POLITECNICO DI TORINO

III Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Monografia di Laurea

HTTPVPN

Using HTTP for asynchronous communication

Marco Elio Gustavo Gario

Settembre 2009

Contents

Introduction

The wide adoption of the NAT technology has led to the invention of numerous ways to connect hosts that do not posses a public IP. Those software are mainly of two types: those that exploit the inner working of the NAT (eg. STUN, TCP/UDP hole punching etc.) and those that relay on a public host (called relay) through which flows the connection between the two original hosts.

Both types of software have limitations. The former highly depends on the implementation of the NAT. Therefore those software aren't a universal working solution. The latter generally requires administrative privileges on the relay to install the software and most of the time they work on non standard TCP/UDP ports that are generally blocked by corporate firewalls.

Aiming at trying to overcome those limitations I developed the HTTPVPN system. This system allows to create a point-to-point connection using a simple webpage as a relay. Thanks to the server side scripting, the UNIX domain sockets and HTTP's chunked encoding, it is possible to establish an asynchronous HTTP connection between two hosts. This way both hosts can receive the data as soon as it is available, without any need to perform time-consuming pollings. This system conforms to the HTTP 1.1 protocol, so it is possible to use it also through corporate proxy servers.

The first part of this report will present the technology and the methods that make the asynchronous communication via HTTP possible. In the second part I will present a working implementation of the VPN developed for Linux in Python whose source code is available in the appendices.

A problem that this report will deliberately not address concerns the security of the "access point": in the implementation that I present there is no control over who is connecting to the relay. This control is hard to achieve without using HTTPS or PHP cryptographic libraries, but both solutions are not commonly available on most of the free webhosting services yet. I then preferred to leave this problem open for further investigation.

Chapter 1

Asynchronous HTTP tunneling

I will now introduce the technologies involved in making the asynchronous tunnel possible. This explanation is not meant to provide an in-depth analysis of each technology, but it will help the reader to grasp how the system works and will provide him with enough references for further studies.

1.1 Technologies involved

1.1.1 HTTP 1.1

HTTP is one of the most present protocol on the internet: its wide deployment is justified by its simplicity and flexibility. One of its key characteristic is that it is session-less. This means that every request must contain all the information needed by the server in order to be processed. In other common protocols, like FTP, POP3 etc, different requests and responses are needed to build, use and terminate a session but in HTTP each request is independent from the previous and following ones. Obviously it is hard to imagine an application where no session is needed, so sessions are maintained by web applications through different means like cookies or url-encoded session identifiers. This leaves the responsibility of the session handling to the web application and not to the web server. Keep in mind this feature of HTTP because it will be important when talking about persistent connections and about the security of the tunnel.

HTTP 1.1 [1] improves HTTP 1.0 by introducing, among others, some important features:persistent connections, chunked transfer coding and pipelining,. While pipelining is an optional, chunked transfer coding and persistent connections are vital for this kind of asynchronous communication. This means that the system that I'm going to present will only work with HTTP 1.1 compliant web-servers and proxies.

Persistent Connection HTTP 1.0 requires a new TCP connection to be established for each request. This can lead to an enormous amount of overhead if a client needs to perform many small consecutive requests. HTTP 1.1 states that the client should assume that the server will not close the connection after sending the response unless a "Connection: close" header is present in the response itself. It is important to note that no relation can be assumed between the different queries performed on the same TCP connection: HTTP 1.1 still remains a session-less protocol.

Chunked transfer coding Every HTTP 1.0 response message, containing some data, has a "Content-Length:" header in which the size of the response is specified. This means that the web server needs to know beforehand how big the response will be. If the response is dynamically generated, then the web server will have to wait for the whole response to be ready before beginning the transmission. Imagine that you want to receive a huge table from a database and that the generation of the table takes 30 seconds, this means that you will have to wait at least 30 seconds to be able to read the first line of the table!!! Chunked encoding was introduced to address this kind of problem. The server can tell the client that it will send little chunks of data as soon as they are ready. Every chunk is preceded by the size of the chunk itself, allowing the client to rebuild the whole response concatenating every chunk. As on most web servers chunked coding is not the default behaviour you might need to force it at the application level.

Pipelining Pipelining was introduced to improve performances over high latency connections. A web page is generally composed by a "master" document containing references to many other resources like images, flash objects, javascript files and css stylesheets. Those resources are generally independent from each other. This means that as soon as we find a reference to a resource in the master document, we can request it to the webserver, and we don't need to wait for the resource to arrive in order to be able to perform another request. Most browsers do open many parallel connections in order to speed up the transfer, but this is not a very good solution when working on networks with a very long round-trip time (RTT) because each new TCP connection requires at least 1 RTT before being able to send the actual request. With HTTP 1.1 it is possible to send many consecutive requests on the same connection without waiting for the response. In Figure 1.1 there is an example in which the latency time is 1t and the elaboration time on the server is 2t. This is an optimistic assumption because the RTT is usually bigger than the elaboration time. Nevertheless, by sending two pipelined requests (on the right), it is possible to conclude the transfer 1 RTT(=2t) earlier than by sending the second one after the acknowledgement of the first one has been received (on the left). This performance improvement comes with a catch: it is not possible to know if all the requests have been processed unless all the responses are analysed. When talking about persistent connections, I mentioned that when the server wants to close the connection, it must send a “Connection: close” header. Unfortunately, if we send 3 pipelined requests and the server decides to close the connection after receiving the second request, it will notify us in the second response and will therefore close the connection. This means that the third request will never be processed and that it will have to be sent again. As we will see in section 3.1.4 (“Internet results”) this makes the protocol unreliable.

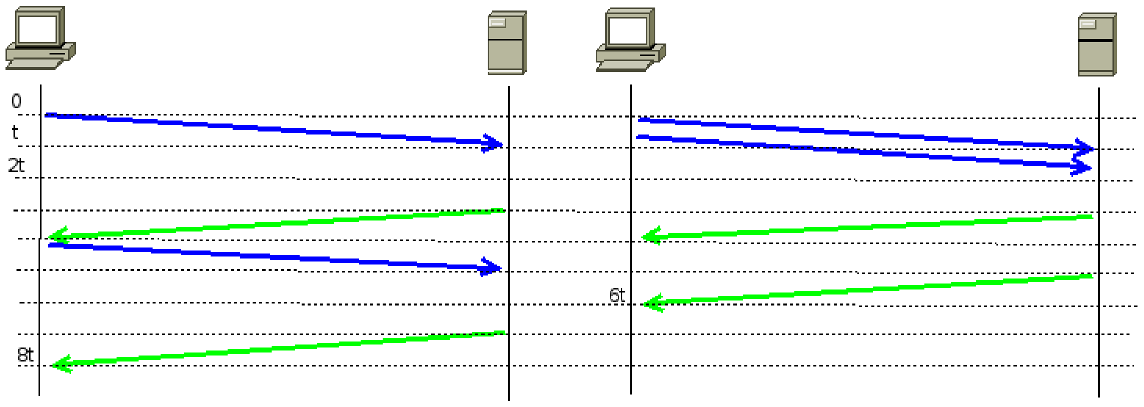


Figure 1.1. Pipeline lattice diagram

1.1.2 CGI / Server side application

There are basically 3 methods to dynamically generate an HTTP response:

Write one’s own web server providing it with all the logic needed to handle our request. Unfortunately, writing a web server is nor simple neither practical. Furthermore it would be extremely hard to offer hosting services if each website required a different web server. The advantage of this solution is that it is possible to write a really small and fast customised web server that only does what we want. This is the solution adopted in most embedded devices that can be configured via a web interface.

Use an external program that is executed by the web server and whose output is sent by the web server as the response. This method is also known as CGI [2] (Common Gateway Interface) and has been widely used because it is simple and platform independent. Indeed it is possible to write a CGI script in any language that the system is able to run and to use it on any web server that does support CGI. The main problem with this method is that it is not very fast. A new process is created every time that a new request arrives and it is not possible to share resources among multiple instances. This means that if your script needs to connect to a database, you cannot perform the connection once and relay on the script always using the same connection. To overcome this limitation FastCGI [3] was invented but, unfortunately, it is not widely adopted, making your script less portable.

Use a well known and widely available web server and extend it with modules. This is the most common case because it is a trade-off between the first two options. The idea is that we can tell to the web server which request it should process using the external module, so that we can focus our module on handling the requests that do matter. The module itself is loaded inside the web server, so it is able to provide better performances but it is tied to the software in use.

I chose to write the software I needed in PHP because it is a widely adopted scripting language and because many free hosting services provide a web server module to handle PHP: `mod_php`. In the few cases where `mod_php` is not available, it is generally possible to run the PHP script as a CGI.

Note that most web servers (especially when working with CGI's) generally buffer the response and wait for the script to finish before sending it. This means that if you want to exploit chunked coding you need to manually flush the buffer when the data is ready. How to actually do this depends on the language in use, but in PHP it is sufficient to flush the output buffer via the `ob_flush()` and `flush()` functions.

1.1.3 Unix domain sockets (IPC)

In order to build a communication tunnel, we need to exchange information between two running scripts. One possible solution could be to use a file or a database table in which both processes write and read data in turns. However, this solution can create race conditions and concurrency problems and doesn't guarantee good performances either. We will use interprocess communication instead. Interprocess communication (IPC) is a vast family of operating system dependent methods that allow the exchange of data between processes. The Unix domain sockets are one type of IPC [5]. The Unix domain sockets are special files that behave like a pipe. They are very simple to use because they can be accessed using standard file I/O functions: this means that no special libraries are needed on the server. Another great advantage of the Unix named sockets is that they provide very good performances because they reside on RAM. Also, when a process tries to read on an empty pipe, it is suspended by the OS, and awakened as soon as some data is available granting the best performances achievable.

Using the Unix named socket limits the use of my solution to *nix web servers but this limitation is justified by their simplicity of use. The same idea can nevertheless be reimplemented using an equivalent method on other platforms.

1.2 Implementation

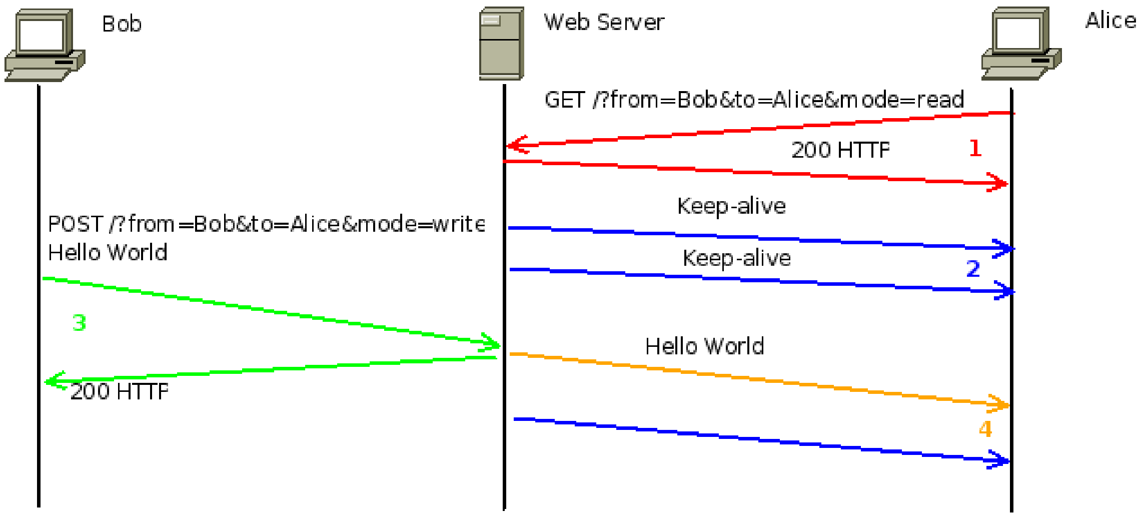


Figure 1.2. Lattice Diagram

In the previous paragraph I introduced all the elements needed to build an asynchronous tunnel using HTTP.

The core of the system will be a script that is able to read and write from and to a pipe. I called this script `mirror.php` and it is available in the Appendix A.

In order to make the explanation clearer, I will describe a scenario in which two clients (Alice and Bob) want to communicate through the web server in an asynchronous way. For sake of simplicity we will describe an unidirectional communication in which Bob will send data to Alice.

First of all, Alice needs to establish a connection with the web server (1) requesting `mirror.php`. `mirror.php` will force the web server to keep the connection open waiting for the data from Bob (2). When Bob sends the data (3) to the the web server, it will be processed by `mirror.php` and sent to Alice (4). This is the idea in a nutshell as presented in Figure 1.2: let's analyse each step in depth.

When Alice opens the connection to the web server, she will specify that she wants to read all the data that Bob is sending to her. The web server will then execute the

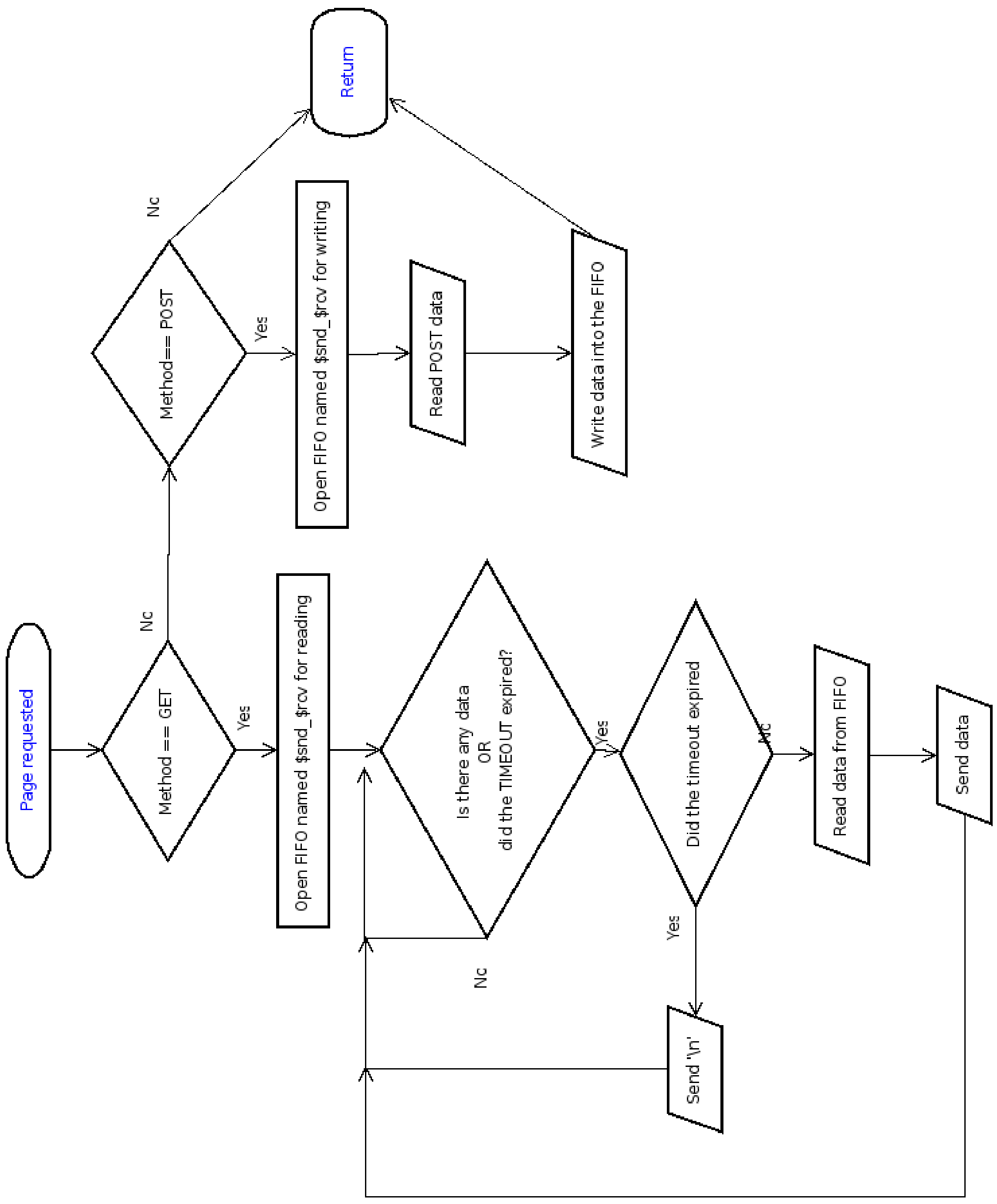


Figure 1.3. mirror.php's flowchart

script which will open a particular pipe (identified by the combination of sender and receiver name, eg. BobAlice) and wait for data on the pipe to be available. This means that the script does not terminate but it is just suspended by the O.S. and will be reactivated when there is data available.

When Bob has data to send, he will send the data specifying that he wants to talk to Alice. Upon receiving this request, the web server will execute another instance of the script which will write Bob's data inside the pipe. This will automatically awake Alice's `mirror.php` process and allow it to send the data to Alice.

In theory those are the simple steps needed to perform an asynchronous HTTP tunnel. Unfortunately there can be a long time frame before Bob sends any data. This can lead to the TCP connection timing out, but this problem can be avoid by making the script send dummy keep-alive messages.

On many web servers there is a time limit for the execution of the scripts. This means that the web server will kill the process if it runs for more than N minutes. Generally, this is not a problem because the script will be sleeping most of the time, and will only be active for the milliseconds needed to copy the data from the pipe. Nevertheless, it is important for the client to keep track of the last message received in order to know if the script was killed. In that case it will be sufficient to connect again to the web server. The data that was sent in the meantime will not be lost because it will be stored in the pipe, waiting to be read. Data loss occurs only when the pipe is filled up. In this case, the web server can be seriously compromised, as the processes that are trying to write to the pipe are suspended by the O.S., in order to let the pipe to empty therefore locking many resources.

Figure 1.3 shows a flowchart summarizing the behaviour of the script.

Obviously, if we want to achieve a bidirectional communication we will just need to quadruplicate the system above illustrated by making 4 tunnels:

1. Alice reading data from Bob (A,B,r)
2. Alice writing data to Bob (B,A,w)
3. Bob reading data from Alice (B,A,r)
4. Bob writing data to Alice (B,A,w)

1.3 Limitations

The method described above can be used to transmit any type of data but it has some limitations.

The client must be able to manage chunked encoding. This (for example) is not the case of python's httplib. It understands chunked encoding but waits for the whole answer to arrive before returning the data, making the chunked encoding useless.

No reliable authentication scheme can be achieved without HTTPS. HTTP doesn't provide a secure authentication system. The best HTTP 1.1 can offer is "Digest" authentication. This form of authentication is stronger than "Basic" authentication, but is susceptible to men in the middle attacks. The absence of a reliable authentication scheme means that anyone can connect to the web server pretending to be Alice or Bob and read or write to the pipe. Actually it is worse than that: as the pipe doesn't duplicate the data, only one Bob (be it the legit or fake one) can receive the data. This can lead to a Denial of Service, where the legit user doesn't get any data at all. If the web server supports Digest authentication it is possible to use it to mitigate the problem, but this solution is far from being perfect.

Chapter 2

HTTP VPN: IP Tunneling through HTTP

The asynchronous HTTP tunneling presented in the previous chapter can be used to send any type of data. In particular I'm going to explain how to use it to tunnel IP packets. To this purpose I will present the details involved in writing a point to point Virtual Private Network (VPN) using HTTP. The explanation of the technologies involved will go in depth enough to allow the reader to understand the source code available in the Appendix B. This source code is a commented example of a working implementation of HTTP VPN.

2.1 Technologies Involved

2.1.1 TUN/TAP

TUN/TAP devices are special virtual network devices available for different kernels. They are useful because they allow to write software that interact with the system at the IP level, pretending to be a network interface. This is the easiest way to write a tunneling driver, without having to worry about kernel programming. Their simplicity is the reason why they are at the base of the famous OpenVPN software.

The API is a simple pipe where one can read and write data. When the O.S. wants to send data through the TUN/TAP device, it will write the IP packets in the pipe. When the process writes data into the pipe, the O.S. will see it coming like a packet on a real network interface. If all you want to do is data forwarding between two different TUN devices, then there is no need to interpret the data you read from the pipe but you just have to copy it from one pipe to the other.

2.1.2 Cryptography

The need to send the data across the internet (and maybe across different proxies) requires the use of some sort of encryption. I decided to use a symmetric encryption to make the system faster. Because the system needs to encrypt every packet, an asymmetric encryption would be too slow. It would be possible to use some key exchange algorithm (eg. Diffie-Hellman) to make the symmetric encryption stronger, but it is far behind the objective of proving the feasibility of the HTTP VPN.

Each packet will be encrypted with Blowfish being it a fast, reliable and free algorithm.

2.1.3 Proxy

The asymmetric HTTP tunnel is fully compliant with HTTP 1.1. This means that there is no reason why the system should not work when adding a proxy between our client and the web server. The proxy needs, of course, to be HTTP 1.1 capable. Unfortunately, this is not always sufficient. When experimenting the system, I found some proxies that claimed to be HTTP 1.1 capable, but did not manage correctly the chunked encoding, returning chunks of data without specifying their length. This is not a problem of the VPN or the protocol itself: but it is a problem of the proxy implementation.

2.1.4 Base 64 encoding

HTTP is a text based protocol. This means that if we want to transfer a binary file, we must first encode it into a text only representation. This is commonly done using the Base64 encoding. In this encoding the data is split into groups of 6 bits and then converted to one symbol in the set 'a' to 'z' (both lower and upper case), digits, '+' and '/'. This encoding implies an overhead of 33% on the actual size of the data being encoded.

2.2 Implementation

To build the VPN we will be using the tunnel presented in Chapter 1. The clients will be able to read and write from and to the TUN/TAP device. As previously stated, we will need four connections to make the communication possible: we will need one connection to send and one to receive on each client. The sending and the receiving part of each client are independent from each other. This means that we can implement them independently, and I will present them as follows to keep the explanation simple.

2.2.1 Sender

The sender will wait for data from the TUN/TAP device, it will then elaborate it and send it to the web server (optionally passing through a proxy).

The elaboration of the data can be divided into encryption and base64 encoding.

Blowfish requires the data to have a length that is a multiple of 8 bytes. This means that it might be necessary to pad the data in order to reach a valid size. In order to allow the receiver to discard the padding, it is necessary to attach to the packet the length of the payload (the actual data).

To simplify the debugging and detection of errors, a checksum is appended to the data. The payload, the padding, the payload size and the checksum constitute the packet that we will transmit via HTTP.

The resulting packet has the following structure:

Checksum (160bit)	Payload size (32bit)	Payload	Padding

Before being able to send the packet, the sender has to encode it in base64. This process is completely reversible, and doesn't impose any restriction on the packet size. Once encoded, the new packet will be 4/3 the size of the original packet. This is one of the most critical overhead that the system presents: to send 1500 bytes of actual data you need to send 2000 bytes of information.

The sender is now ready to send the new packet of data. It will check if an HTTP connection to the server already exist otherwise it will create a new one. After the connection is established (whether we are using a proxy or not), it can send the data using HTTP's POST method. After sending the data, the sender must wait for the web server to acknowledge the transmission. We already discussed the problems related to waiting for an acknowledge when talking about pipelines. Waiting for the acknowledgement before proceeding to the next transmission is the only way to have a reliable communication.

A flow chart summarizing the sender work is presented in Figure 2.1.

2.2.2 Receiver

The receiver logic is slightly more complicated because it not only has to read data from HTTP, elaborate it and write it on the TUN/TAP device, but it also has to ensure that the receiver itself is still connected to the web server.

In the section 1.2 ("Implementation") I introduced the need for a keep alive to be sent periodically to the client. We will exploit this keep alive message to reset an internal timeout. In particular we will reset the timeout each time we receive data from the web server. Obviously, if the timeout expires we will have lost the connection to the web server and we will have to initiate a new one.

Suppose that we already established the connection and that we are waiting for the data. When data is available, we must be able to tell the difference between real data and keep alive messages. This is easy to do because I decided to use a line feed ('\n') as a keep-alive message, so that the chunk will be only 1 byte long. Because of the different elaboration we did on the sender before sending the packet, we are sure that no meaningful data can be transmitted in a chunk of 1 byte. So we are able to recognize the keep alive messages and now we can focus on how to restore the useful data.

In theory, getting the real data should only consist in reading one chunk of data. Unfortunately, experiments have shown that this is not the case: some web servers only transmit chunks of up to 8192 bytes. If you want to send more data you will have to split it into different chunks. This constitutes a problem because there is no easy way to tell if you received a valid packet of 8192 bytes or a part of a bigger packet. This problem can be addressed in two ways. First, it is important to note that if you are using an MTU for the tunnel that is less than 6000 bytes (see 3.1 - "Experimental results"), you cannot have a packet bigger than 8192 bytes. So when using "small" MTUs there's no need to worry. If we want to use bigger MTUs we can simply instruct the receiver to append a linefeed ('\n') at the end of every packet sent. The linefeed is not a valid base64 character. When found we will easily understand that it indicates the end of a packet so, it will only be a matter of concatenating the previous chunks in order to obtain the full packet.

Getting the original data, after rebuilding the packet, is only a matter of reversing the operations done by the sender. This means that the receiver has to base64 decode the packet and decrypt it using the same key as the sender. After the data is in clear we can compute the checksum and compare it with the one stored inside the packet. This step is important because we need to know if the data we decrypted is the same that was encrypted.If the data passed the checksum check, the padding will be removed, and the data will be written into the TUN/TAP device.

I would like to spend some words about the utility of the checksum, as at first sight it might not be clear enough. The checksum was introduced to facilitate the debugging of the protocol. Knowing at a glance that the data sent was the same as the one received was a quick way to detect trivial errors. Also, it is very useful to know if the packet was modified during the transmission. Because we are encapsulating everything on top of a TCP connection, we can be quite sure that no accidental modification happened (HTTP doesn't impose the use of TCP but TCP/HTTP is a de facto standard). Suppose that our packet is traveling on the internet and someone tries to modify it. Without knowing the encryption key, it will be hard to modify the packet in a meaningful way. Unfortunately, when using blowfish a change in the centre of the message will provoke some noise near the centre of the cleartext. This happenes because blowfish is a block cipher, and each block is independent from the others. If we don't have a way to compute the checksum of the entire message, we might write garbage to our TUN/TAP device. This, depending on the data traveling, can be quite dangerous. This is the reason that led me to keep the checksum. The only disadvantage is that it slows down the system. For each packet the whole checksum has to be computed: especially on small consecutive packets it can cause a bit of latency.

We started the explanation supposing that the HTTP connection was already established. When we perform the connection we need to take into account at least two things. First, we must check if the web server (or the proxy) supports HTTP 1.1 otherwise we cannot use the chunked encoding and the system will not work. Second, we must ensure that our system will work on ad-sponsored hosting sites. Those sites generally place advertising on top or on the bottom of the response. We will never reach the end of the page, but we must not manipulate the top advertising as data. To do so we need to modify our PHP script in order to make it send a message saying that it is ready to transmit. This means that our connection procedure will wait for an acknowledgement discarding all the data it received before.

A short flowchart summarizing the Receiver's behaviour is presented in Figure 2.2

2.3 Limitations

The system just described works in a very reliable way but it presents two limitations that are related to the limitations of the HTTP asynchronous tunnel, such as high latency and insufficient authentication of the entry point.

2.3.1 Startup time

Each time that a packet is sent, the web server needs to execute the php script in order to handle the request. This adds latency to the network link and is also the major bottleneck of the system (see 3.1.3 - "Local results"). Some alternatives to improve the startup time (like FastCGI or an ad-hoc webserver) were already introduced in the section 1.3 ("Limitations"). In general those technologies are not widely available so it doesn't make sense to rely on them, although it would be interesting to see how fast the system can be when using them.

2.3.2 Pipelining

Although we cannot reduce the startup time, we can still reduce the latency by pipelining the requests. By using the the pipeline we gain time but we are not sure whether the server received our request. This problem was introduced when talking about the sender: if we want to establish a reliable channel, we need to wait for the acknowledgement. Pipeline makes sense when we are tunneling TCP connections because TCP is a reliable protocol. This means that if a packet is lost, we don't need to resend it, because the connection we are encapsulating will do it for us.

2.3.3 HTTPS

As discussed earlier the easiest way to make the tunnel secure is by using HTTPS. So, whenever possible, it should be preferred to plain HTTP. I would like to point out that using HTTPS doesn't make the use of Blowfish encryption pointless. HTTPS is used to protect the communication between the client and the server but the packet is going to be elaborated in clear on the server. The blowfish encryption grants that whoever gets access to the web server will not be able to see our traffic. The software I developed doesn't support HTTPS but fortunately there is a very useful application available for most O.S.: stunnel [11] . This application allows to wrap any TCP connection inside SSL, making it very easy to test the HTTPVPN on HTTPS servers.

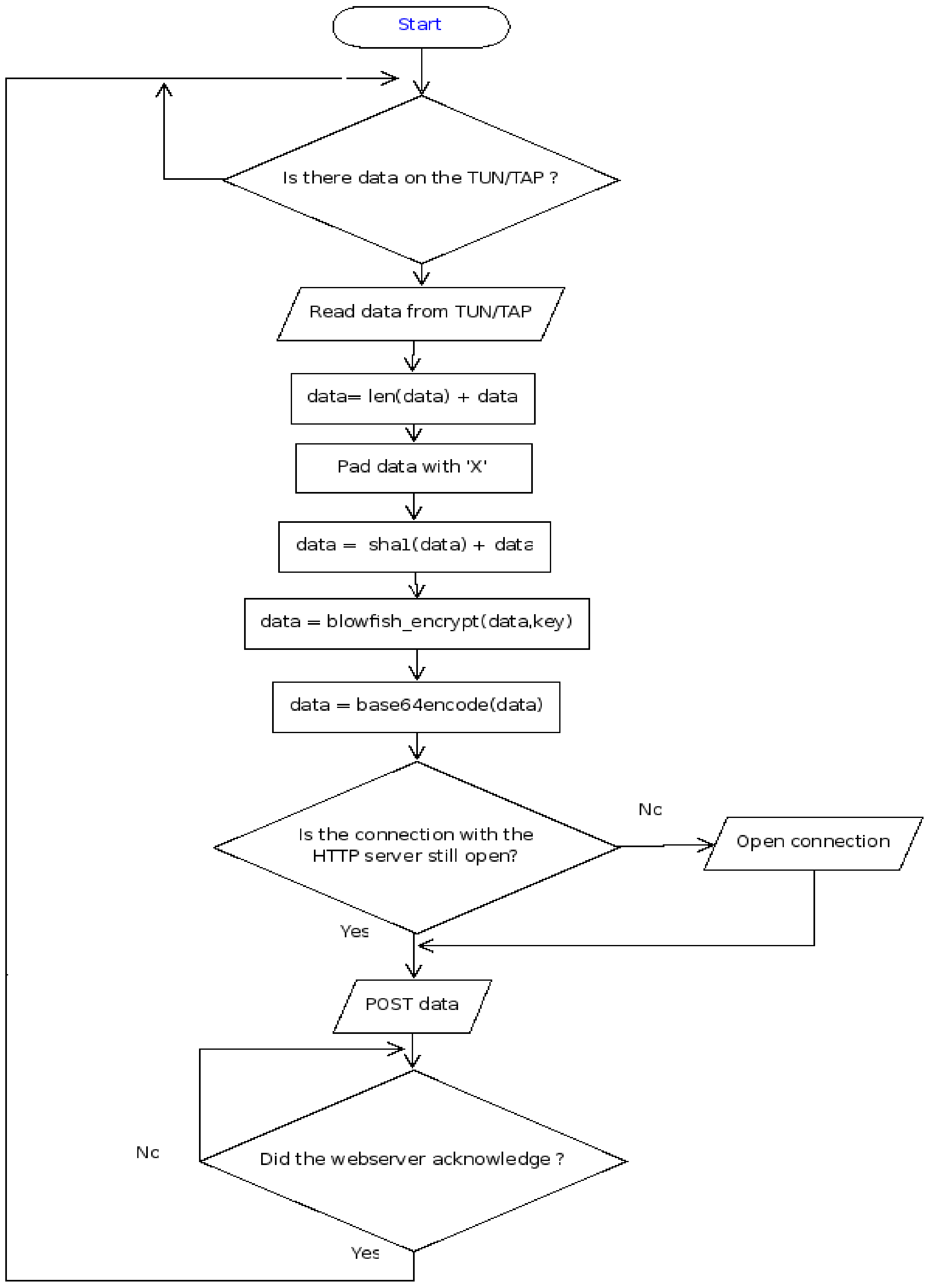

Start
Is there data on the TUN/TAP ?
Read data from TUN/TAP
data= len(data) + data
Pad data with 'X'
data = sha1(data) + data
data = blowfish_encrypt(data,key)
data = base64encode(data)
Is the connection with the HTTP server still open?
No
Open connection
Yes
POST data
No
Did the webserver acknowledge ?
Yes

Figure 2.1. Sender's flowchart

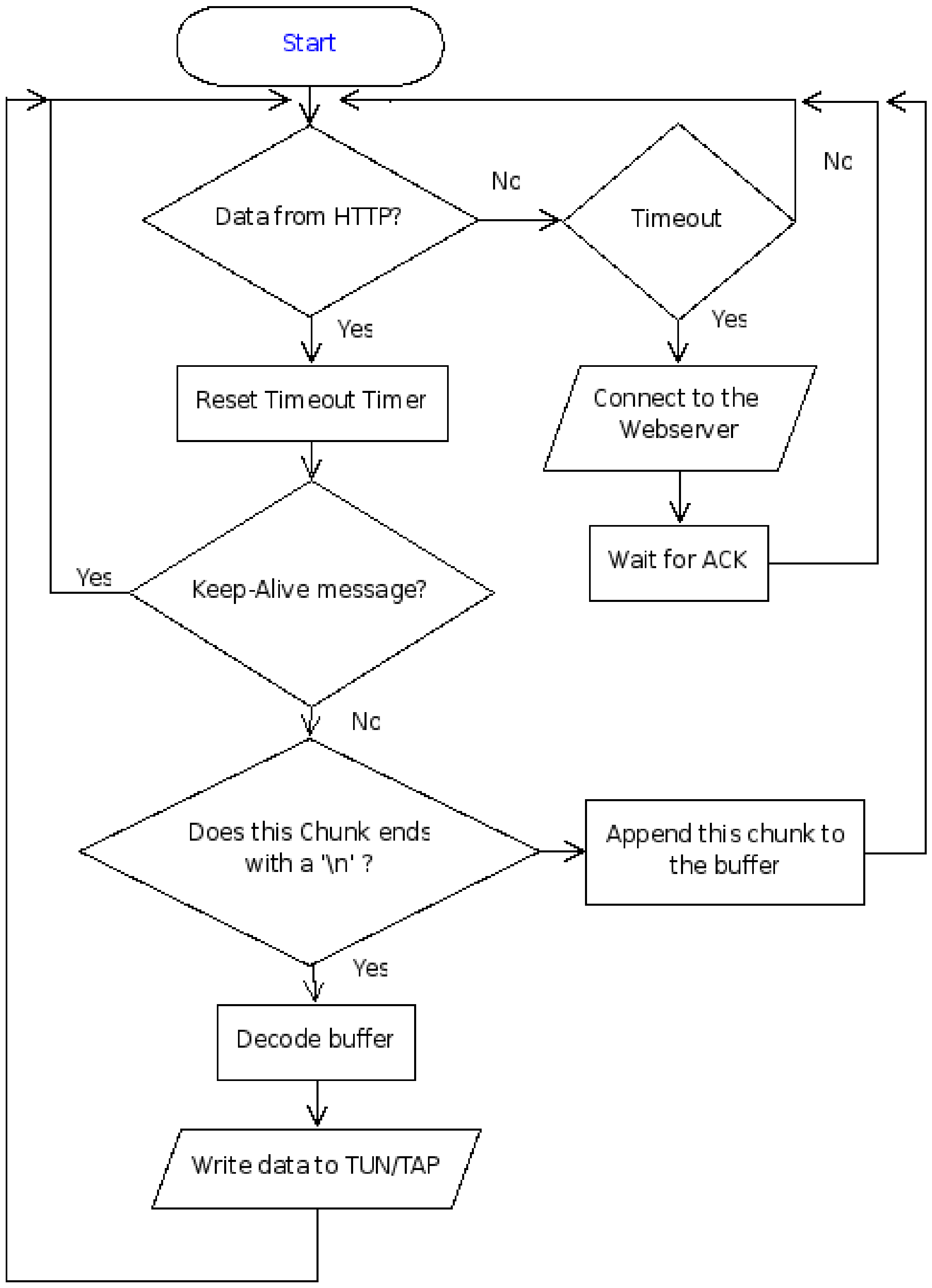


Figure 2.2. Receiver's flowchart

Chapter 3

Conclusions

The idea presented, and its implementation, have been tested in different environments and with different traffic types. This chapter will summarize the results of those tests, and will make suggestions to give enough ideas for the interested reader to improve this project.

3.1 Experimental results

This software was written to overcome the limitation imposed by NAT and firewalls. Two main configurations were tested:

1. Both clients are behind a NAT/Firewall and the web server is on the internet: this is a real life scenario and the one through which most of the test on reliability have been done. In some cases, a proxy was added between a client (or both) and the web server to compare the latency time of the network.

2. The two clients and the web server are on the same network: This scenario was adopted to measure the performances of the system.

3.1.1 Tools used

The speed and latency of the VPN have been analyzed using ping, Netperf [6] and Pathrate [4].

Ping. The round trip time (RTT) is very important especially when using interactive protocols like SSH or VoIP and is not related to the bandwidth of the channel. The ping utility sends a packet to the destination host, waits for the answer and then measures the time elapsed.

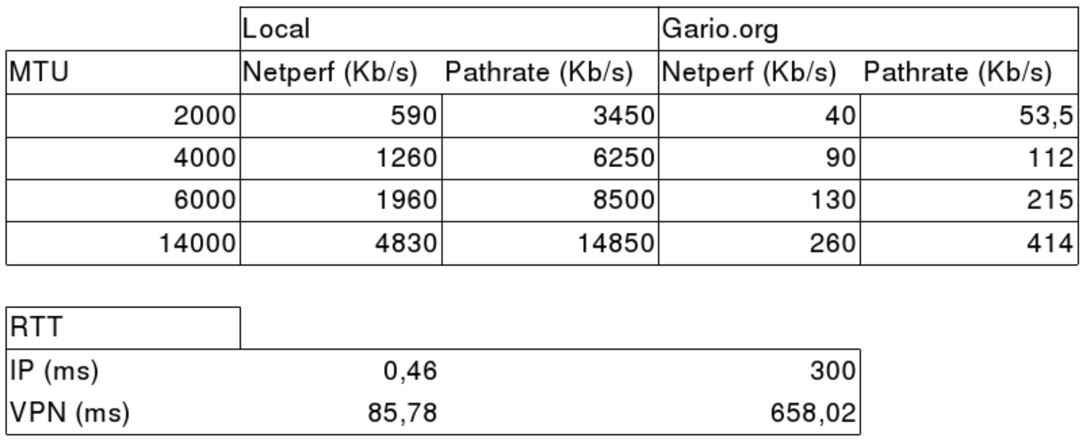
Pathrate. The capacity of a channel is the maximum throughput achievable on the network when there is no cross traffic. In general, the capacity of a channel, is lower or equal to the lowest capacity of the links that compose the channel. It might be possible that the capacity of a channel is lower than the capacity of the slowest link, because it can be influenced by the elaboration time of each hop. There are different techniques used to measure the capacity of a channel. Pathrate uses an end to end method known as "Packet Dispersion" [6] to give an estimation of the capacity of the channel. The idea is an evolution of the packet pair technique. Two packets are sent back to back from one host to the other. When the receiver receives the packets, it measures the delay between the end of the first packet and the start of the second. This delay was introduced during the transmission by the intermediate hops that were working between two links with different capacities. In particular, the capacity of the most tight link is proportional to the delay introduced. This simple method is not very accurate because it can suffer from different noises. Pathrate uses trains of packets of different size to determine the Asymptotic Dispersion Rate (ADR) of the channel. The ADR is not an accurate indication of the channel capacity, but it is a lower bound to the capacity itself [8]. This means that with this tool it is possible to give a pessimistic guess on the capacity of the channel.

Netperf. The throughput of a channel is the speed at which we can transfer data. It is very important for bulk transfers like p2p or streaming. Measuring throughput is a challenging task because it is influenced by every router in the path. The routing path, the routers buffer size, routers QoS, and possible congestion of the network are only just some of the elements that could influence the measuring. Ideally, if there is no other traffic on the channel, the throughput should equal the capacity, but unfortunately the throughput depends on the protocol in use (TCP, UDP etc.). Netperf measures the achievable TCP throughput.

3.1.2 Experimental data

Measuring performances on the internet is quite difficult, because there is little information about what is going on: crosstraffic, server load, routing path etc. On the other hand, in order to test the compatibility of the software with different web servers and infrastructures, it is not possible to relay only on ad-hoc scenarios. This is the reason why I will try to present here the results of the test that do make sense in both scenarios, while I will analyze specific issue in the following sections.

In table 3.1 it is possible to see the different results achieved both locally and with a server hosted by 3ix.org. Bandwidth and throughput have been measured with different MTUs for the VPN. This means that the MTUs of the TUN/TAP interface was bigger than 1500 bytes, while the MTU of the underlying interface was 1500 bytes. At first sight it doesn't make sense to encapsulate bigger packets into



	Local		Gario.org	
MTU	Netperf (Kb/s)	Pathrate (Kb/s)	Netperf (Kb/s)	Pathrate (Kb/s)
2000	590	3450	40	53,5
4000	1260	6250	90	112
6000	1960	8500	130	215
14000	4830	14850	260	414

RTT		
IP (ms)	0,46	300
VPN (ms)	85,78	658,02

Figure 3.1. Experimental Data

small packets but, as the data proves, using a bigger MTU leads to performance improvements. This result suggests that the major bottleneck is the startup time of the script: if we send fewer but bigger packages we get a throughput improvement. Note that the MTU should influence only the throughput of the channel and not its capacity. Unfortunately, the experimental results show that also the capacity measured by pathrate seems to grow with the MTU size. I contacted the author of the software to try to understand this behaviour but we couldn't find what the problem was. Probably the startup time is influencing the "measure" of the capacity. It is interesting to see that while the TCP throughput measured by Netperf (both locally and on the internet) and the capacity measured on the internet grow roughly linearly with the MTU size, the capacity measured locally seems to grow asymptotically. This characteristic can be much better appreciated in the Figure 3.2. Unfortunately it was not possible to test the software with any MTU bigger than 14000 bytes because Pathrate was not able to handle such long trains of packets. In the Figure 3.2 we see that we might also not consider the capacity of the channel (as measured by Pathrate) as a good indicator of the performances of the tunnel, because the Netperf test is way smaller then the capacity. Theoretically, being the netperf test the only traffic on our VPN, we should expect to use all the capacity of the channel. This proves that the VPN capacity cannot be measured with traditional software like Pathrate, probably because we are using high-latency nodes.

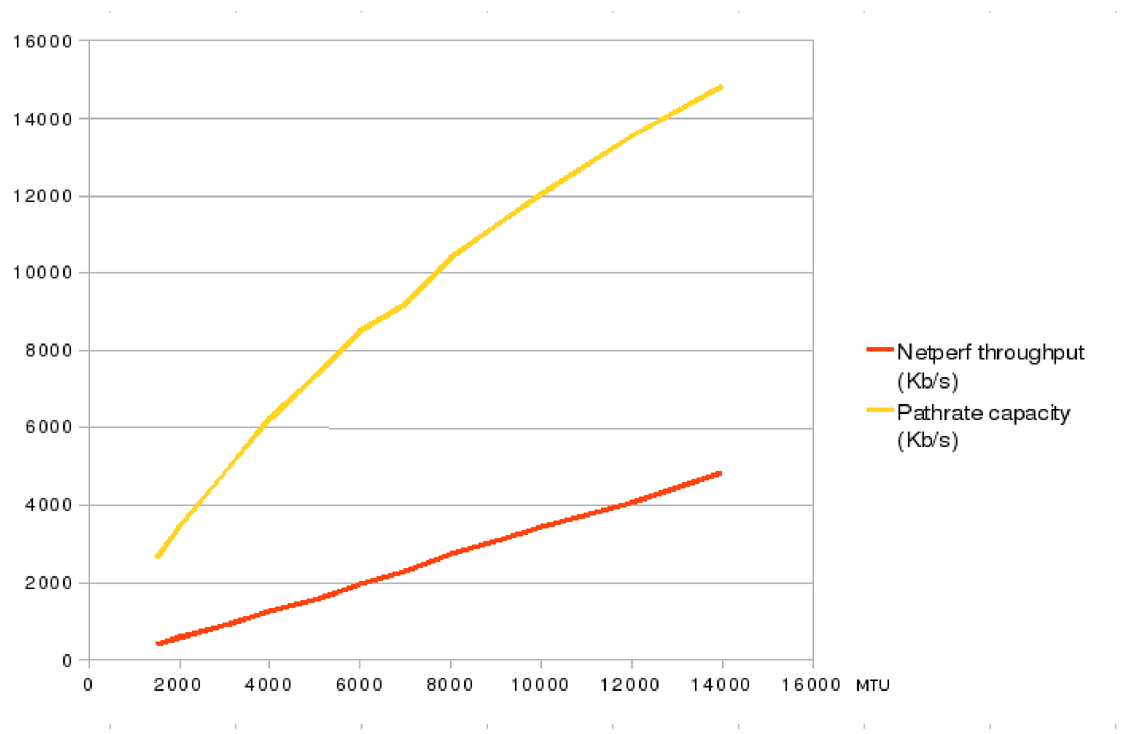


Figure 3.2. Capacity and Throughput diagram

3.1.3 Local results

With the local configuration it was possible to measure both the startup time and the capacity ratio of the VPN.

I already explained why the startup time of the script is important, and in the previous section I underlined its correlation to the performances of the VPN. With the data in the table 3.1 we can try to measure how long this timeframe is. The web server was configured as a router and the two clients were configured on different IP networks. A ping test was then performed between the two clients with and without the vpn. The results are the RTTs reported in the table 3.1.

The startup time can be calculated as: $T_s = \frac{(RTT_{VPN} - 2*RTT_{IP})}{2} \approx 42ms$

This formula assumes that: a) a ping requires one packet to be sent from each client, b) the encapsulated ping can be transmitted using only a single TCP packet, c) the path from the host to the router/web server is symmetric between both hosts.

Those three conditions were met in my test environment, so it is possible to say that I had a startup time of 42ms. Please note that this value will be different in different networks and on different servers, but I thought it was important to give an idea of the entity of this delay.

After measuring the startup time, I was interested in seeing what the maximum achievable transfer rate was. I therefore decided to compare the results obtained with Netperf and Pathrate with and without the VPN. All the local measurements were performed on a 100Mbit network, without traffic. On this network, Pathrate measured (correctly) 98 Mb/s and Netperf measured 92 Mb/s . If we consider the results obtained with an MTU of 14000 bytes as the "maximum values", we can then say that the capacity of the VPN is the 15% of the capacity of the underlying channel and that the throughput is just the 5% of the original throughput. Those results suggest that the VPN should not be used to perform bulk transfers because, even when using a fast network, the transfer rate will be very low.

3.1.4 Internet results

As the system was originally designed to work on the internet, I considered it necessary to measure the latency and the throughput of the channel also on it. Measuring the capacity of the VPN on the internet with Pathrate was an extremely long process because many tests didn't provide good results: there was too much noise to obtain a clean value. On the other hand, Netperf provided quite good results, stating that the throughput was 260Kbit/s (when the MTU was 14000 bytes). I then decided to stress the hosting site and to transfer a big file (26MB) using a raw TCP connection to see if it was possible to really achieve 260Kbit/s of transfer rate. Unfortunately, the transfer rate achieved in this test was slightly above 39Kb/s. This result confirms what stated earlier: the VPN is not suitable for bulk transfers. The difference between the throughput measured by Netperf and the one achieved through the file transfer is probably caused by the length of the test. The Netperf test is set to last only 10 seconds, while the file transfer took almost 2 hours to complete. During those 2 hours the connection with the webserver was re-established many times by both clients, and this is probably the cause for the big delay.

In order to improve the speed of the system I implemented the pipeline. The implementation was very simple: the sender only had to send many requests together and it relayed on the TCP's timeout to send the packages that were not acknowledged. Unfortunately, this solution didn't improve the overall performances, and in particular the transfer rate. On the contrary, it did improve the ability to send two ping packets very close to each other. Actually it made the channel so unreliable ($\sim$ 30% packet loss) that the transfer of the 26MB file took twice the time needed without the pipeline. The unreliability of the channel also made it impossible to use Netperf to measure the TCP throughput. Those results suggest that a simple implementation of the pipeline does not improve performances and that a control over which packets have been correctly received is necessary in order to really improve the performances.

Once ascertained the inability of the system to perform bulk transfers, I decided to test the latency of the connection, to see how interactive applications would behave when used through the VPN. The RTT was measured to be around 660ms. This latency is quite high, but it is not high enough to make SSH unusable. Unfortunately, when testing the latency with different public proxies (in different countries), the RTT grows up between 1 and 3 seconds, making it really hard to use interactive programs with the VPN when using a proxy.

Those experimental results show how difficult it could be to use this VPN for daily use. Nevertheless, the VPN is still useful to reach a host that would otherwise be unreachable. For instance, it can be used to “open” a connection through a fastest channel (dial-up or ADSL), or to publish simple status information about the host. Indeed, the goal was to make a communication between hosts behind NATs possible, and this objective was fully achieved.

3.2 Alternatives

Asynchronous HTTP communication is called with different names in different contexts. When I was realizing the first version of my software, I couldn’t find any information on how to achieve an asynchronous HTTP communication. Nevertheless a couple of months ago, after releasing the first versions of my software, I found some documents talking about something very similar: HTTP pushing [9]. HTTP pushing is the server-side equivalent of AJAX. This idea is also commonly referred to as Comet [10] and generally uses a technique known as Long Polling. The background idea is that a request is performed by the client but the server doesn’t respond immediately. On the contrary, the server responds only when it has data to send or when the request is going to expire. When the request expires, a new request is performed and so on. This means that the request is kept on hold and this is the reason why it is called long polling. HTTP pushing is oriented to browser communication and generally multi-part MIME is preferred over chunked coding. Not only it is easier to get multi-part MIME working in AJAX applications, but it also avoids the never ending “Page loading...” effect.

3.3 Further developments

Some of the possible improvements that can be done to the software have been already discussed when talking about the limitations of the software itself. Nevertheless improving security and speed are not the only possible developments for this project.

Proxing HTTP requests The asynchronous HTTP tunnel is the ideal solution to write a proxy. In the solution I presented the single requests where forwarded without being interpreted. If we slightly modify the logic of the server side script, it should be easy to write a proxy forwarding HTTP requests. To make it possible, it is necessary that the web server, where the script is hosted, allows outgoing connections. From my experience there are quite a lot of hosting sites that allow it, but it would be interesting to find some data to prove the feasibility of the idea. This way it would be also possible to make a multi-hop connection travel through different web sites. This could be very useful, for instance, to build anonymity networks to provide access to information in those countries where strong censorship makes it hard to access information websites and social networks.

Multiple users. The original idea was to establish a connection between two well known hosts. While it represent a common VPN usage, another common usage is the road warrior, or mobile user. The idea is that a user should be able to connect to the VPN from any place at any time. The system I described requires a connection to be established beforehand with the webserver for each client. This means that if I want to allow 10 people to connect to my computer, I need to establish 10 connections from my computer to the webserver. It would be interesting to study a way to tell a host to "open" a new connection.

Port the server to Windows. The implementation I presented only works on *nix, and a port to Windows would be an interesting challenge. The script takes advantage of the Unix domain sockets that (obviously) are not available on Windows. It would be interesting to find and compare some alternatives to Unix domain sockets on Windows, trying to avoid IIS specific functions.

Appendix A

mirror.php

```
<?php

// mirror.php
// v 0.2

$DEBUG=0;
/* Where should the named socket be created?
 * Note: This directory shouldn’t be readable by other users
 */

$WRITE_DIR="/tmp/";

/* How many seconds do we wait before sending the Keep-Alive */
$KEEP_ALIVE=4;

// Get the tunnel info like mirror.php?s=A&d=B&m=r
$src=$_GET[’s’];
$dst=$_GET[’d’];
$mode=$_GET[’m’];

if(isset($_GET[’conf’])){
 $err=0;
 echo "<h2> Configuration menu </h2>";
 echo "<ul>";

 echo "<li>posix_mkfifo()...";
 if(function_exists("posix_mkfifo"))
   echo "<font color=\"green\">Ok</font>";
 else{
   echo "<font color=\"red\">NO</font>";
   $err=1;
 }
 echo "</li>";
 echo "<li>".$WRITE_DIR." writable...";
 if(posix_mkfifo($WRITE_DIR . "test",0600))
   echo "<font color=\"green\">Ok</font>";
 else{
   echo "<font color=\"red\">NO</font>";
   $err=1;
 }


 exit(0);
}

// Security check
if(!( ereg("^[a-zA-Z]*$",$src) )){
  echo "Error";
  exit(0);
}
if(!( ereg("^[a-zA-Z]*$",$dst) )){
  echo "Error";
  exit(0);
}
if(!( ereg("^[rw]$",$mode) )){
  echo "Error";
  exit(0);
}



if($DEBUG){
 $log=fopen("log","a+");
 fwrite($log,"req: " . $src . ":" . $dst . ":" .$mode ."\n");
}
/*
 4 tunnel are to be used:
 A:B:r (used by B to read data)
 A:B:w (used by A to write data)
 B:A:r (used by A to read data)
 B:A:w (user by B to write data)
*/


$path_s= $WRITE_DIR . "pipe" . $src . "_" . $dst;

/*
 Create the fifo
*/

if(! file_exists($path_s)){
  posix_mkfifo($path_s,0600);
  if ($DEBUG)
```

```
    fwrite($log,"Pipe: " . $path_s . " created" . "\n");
}

/*
 Open the fifo
 ( Mode is append [r+|w+] )
*/

$pipe = fopen($path_s,$mode."+");

if ( $mode=="r" ){

  $w = $e = NULL;

  /*
   Print content or keep-alive)
  */

  header('Content-type: text/html');
  ob_flush();
  print("\n");
  /* Force flush to ensure chuncked encoding */
  flush();

  while (1){
    $read=array($pipe);
    // We set a Timeout to keep-alive
    $n = stream_select($read,$w,$e,$KEEP_ALIVE);

    if ( $n > 0){
      $data = fgets($pipe);
      print($data);
      flush();
      if($DEBUG){
        fwrite($log,"read < ". $data);
        fflush($log);
      }

    }else{
      // Keep Alive
      print("\n");
      flush();
    }
  }

}else if($mode == "w") {
   $s = fopen("php://input","r+");
   $data = fgets($s);
   fwrite($pipe,$data);
   fflush($pipe);
   if($DEBUG){
     fwrite($log,"write > ". $data);
     fflush($log);
   }
   /* Return an acknowledge that data was written */
   print("Ok\n");
}
?>
```

Appendix B

tun.py

```
#!/usr/bin/python

# v 0.1.10


# OPTIONS:

# HOST is the address of the server. It can be the same as HOST_URI.
HOST="192.168.0.2"
PORT=80

# HOST_URI is passed in the header "Host:" during the HTTP requests.
HOST_URI="httpvpn.example.org"

# URL where the mirror script resides. Remember the / at the beginning
URL="/mirror.php"

# MODE indicates which type of script is in use: php or cgi.
MODE="php"

# ME is the localhost "name". This name is used to created the pipe on the server and
# as an address to this host, so it must be unique on this server.
# It can be any combination of letters that can be used for an URL.
ME="B"

# PEER is the name of the client we want to connect to. See ME for details.
PEER="A"

# L_IP and L_NETMASK are the local IP and netmask of the tunnel interface.
L_IP="10.0.0.1"
L_NETMASK="255.255.255.0"
# MTU is the MTU to be used in the tunnel interface.
MTU=2000
```

```
# KEY is the Shared key you are using in the Encrypt/Decrypt.
# When using Blowfish encryption, the key size must be a multiple of 8
DISABLE_ENCRYPTION=0
KEY="HTTPVPN12345!!!!"

# PROXY specifies the IP address of the proxy. In order for this option to
# have effect, PROXY_ENABLE must be set to 1

PROXY_ENABLE=0
PROXY="194.176.176.82"
PROXY_PORT=8080

# TIMEOUT is the timespan (in seconds) before sending a keep-alive to the http server.
TIMEOUT=9

# DEBUG specifies the output level of debugging information (0=none,1=normal,2,3..)
DEBUG=0

# Trace enables live debugging
TRACE=0

### CRYPTOGRAPHIC CODE ###
# Note: This code was intentionally separated from the VPN itself to give the ability to
#       anyone to change and experiment with different encryption functions.


from Crypto.Cipher import Blowfish
from Crypto.Hash import SHA

def EncryptData(s,key):

  # Save data length
  data_len = hex(len(s))[2:].zfill(4)

  s = "%s%s" % (data_len , s)

  # Padding data to be multiple of 8
  l=len(s)+40
  pad = (8 - (l%8) ) + 1
  s = s.ljust(8-len(s)%8+len(s),"X")


  # Calculate the digest
  digest = SHA.new(s).hexdigest()

  # put all together
  s = "%s%s" % (digest, s)

  blowfish = Blowfish.new(key)
  data=blowfish.encrypt(s)
  return (data,0)

def DecryptData(s,key):
  blowfish= Blowfish.new(key)

  try:
    data = blowfish.decrypt(s)
  except:
    print "Error: DecryptData "
    return("",1)

  digest = data[0:40]
  data=data[40:]
  new_digest = SHA.new(data).hexdigest()

  if( new_digest == digest):
     l = int(data[0:4],16)
     data = data[4:l+4]

     return (data,0)
  else:
    print "Digest don't match: " + new_digest + " vs " + digest
    return ("",1)


### CODE ###
import os, sys
from socket import *
from fcntl import ioctl
from select import select
import getopt, struct
from base64 import b64encode,b64decode
from time import time
import httplib

from md5 import md5

if(TRACE):
  import pdb

# This are just constant needed by the TUN/TAP driver
TUNSETIFF = 0x400454ca
IFF_TUN   = 0x0001
IFF_TAP   = 0x0002

TUNMODE = IFF_TUN
#TUNMODE = IFF_TAP
```

```
"""
   Connect the Receiver to the webserver
"""
def Recv_Connect(url):

  recv_s = socket(AF_INET, SOCK_STREAM)
  if (PROXY_ENABLE):
    recv_s.connect((PROXY,PROXY_PORT))
  else:
    recv_s.connect((HOST,PORT))

  recv_s.send("GET " + url + " HTTP/1.1\r\n")

  if(PROXY_ENABLE==0):
    recv_s.send("Host: "+HOST_URI +"\r\n")

  recv_s.send("\r\n")
  # Convert the socket to a file to make it easier to use, also note that
  # the buffer's lenght equals the MTU + headers + base64 encoding ratio
  recv_f=recv_s.makefile("r+",((MTU+20+40+8)*8/6)+1 )

  return recv_f


"""
 Main
"""

# Create the tun device
f = os.open("/dev/net/tun", os.O_RDWR)
ifs = ioctl(f, TUNSETIFF, struct.pack("16sH", "tun%d", TUNMODE))

ifname = ifs[:16].strip("\x00")
print "Allocated interface %s." % ifname

# Configure the interface
if (L_IP):
  os.system("ifconfig " + ifname + " " + L_IP + " netmask " + L_NETMASK + " mtu " + str(MTU) )
  print L_IP + "/" + L_NETMASK + "(mtu " + str(MTU) + ") assigned."


"""
 Preparing URLS
"""
if (MODE=="php"):
  recv_url=URL+"?s="+ PEER+ "&d=" + ME +"&m=r"
  send_url=URL+"?s="+ ME + "&d=" + PEER +"&m=w"
else:
  recv_url=URL+"?"+ PEER+ ":" + ME +":r"
  send_url=URL+"?"+ ME + ":" + PEER +":w"

if (PROXY_ENABLE):
  recv_url="http://"+HOST_URI+recv_url
  send_url="http://"+HOST_URI+send_url

recv_f=Recv_Connect(recv_url)

if(DEBUG):
  print "Recv_url:"+recv_url
  print "Send_url:"+send_url

"""
 Connect to send data
"""
if(PROXY_ENABLE):
  print "Using Proxy " + PROXY + ":" + str(PROXY_PORT)
  send_h = httplib.HTTPConnection(PROXY,PROXY_PORT)
else:
  send_h = httplib.HTTPConnection(HOST_URI)

if(DEBUG >=3 ):
  send_h.set_debuglevel(2)

# recv_status=0 means that we are waiting for the webserver acknowledge
recv_status=0

try:
  pid = os.fork()
  if pid > 0:
  # Parent process: Receiver
    while 1:
      r = select([recv_f],[],[],TIMEOUT)[0]
      if ( not r ):
      # Timeout: We didn't received any packet in TIMEOUT seconds.
      #          Normally the keep_alive on mirror.php is shorter than this timeout.
      #          This probably means we lost the connection with the webserver
        print "recv: Timeout. Connecting again to " + recv_url
        recv_f=Recv_Connect(recv_url)
        recv_status=0

      elif (r[0]==recv_f):
        try:
        # Data from HTTP
          data=recv_f.readline()
          if ( recv_status ):
            # We need to know when we received the last packet to know if
            # we lost the connection with the webserver
            last_pkt_timestamp=time()

            if(int(data,16)==1):
              # Skip keep-alive packets
              if (DEBUG):
                print "skip keep-alive:" + data
              recv_f.read(3)    # \n\r\n
            else:
              # Get chunk size
              # Most server limit chunk size to 0x2000 (8192 byte)

              c_len = int(data,16)
              data=recv_f.read(c_len)
              if(c_len==8192):
                while(data[-1]!='\n'):
                  if(DEBUG):
                    print "Fetching more chunks"
                  recv_f.read(2)
                  c_len=int(recv_f.readline()[:-1],16)
                  data="%s%s"%(data,recv_f.read(c_len))

              # Get rid of CRLF
              recv_f.read(2)

              if(DEBUG):
                print "Data from HTTP length=%s:%s"%(str(len(data)),md5(data).hexdigest())

              clear_data=""
              try:
                clear_data=b64decode(data[:-1])
              except:
                if(TRACE):
                  pdb.set_trace()

              err=0
              if(not DISABLE_ENCRYPTION):
                (clear_data,err)=DecryptData(clear_data,KEY)
              if(not err):
                if(DEBUG>=2):
                  print "Decoded %s:%s" % (str(len(clear_data)),md5(clear_data).hexdigest())
                # Writes data to tun/tap
                os.write(f,clear_data)
              else:
                print "An error has occured while Decrypting Data: "+ str(err)

          else:
            # We are waiting for the server ack
            if( data[:15] =="HTTP/1.0 200 OK"):
              print "WARNING: HTTP/1.0 The system MAY not work"
```

```
          # Check if answer is HTTP/1.X 200 OK
          if( data[:7] == "HTTP/1." and data[9:15] =="200 OK" ):
            if(DEBUG):
              print "Waiting for recv_ack"
              print data

            d=recv_f.readline()
            # Discard Headers
            while (d!="\r\n"):
              if(DEBUG >=3):
                print len(d)
                print d
              d=recv_f.readline()

            # Read chunk size and convert to decimal
            c_len = int(recv_f.readline(), 16)
            d=recv_f.read(c_len)

            # Get rid of ending CRLF
            recv_f.read(2)
            recv_status=1
            print "recv: Connected"
      except:
        # Reset the connection
        print "Error: HTTP data"
        recv_f=Recv_Connect(recv_url)
        recv_status=0


else:
# Child process: Sender
  while 1:
    r = select([f],[],[],TIMEOUT)[0]

    # Data from TUN/TAP
    data = os.read(f,MTU + 20)
    if (DEBUG):
      print "Data From TUN length=%s:%s" % (str(len(data)),md5(data).hexdigest())

    err=0

    if(not DISABLE_ENCRYPTION):
      (data,err) = EncryptData(data,KEY)

    if (not err):
      encoded_data = b64encode(data)+"\n"
      if(DEBUG >= 2):
        print "Encoded %s:%s" % (str(len(encoded_data)),md5(encoded_data).hexdigest())
    send_h.request("POST",send_url,encoded_data)
    try:
      res = send_h.getresponse()
    except httplib.BadStatusLine:
      if(DEBUG):
       print "BadStatusLine reconnecting..."
      if(PROXY_ENABLE):
       send_h = httplib.HTTPConnection(PROXY,PROXY_PORT)
      else:
       send_h = httplib.HTTPConnection(HOST_URI)
      send_h.request("POST",send_url,encoded_data)
      res = send_h.getresponse()

    if (res.status == 200):
     res.read()
    else:
     print "An error has occured while sending data: " + str(res.status)
     print res.read()
   else:
    print "An error has occured while Encrypting the data: "+ str(err)

except OSError, e:
 print >>sys.stderr, "fork #1 failed: %d (%s)" % (e.errno, e.strerror)
 sys.exit(1)
```

Bibliography

[1] Fielding, et al., *Hypertext Transfer Protocol - HTTP/1.1*, IETF RFC 2616, June 1999.

[2] *CGI 1.1*, "http://hoohoo.ncsa.illinois.edu/cgi/"

[3] *FastCGI*, "http://www.fastcgi.com", 2007.

[4] *Pathrate*, "http://www.cc.gatech.edu/fac/Constantinos.Dovrolis/bw-est/pathrate.html", September 2006.

[5] B. Hall, *Unix Sockets*, "http://beej.us/guide/bgipc/output/html/multipage/unixsock.html", 2009.

[6] *Netperf*, "http://www.netperf.org/netperf/", February 2007.

[7] C. Dovrolis, D.Moore, P.Ramanathan, *What Do Packet Dispersion Techniques Measure?*, in Proceedings of the 2001 Infocom, Vol. II, p. 905-914, Anchorage AK, April 2001.

[8] R. Prasad, C. Dovrolis, M. Murray, K. Claffy, *Bandwidth estimation: metrics, measurement techniques, and tools*, in IEEE Network, Vol XVII, Issue 6, p. 27-35, 2003.

[9] C. Musciano, B. Kennedy, *HTML: The Definitive Guide - Server-Push Documents*, "http://web.deu.edu.tr/doc/oreily/web/html/ch14_03.html", 1997.

[10] Alex Russell, *Comet: Low Latency Data for the Browser*, "http://alex.dojotoolkit.org/2006/03/comet-low-latency-data-for-the-browser/", March 2006.

[11] *Stunnel – Universal SSL Wrapper*, http://www.stunnel.org/