

Conception of MPPX protocol Making a new MPPX protocol will help us to avoid disadvantages of current protocols and gives us much flexibility. We could add sequence numbers for MPPX packets to control the sequence of UDP datagrams, distribute an identity number to each client to give them accessibility to the MPPX server.

Implementation in Android phones The tests were done on a Linux machine, this proxy should also work for Android phones.

Bibliography

[1] Ramón Agüero et al. “OConS: Towards Open Connectivity Services in the Future Internet”. In: *MONAMI 2011, Third International ICST Conference on Mobile Networks & Management*. Ed. by Susana Sargento and Ramón Agüero. Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering. Aveiro, Portugal: Springer-Verlag Berlin, Sept. 2011.

[2] Mark Allman, Vern Paxson, and Ethan Blanton. *TCP Congestion Control*. RFC 5681. Fremont, CA, USA: RFC Editor, Sept. 2009. URL: http://www.rfc-editor.org/rfc/rfc5681.txt.

[3] Juliusz Chroboczek. *The Polipo Manual*. Université Paris Diderot. Paris, France, 2006. URL: http://www.pps.univ-paris-diderot.fr/~jch/software/polipo/polipo.pdf.

[4] Luis Diez et al. “Design and Implementation of the Open Connectivity Services Framework”. In: *MONAMI 2012, 4th International Conference on Mobile Networks and Management, OConS workshop*. Ed. by Lucio S. Ferreira and Lucian Suciu. Lecture notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. TUHH, EAI. Hamburg, Germany: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Sept. 2012.

[5] Roy T. Fielding et al. *Hypertext Transfer Protocol — HTTP/1.1*. RFC 2616. Fremont, CA, USA: RFC Editor, June 1999. URL: http://www.rfc-editor.org/rfc/rfc2616.txt.

[6] Eddie Kohler, Mark Handley, and Sally Floyd. *Datagram Congestion Control Protocol (DCCP)*. RFC 4340. Fremont, CA, USA: RFC Editor, Mar. 2006. URL: http://www.rfc-editor.org/rfc/rfc4340.txt.

[7] Olivier Mehani et al. *An Instrumentation Framework for the Critical Task of Measurement Collection in the Future Internet*. Under review. 2012.

[8] Jonathan B. Postel. *User Datagram Protocol*. RFC 768. Fremont, CA, USA: RFC Editor, Aug. 1980. URL: http://www.rfc-editor.org/rfc/rfc768.txt.

[9] SAIL project. *Architectural Concepts of Connectivity Services*. Deliverable FP7-ICT-2009-5-257448-SAIL/D-4.1(D-C.1). EC Information Society Technologies Programme, July 2011. URL: http://www.sail-project.eu/wp-content/uploads/2011/08/SAIL_D.C.1_v1.0_Final_PUBLIC.pdf.

[10] SAIL project. *Architecture and Mechanisms for Connectivity Services*. Deliverable FP7-ICT-2009-5-257448-SAIL/D4.2(D-C.2). EC Information Society Technologies Programme, July 2012.

[11] SAIL project. *Description of Project Wide Scenarios and Use Cases*. Deliverable FP7-ICT-2009-5-257448-SAIL/D2.1(D-A.1). EC Information Society Technologies Programme, Apr. 2011. URL: http://www.sail-project.eu/wp-content/uploads/2011/09/SAIL_DA1_v1_2_final.pdf.

[12] Randall R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. Fremont, CA, USA: RFC Editor, Sept. 2007. URL: http://www.rfc-editor.org/rfc/rfc4960.txt.

Acronyms

DCCP	Datagram Congestion Control Protocol
DNS	Domain Name Server
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
MPPX	multi-protocol proxy
NAT	Network Address Translation
NICTA	National ICT Australia Ltd
OSI	Open Systems Interconnection
SAIL	Scalable and Adaptive Internet Solutions
SCTP	Stream Control Transmission Protocol
SNMP	Simple Network Management Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
URL	Uniform Resource Locator
WWW	World Wide Web

List of Figures