Building an Anonymous VPN

Bernhard R. Fischer

2048R/5C5FFD47 <*bf@abenteuerland.at*>

Diploma Thesis

Submitted to the St. Pölten University of Applied Sciences

for the degree of

Diplom Ingenieur

Supervisors: Dipl. Ing. Johann Mühlehner and Dipl. Ing. Johann Haag

Institute for Communication Networks

April 2009

Version 499, April 26, 2009

This document has been produced using OpenSource software exclusively. It is created with LaTeX using Texlive on Debian Linux. All figures are drawn with Dia, the UML diagrams have been created with Umbrello. My picture as well as the *Good Smelling Snail* on the acknowledgments page have been resized with the Gimp. The text was solely edited with vim. Spell check was done with Ispell. The PDF file was produced with Ghostscript.

To my children Corina and Patrick

in the hopes to give them a network of freedom.

Declaration

I declare that

- this thesis is all my own work, that the list of sources and aids is complete and that I have received no other unfair assistance of any kind,
- this thesis has not been assessed either in Austria or abroad before and has not been submitted for any other examination paper.

This piece of work corresponds to the work assessed by the appraiser.

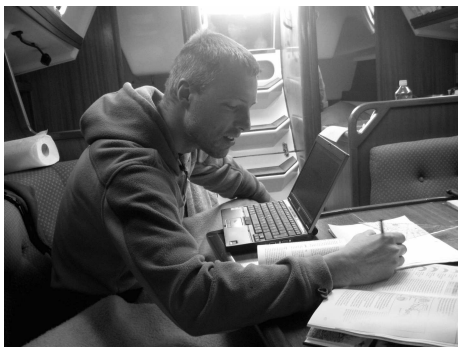
Place, Date *Signature*

"Freedom is never voluntarily given by the oppressor; it must be demanded by the oppressed."

Martin Luther King, Jr.

Acknowledgments

The beginning of this work dates back to December of 2007. It was at the 23rd Chaos Communication Congress in Berlin when we started to talk about it. Therefore I would like to thank Daniel Haslinger for being instrumental in developing the first ideas. Specifically in the last months I received very much support for issues surrounding this project. I would like to thank Ferdinand Haselbacher for everything he did until now to make this project even more attractive.



A good project can never be carried out alone. Thus I want to appreciate my mate marshallbanana of the anonymity community for believing in this project and being an excellent partner for discussion of many issues regarding this topic.

I want to thank my academic supervisor Johann Mühlehner for giving me an even more comprehensive knowledge about operating system internals and for being very patient while waiting for me finishing this work.

Last but not least I want to appreciate my girl-friend Petra for giving me much mental support in many many hours I was writing this document and the source code, and helped me in finalizing this document by correcting errors. The picture shows me working on the document on a sailing yacht in Croatia.

Bernhard R. Fischer, April 26, 2009

Abstract

The Internet is a continuously growing network which reaches more and more areas of everybody's life independently of their occupation. In particular the network grew extraordinary in the last two decades and heavily influences the daily business of individuals and organizations. Data is collected everywhere in a seemingly uncontrolled manner for many plausible and implausible reasons. Specifically in the last years topics like *data retention* and *surveillance* appeared in the public, those affecting individuals in particular. But traffic monitoring may not influence just individuals. This also covers industrial espionage and might affect even the sovereignty of states when considering surveillance of governmental or military data exchange.

As a consequence many efforts are undertaken to build so-called anonymizing networks on top of the Internet, being *Tor* such a network as an example. Their purpose is to hide the fact of who is communicating to whom. Technical speaking this means hiding IP addresses and locations from observers.

The Tor network provides an interface which has to be used in order to gain anonymity. This interface has some limitations. Although it is possible to setup peer to peer connections between participants, it does not allow transmission of IP packets directly.

We are looking for a way to make it capable to transport IP packets and create peer to peer connections between participants in a completely dynamic way, thus creating an anonymous Internet overlay.

This document gives a sound basis on cryptography, networking, and operating systems to fully understand the difficulties and possibilities when creating such an overlay. Finally a solution and a software ready for use is presented in detail.

Contents

List of Figures

List of Listings

1 Introduction

Nowadays, the Internet is used everywhere, in daily business of organizations and companies, regardless of their size. The Internet even found a way into everybody's private life.

This is because it provides features which makes many things of daily life more convenient. It makes it more easy to distribute information. Thus, information is accessible by people that would not have had access to it before. It also makes many things more transparent. The transparency and information distribution property boosts the economy. In part, this is due to competition being increased by those properties previously mentioned. Once commercial establishment found out that something may increase profit, money is invested. Thereby the network further grows.

It attracts the interest of more and more people independently of their occupation. It comes to governments, military, law enforcement, good guys, bad guys, and many more.

The new digital world and the Internet in particular has downsides too.[1] Digital information may be copied without information loss. Once published in the network, it may be spread everywhere, many times. And this paragraph is not about copyright violations. It means every kind of information being published, independently if it was published deliberately or not. For example, this includes web pages, emails, or other personal information of individuals. And information being published once can never be deleted again. It is preserved on many Internet storages, Google as an example.

Information travelling through the network may be monitored by observers, for whatever reason. Specifically in respect to the sovereignty of states and diplomatic efforts, information leakage may have disastrous consequences. Cryptography is being used to avoid such cases. Simple methods (from today's point of view) have been used since ancient times

[1] It is not being said that permanent growth of economy is an adventage. The reader might decide himself.

but specifically the last 50 to 100 years brought many advancements into this field.
But cryptography does not solve all problems. In reality it covers just a small field of problems which exist in respect to information distribution, although it is used in many applications. Cryptography hides the content of something being distributed. It does not hide the fact that communication takes place and who communicates to whom.
With those considerations, a new field of research – the field of *anonymity* – emerged in the late 70's of the last century. A paper about untraceable electronic mail [3] was published in 1981 by David Chaum, but it can be said that this field got much attraction not before 2000. It does not cover just anonymity. The term "anonymity" in particular is one aspect of objectives being explored within this field. A list of papers is found in the *Free Haven's Bibliography* [15].
The property that information within the Internet is traceable is used by governments to monitor their citizens. It is a basic goal of any government to achieve as much control on their citizens as possible. Nowadays, we are told on one hand that governmental or military traffic must not be monitored by opponents to preserve the world's peace, but on the other hand an individuals traffic should be as transparent as possible, for security reasons. There are actually discussions about federal Trojans getting installed on individuals computer systems. This sounds very odd to the author of this paper, hence, the motivation came up on how to prevent from being monitored while not loosing the network's freedom, which existed in the Internet (and also in reality) some time ago.
Several techniques have been developed over the years to prevent information being traceable while carried within the network. Tor, as an example, uses one of those techniques. It is a network based on the Internet which is capable of hiding participants and their activities. It allows users to resist against traffic monitoring and surveillance. It provides an application interface, but this has some limitations.

This paper is about creating a completely transparent application interface and building a network layer with the anonymity property, based on Tor for a start. It can be seen as an anonymous network within the Internet – an anonymous Internet overlay.
With this, a new network can evolve, providing the same information and flexibility as the Internet does, with the advantage of anonymity in respect to network addressing.
A network within a network usually is referred to as VPN. It utilizes the same protocols as are used for the Internet. Thus, all types of services that work within the Internet also work within such a VPN.

Within this paper it is explained how to create an anonymous VPN, which uses a completely decentralized approach. This type of approach is usually being referred to as *peer-to-peer* networks. As it is true for all types of networks, a method of addressing is necessary to identify participants. The method of choice provides unique identification but hides a client's real identity in respect to Internet addresses.

This document is split into two parts. Part I explains basics on cryptography in Chapter 2 and gives a detailed view on fundamental network protocols in Chapter 3. Chapter 4 is about operating systems and their interaction with networks. The thoughts covered herein are based on the Tor network, hence, Chapter 5 is about it and its *hidden services*.

Part II describes in detail how the anonymous VPN is built. This is covered in Chapter 6. It also includes the description of the addressing method which is crucial for such a network. Chapter 7 is about a software implementation in general and explains the implementation of the reference project called *OnionCat*. Chapter 8 shows some use cases and gives detailed installation and configuration instructions for OnionCat. The project was published under the GNU GPLv3 license. Further project documentation as well as the complete source code and some packages are found at [13] www.cypherpunk.at/onioncat. During this development several new topics emerged. Those are documented in the last Chapter 9.

Part I

Prerequisites

2 Cryptographic Building Blocks

Cryptography has been used since thousands of years to hide secrets from adversaries. Considering the famous *Caesar's Cipher* [39] used by the Roman emperor Julius Caesar to securely communicate with Cleopatra, Queen of Egypt, as the first documented cipher. It is a simple substitution algorithm which can easily be broken today even from people not being cryptanalytic experts.

With further development of human knowledge about mathematics, physics and similar scientific fields, also the complexity of ciphers signifficantly increased. The *Enigma Machine* was a mechanical implementation of a fairly complex algorithm used by the Germans during World War II. Great efforts had been necessary and were undertaken by the allies to break the code. But first the Polish later the British were able to break intercepted messages within hours, at least for the weaker codes [39]. Alan Turing was one of the mathematicians involved in code breaking.

Since the 1970s really strong algorithms were developed although some of the concepts used date back several centuries. But the still increasing computational power made things possible that have not been possible before. This computational power permitted to pick up older concepts again which were originally dropped because of too high complexity then.

Cryptography is one of the most fundamental prerequisites for anonymization networks. In the following I will explain cryptographic building blocks which are widely used today as far as necessary under this topic. More detailed explanations about cryptography can be found in [12] and on a much more technical basis in [39].

2.1 Encryption and Decryption

The basic idea is that two partners, *A*lice and *B*ob, want to communicate secure. That is that everybody in between – like the *e*avesdropper *E*ve – who reads the message cannot deduce the original meaning of the text. Let's assume that A and B already exchanged a "small" secret K_e, further known as the key. A encrypts her message m using an encryption algorithm together with the key K_e to produce the cipher text c. You can think of the encryption algorithm as the mathematic function $E(m, k)$ taking two arguments: the message m and the key K_e. The encryption algorithm applies the secret K_e to the message in a way that the whole messages turns to be secret.

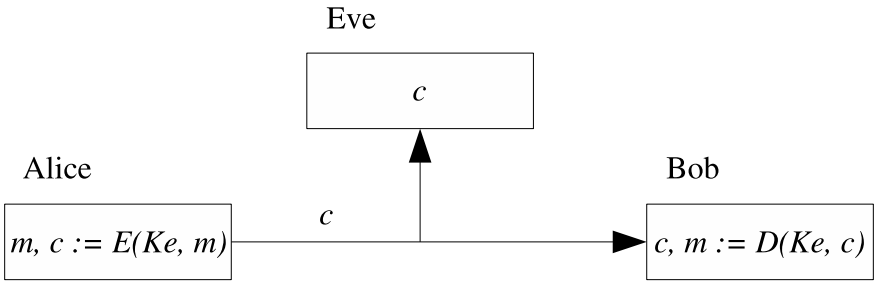


Figure 2.1: Symmetric encryption.

As shown in Figure 2.1, A calculates $c = E(m, K_e)$ and sends c across an insecure channel. B receives c and calculates $m = D(c, K_e)$ to retrieve the original message m. The decryption algorithm removes the secret from the cipher text c. Note that c is not revertable to m if they either use a different key K_e or c delivered to B is not the same as A sent. That is, c was altered during transmission and that is the lack of encryption. To detect modifications of cipher text we need a concept called authentication. See 2.2 for details. This type of algorithms are called *symmetric ciphers*, because both parties A and B use the same key k. Symmetric ciphers are usually very fast compared to their counterparts the *asymmetric ciphers*. Symmetric ciphers internally use a very basic set of operations to accomplish the task of encryption and decryption. Those are simple additions and bit operations like bit shifting, and boolean operations like AND, OR, XOR, and so on.

Asymmetric ciphers use two different keys. One for encryption and one for decryption. Those keys are dependent on each other, hence, together they form a key pair. The "dangerous" one is the decryption key because one who holds it can decrypt ciphered text.

Thus, it should be kept secret and is therefore called secret key or more frequently the *private key*. The other one – the encryption key – is called the *public key*. It can be made public because it can only be used for encryption, but not for decryption and, additionally, it is not possible to derive the private key from it (but the public key is derived from the private one). This type of algorithms belong to the field of *public key cryptography*.

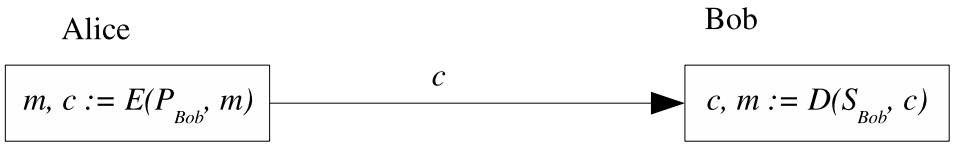


Figure 2.2: Asymmetric encryption.

Figure 2.2 shows the pretty similar picture of encryption with asymmetric ciphers, like Figure 2.1 does for symmetric ones. *A*lice encrypts the message m with *B*ob's public key P_{Bob} using the encryption function $c = E(P_{Bob}, m)$ and sends the cipher text c across the insecure channel to *B*ob. He can then decrypt the cipher text c with his private key S_{Bob} using the decryption function $m = D(S_{Bob}, c)$.

As it is with symmetric ciphers, *B*ob cannot detect modifications of the cipher text. Asymmetric ciphers are slow because internally they use fairly complex mathematics with really huge numbers which is not a cakewalk even for todays computers.

The big advantage of public key cryptography is that it simplifies the problem of *key exchange*.

Those two Figures (2.2, 2.1) from above both assume that the keys are already exchanged. Once they are exchanged, secure communication across an insecure channel is no problem. But exchanging keys is really difficult and not sufficiently solved today. Witfield Diffie and Martin Hellman founded the basic fundamentals of public key cryptography and key exchange in 1976 with their famous paper *"New Directions in Cryptography"* [10]. They developed the *DH key exchange protocol*[1] which is still used today in combination with public key cryptography for key exchange.

A further concept necessary for building trust relation ships is *authentication* which is described in the following Section 2.2.

[1] Actually it does not exchange keys. It rather generates a key across an insecure channel.

2.2 Authentication

Authentication is a concept to prove that something or somebody is authentic. In the field of digital data transmission that means that you can check if a piece of data that arrived is as it was before sending. That is, the data was not altered during transmission. With some limitations it is even possible to identify who sent the data.

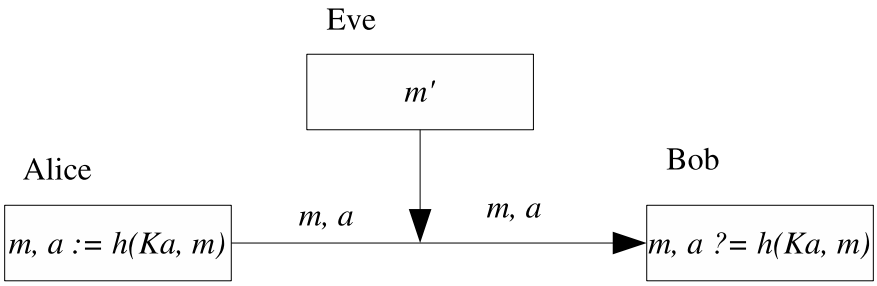


Figure 2.3: Message authentication.

Figure 2.3 shows the concept of *message authentication*. Before message authentication can take place, again, *A*lice and *B*ob have to agree on a shared secret, the message authentication key K_a. *A*lice can then calculate a *message authentication code* (MAC) a using the function $a = h(K_a, m)$ which takes the key K_a and the message m as arguments. Next she sends the message m together with its MAC a to *B*ob.
*B*ob can verify that no modification happened during transmission. He calculates his own MAC using the same function $a' = h(K_a, m)$ together with the received message m and the key K_a that he agreed with *A*lice. Then he compares if his calculation results in the same MAC as the one he received: $a' = a$. If the comparison evaluates to *true*, the MAC was calculated by someone who owns K_a and the message was not altered after calculation of a. This implies that the message was sent by *A*lice, if we assume that K_a is owned exclusively by her (and *B*ob). If the equation evaluates to *false* either the MAC a or the message m was modified, or a was not calculated with the right authentication key K_a.

Now look a little bit more into details of MAC calculation. The message authentication code usually is a small piece of data in the range of 128 to 512 bits (16 to 64 bytes). But the message itself could be really huge, for example a file of several giga bytes. Thus, some kind of data "compression" is necessary. We need a function that projects an theoretically infinite set of input data into a limited set of output data. This is what hash functions do.

Hash functions have a wide range of applications in information technology, but not all of them are applicable to be used in cryptography. A property of all hash functions is that they have *collisions* because of the infinite number of possible inputs but finite number of outputs. It is called a collision if two different inputs result in the same output value. Thus they are indistinguishable after hashing.

There is a class of specific hash functions, the *cryptographic* hash functions [12] whose collisions are unpredictable. This is that they still have collisions but we are unable to consciously produce one. Very well known and most widely used hash functions are MD5 (Message Digest Version 5), SHA-1 (Secure Hashing Algorithm Version 1)[2], and its newer versions based on the same algorithm with a greater output set: SHA-256 and SHA-512. Both, MD5 and SHA-1 are internally based on a similar algorithm. MD5 has a result of 128 bit, SHA-1 results in a 160 bit wide value, SHA-256 in 256 bits, and SHA-512 in 512 bits, obviously.

With those functions we are now able to calculate "fingerprints" of messages by applying a hash function to the message: $d_m = H(m)$. This d_m is called a *message digest* or a *fingerprint* of a message. The message digest d_m is unique to the message,[3] hence, it may be used as an identifier for the message m. Even if changing a single bit in m, several bits in d_m will change randomly. Of course it is not possible to retrieve the original message m out of the digest d_m.

With this cryptographic hash functions we can create message authentication codes by simply appending the authentication key K_a to the message m and calculating d_m, as an example. As explained above, without having K_a no one will be able to generate a valid message digest. Ferguson [12] explains that digests with simple appended keys may be broken easily and more sophisticated algorithms like HMAC (Hashed Message Authentication Code) should be used instead.

MACs are based on symmetric cryptography. If we apply the concept of public key cryptography to it, they become *digital signatures*. It is necessary to use a public key algorithm which is able to encrypt with the private key and decrypt with the public key. This is different from what we need for normal encryption as has been described in Section 2.1.

[2] It is the slightly adapted successor of SHA Version 0.

[3] As explained above in reality we are not able to find a different message m' with $d_{m'} = d_m$. But there are some approaches to generate collisions. [47]

It can be done with RSA or the *digital signature algorithm* (DSA), as an example. Both are described in [39].

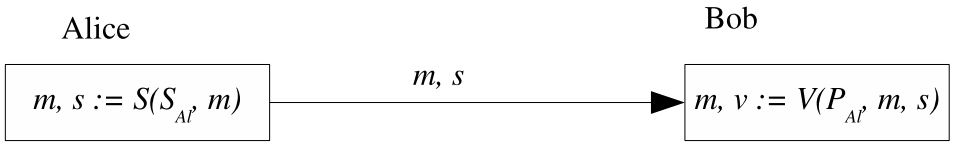


Figure 2.4: Digital signatures.

The procedure is shown in Figure 2.4 on page 10. *A*lice calculates a hash value of the message and encrypts it with her private key S_{Al}. This happens within the digital signature function $s = S(S_{Al}, m)$. The signature s is sent to *B*ob along with the message m. Bob receives it and generates in turn the hash value h' of the message. Then he decrypts the signature using *A*lice' public key P_{Al}. This results in a hash value h which *A*lice originally computed. *B*ob then compares if h equals h'. All together this is done within the verification function $V(P_{Al}, m, s)$. It returns true or false. If it is true, he knows that the message was unmodified and it was signed by *A*lice. Otherwise either the signature or the message was altered during transmission. *B*ob cannot forge *A*lice' signature because s cannot be computed with the public key.

2.3 Summary

In this Chapter I briefly introduced the concepts of encryption. It is used by the Tor network to hide data from observers. Tor is the anonymizer of choice for this project. I further introduced the cryptographic mechanisms of hash functions and authentication. These are used by the hidden services, which are services provided by the Tor network. They are used as a communication basis for this project.

3 Networking Basics

I will discuss several fundamental network basics here which are vital for understanding several mechanisms regarding Tor and this project about building a VPN on top of it. Network protocols are usually classified using the *OSI*[1] *7 Layer Model* [42]. As the name already tells, it is structured into seven layers starting with the *physical layer* as the lowest, describing electrical parameters like voltages, currents, frequencies, cables, and so on. The 7th layer describes application behavior, like sending emails. Thus, it is called *application layer*. The layers in between deal with various aspects of network communication. One major property is that those layers are dependent on each other. The higher ones depend on the lower ones. That is, for example, if layer 3 does not work for some reason then also the layers 4 to 7 will not work. This is absolute essential.

Figure 3.1 on page 12 is a simplified view of the OSI layer model. It shows only five layers symbolized as boxes. The lowest at the bottom is layer 1 and represents the physical layer, as already mentioned. The physical layer ensures to transmit a single bit. The assumption is, that if one can transmit one bit he can transmit any number of bits. The gray shaded arrows mean that every lower layer offers an interface to the upper layer. And the upper layers use those interfaces. These interfaces are well defined and can only be used if the upper layer knows exactly how they work. That does also mean that, for example, the transport layer (4) cannot directly use the data-link layer (2), for two reasons. First it breaks the model because layer 4 shall be based on layer 3 and second, layer 4 does not "know" the interface of layer 2 – mainly because of the first reason.

Layer 2 is the *data-link layer*. It makes sure that the virtually endless stream of bits gets framed which means it forms data structures. The bit stream in reality is not endless. It is a flow of packets – in layer 2 those are called *frames* – which have beginnings and ends. Additionally layer 2 has addressing capabilities. That is of major importance if a

[1] Open Systems Interconnection

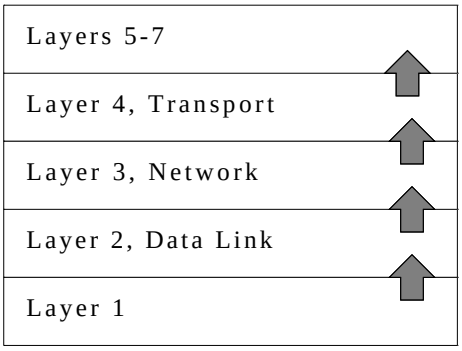


Figure 3.1: The protocol layer model.

multi-access protocol is used. Different from *point-to-point protocols*, more than two hosts can be connected together at the same time on the same physical segment. This obviously needs some kind of addressing to designate frames to specific hosts – and not to any of the connected ones.
One of the most widely used and under this topic important layer 2 protocol is *Ethernet*. I will discuss the necessary issues in Section 3.2.

Layer 3 is the *network layer*. It is designed to create *logical addressing*, do *"best" route selection*, and *packet routing*. It transports several basically independent packets across a "large" network consisting of several network segments, specifically several layer 2 segments. They could be of different type, for example one segment could be Ethernet, another one could be HDLC.[2]
Logical addressing means network design oriented address assignment. It enables network designers or administrators to form logical units of hosts in the network, those belonging together. This is different from layer 2. Its addressing method just has the function to connect nodes being addressable on one physical network segment. Physical, because it is based on the physical layer. Logical addressing is user-chosen, hence, we need routing. Routing is the term for finding a path through a network to a designated destination.[3] And if there is more than one path available we need some kind of route selection.
Within this context we use the *Internet Protocol Version 6* (IPv6). The reasons why IPv6

[2] At this point the *high-level data-link control* (HDLC) is a randomly chosen example for a protocol within the data-link layer. But it is not of relevance any further so I will not discuss it subsequently.

[3] Layer 2 has no routing capability because all hosts are connected physically. Thus they can "see" each other and frames of layer 2 are not intended to get passed over to another different network segment.

is used will be explained later in Section 6. Mechanisms associated with layer 3 and IPv6 are described in Section 3.1.

Layer 4 is the transport layer. Its ability is to associate packets together thereby forming data streams, and it provides multiplexing. The latter is the ability to have several different services running on a single logical layer 3 address. Obviously, the task of multiplexing also needs an addressing method. Layer 4 addresses are usually called *port numbers*. There are many transport protocols available, like UDP or TCP. I will discuss them in Section 3.3.

3.1 The IP Protocol Version 6

The IP protocol belongs to layer 3 – the *network layer* – of the OSI model. Nowadays, if the term "IP protocol" is used it usually refers to the IP protocol *version 4*. It was developed in the late 70s and published in RFC791 [34] in 1981. Various reasons required protocol enhancements which led to the IP protocol *version 6* which was first published in RFC1883 [6] in 1995 and was slightly changed and republished in RFC2460 [7] in 1998. A comprehensive overview of many protocol aspects is given in [26].
Within this context the most important protocol change is the address space. IPv4 has an address space of 32 bits but IPv6 has 128 bits which is really huge. The reason why such an address space is needed will be discussed later in Section 6. First the method of addressing is explained.

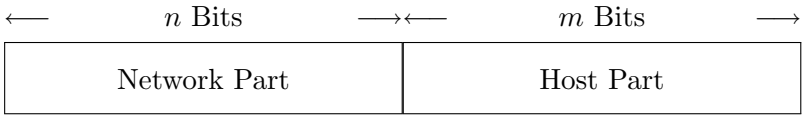


Figure 3.2: IP address format.

As shown in Figure 3.2, an IP address can be split into two parts, independent of the protocol version. The network part is used to address groups of hosts and the host part addresses the hosts within a specific group. The network part usually is referred to just as the *network* or specifically for IPv6 as the *prefix*. Concatenated, being the prefix the most significant and the host part the least significant bits, it forms an IP address of length $n+m = 128$ for IPv6. The length of the prefix is not pre-defined. It may vary. Thus, it must be specified

when configuring a network device. The notation used is called the *CIDR*[4] *notation* [16] and specifies the prefix length with a slash behind the IP address. Generally, the notation of IPv6 addresses is done in hexadecimal number system as eight 16 bit blocks separated by colons. As an example, the address fd87:d87e:eb43:938c:6923:402d:6ca3:1777/48 has a prefix length of $n = 48$ bits, being the prefix part underlined. The prefix length also defines the size of a network which is the maximum number of hosts which a network may contain. Obviously, the size of a network with a given prefix length n is 2^m which is 2^{128-n} for IPv6. For example, a network with a prefix length of $n = 48$ has a size of $2^{80} \approx 10^{24}$. A different concept of notation of prefix lengths is the *network mask* or just *netmask*. It is denoted as IP address with all bits of the network part set to one and all bits of the host part set to zero. It was widely used for the IPv4 protocol but not for IPv6. It is still mentioned here because those bit masks are useful for the route selection process, as described below.

IPv6 host address	`fd87:d87e:eb43:938c:6923:402d:6ca3:1777`
netmask (prefix length = 48)	`ffff:ffff:ffff:0000:0000:0000:0000:0000`
logical AND	`----------------------------------------`
IPv6 network address	`fd87:d87e:eb43:0000:0000:0000:0000:0000`

Figure 3.3: Network address calculation.

Routing is the process of determining to which network interface an IP packet should be forwarded. That is necessary because a host attached to an IP network might have more than just a single network interface. The kernel has to decide where to deliver the packet to, once it arrived (see Chapter 4 for more details on OS interaction). The route selection is based on a routing table. It usually consists of at least three columns: the prefix, the prefix length, and a destination. For now, the destination can be considered to be a physical network interface like an Ethernet interface. The kernel tries to match the destination IP address of a packet to a route of the routing table. This is done with simple logical operations. It takes the destination IP address of the packet and applies a logical AND operation with the prefix length in its netmask equivalent of an entry of the routing table to it, as shown in Figure 3.3. Then it compares the result with the prefix of the same entry of the routing table. If the comparison evaluates to true, the route matches and the

[4] Classless Inter-Domain Routing.

Prefix	Length	Destination
`::1`	128	Loopback
`2001:6f8:3c4:1::`	64	Ethernet 0
`2001:6f8:3c4:17::`	64	Ethernet 1
`fd87:d87e:eb43::`	48	Tunnel 0
`::`	0	`2001:6f8:3c4:1::1`

Figure 3.4: Routing table example.

packet is forwarded to the destination interface of this routing table entry.

A typical routing table is shown in Figure 3.4 on page 15. It contains five entries. The first one is the local loopback route. It is a host route[5] of the address `::1` to the loopback device. Being a special definition, this address always identifies a host itself from a local point of view. The second to fourth routes are routes to physical interfaces. The fifth route is the so-called *default* route and can be seen as a "catch all" route. This means that any packet whose destination is not kept within one of the other routes will always match this one. This is because it has a prefix length of zero, thus the network part of this address does not exist. It only has a host part with a length of 128 bit which actually matches all possible hosts. The default route is not physical but logical because its destination is in turn an IP address, but this is not of real importance here.

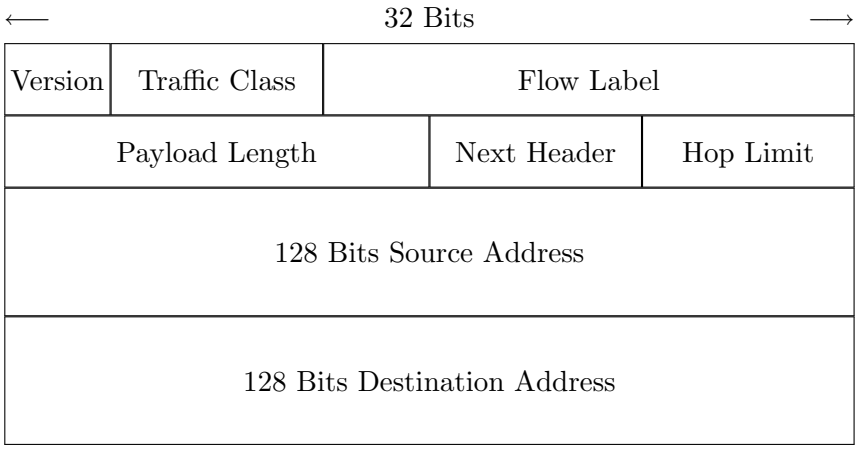


Figure 3.5: The IPv6 packet header.

[5] A host route always is a route with greatest possible prefix length which is 128 for IPv6.

Now I will examine the packet header in more detail. As shown in Figure 3.5 on page 15, the IPv6 packet header has a fixed length of 40 bytes and contains eight fields, as defined in RFC2460 [7]. The *version* field is always set to 6, the *traffic class* and *flow label* are mainly used for quality of service (QoS) which is not of further importance within this context. The *payload length* contains the length of the packet's payload (the data body). The *next header* field is a one-byte value and contains a number which defines the type of data contained within the payload. As an example, a typical value is 58 for the *Internet Control Message Protocol Version 6* (ICMPv6), which usually is defined as the C preprocessor macro IPPROTO_ICMPV6 in the header file netinet/in.h.[6] The *hop limit* field prevents the packets having an endless life. Every layer 3 device decreases the hop limit counter. If the value reaches zero, the packet is dropped.
All bytes of the header are encoded in *network byte order* which is *big endian*. This means that the most significant byte is sent first. This is usually true for all other network protocols, like Ethernet, IPv4, ICMP, TCP, UDP, and so on.

3.2 Ethernet

Ethernet is a protocol of the data-link layer and is most widely used for local area networks. Data units of this layer are called frames and in turn this is true for Ethernet. For Ethernet in particular, several different frame formats are defined. The important one in respect to this topic is called *Ethernet II* [42] or also *DIX-Ethernet*. It consits of a 14 bytes MAC[7] header, the payload (the frame body) and a 4 byte checksum field. The latter usually is checked and stripped off by the network interface hardware, hence, the kernel just receives the header and the payload.

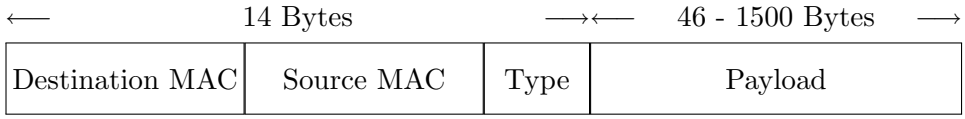


Figure 3.6: Ethernet II frame.

As shown in Figure 3.6, the MAC header contains three fields followed by the payload: a *destination MAC address*, a *source MAC address* and a *type* field. A MAC address has 48

[6] Unfortunately the location of the macro definition may vary from OS to OS. Thus, it is always a good idea to use the GNU autotools [46] for generating portable source code.
[7] Media Access Control.

bits (6 bytes) and is a unique address, defined by the hardware vendor. The uniqueness of the address is only of importance within a single physical network segment. This is because frames (the headers) never traverse layer 3 devices like IP routers. The 16 bits wide type field defines the type of data contained within the frame. For example 0x0800 is set if the payload contains an IPv4 packet and 0x86dd is used if the payload is an IPv6 packet. Figure 3.6 also shows the payload length with a limited range of up to 1500 bytes. This limitation actually depends on the capability of the physical layer below and is called the *maximum transfer unit* (MTU) size. For 10 and 100 MBit Ethernet this is defined to be 1500 bytes [42] but it may differ on other physical layers. But for all of them is true, that the MTU size of the sender and recipient interface must be equal. Thus, it is a good idea not to change this unless you exactly know what you are doing.

There is no routing mechanism in Ethernet. This requires all hosts to "see" all frames of the other ones. Thus, they must be connected together in an appropriate manner. Of course, every host knows its own MAC address. Frames are accepted locally, only if the destination address matches this own address. Other frames are ignored with one exception. If the 8th most significant bit is set, frames are always accepted. Those frames are so-called *broadcast* frames. In that case also those frames are further investigated. The process of distinction what to do with such frames depends one several cases. Mostly this depends on some additional bits of the destination MAC address. This is, if the address has the value 33:33:xx:xx:xx:xx[8] [4] and the type field is set to 0x86DD, it is meant to be an IPv6 multicast destination which is accepted by all hosts capable of the IPv6 protocol. In other words, those packets are forwarded to all hosts on the Ethernet because it is an Ethernet broadcast message but it is only accepted by the IPv6 capable ones. Thus, they can be considered to be multicast frames, even if Ethernet does not know about multicasting.[9]

If two hosts intend to communicate on an Ethernet, they inevitably need to know the MAC address of its opponent. Nowadays, every host is assigned an IP address to. But IP belongs to the network layer which is layer 3. Thus, they additionally need to be able to form layer 2 frames, as a protocol dependency, because layer 3 packets are encapsulated

[8] Similar to IPv6, MAC address notation is done in hexadecimal numbers of six groups of bytes separated by colons. The 'x' means any value.

[9] Different from *broadcasting*, the term *multicasting* referres to an addressing mode which just targets a group of specific hosts but not all available ones.

within layer 2 frames. This was explained at the beginning of Chapter 3. Even if they know the opponents IP address they will not be able to form valid frames because they usually will not know the desired MAC address. Hence, they need a way to determine those MAC addresses. This is what the *neighbor discovery protocol* (NDP) is good for. This is described in the following Section 3.2.1.

3.2.1 The IPv6 Neighbor Discovery Protocol

The neighbor discovery protocol addresses different problems. RFC2461 [29] says the following:

> "IPv6 nodes on the same link use Neighbor Discovery to discover each other's presence, to determine each other's link-layer addresses, to find routers and to maintain reachability information about the paths to active neighbors."

Within this context just the second objective is of importance: determining an opponent's link-layer address. The reason for that was explained in Section 3.2.
The NDP is an extension to ICMPv6 and this in turn is carried within IPv6 (see Section 3.1). Briefly explained, ICMP enhances traffic flow in an IP network by exchanging control information about network states.

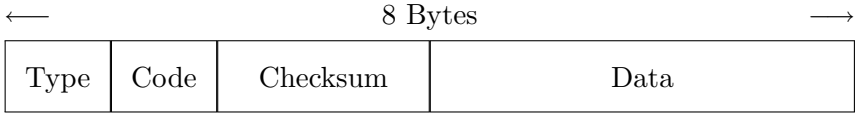


Figure 3.7: ICMPv6 message format.

The ICMPv6 message format is rather simple, as shown in Figure 3.7. It is an 8 byte message with four fields. The *type* defines the type of message and the *code* field may be considered to be an extension to the message type. This means that some messages have the same type but are differentiated further by the code field. The *checksum* allows a recipient to verify if transmission errors occurred and the *data* field may contain additional information dependent on the type. Note, that some messages may have additional data or specific options appended. Thus it is vital to check if the payload length of the IPv6 header is greater than the length of the ICMPv6 header. In that case it has options.

Neighbor discovery means finding the corresponding link layer address[10] to a known IPv6 address. This is done by sending out a *neighbor solicitation* message using an ICMPv6 packet. The source IP address of the IPv6 header is set to the address of the sender, the destination IP address is set to a specific multicast address. This is FF02::1:FFxx:xxxx where 'x' is replaced by the least significant three bytes of the IPv6 address of the desired receiver. This is necessary because the receiver's MAC address is unknown, hence, the destination MAC address of the Ethernet frame must be an Ethernet broadcast message, more specifically an Ethernet broadcast with IPv6 multicast destination, as has been explained in Section 3.2. This is 33:33:xx:xx:xx:xx being 'x' the lowest four bytes of the IPv6 address of the desired receiver. The ICMPv6 message is of type ND_NEIGHBOR_SOLICIT[11] (=135). To the ICMPv6 header always the complete 16 byte IPv6 address of the desired destination is appended. Behind that, usually a single option is appended. It is a *neighbor discovery options header*. The header consists of two single one-byte fields. The first contains the type of option which is ND_OPT_SOURCE_LINKADDR (=1) in this case. The second contains the length of the option in multiples of eight. In this case it is set to 1, which means 8 bytes. The option data behind the option header contains the MAC address of the sender. The MAC address just has 6 bytes. If this option is the last one, the missing two bytes may be omitted. The payload length of the IPv6 header must contain the actual amount of bytes of the packet.

After the message is compiled it is sent to the Ethernet. If the desired receiver exists it will receive it because it is a broadcast message on layer 2 and an appropriate multicast message on layer 3. The recipient identifies itself by checking the IPv6 address contained in the message behind the ICMPv6 header. In turn it will respond with a *neighbor advertisement* message.

The advertisement looks very similar to the solicitation with some differences. First, the message isn't a broadcast/multicast message anymore. The destination MAC address of the response contains the MAC address of the sender of the solicitation as well as the destination IP address is the one of the solicitation sender. The source addresses are those of the advertisement sender. The ICMPv6 type is set to ND_NEIGHBOR_ADVERT (=136) and the additional data field of the ICMPv6 header contains the flag ND_NA_FLAG_SOLICITED (=0x40000000) which means that this message is sent in response to a solicitation. The

[10] On an Ethernet this is a MAC address.

[11] On Linux this is defined in the C header file netinet/icmp6.h.

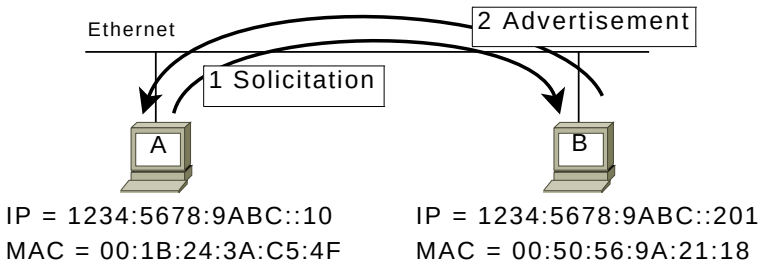


Figure 3.8: Example of a neighbor discovery process.

ICMPv6 header is followed by the IPv6 address of the advertisement. Behind that, the message might be followed by an option. In this case it is the option ND_OPT_TARGET_LINK-ADDR (=2) with the length field set to 1. Behind the option header the MAC address of the advertisement sender follows.

Figure 3.8 on page 20 shows a simple example of a neighbor discovery process. There are two hosts A and B. A would like to communicate with B, but A does not know the MAC address of B, but only B's IPv6 address. Thus it needs to send out a neighbor solicitation first. B will receive it and will respond with a neighbor solicitation message. Appendix A on page 83 contains two example messages and gives in turn a little bit more explanation.

3.2.2 IPv6 Header Checksum

The calculation of the IPv6 header checksum is a little bit different from what we know of IPv4. This is because the IPv6 header does not contain a checksum as shown in Figure 3.5 on page 15. The missing checksum is a performance improvement compared to the older protocol version. This is because the IPv6 as well as the IPv4 header both contain the hop limit field which must be decreased by every layer 3 device when forwarding packets. But decreasing means modifying the header which renders the checksum of IPv4 invalid. Thus, all layer 3 devices are forced to additionally recalculate the header checksum for all version 4 packets.

Obviously, the missing checksum would decrease the detectability of transmission errors. This is circumvented by including parts of the IPv6 header information into the checksum of encapsulated protocols like ICMPv6. Including the full header would destroy performance gain because of the hop limit field, hence, a special *pseudo header* is defined for this purpose in RFC2460 [7]. The pseudo header contains the source and destination IP addresses, the payload length, and the next header field. For checksum calculation, this header is

prepended to the IPv6 packet's payload instead of the original IPv6 header.
The algorithm for checksum calculation is a *16 bit one's complement* and is defined in RFC1071 [2]. It is the same algorithm as it is used for IPv4. An interesting and very important aspect of this algorithm is that it is immune against changing endianess.

3.3 Transport Protocols

Transport protocols are based on layer 3 which means they are located at layer 4. This also means that it makes no difference for a transport protocol if it is based on IPv4 or IPv6. According to Tanenbaum [42], transport protocols have at least two objectives: *multiplexing* and *creating sessions*.
Multiplexing gives the possibility to discriminate between several services on a host. This is necessary, because with an IP address usually a host is identified, but this host may run more than just one service. This distinction is done within layer 4 and is another method of addressing. Most layers have specific methods of addressing and they usually vary from one protocol to another one within the same layer. As explained in the previous sections, addressing on layer 3 is done with IP addresses for the IP protocol and with MAC addresses for the Ethernet on layer 2. Specifically for the *transport control protocol* (TCP) [35] and the *user datagram protocol*[12] (UDP) [33] the addresses are called *port numbers*. These two protocols belong to layer 4. Port numbers are 16 bit values. Thus, they range from 0 to 65535.
Building session means creating virtual circuits.[13] This is different from just sending sole packets because packets in a network may get lost. A session is a communication channel which seems to be reliable, although it is still based on packets. This means that such protocols that *establish* sessions must keep track which packets did arrive and which did not, thereby resending lost packets.
TCP is capable of multiplexing and creating sessions. It has several handshake mechanisms and uses programmable retransmission timers. The session that TCP virtually creates is a stream of bytes. Applications accessing TCP streams[14] usually do not "see" or "know" about packets any more. This is similar to files: bytes are read and written sequentially one after the other. Obviously, TCP sessions have two ends: the writer end (the sender)

[12] The next header value of the IPv6 header is 6 for TCP and 17 for UDP.
[13] Some aspects of virtual circuits will be discussed later in Section 3.4.
[14] Obviously, access to TCP sessions is done from above layer 4 which are the layers 5 to 7.

and the reader end (the receiver).[15] A nature of TCP sessions is that the most atomic data unit visible to the upper layers is one byte. This implies that blocks of bytes transmitted by one atomic operation on one end must not necessarily arrive at the recipient as the same block. TCP may split this up into smaller parts or may concatenate it with other blocks that have been transmitted before. It is absolute vital to not rely on specific application built data units, if they are greater than one byte.

Different from TCP, UDP does not create sessions. It only provides multiplexing. Because of the missing session feature, data units are sent as they arrive from any application. This implies that they arrive on the receiver end in exactly the same block size. There is no such thing like byte streaming as it was explained for TCP before. Because of the missing session creation, there are also no handshaking mechanisms regarding packet delivery. That means that there is no guarantee that packets really will arrive. Packets may get lost in networks. If using UDP, the applications must keep track on data transmission integrity (as far as necessary).

3.3.1 The SOCKS Protocol

TCP establishes an end-to-end communication, usually based on IP. This requires that the IP packets between two communication endpoints can flow bidirectionally without any barrier.[16] Nowadays, it is the normal case that firewalls are applied in many places. Firewalls filter IP packets based on a set of rules. This avoids usual packet flow, hence, TCP sessions might not get established. Obviously, that is the whole purpose of a firewall. But in some cases some users should still have TCP access to some servers which others should not have. This may be accomplished by adding specific rules to the firewall for those users. But this only works if the destinations are static and do not change, but it does not work if the destinations are dynamic which means they are not known in advance. Therefore the SOCKS protocol [24] was designed. SOCKS is a simple protocol for proxying TCP sessions and it has limited support for authentication. A SOCKS service is usually run on a firewall or similar gateway to the Internet.

The client establishes a TCP session to the SOCKS server and sends a SOCKS request message. This message contains the desired remote destination IP address and port number

[15] Actually sessions have four ends, because they are bidirectional. This means that every communication partner can send and receive data.

[16] Of course this is also true for UDP.

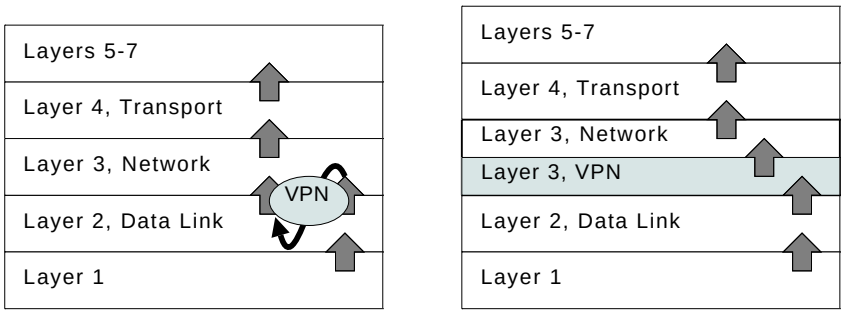
and an identifier. The SOCKS server uses the identifier to authenticate the client. This means it looks up the identifier in a database and checks if the client has permission to access it. Then the SOCKS server in turn opens a TCP session to the destination contained in the SOCKS request message. After the session is established it sends a SOCKS response back to the client. Thereafter all data is forwarded transparently between the client and the destination endpoint by the SOCKS server in both directions.
This protocol was slightly extended in version 4a [23]. It allows that the request message might also contain a hostname instead of an IP address. This is useful if the client is not able to do DNS lookups himself. The SOCKS server will resolve the hostname to an IP address before establishing the external TCP session.

3.4 Virtual Private Networks

One of the most generic definitions is found in [22]:

> "A Virtual Private Network is a network of virtual circuits for carrying private traffic."

I will refine and explain these terms more specifically. Virtual circuits are connections between nodes. Those connections do not exist physically, but virtually. TCP sessions, for example, could be seen as virtual circuits (see also Section 3.3). While browsing the web, a connection between the local computer and the webserver seems to exist but this is just a virtual connection. Of course, those two are not connected physically. But still the connection carries some information, as it is the case for web pages.
What Kosiur ([22]) further says is that those circuits carry *private* traffic. This does not necessarily mean that the traffic is always personal or secret to somebody. This might be the case but it is not a must. In some cases those connections could carry both types of information – secret and non-secret – and very often it is only a matter of definition what is private and what is not.
Many VPNs share a similar concept, which is carrying traffic of a specific type across a network of the same type. In most cases, VPN applications like the Microsoft VPN, Cisco's VPN and OpenVPN carry IP packets. Usually they achieve the same goal, which is to access some kind of "private" network. A prime example would be a company's internal network. It is designed to work for people who have Internet access. Obviously, Internet



(a) Insertion of the VPN layer. (b) The VPN layer enrolled.

Figure 3.9: VPN intermediate layer.

is based on the *Internet Protocol* (IP). Thus follows that those types of VPNs carry IP packets within IP packets. Expressed in a different way, IP packets get encapsulated within IP packets. Figure 3.9 on page 24 shows a simple diagram of how VPNs fit into the OSI layer model.[17]

The Figure highlights three layers. Ethernet is contained within the data-link layer. Usually we carry IP within Ethernet, hence, IP is one layer above in the network layer. On top of IP we have protocols, such as TCP and UDP, and categorize them into layer 4 – the transport layer.

If a VPN is in use, IP is encapsulated into IP and not, for example, TCP into IP as the layer model suggests. Thus, from an architectural view, it inserts a second IP layer. That is what Figures 3.9(a) and 3.9(b) depict. Above layer 2 the VPN layer follows, which might also be IP. On top of that layer an IP layer follows. In Figure 3.9(a) it is shown where the VPN layer is inserted, while in Figure 3.9(b) you see the same but being the VPN layer graphically enrolled.

If a VPN is implemented, there is always some kind of VPN layer. The VPN layer creates the virtual circuits. The difference between various VPNs is where they insert the VPN layer in respect to the OSI model. Figure 3.9 gives just an example of encapsulating IP within IP, but it is also possible to encapsulate TCP within TCP or UDP. Then the VPN layer would be located one layer above.

What one might also associate when thinking on VPNs is encryption. But this is not what makes up a VPN, even if most VPNs provide encryption. The most fundamental issues of a VPN was explained above. But with insertion of a VPN layer, additional data encryption

[17] See the beginning of Chapter 3 for more details on the OSI model.

may simply be included within it. At one end of the VPN communication packets get encapsulated into the VPN layer and encrypted. At the other end data is decrypted and unwrapped from the VPN layer. The VPN participants do not "see" neither the VPN layer nor the encryption.

Because of the higher speed of symmetric algorithms as it was explained in Section 2.1, they usually get used in that application. This implies a need for key exchange in advance. Several different methods are available and often depend on the actual implementation. To anticipate regarding the project documented here, encryption is not used explicitly. This is because part of the VPN layer being used already does encryption and key exchange. This will be described in detail in Section 5.2, respectively.

3.5 Summary

In this Chapter I gave an overview on the OSI layer model and described several protocols of its lower layers. Those are Ethernet and IPv6. Ethernet creates a basic connection between hosts and IPv6 is capable of forming logical network groups. Some protocol details have been explained about the interaction between Ethernet and IPv6. This is the neighbor discovery protocol (NDP). It associates link layer addresses with network layer addresses. A brief overview about transport layer protocols was given, TCP in particular, because it is used as a transport for this project.

The SOCKS protocol was discussed. It is used for proxying TCP sessions.

Based on those protocols, virtual private networks (VPNs) are implemented. Being VPN one of the major aspects under this topic, it was explained thoroughly.

4 Operating System Interaction

This Chapter shall clarify the flow of packets, IP routing, and the required interfaces in respect to the *operating system* (OS). It is not about kernel or OS architectures.
In the following I will often use the term *kernel*, although from a computer science point of view its use is not perfectly correct here. According to Stallings [40], the kernel is the innermost part of an OS. Different architectures exist in respect to the functionality and size of a kernel, and which parts of an OS are actually implemented in the kernel, and which parts surround it. Different from most Unix OSes, Windows for example, uses a *micro kernel architecture*. Most parts like the TCP/IP stack (see below) are located outside this "small" kernel. But this makes no real difference under this topic, hence, I will still use the term kernel beyond this paragraph, consciously that it is not completely true for all OSes.

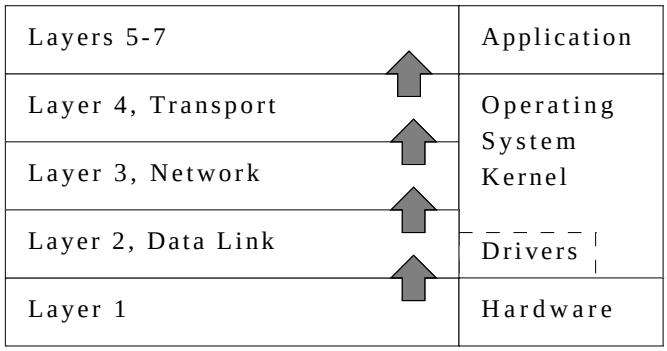


Figure 4.1: The operating system in the OSI model.

To understand which tasks in respect to networks an OS accomplishes, first, the layers in respect to the OSI model (see Chapter 3) into which an OS fits, need to be defined. More specifically, this is about the kernel of an OS. Figure 4.1 gives a brief overview. It is a simplified picture of what is found in [40]. Layer 1 is the physical layer, thus all parts of it are usually implemented in hardware like, for example, a network interface card. Layer

2 may be implemented in hardware as well as in software. Nowadays, parts of it are done in hardware but allow software interaction. This means it is completely programmable or may solely be done in software. Software that deals with hardware in that way is a low-level module from a software architectural point of view. Those modules are usually referred to as *device drivers* or just *drivers* and are always part of the kernel. Layer 3 and 4 which contain the IP protocol, the IPv6 protocol, ICMP(v6), the TCP and UDP protocol (see Chapter 3), and others, are usually also part of any network enabled kernel. The module which implements those protocols is referred to as *TCP/IP stack*.

All layers above 4, these are layers 5 to 7 are implemented in applications. Every software running outside the kernel is referred to as *userland* or *user space* applications. Those running in userland have many restrictions compared to the kernel. For example, they are unable to access the hardware directly. Thus, the kernel offers an interface to userland applications. This interface allows software to access lower level functions of a computer system under the control of the kernel. This usually implies that the kernel deals with misbehavior of applications or rejects any invalid use of this kernel interface.

Typical applications are web browsers, editors, shells, but also web servers or anonymizers, like Tor, OnionCat, or I2P.

Kernels provide several interfaces to userland applications. A traditional way to access the kernel is to use specific function calls. They are called *system calls* and are low level calls, usually based on C functions with stack-passed[1] parameters.

System calls can be grouped together dependent on their functionality. One group of functions is dedicated to *socket* operations. Socket operations access the TCP/IP stack of the kernel. As already mentioned, all layers below 5 are part of the kernel, hence, an application may access a TCP-based byte stream (see Section 3.3) but it will not be able to create IP packets and send it to the network through the hardware. This is because of two reasons: first, userland applications are not allowed to access the hardware directly, which means they are not able to access it, under no circumstances.[2] Second, the socket interface provides only a limited way or even no way to access the network layer, depending on the type of operating system being used. If such access exists as it does on Linux, it bypasses the routing process. Packets arriving are directly passed from a network interface to such

[1] The *stack* is a temporary memory space for programs. It is mainly used for passing function parameters, return values and intermediate operation results.

[2] This is true if we assume that there is no software bug within the kernel code.

a socket, and from the socket to the network interface. On Linux such a socket may be created by using the address family AF_PACKET.

Various other kernel interfaces exist. On Unix-like OSes a widely used way to access kernel structures are *character devices*. Those are special files which virtually reside within the file system, most commonly in the /dev directory. But actually they are interfaces to the kernel and not to data on a hard disk. They can mostly be opened, closed, read, and written, like regular files, even by using the same system calls. Special about character devices is the structure of the content of data that is read or written. This is because they are used to control some kind of kernel feature. One of those devices is the *tunnel device*. It gives access to the layer 3 including the routing process, or even to layer 2. This device is a major building block for VPNs, hence, it is explained in the following Section 4.1 separately.

At this point, it is essential to know that Windows does not have such kind of kernel interface. Access to the kernel is mostly done using system calls. Furthermore, there is no tunnel device available in that kind as explained above. For those reasons it cannot be accessed through a character device. The tunnel device must be implemented as a driver. This is done using the *user-mode driver framework* which appeared in Windows XP. A user-mode driver can then be accessed like a file through Windows' character-device-like file system with regular system calls. This will be explained in Section 4.2.

4.1 The Tunnel Device

A tunnel device gives access to the layer 3 or layer 2 of the OSI model. On Unix-like operating systems the tunnel devices is implemented using a character device. Thus, the userland interface is still above layer 4. But the other end of the device is attached to the kernel's routing process. That is, the device, or the tunnel driver rather, delivers packets directly out of the routing process to the userland, back and forth.

Figure 4.1 on page 26 gave an overview of how an OS fits into the OSI model. Figure 4.2 on page 29 now shows a more detailed view of the kernel internals. At the bottom are shown the network device drivers and the network devices. In this picture, as an example, an Ethernet device which is connected to a network switch and a PPP[3] device connected to a

[3]The *Point-to-Point-Protocol* is oftenly used for low bandwidth network uplinks, like digital subscriber lines (DSL) or mobile connectivity.

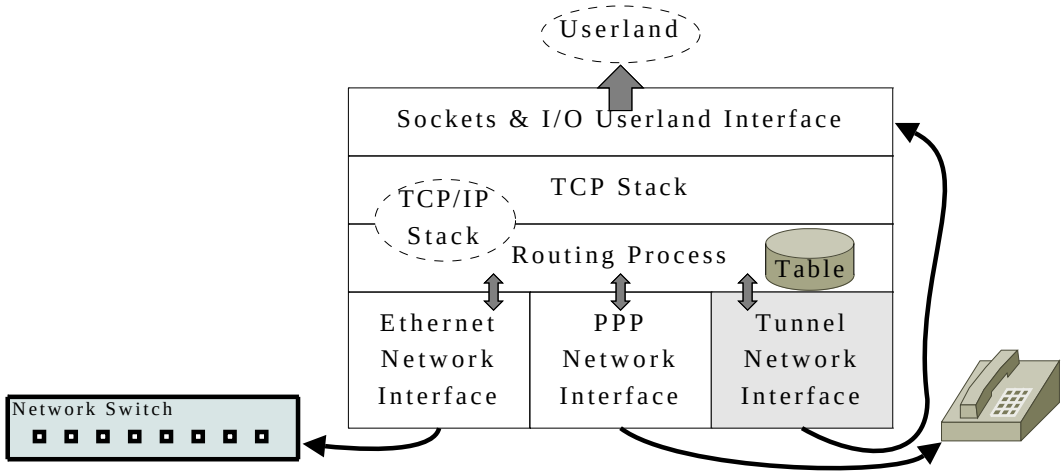


Figure 4.2: Tunnel device integration into the kernel.

modem (the telephone icon). The third gray shaded device shows the tunnel device. The notable fact of this device is, that it is connected back to the kernel's userland interface and not to some physical device.

Those devices are usually operating on layer 2, from a software point of view. All frames received are decoded and the payload is unwrapped. The payload usually carries a layer 3 packet. Dependent on the payload's protocol type the packet is forwarded to the appropriate routing process. This means that every layer 3 protocol has its own routing process, in case the protocol supports routing at all. But this is true for IP and IPv6. The routing process makes decisions where to forward packets based on a routing table as it was explained in Section 3.1. Since the routing process has a top and a bottom end, it may forward packets either back downwards to another (or the same) network device or upwards to the TCP stack. The routing process and the TCP stack form together the TCP/IP stack, as it ways explained in the introduction to the OS Chapter 4. If it is sent to the network device, the layer 2 information is recreated and then it is sent (layer 1). If it is delivered upwards it is unwrapped again because the TCP stack operates on layer 4. The picture just shows the term TCP stack but actually this kernel module handles UDP as well as other layer 4 protocols. This modules handles all layer 4 information and hands over payload information in an appropriate way to the sockets & I/O interface. At this interface, userland applications can receive data from and send data back to.

Figure 4.3 on page 30 shows packet flow examples. These are the two bold arrows. The one shows a packet arriving on the Ethernet and leaving again on the PPP device. This

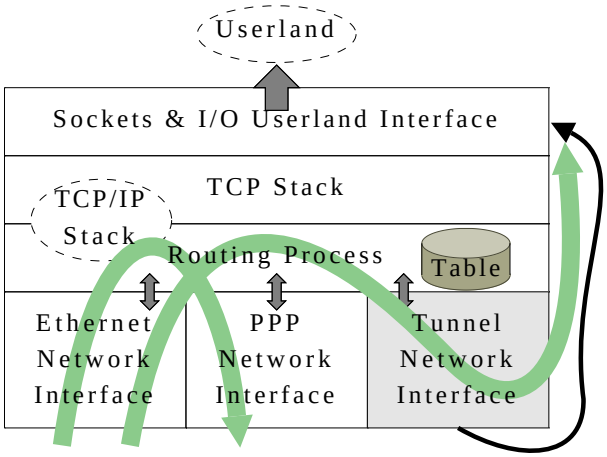


Figure 4.3: Packet flow in respect to the tunnel device.

is due to a routing table based decision of the routing process. The second packet also arrives on the Ethernet device but exits again on the tunnel device. This also happens due to a routing process decision. That is, a packet entering the tunnel device from the top (referring to Figure 4.3) has already been routed. And further, the packet will be delivered to the userland, but not through the TCP stack. As explained before, delivery is done by a character device. As a matter of fact that the packets do not traverse the TCP stack, they arrive at the character device as is, in their raw condition.

This work flow is bidirectional, meaning that a packet delivered to the tunnel device by writing to the character device will be delivered further to the routing process. This implies that, first, the packets must be well-formed because there is no intermediate layer like the TCP stack and, second, the packet will be routed. This in turn means that the routing process makes a decision based on its routing table before the packet is forwarded. It also implies that, if there is no entry for the desired destination, then the packet will be dropped and an ICMP error message is replied.

The bottom end of the tunnel device is connected to the userland, as has been said before. But from a software modularity point of view it acts like, for example, the Ethernet device. On the top end is the routing process of the kernel, on the bottom end may other devices be connected. In case of Ethernet this is a hub, or a switch,[4] or another host directly. Again, connected to those are Ethernet devices from other computer systems. This figure is true for the tunnel device, too. The well-formedness relates to the packet format that

[4]Hubs and switches are Ethernet distributors. They enable several nodes to be connected to a single physical Ethernet segment.

the routing process on one end and the connected devices on the other end of the tunnel device expect.

At least all Unix-like OSes provide a tunnel device which may operate in two different modes:

- *TAP mode* is a real Ethernet emulation device implemented in software. It expects that devices attached to it at the bottom by the character device act like real Ethernet devices. That is, they are requested to send and receive layer 2 frames with all requirements that Ethernet has. This includes, for example, neighbor discovery capability for IPv6 devices. TAP mode can be seen as an Ethernet emulation device implemented in software.

- *TUN mode* is a mode which acts at one layer higher than TAP mode. This is layer 3. This implies that there are no layer 2 protocols like NDP, hence, it is always a point-to-point connection in respect to layer 2. This means that, different from TAP mode, always just a single device may be attached. Layer 2 is completely missing.

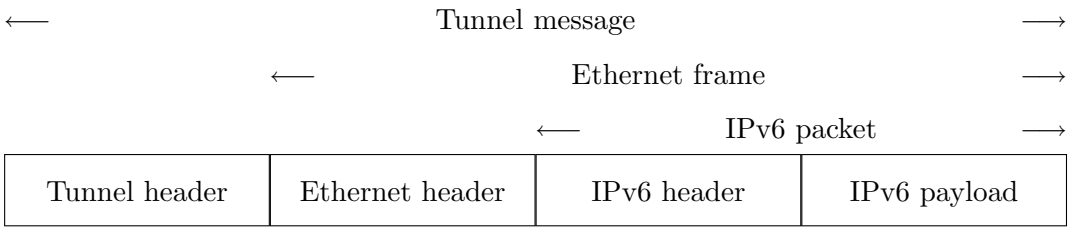


Figure 4.4: Tunnel device message format.

The tunnel device has a specific message format as shown in Figure 4.4. The total message size depends on the MTU size (see Section 3.2) of the tunnel device. To be as much compatible to Ethernet as possible, the MTU size of all[5] tunnel devices defaults to 1500 bytes. This does not include the tunnel header. The tunnel header is a 4 byte integer value, thus, the total tunnel message length is 1504 bytes when using the default MTU size. As already mentioned in Section 3.2, it is not a good idea to change the MTU size because it would break the interoperability. Devices with different MTU sizes will not be able to communicate properly. The Ethernet header is based on the Ethernet II type,

[5]To be more precise, I am talking of all tunnel devices that have been observed. Those include Linux kernels 2.6.18 to 2.6.27, FreeBSD kernels 6.3 to 7.0, OpenBSD 4.3 to 4.5, MacOS X 10.4 and 10.5, SunOS 5.10, and WindowsXP based on the OpenVPN TAP driver 2.1.

```
// fd is an integer containing a valid file desciptor
// of the open tunnel character device
int prm = 1;
if (ioctl(fd, TUNSIFHEAD, &prm) == -1)
    log_msg(LOG_EMERG, "could not ioctl:TUNSIFHEAD: %s",
            strerror(errno)),
    exit(1);
```

Listing 4.1: Activating the tunnel header.

hence, its length is 14 bytes. The IPv6 header has a fixed length of 40 bytes. Thus there are 1446 bytes left at a maximum for the IPv6 payload.[6]

The difference between TUN and TAP mode in respect to the tunnel message format is that the Ethernet header is missing.

The tunnel header is a specific integer value in network byte order (big endian) which may help identifying the payload of the tunnel message. Unfortunately, this differs completely on every operating system that have been investigated. On Linux it defaults to ETHERTYPE_IPV6[7] which is 0x000086DD. On the BSD based OSes (this includes FreeBSD, OpenBSD, and MacOS X) it defaults to the value of the socket's IPv6 address family macro AF_INET6.[8] The integer value of the macro differs even within those OSes. For example on FreeBSD it is 0x0000001C, on OpenBSD it is 0x00000018. Thus, it is strongly recommended to use the literal preprocessor macro rather than one of those integer values, to make the code more portable.

As a further pitfall, the tunnel devices are configurable in several ways and also this differs on most OSes. Some of them offer to activate or deactivate the prepended tunnel header, some others are able to change the behavior of the tunnel header. For writing portable code it might be a good idea to configure all tunnel devices to behave in a similar way. This is, activating the 4 byte tunnel header as explained before. This is done by issuing an I/O control command using the system call ioctl(). On Linux and OpenBSD the tunnel header is on by default, on FreeBSD and MacOS X this is done with the low level command TUNSIFHEAD, as shown in Listing 4.1.

[6] Of course, the tunnel device is capable of receiving IPv4 packets, too. But for a specific reason which is explained later in Section 6, we do not bother with version 4.

[7] It is defined in net/ethernet.h.

[8] It is defined in sys/socket.h.

```
struct ifreq ifr;
memset(&ifr, 0, sizeof(ifr));
// use_tap is set to 1 if TAP mode should be used
ifr.ifr_flags = use_tap ? IFF_TAP : IFF_TUN;
strncpy(ifr.ifr_name, "tun0", IFNAMSIZ);
if (ioctl(fd, TUNSETIFF, (void *) &ifr) == -1)
    log_msg(LOG_EMERG, "could not set TUNSETIFF: %s", strerror(errno)),
    exit(1);
```

Listing 4.2: Setting TUN/TAP mode on Linux.

The character device is opened with a simple call to open(). The location of the device is OS dependent, too. On the BSD based OSes it is usually located at /dev/tun$x$ for opening a device in TUN mode and /dev/tap$x$ for opening it in TAP mode, where x is an integer number starting with 0. The device is opened exclusively which means it can be opened only once. On Linux it is located at /dev/net/tun and it is a *clone device*. This means it can be opened more than once and enumerates the actual network device automatically. It results in being every device opened exclusively, too. In the latter case there is no such thing like a TAP character device, hence, configuring TUN or TAP mode must be done by issuing the I/O control low level command TUNSETIFF. This introduces a little bit more work as shown in Listing 4.2.

Once the device is opened and set up correctly, an IPv6 address may be assigned to it and configured as being "up". This might be done using the ifconfig[9] system command on Unix, or netsh on Windows. Once an IP address is assigned, the kernel will add at least one entry into its routing table, but usually two.

The first one is a route to the whole prefix with the tunnel device being its destination. Thus, all packets entering the routing process, having an IP destination address which is included in the new route, will be delivered to the tunnel device. This was explained earlier in this Chapter at Figure 4.3 on page 30. Conversely, packets sent to the tunnel device from userland through the character device are forwarded to the routing process before they are delivered further. Again, this means that the routing table is applied before. If no routing entry exists for the desired IP destination, packets are dropped and replied with and ICMP error message.

[9] Look at the appropriate man page for the correct syntax. Some examples are given in Chapter 8.

The second new routing entry will be a route to the local *loopback device* for the newly assigned IP address. This is done for two reasons. First, it avoids that packets being designated for the host itself are delivered to the tunnel device. Locally assigned addresses are always designated for the host itself, hence, those packets must be kept locally and not sent out again. Second, because of the first reason, the routing process needs some way to determine which packets are destined for the host itself. Again, this is done with a route for the desired IP address to the local loopback device.
The loopback device is very similar to the tunnel device. Its bottom end (referring to Figure 4.2 on page 29) is connected not physically but OS internally, namely to the TCP/IP stack.[10] In other words, it ends within the kernel, and not within the userland, as it is true for the tunnel device.

All OSes make sure that a tunnel message is read at once, and only once, and that the message is read completely. Thus, the read buffer provided by the application shall be large enough. At the default setting this is 1504 bytes. If the buffer is not large enough the OS will try delivery it as a whole again at the next read. Sequential partial reads are therefore impossible. This applies also for writes. Tunnel messages must be written at once at a whole, including the correct tunnel header. Thus, the tunnel device may be considered as a *packet stream*. Of course, the written message must be well-formed. If this is not the case, the character device would still accept the message but will drop it later as a consequence.
Using TAP mode means that the data-link layer exists. This implies that designating messages to another node requires not just the knowledge of the IP destination address but also its link-layer address. This is a MAC address for Ethernet (see Section 3.2). It is also true for the TAP device because it is an Ethernet emulation device.
MAC addresses are usually not known in advance, thus, neighbor discovery is used as it was explained in Section 3.2.1. This implies that, if a software device is connected using a tunnel device in TAP mode, it then must provide neighbor discovery capability. This is true in case the software device should act as an end node like, for example, a computer connected via Ethernet. It is not true for intermediate devices like hubs or switches, even if implemented in software.

[10] For simplicity, the loopback device is completely missing in Figure 4.2. As has been explained, it is located between the routing process and the TCP/IP stack.

4.2 A Tunnel Device for Windows

A userland (user-mode) driver for a tunnel device exists. It was implemented by the OpenVPN project [32] and works on Windows XP and the newer releases also on Windows Vista. It provides an interface to the routing process on one end and to the userland on the other end. This is very similar to that what have been explained in the previous Section 4.1. Thus, everything applies as said before with a few differences.
Windows uses the *registry* to store all kinds of system settings. Thus, also device drivers are registered there. The access to the driver from userland is done using the device file system of Windows. Similar to Unix, Windows maintains a naming hierarchy where all kinds of files (pipes, devices,...) can be found, starting at the root '\\'. Windows exposes special files, like devices, at the path '\\.\Global\<*deviceID*>'.[11]
Once the OpenVPN TAP driver is installed, it can be accessed through the Windows API. This is done by the following steps.

1. Locate the driver in the registry.
2. Retrieve the device ID.
3. Open it through the device namespace.
4. Assign an IP address.
5. Add a route to the routing table.
6. Read/write data.

Locating the driver in the registry is done by opening the appropriate registry directory and iterating through it until the desired entry is found. All network drivers are stored in the HKEY_LOCAL_MACHINE directory, in a special directory named SYSTEM\CurrentControlSet\Control\Class\{4D36E972-E325-11CE-BFC1-08002BE10318}. All entries contain a string variable called ComponentId. The string of the TAP driver should have the value "tap0901", or "tap0801" for the older releases of the driver. If such an entry is found, the device ID can be retrieved. It is contained in the string variable NetCfgInstanceId.

[11] They cannot be browsed with the Windows Explorer because that is only capable of browsing files of the *files namespace*, which is located below the global root. Russinovich provides the tool WinObj [38] for browsing the whole namespace.

Opening a registry entry is done with the system call RegOpenKeyEx(), iterating through the entries is done with RegEnumKey(), and querying specific entries is done with RegQueryValueEx(). All Windows system calls, and more, is found in Microsoft's very comprehensive online developer network at [28].

Once the device ID is retrieved, the device can exclusively be opened with a call to CreateFile(). As already mentioned, the path to the device is "\\.\Global\<*device ID*>". Once device is opened, it can be written and read using the system calls ReadFile() and WriteFile() as shown in Listing B.1 on page 85. The Listing has been moved into Appendix B because of its length.

Reading and writing is different from what one might expect. It behaves like non-blocking I/O (see Section 4.3.1 about non-blocking behavior on Unix-like OSes). Non-blocking means that calls return immediately and do not wait until the request is completed. To wait for completion, Windows provides the call WaitForSingleObject(). After it finished its request, status shall be retrieved with GetOverlappedResult().

This is what Listing B.1 shows. First, the read request is posted with ReadFile(). Second, the function waits in a loop for the request being finished. In that case, a time period is supplied additionally. This protects the program to block forever in case something goes wrong. After successfull completion, the result is retrieved with GetOverlappedResult(). As also shown in the sample Listing, additional error checking should be done very careful.[12]

There are two more differences compared to the Unix tunnel devices. First, the message format of the tunnel device has no tunnel header as shown in Figure 4.4 on page 31. Second, the tunnel device can only be operated in TAP mode.[13] It always operates like a virtual Ethernet device. Thus, for a software attaching to it, it may be required to implement also layer 2 protocol requirements like, for example, the NDP protocol, as has been described in Section 3.2.1.

4.3 Doing Several Tasks in Parallel

Userland processes are usually dedicated to a single CPU. If not designed in a specific way, every process (application) is always only capable of doing one step after the other. This is sufficient for many applications but especially when it comes to network programming,

[12] The original code has some additional error checking. It is stripped off in this example to make it more readable.

[13] Eventually, that is the reason why it is called *TAP driver*.

it may be required to do several things in parallel. This is easily explained. A web server, for example, should serve several clients at the same time, but not just one after the other. Simply adding CPUs into a server does not circumvent the problem. There are several models available for parallelizing tasks.

- *Multi-processing* means running several processes in parallel. This is what every OS usually does by default. Actually, always only one process is executed at a given time, but the OS *schedules* one after the other for a small amount of time. In other words, they are executed in dedicated time slices. This has the effect that it looks like if they are executed in parallel, even if they are still execute one after the other. But it still gains some execution time, because often a process has to wait for some events. As an example, a process may wait for new data arriving on the network card. During this time another process may complete some instructions.

  Processes are dedicated to the CPU by the OS using *scheduling algorithms* to be as efficient as possible. Several algorithms are available as described in [40]. Multi-processing is expensive in respect to memory and CPU time and it requires some communication methods. Obviously, because usually tasks are interdependent on each other, in particular if they should complete a specific job together in parallel. This communication methods are usually summarized by the term *inter-process communication* (IPC) [40]. These methods are expensive because every process has its own dedicated memory space and switching from one to another (*context switching*) is time consuming.

  If a computer system commands several CPUs, processes might really run in parallel. Multi-processing is the traditional method of parallelizing.

- *Multi-threading* allows execution of several instruction streams within a process in parallel. This has several benefits compared to multi-processing. It is much cheaper in respect to memory and execution time and it may utilize more simple intercommunication methods like shared variables. It does not involve such expensive context switches, even if the OS still has to switch from one thread to another. But it has not to change the whole process context. Multi-threading has a downside, too. It requires a program to be written very thoroughly, particularly in respect to shared resources like variables. This is because the OS does not sanitize access to those. This is different to multi-processing because, if using IPC in a proper manner, the

OS will protect shared resources. Of course, IPC may still be used, but often also this is expensive and it may reduce performance gain of multi-threading.

If a computer system commands several CPUs, also multi-threading enables the use of all of them in parallel at the same time.

- Specifically for network programming or I/O intensive processes in general, it is possible to *select on non-blocking I/O channels* that are ready for reading or writing and dedicate short instruction streams to those being ready, one after the other. This creates virtual parallelism, even if those instruction streams are executed sequentially in reality. This has the benefit that it does not involve expensive process or thread switches, it does not need any kind of IPC and, because it has no real parallelism, it does not involve the problem of accessing shared resources at the same time. The downside is that it does not utilize more than one CPU, even if they are available, and it may involve complex conditional branching. Selecting on non-blocking I/O is explained below in Section 4.3.1.

A good choice, particularly for this application, is to use multi-threading and selecting non-blocking I/O channels together. This is a good tradeoff of enabling a program to utilize several CPUs and not spending to much time and memory for context switches. Multi-threading should be done by the standardized POSIX[14] threads interface (Pthreads). A very good literature to Pthreads programming is found in [30]. The Pthreads interface provides function calls for thread creation and deletion, and it is capable of dealing with access to shared resources. Generally, this is referred to as *thread locking*. It includes conditional and unconditional *mutexes*.

4.3.1 Selecting on Non-blocking I/O

Selecting on non-blocking I/O needs some additional explanation. POSIX-like OSes provide a wide range of I/O channels. These are communication channels to or from which data can be written or read. Traditional I/Os are files, pipes, and sockets. Sockets have been mentioned already at the beginning of this Chapter. Once those I/O channels are opened, they can all be read or written with the same system calls. Obviously, opening an I/O channel is different and depends on its type. To open a file, for example, a file

[14] Portable operating system interface for Unix.

name is needed. Opening a socket probably requires a port number, an IP address and a protocol type, like UDP or TCP. In case of success, the opening function always returns a *file descriptor* which is an integer number identifying this specific I/O channel.

After opening, all of them can be read or written with the same set of system calls. The most basic ones are recv() and send().[15] Some more are available but it will not be elaborated on them as it is not necessary here. All system calls are documented very well. They are found in the developers manual pages, in Lewine's comprehensive book on writing portable code [25], and online at the Open Group site [45].

I/O channels may be *blocking* or *non-blocking*. The default setting for most channels is blocking. This means, that system calls are waiting as long as the operation is finished while manipulating the underlying I/O. For example, writing (with send()) data to a socket may take some time.[16] On the other hand, reading some bytes from a socket might last even much longer. This is because the receiver does not know when the next bytes will arrive.

This leads to the case that blocking reads or writes block the whole process at specific locations within the code. This implies two things. First, as mentioned before in Section 4.3, processes usually can always execute just one instruction at a specific time. Thus, a process cannot read and/or write two or more I/O channels at the same time. Second, if a process is in a state waiting for something, this suggests to do something else in the meantime.

To avoid that a process may be stuck somewhere because of an I/O operation blocking too long (probably infinite), the I/O channel's behavior may be changed to non-blocking. This is done by manipulating it using its file descriptor with the system call fcntl() as shown in Listing 4.3 on page 40. First the current flags are retrieved with the low level command F_GETFL. Then the non-blocking flag is set by logical or'ing them with the flag O_NONBLOCK. Then the flags are set again with the command F_SETFL. Prototypes and flags are usually defined in the C header file fcntl.h. Another method is opening the I/O channel in non-blocking mode, a priori. But not all opening functions support this.

Operations on non-blocking I/O behave different. This has two consequences. First, calls will always immediately return, independently if they were successful or not. Second,

[15] The reader might be more familiar with the function calls read() and write(). They are very similar but send()/recv() allow to additionally pass some flags to the underlying I/O channel.

[16] Even if this is just 1 ms it is "some" time compared to the instruction execution time of a modern CPU.

```
// fd is the file descriptor of an open I/O channel.
void set_nonblock(int fd)
{
    long flags;

    if ((flags = fcntl(fd, F_GETFL, 0)) == -1)
    {
        log_msg(LOG_ERR, "could not get socket flags for %d: \"%s\"",
                fd, strerror(errno));
        flags = 0;
    }
    if ((fcntl(fd, F_SETFL, flags | O_NONBLOCK)) == -1)
        log_msg(LOG_ERR, "could not set O_NONBLOCK for %d: \"%s\"",
                fd, strerror(errno));
}
```

Listing 4.3: Changing I/O to non-blocking.

detecting errors is different. On blocking I/O errors are directly returned as soon as they occur. This is not always possible with non-blocking I/O since function calls do return immediately, but errors may occur later. Obviously in that case, they can impossibly be returned before. Thus, errors must carefully be detected later.

Beside those two issues also the semantics of a program should be changed. This is because using this technique, no point of waiting exists. This is important because usually I/O channels are much slower than CPUs may execute programs. With non-block I/O a program might read (or write) from several I/O channels virtually at the same time. Virtually, because they are still read (or written) one after the other.

But what to do if, for some time, no bytes arrive on a socket? A very unclean solution is to sleep for some time and then try reading (or writing) from all channels again. This is unclean because mainly for two reasons. First, if bytes arrive or a channel gets ready for writing before this time period elapses, the program will not be able to read immediately. The sleep blocks it until the time period elapses. This means it may loose some time. Loosing time means data transfer, or whatever, is getting slower. Second, the program has to try reading (or writing) all I/Os. This would include I/Os actually not being ready. Again, this wastes valuable time.

```
int so_err;
socklen_t err_len = sizeof(so_err);
// fd is a file descriptor to test for errors
if (getsockopt(fd, SOL_SOCKET, SO_ERROR, &so_err, &err_len) == -1)
    log_msg(LOG_ERR, "getsockopt failed: \"%s\"",
        strerror(errno)),
    exit(1);
if (so_err)
    log_msg(LOG_ERR, "getsockopt returned %d (\"%s\")",
        so_err, strerror(so_err)),
    exit(1);
// everything ok, continue...
```

Listing 4.4: Testing socket file descriptor for errors..

To solve this problem nicely, POSIX defines the system call select().[17] The function call takes some arguments. The most important ones are the read and write sets. These are sets of file descriptors which the select() command should wait for to get ready. Select() will block until at least one of the file descriptors contained in the sets gets ready.

To call select(), the sets have to be set up correctly. A set is of type fd_set. It should be cleared initially with a call to FD_ZERO().[18] File descriptors may then be added with FD_SET(). Detailed descriptions are found in the manual pages or in [45, 25]. Prototypes and macros are found in the C header file sys/select.h. Select() returns the number of file descriptors that are ready. Those being ready can be tested for their readiness with calls to FD_ISSET().

Beside reading or writing, select() can also be used to wait for other events. Particularly in respect to network and socket programming, these events are creation of socket connections in both directions. This means either connecting to a remote socket with connect() or accepting an incoming connection with accept().

For outgoing connections in particular, it is important to test the file descriptor for errors. Listing 4.4 shows how to test for socket errors with getsockopt(). It can be used for various socket manipulations. To test for errors it shall be called with option SOL_SOCKET and SO_ERROR. Prototypes are defined in sys/socket.h.

[17] POSIX defines also the call pselect(). Its main difference to select() is that it allows to influence signal behavior while the command blocks. But this is beyond the scope of this document.

[18] According to the POSIX standard, it might also be implemented as a macro.

4.4 Summary

In this Chapter a brief overview on the interaction of operating systems and their attachment to the network was given. The relation between an OS and the OSI layer model was shown. It is important to understand the process of routing and packet forwarding, which has been explained thoroughly. Using that picture of an OS, it can clearly be defined were userland processes and where the kernel is located in respect to the model.
The tunnel device is a kernel driver and thus located at a very low layer. The flow of packets through the tunnel device and the interaction with the kernel's routing process has been discussed – for Unix-like OSes and for Windows.
In the Sections behind, the problem of executing several instruction streams in parallel within a process has been pointed out. One pseudo-parallelism is created by using `select()` and non-blocking I/O. This was briefly explained.

5 Tor – An Anonymizing network

Tor [43] is an anonymizing network. It is based on the Internet and consists of several nodes capable of forwarding TCP/IP sessions through it. Thereby it hides the origin at the destination endpoint. The location of a user, that is his IP address, is hidden at the remote site. For example, the IP address of a user accessing a web service will not be disclosed in the server's log files. Instead, the IP address of a random Tor exit node will appear. An exit node is a Tor node at which a TCP/IP session leaves the Tor network.
This is a great feature because it improves a person's privacy, specifically if somebody resides under aggravating circumstances. Unfortunately, Tor is not only used in the "right manner". Someone could also missuse it, and if done right, nobody will every discover who did wrong, because even for the Tor network itself it is impossible to find out the originating IP – deliberately it is a design feature. Always only the IP addresses of exit nodes appear in the public and depending on the law of the country where an exit node resides in, it could lead to a law-enforced service shutdown or even something worse. That is why people are usually not willing to run exit nodes.[1]

5.1 Introduction to Hidden Services

The counterpart of a user willing to hide his location is a service which should be hidden. This is a service which is known to exist and it is known how to access it but it is unknown where it is. This includes that also the service's IP address is unknown and the site maintainer is unknown, as long as he does not reveal himself. Basically, it could be any type of service, a web service as a prime example.
In traditional Internet this is more or less impossible because an IP address can always be traced back to an Internet provider and finally to a user or a company. Hidden services

[1] This is not the only reason but probably the most important one.

[11] are services which exist only within the Tor network. Different from what we know from Internet, they are not identified by an IP address but by an onion-URL. The Tor network is able to find the right path to it, but neither the user nor the Tor network can detect the IP address.[2] Details on onion-URLs are given later in Section 5.2.

Beside location hiding there is a second great benefit: connections to hidden services do not leave the Tor network. No single exit node is needed. This is perfect because, as already mentioned, exit nodes are rare and because of that they are permanently overloaded with traffic. This results in a high latency.

Another benefit is that Tor guarantees end-to-end encryption from the client to the hidden service which is not true for connections to the Internet, even when using Tor. Unfortunately this is quite often misunderstood. According to the Tor project page [43], many users believe that everything gets encrypted just because they use Tor.

This is why the use of hidden services is really interesting. Providing them increases the privacy of users and service providers.

Unfortunately, these onion-URLs look like random numbers and characters – and in fact they are more or less random – which makes them really hard to remember, even harder than IP addresses, because they have 16 digits.

But who really needs to remember IP addresses? Nowadays, everybody uses hostnames, www.cypherpunk.at as an example. There is the *domain name system* (DNS) [27] which resolves names to IP addresses. In traditional Internet, name service is one of the most important ones. Nearly every user and every service uses names instead of IP addresses while using the network. The introduction of DNS in the 80s – a distributed name resolution service – made the Internet more usable and opened it to a wider community.

But within Tor there is currently no resolving mechanism available for translation of names to onion-URLs. Traditional DNS cannot be used that easy because it is IP-based[3] (specifically the Internet class IN). Hidden services are onion-URL based. They cannot be simply exchanged with IP addresses.

From the Tor network's point of view those URLs are already names. Theoretically, an approach could be to use canonical names (CNAME) pointing to onion-URLs but this would break authentication. Unlike IP addresses, onion-URLs provide authentication. This means, by using the onion-URL, a user can verify that a service really is the right

[2] Of course only if the service is configured correctly.

[3] That's not a matter of design but a matter of the real (IP) world.

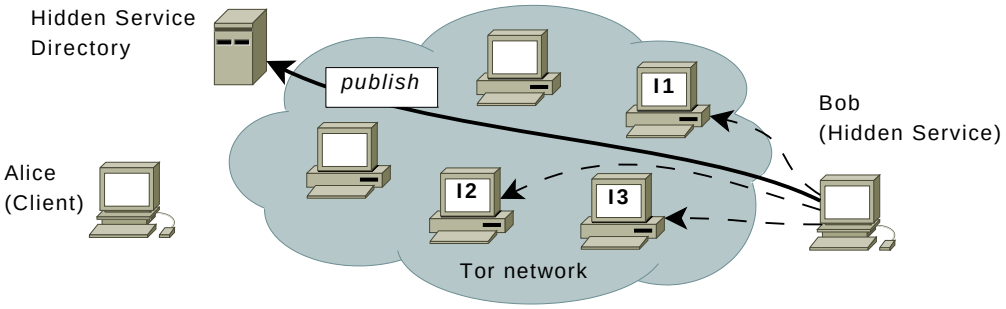


Figure 5.1: Hidden service initialization.

hidden service and not any other one that pretends to be the right service. How this authentication works is explained later in Section 5.2.

DNS basically does not interact with services that are associated with names. That is, it cannot provide authentication as it is used for Tor and the security of users and services. Even if someone deals with those onion-URLs, it is still not easy to use hidden services because the interface between an application and Tor is SOCKS. SOCKS has been explained in Section 3.3.1. From a software modularity point of view it is a good idea to use SOCKS because it is a well standardized interface and many applications support it. But many do not. And every application that supports it needs user interaction for setting up the right configuration for SOCKS. A user should be able to use hidden services without any differences to regular Internet services.

5.2 Hidden Service Internals

As already mentioned, the Tor network consists of several nodes. All nodes have their own identifiers. They are listed in central directories – the *Tor directory* – together with their IP address and some additional information. Hidden services are TCP session endpoints (TCP server sockets) on a Tor node. The goal is to connect to this TCP server socket without the possibility to put the hidden service with the node that it resides on into context.

Therefore every hidden service generates its own asymmetric key pair.[4] At the current versions of Tor this is a 1024 bit RSA[5] key. The public key PK is hashed with the SHA-1

[4]Public key algorithms have been discussed in Section 2.1.

[5]RSA [39] is a well known public key algorithm. It can be used for encryption and authentication.

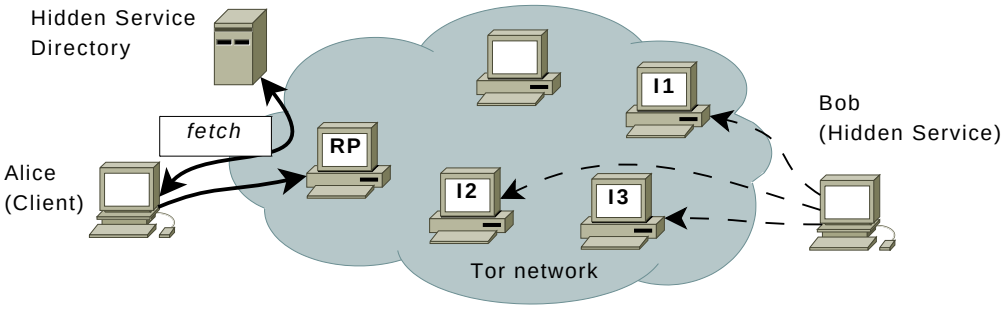


Figure 5.2: Creating a connection to a hidden services.

cryptographic hash function.[6] It results in a 160 bit value $H = SHA(PK)$. Then it takes the most significant 80 bits. This is the *permanent ID*. Furthermore, the hidden service randomly selects three Tor nodes and builds virtual circuits to them, as shown in Figure 5.1 on page 45. Those are the *introduction points* (I1, I2, I3). Finally, it publishes its permanent ID together with the identifiers of the introduction points, the public key, and a digital signature to the *hidden service directory*.

This set of information is called *hidden service descriptor*. As part of the nature of the Tor network, the introduction points do not know who the originator of the circuit is. The permanent ID is then base32 encoded as described in RFC3548 [1] and the string ".onion" is appended.

The RSA private key contains some randomly chosen prime numbers.[7] The public key is derived from those numbers, hence, also the public key is random. The hash function deterministically calculates a value with a fixed width of 160 bit. Obviously, because the input is a random number, the result is reflected in a random item of the set of all numbers of 160 bit length. Cutting of 80 bits does not change anything of its randomness. It just reduces the size of the set. Base32 encoding can be considered as a transposition into another number system with a base of 32. This also does not change randomness. A real example for an onion-URL is "ejbv7bc3cj53lo6c.onion". Its permanent ID in hexadecimal representation is 0x22435f845b127bb5bbc2.

Bob, the hidden services provider, tells Alice about the onion-URL of his hidden service. This happens out-of-band in respect to the Tor network. This could be in a personal dialog, through a weblog, a newspaper, or whatever.

[6] Hash functions have been discussed in Section 2.2.

[7] See [39] for protocol details.

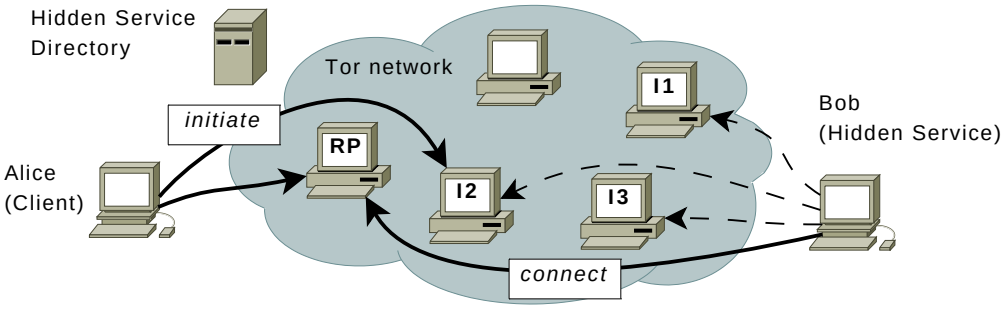


Figure 5.3: Finalization of the connection setup to a hidden services.

Using this information, Alice opens a SOCKS4a connection (see Section 3.3.1) to her Tor proxy and requests a connection to the desired onion-URL. As mentioned at the beginning of Chapter 5, SOCKS is the client interface of Tor. As shown in Figure 5.2 on page 46, Alice' Tor proxy fetches the hidden service descriptor out of the hidden service directory. The Tor proxy can verify that the onion-URL and the hidden service descriptor belong together, and that the descriptor is unmodified. Therefore it verifies the digital signature of the service descriptor using the public key which in turn is contained in the descriptor. Then it computes the hash value of the public key as described above and compares it to the onion-URL. If they are equal everything is ok. Next, it randomly chooses another Tor node being the *rendezvous point* (RP). The rendezvous point cannot detect the identity of Alice's proxy because of the nature of Tor's virtual circuits.

Now the finalization of the connection setup to the hidden services starts. The steps are shown in Figure 5.3. Alice first *initiates* it by connecting to one of the hidden service's introduction points. There she posts a message to the hidden service. The message is encrypted with the public key of the hidden service. It contains the identifier of the chosen rendezvous point and a random one-time secret. This is a secret piece of data used only once. The introduction point delivers the message to the hidden service through the virtual circuit.

The hidden service in turn decrypts the message, *connects* with a virtual circuit to the rendezvous point, and posts a message together with the one-time secret to Alice. The rendezvous point delivers the message and connects the two circuits together. Alice and Bob can now securely exchange data but both do not know about the identity of each other. No other element of the network has a knowledge about that, too.

The connection is end-to-end encrypted because the initial message from Alice to Bob

through the introduction point and the response from Bob to Alice through the rendezvous point are used to complete a Diffie-Hellman key exchange.[8] This key is then used as a symmetric key for Alice and Bob for data encryption.

5.3 Summary

This Chapter briefly introduced the Tor network. It is designed to provide anonymity in respect to the underlying method of addressing. These are IP addresses.
Beside that, it provides hidden services. That is, hiding services within the Tor network. Hidden services are addressed using onion-URLs. The nature of hidden services and onion-URLs has been discussed in detail.

[8] See Section 2.1 and [39] for more information about Diffie-Hellman.

Part II

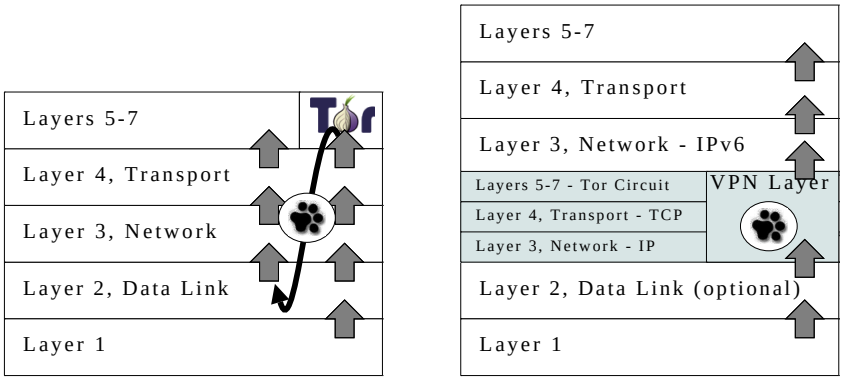
Building an Anonymous VPN

6 Creation of a VPN Layer

Building an anonymous VPN basically addresses two problems. First, anonymity and second, building a VPN. Creating anonymity is not an easy task but is done well by several systems like Tor, as has been explained in the Section 5. Reinventing the wheel is not an option, hence, we rely on the anonymity that such systems provide. This is Tor in particular but it is not limited to.
Based on the theoretical work elaborated within this document, a project evolved. It is a connector using a kernel-to-userland interface on one end and Tor with its SOCKS interface on the other end. First trials were made with the software called socat [37]. It implements the features that are necessary for a trial in a manual setup. Socat is based on the ideas if netcat.
The project described here is in parts similar to socat but it is designed to work specifically with Tor. The routing mechanism of Tor is called *onion routing* because messages are repeatedly encrypted like the peelings of an onion. Thus, this project got the name OnionCat. In the following, this term will be used to refer to that project.

As has been explained in Section 3.4, VPNs consist of two fundamental parts. The virtual circuits and the traffic they carry. Both can be fitted into the OSI model and both may not be of the same layer.
OnionCat does not create a completely new type of virtual circuit. It uses circuits which are created by anonymizing networks upon request, like Tor does.
Tor's virtual circuits, which are relevant for OnionCat, exist only within the Tor network and connect two Tor nodes. As it is the case for most virtual circuits, one end initiates the connection and the other end accepts it. The latter usually is referred to as *server*. Again, within Tor nomenclature this server is called *hidden service* as explained in 5.1. The circuits are based on TCP. As such they are above layer 4 in respect to the OSI model.

Part of the nature of anonymizing networks calls for the capability of two nodes (connected by a virtual circuit) to open up communication channels, but not know who or where the other node is. After circuit setup Tor does not care about data carried within it. It just manages that bytes piped into it at one end drop out at the other end and vice versa.
For all virtual circuits, addressing is required to designate a connection to a specific server. For TCP/IP sessions, addressing is achieved by an IP address and a port number (see Sections 3.1 and 3.3). Tor uses onion-URLs to address a specific hidden service (see Sections 5.1 and 5.2).



(a) OnionCat layer insertion.

(b) OnionCat's VPN layer enrolled.

Figure 6.1: OnionCat in layer model.

OnionCat requests Tor to build such virtual circuits and sends raw IP data across. In this application the virtual circuits carry IP data as it is true for most VPNs. Figure 6.1(a) shows how OnionCat fits into the architecture of network protocols as defined by the OSI model. In the upper right corner it shows Tor's virtual hidden service circuits. They are based on TCP, hence, they are located above transport layer within the model. OnionCat (the cat's paw) inserts the VPN layer, but it is not just an insertion as it was shown in example Figure 3.9 on page 24. OnionCat actually makes a bridge from above the transport layer down to the network layer. If this pictures is enrolled it looks like in Figure 6.1(b).
OnionCat creates a VPN layer (gray shaded area) on top of layer 2. The VPN layer is based on Tor circuits, hence, it uses several layers. These are layers 3, 4, and those above 4. The VPN layer carries IPv6 packets. Thus it provides a layer 3 interface on the top end. As it is true for all VPNs, the users (clients) of the VPN do not "see" the VPN layer. This means it is completely transparent.

What Figure 6.1(b) does not show is, that Tor circuits themselves are a VPN layer in respect to the Tor network. It provides a TCP interface (through SOCKS) and is based on TCP itself. Thus, they gray shaded layer 4 box could be further subdivided.
The downside of this high stack of layers is that it introduces much overhead. Basically every inserted layer introduces overhead but, obviously, more layers introduce more overhead. In this case the VPN layer overhead is at least the IPv4 header of 20 bytes, the TCP header of at least 20 bytes, and the cell overhead of Tor's circuits. According to [11] this is at least 3 bytes. Tor's circuits are based on *transport layer security* (TLS) [9]. TLS overhead is hardly determined as it is a fairly complex protocol but it may be assumed with at least 5 bytes. This are at least 48 bytes at a total.

6.1 OnionCat Addressing

A typical configuration for most kinds of VPNs is that they are setup in a static way. An example of this would be how their virtual circuits are addressed. It is common for organizations to run a centralized VPN entrance point to which all VPN participants connect. This setup is easy and usually matches all requirements for such a private VPN. But it is not suitable for an open anonymous network for several reasons.

1. The person or organization that runs the entrance point probably will not stay anonymous. Even if they never appear in the public, such an entrance point might be revealed due to the fact that it is a traffic sink.

2. A centralized service is always a single point of failure.

3. The service provider might enjoy unlimited trust of its users which obviously would never be the case in today's world.

4. That kind of service could attract certain interest of various organizations like intelligence services.

Hence, the approach is to distribute it. To connect a Tor client to a hidden service, an example being establishing a virtual circuit within Tor, it is required to use Tor's addressing method of choice for hidden services. As has been explained in Section 5.1, Tor uses onion-URLs which are 80 bit long addresses. If we assume that every client runs his own hidden service, then all of them also get a unique hidden service address – an onion-URL. This

leads to the interdependency that every client can connect to every other client since every client now also is a uniquely identifiable server.[1]

The difficulty now arises from layer discrepancy. OnionCat lies between Tor on one end and the operating system on the other end. In respect to the layer model (see Figure 6.1) Tor (the hidden service) operates above layer 4. The other end of the VPN layer which OnionCat creates is at layer 3, which is the IP layer. Every layer has its own addressing method. Hidden services use the 80 bit long onion-URL and the IP layer obviously uses IP addresses. A static configuration, one example is a configuration file, would solve that problem but not in respect to the requirement from above of not being static.

7 bits	1	40 bits	16 bits	64 bits
Prefix	L	Global ID	Subnet	Interface ID

Figure 6.2: Unique-local address format.

We looked for a complete dynamic solution which does automatically exclude some kind of "configuration file update service". The solution lies within the IPv6 protocol.

IPv6 uses 128 bit long addresses. This is a huge address space and obviously greater than 80 bits. Because OnionCat should act as a private network with public access, we chose a network prefix of the *unique local IPv6 unicast addresses* according to RFC4193 [17]. These addresses are similar to those of RFC1918 [36] for IPv4. The basic address format is shown in Figure 6.2.

It has a fixed minimum prefix length of at least 48 bits, additionally 16 bits being variable for subnetting and 64 bits for the interface ID (host part). We do not need any subnet so we can add the full subnet part to the interface part resulting in an 80 bits wide host part. The prefix length for those addresses is 48 bits. According to [17] we set the "L"-bit to 1 and generated a global ID thus resulting in the new unique-local IPv6 prefix FD87:D87E:EB43::/48 – the OnionCat prefix.

With this, we are now able to translate an onion-URL into an IPv6 address. This translation is done as follows: base32-decode the onion-URL and insert those 80 bits into the host part of the IPv6 address as shown in Figure 6.3 on page 54.

[1] Specifically for Tor it is true that running a hidden service does not require it to be a transit node which eliminates the headache of attracting huge amounts of traffic. This is of high importance for users with low bandwidth Internet connectivity.

48 bits	80 bits
FD87:D87E:EB43	onion-URL

Figure 6.3: OnionCat addressing scheme.

For example, decoding 7fd22jhmqgfl45j6.onion leads to 0xf947ad24ec818abe753e. Putting this together with the OnionCat prefix, it results in the IPv6 address fd87:d87e:eb43:f947:-ad24:ec81:8abe:753e. It perfectly meets the requirements for OnionCat.
Using this configuration, OnionCat can translate IPv6 addresses to onion-URLs and vice versa. If an IPv6 packet arrives from the operating system, OnionCat extracts the lowest 80 bits from the packet's destination IPv6 address, translates it back into an onion-URL, and requests Tor to open a virtual circuit to the desired destination.
After the connection is setup, OnionCat starts forwarding all packets through this virtual circuit. On the other end of the virtual circuit, OnionCat receives the packets from Tor and forwards them to the operating system. The operating system then in turn does with IP packets what has been said in Section 4. From the operation system's point of view, there is no difference if a packet arrived on a physical Ethernet interface or from OnionCat's virtual tunnel interface.
This method perfectly distributes the VPN entrance point. In this configuration every client is an entrance point. Summarized for network users, all one needs to know about is the destination IP address (or an onion-URL).

6.2 OnionCat Message Format

There are basically two questions associated regarding the data carried across the circuits.

1. Data based on which layer should be carried? Particularly, should only be layer 3 packets, which are IP packets, be carried, or should also layer 2 information, this is Ethernet, be forwarded? Layer 2 forwarding may be accomplished by using TAP devices (see Section 4.1) on both ends and send/receive raw data from the device to the remote end and vice versa. Pure layer 3 forwarding may be accomplished by using a TUN device instead.

2. Should the data be packaged in some way? This means should the data be prepended by some kind of header?

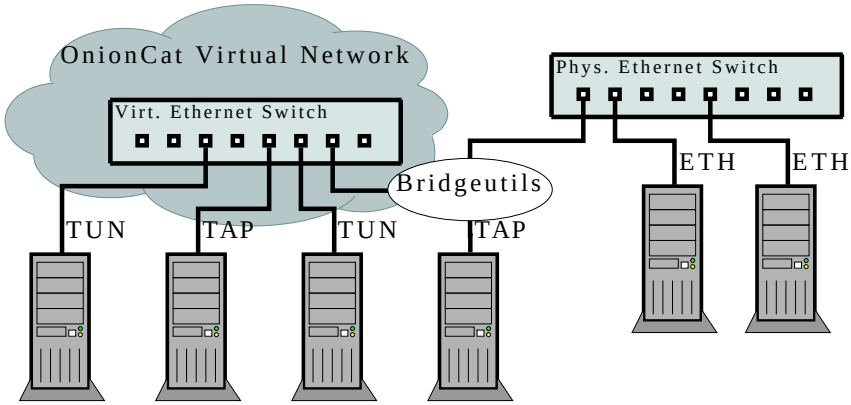


Figure 6.4: OnionCat acts like an Ethernet switch.

The first questions has several considerations. Carrying layer 2, this means whole Ethernet frames, is the most transparent way. Although computers are connected just virtually, when carrying Ethernet frames it would behave like if they were connected directly with a cable or an Ethernet switch. This is depicted in the left part of Figure 6.4. It shows the virtual OnionCat VPN cloud acting like a virtual switch. Nodes are connected through the TAP or TUN device.

Unlike common switches do, the IP headers (layer 3) have still to be analyzed, because they contain the link information of the virtual circuits of the VPN. This is the destination IPv6 address which is the onion-URL. This was explained in Section 6.1. But this kind of layer 3 investigation is part of the VPN and the VPN layer. It does not interfere with the data flow itself. The Ethernet header is generated by the kernel and forwarded to the user space when using a TAP device.

But carrying layer 2 information has downsides, too. First, it would introduce additional overhead. This is the Ethernet header of 14 bytes length per frame and thereby per packet, since every frame carries one packet or packet fragment. Second, it introduces additional overhead and network interaction, beside those 14 bytes.

As has been explained in Section 3.2, Ethernet has its own method of addressing and utilizes some specific protocols. One of these is NDP. Communication on Ethernet requires NDP handshakes beforehand. Even though this is of negligible effort for a local high bandwidth network it may introduce considerable traffic for a very low bandwidth wide area network.[2] Furthermore, protocols designed for Ethernet rely on its low latency, hence,

[2]Observations during this research turned out that the average bandwidth of a hidden service circuit is somewhere between 2 and 10 kbytes/s at the current state of the Tor network. It is not part of this

they may behave incorrect on networks with poor timings.[3]

Those arguments suggest to not forward layer 2 information and to use TUN devices only. They do not have any layer 2 information at all, as has been explained in Section 4.1. But also this has downsides. First, at least for Windows no TUN device exists (at least at the moment). Only a TAP device is available (see Section 4.2). It would break interoperability because a TAP device expects an Ethernet header but a TUN device does not. Thus, they are not compatible.

Generally, a TAP device provides another benefit: it allows *bridging* which a TUN device does not. Bridging means connecting two Ethernet segments together. An Ethernet bridge is nothing else than a switch with only two ports. Bridging is a feature of layer 2, hence, it requires layer 2 information which TUN devices do not provide. Although a TAP device is a virtual software device within a computer system, it can still be bridged, for example, to another virtual Ethernet device or even to a physical Ethernet device. For Linux the project *Net:Bridge* [41] (aka bridge-utils) provides a software bridge capable of bridging all types of Ethernet devices together. This is also shown in the right part of Figure 6.4 on page 55. When using a TAP device, it is possible to bridge it to the outside of a computer using the bridge-utils. Thus, connecting a physical switch and other computers with their real Ethernet interface (denoted as "ETH") becomes possible. Furthermore, those two physically connected computers do not need to know about OnionCat or Tor.

To summarize again the pros and cons of those issues:

- Transparent use of TAP devices requires layer 2 information to be forwarded. This increases the overhead and might reduce reliability because of critical timing.
- Transparent TAP usage avoids interoperability to TUN clients.
- Using TAP devices intransparently might require Ethernet protocol interaction like NDP. But this in turn allows TUN/TAP interoperability.
- TAP devices also involve more local protocol overhead and OS interaction because of the layer 3 protocols.
- TAP devices allow (real) Ethernet bridging.

study, thus this was not investigated thoroughly.

[3] Round trip times on hidden service circuits have been observed between 1 and 10 seconds. This is about 1000 times higher than it is on a typical Ethernet.

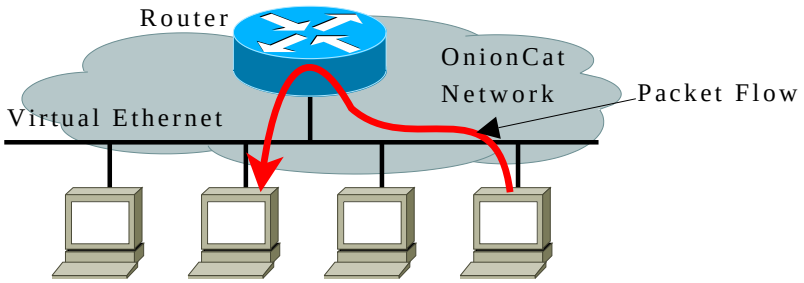


Figure 6.5: OnionCat's hyprid layer 2/3 model.

- Transparent use of TUN devices has less overhead but avoids TAP interoperability.
- TUN devices have less local overhead, too.
- TUN-only implementation would break Windows interoperability.

Apparently, the choice is to forward layer 3 information only to reduce overhead but it should allow TUN as well as TAP devices. This requires OnionCat to deal with layer 2 protocols locally, because layer 2 information is not forwarded across the virtual circuits. This is a hybrid approach. It allows the attachment of layer 2 devices and virtual layer-3-only devices but forwards only layer 3 information which are IPv6 packets. This could be compared to a router doing only packet switching within a single virtual Ethernet segment as it is shown in Figure 6.5. It is not a real router because it does not connect different layer 2 segments. This also implies that even though layer 2 devices are attachable, the network is not layer 2 transparent. No layer 2 protocol is forwarded, being the *spanning tree protocol*[4] (STP) [42] an example.

Nevertheless, this approach has several benefits. First, layer 2 protocol handshakes, NDP in particular, are done locally which completely mitigates the timing problem. Second, it prevents a possible information leak. MAC addresses are usually created by the local OS or the TAP driver. Thus, sending them across might leak information.[5] Third, it allows TUN/TAP interoperability.

The downside of this choice is that the software is required to have NDP implemented. As said in the beginning of Section 3.2.1 on page 18, NDP addresses several problems. Neighbor discovery and link-layer address determination, router discovery, and path main-

[4] STP primarily allows loop detection within Ethernet segments.

[5] Although this was not investigated, it might allow conclusions to the type of OS of a sender if MAC generation algorithms of some OSes in repsect to their tunnel devices are known.

tenance. As mentioned in that Section, only the link-layer address determination is important. OnionCat is not a router, hence, it may not provide router discovery and similar features.

For outgoing connections it has to intercept neighbor solicitation messages, extract the IPv6 destination address and demand a virtual anonymous circuit if the address is of the format as has been specified in Section 6.1. Simultaneously it has to respond with a solicited neighbor advertisement message.

As a consequence of incoming virtual circuits, OnionCat sends a neighbor solicitation message to the kernel end of the TAP device and awaits an advertisement. After that handshake packets can be forwarded.

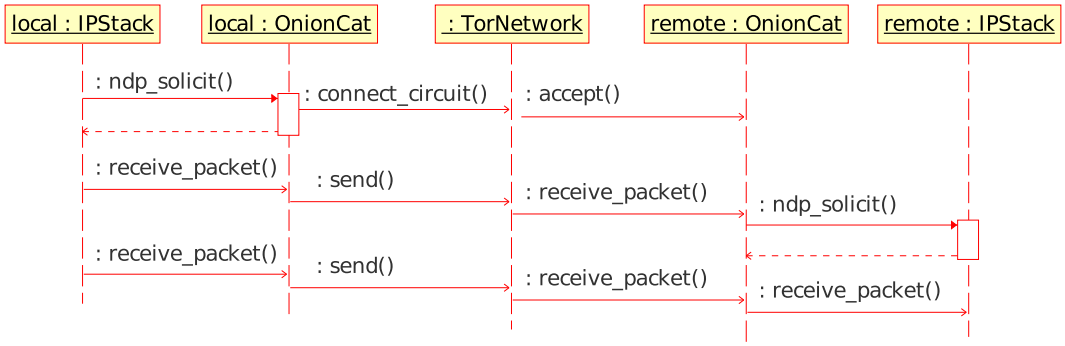


Figure 6.6: Packet flow sequence diagram with TAP devices.

Due to the fact that only layer 3 information is carried across the virtual circuits, OnionCat has to create the layer 2 header to have a correct format as required by TAP devices. For the outgoing direction the layer 2 header is stripped off. This timing behavior is shown in Figure 6.6. On the right side is the sender OS (local:IPStack) connected by a TAP device to the local OnionCat (local:OnionCat). This connects to the Tor network, drawn in the middle. On the right side is the remote OS and remote OnionCat.

The local OS tries to send an IP packet. Thus, it first has to complete an NDP solicitation. The local OnionCat immediately responds and requests a virtual circuit from the Tor network in parallel. The remote OnionCat will accept the connection. If packets are sent before the connection is established, they are dropped. The local OnionCat does not buffer any packet. The local OS sends packets through the TAP device to OnionCat, this in turn will send them through the circuit to the remote OnionCat. As a consequence, the remote OnionCat will complete an NDP solicitation. Once completed, packets are passed from the remote OnionCat to the remote OS.

An Ethernet header contains two MAC address fields. This is a source and a destination as described in Section 3.2. One field is the local TAP device's address or the address of a physically bridged device, the other address is the communication counterpart. Since layer 2 information is not forwarded but locally acknowledged, OnionCat uses its own "randomly" generated MAC address.

A second question was mentioned in item 2 on page 54: should the data packets travelling through the virtual circuits be prepended by an additional header? At a first glance this might make sense. Because an additional header could introduce some modularity. The header could contain information of the type and length of the packet carried within the payload, or the protocol version used by OnionCat, as an example. But the downside of an additional header is – again – additional overhead.
Furthermore, the packets actually already have a very comprehensive header anyway – the IPv6 header. This header has been shown in Figure 3.5 on page 15. OnionCat has to deal with the IPv6 header anyway because it has to decode at least the destination IP address to be able to request virtual circuits. Thus, we decided to not prepend any additional header.
The first four bits of the IP header contain the protocol version. However, if it gets necessary to prepend a header some time, it could be accomplished by using the IP header's version field to distinguish between packets with and without header.
Version number 5 might be a good choice because IP protocol version 5 never existed in real networks. It is very unlikely that it will exist ever because version 6 is already deployed. Also we do not believe that an additional header gets necessary, because even control information might be carried within IPv6.

6.3 Summary

In this Chapter it was explained how the basic building blocks being the concepts of a VPN and an anonymizing network, Tor in particular, are sticked together. The layer discrepancy was discussed and the method of address translation was explained in detail. It is translation of onion-URLs into IPv6 addresses and vice versa.
Furthermore, the message format of the VPN layer was presented and the method of TUN/TAP interoperability. VPN messages contain only layer 3 information. The layer 2 header is generated locally, if desired.

7 Software Architecture

Before designing a software, we have to take into consideration what functionalities it should have and which non-functional requirements it should fulfill.

Basically it bidirectionally shall forward IP packets, these are IPv6 packets in particular. On one end it should send and receive packets to and from a tunnel device as has been described in Sections 4.1 and 4.2. On the other end it shall communicate with Tor, or more specifically with a Tor proxy. This communication shall be bidirectional, too.

Different from the tunnel device which is opened once at startup, the connections are set up on demand. Furthermore, there are two types of connections in respect to the Tor proxy: outgoing and incoming ones. The outbound connections are requested through the SOCKS4a interface (see Section 3.3.1) of the Tor proxy. Those SOCKS requests create virtual circuits to hidden services as has been described in Section 5.2. Incoming connections are such connection which are requested by some remote OnionCat client. They must be accepted locally.

For every active connection to another OnionCat VPN member a TCP session must exist, independently if inbound or outbound. Data should be read and written from and to them, concurrently. Every session is associated with a specific client or hidden service, identified by an onion-URL or an IPv6 address respectively. This has been described in Section 6.1. Thus, it has to maintain a *peer list*. Every list entry has to contain at least a TCP session identifier and the IPv6 address to which it is associated. To prevent the peer list from growing infinitely in the course of time, it should be cleaned up periodically. Details on the peer list are found later in Section 7.1.1.

Non-functional requirements are memory and CPU efficiency to make it possible to be run on tiny systems like embedded controllers. It should also be able to utilize several CPUs if it is used in heavy loaded high performance environments. It should be as much portable as possible to become used by many people on different OSes.

7.1 Internal Design

Being a piece of software acting as a client and a server handling several different connections at the same time, it has to deal with concurrency. As has been described in Section 4.3 several models for parallel processing are available. The requirements previously mentioned suggest a hybrid model of using multi-threading and selecting several I/O channels (see Section 4.3.1). This seems to be a memory friendly approach by still offering multi-core CPU support.

The normal mode of operation will be forwarding data which is coming in on the tunnel device on one hand, and forwarding data which is coming in on the virtual circuits on the other hand. Thus, two threads are supplied, each as a packet forwarder on both ends. Two more threads are used for creation of outgoing circuits and acceptance of incoming ones. They are selecting on multiple I/Os. Of course this could be done within a single thread but it is chosen in that way due to increase of software modularity. Another separate thread will do data housekeeping, cleaning the peer list as an example.

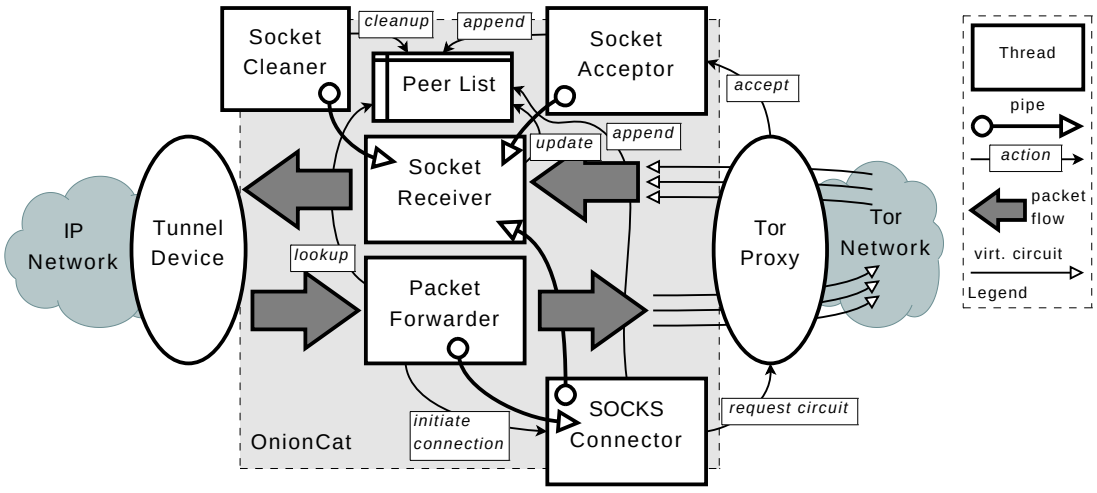


Figure 7.1: OnionCat block diagram.

Figure 7.1 gives a brief overview on the internal architecture. The main action controlling the packet flow from the tunnel device to the Tor network and vice versa is done by the two threads depicted in the middle. These are the *Packet Forwarder*[1] and the *Socket Receiver*. The first handles the outbound direction, the second one the inbound direction in respect

[1]The thread names might be somehow misleading. They just arose during writing the code. The names chosen in this description directly reflect the names of the C functions of the code.

to connections which are assumed to be established already.
The diagram shows three additional threads. This was previously discussed in brief. Those threads support setup of connections, incoming as well as outgoing, and the peer list maintenance. A brief introduction follows here. Some elements of the software are discussed later in more detail.

Packet reception on the tunnel device is handled by the Packet Forwarder thread. It extracts the destination IPv6 address of the incoming packet and *looks up* wether a peer with this address exists in the peer list or not. If so, it forwards the packet directly to the peer's file descriptor[2] and continues receiving packets on the tunnel device.
If there is no peer in the peer list it *initiates a new connection* by triggering the *SOCKS Connector* thread. This is done by sending a message to it through a pipe. The packet itself is dropped. Afterwards it continues receiving packets on the tunnel device.
The SOCKS Connector tries to *connect* to the hidden service through the Tor proxy's SOCKS4a interface. If it was successful it makes a new entry into the peer list and signals the Socket Receiver to indicate that the peer list has changed. This is done by sending a message through a pipe to it. If the connection failed, it retries to connect again several times. After that, the request is dropped.
Data reception from Tor is done by the Socket Receiver thread if connections are already established. If data is received it is appended to the defragmentation buffer of the appropriate peer. Every peer has its own defragmentation buffer. The buffer is discussed in more detail later in Section 7.1.2. If the buffer contains at least one complete packet, the source IPv6 address is extracted from the header. Then it *updates* this peer's address field while it is still empty. Next, it forwards the packet to the tunnel device and deletes it from the defragmentation buffer.
The diagram shows three pipes pointing to the Socket Receiver. Actually this is a single pipe having three threads writing to it.
New incoming connections to the hidden service from the Tor network are handled by the *Socket Acceptor* thread. On program startup it creates a listening TCP socket and waits for incoming connections. We chose 8060 to be the default port. According to the IANA assigned port numbers list [20], the range from 8060 to 8073 is unassigned yet. Once a connection comes in, it accepts it, creates a new entry in the peer list and continues

[2]File descriptors are handles for I/O channels like TCP sessions. This was discussed in Section 4.3.1.

accepting connections. The Socket Receiver is signalled through a pipe if the peer list changed.

At the time the connection arrives, OnionCat does not know about the originating address (onion-URL/IPv6) because those TCP sessions are always initiated by the local Tor proxy. Its source address is 127.0.0.1. Furthermore, it is just the transport. OnionCat (and every other hidden service) just uses the payload of those circuits (see Section 6). Outbound packets cannot be sent to this new peer as long as it is not identified.

Identification happens immediately at reception of the first IPv6 packet (see above). Unfortunately, this is a known security weakness. One might spoof the source IPv6 address to impersonate someone else, or to receive packets destined for someone else. This could be solved by initiating an outgoing connection based on the supposed IP address. Outgoing packets should then be sent only to this circuit and not the incoming one.

Impersonating a hidden service is impossible, because of the relation between the onion-URL and its key as has been explained in Section 5.2. Initiating outgoing connections as a consequence of an incoming one implies that all circuits will be used only unidirectional.[3] However, it is not implemented in that way yet. This is because the setup time of a virtual circuit may be even one minute or longer. This unidirectional approach needs two circuits, hence, the setup time is spent two times.

The *Socket Cleaner* thread wakes up periodically. It iterates through the peer list as described below in Section 7.1.1 and checks if a peer was not used for a given amount of time. This time is chosen to be 3 minutes. If this idle time elapsed, the virtual circuit is closed and the peer is removed from the list. This is done in order to not load the Tor network more than necessary.

7.1.1 The Peer List

The peer list keeps track about active peers. A peer within this context is defined to be a virtual circuit together with its destination IPv6 address. While activating a peer it runs through several states. Thus, a peer is associated with a number of fields. Obviously, this is a file descriptor of the TCP socket and an IPv6 address field. Furthermore it contains a connection retry counter, a pointer to the defragmentation buffer (see Section 7.1.2) and an integer value containing the number of bytes kept in the buffer, and an idle timer which

[3] Such an implementation shall carefully request outgoing connections based on the first packet only. Otherwise it may be subject to a potential *denial of service* (DoS) attack to the Tor network.

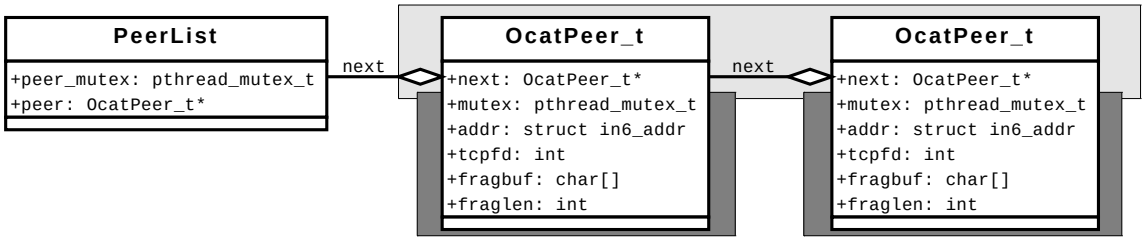


Figure 7.2: The peer list.

actually is the timestamp of the last data being sent or received. Additionally, it contains some fields for maintaining statistics like byte counters. It contains a mutex for thread locking and a pointer to the next peer.

As show in Figure 7.1 on page 61 all threads access the peer list, hence, thread locking must be done very careful. To maximize concurrency of threads, two types of mutexes exist to maintain the list. This is shown in Figure 7.2. The list is a single linked list of peers of type OcatPeer_t. Every peer has its own mutex.

As shown on the left side of the diagram, a start pointer to the first element exists and an additional list mutex peer_mutex. The list mutex is used to lock the list pointers. This is depicted with the light gray shaded area. To preserve from deadlocks or other kind of locking failures, the software must adhere to the following rules in that order without any exception before accessing any data within a peer structure.

1. Acquire a lock to the peer list mutex (peer_mutex).
2. Set the current peer element to be the first element of the list.
3. Acquire a lock to the current peer element (OcatPeer_t).
4. Test if the element is the desired one. If true, continue at step 9.
5. Set the current peer element to be the next element.
6. Release the peer's mutex.
7. If the next element does not exist, release the peer list mutex and stop iterating through the list.
8. Continue at step 3

9. Release the peer list mutex.
10. Read/modify the peer variables.
11. Release the peer's mutex.

Once a peer mutex is released, it must never be reacquired directly without following the steps above.

7.1.2 Buffering and Defragmentation

When writing network applications, it comes always to the question if data should be buffered in some special situations. In respect to OnionCat a typical example is what to do with packets while the connection setup is in progress. As long as the connection is not established, data cannot directly be forwarded. As discussed in Section 6.2, OnionCat acts like a switch on the virtual Ethernet that it creates. Under this aspect it is easy to decide if data buffering should be done or not. Ethernet switches do not buffer data at all, or at least not more than absolutely necessary. We apply this rule to OnionCat.
However, when analyzing protocol behavior, it suggests to not buffer anything, too. OnionCat forwards IP packets. It is a fact that IP packets may get lost in networks. The IP protocol and the protocols based on IP, like TCP or UDP, are designed specifically to deal with packet loss.

The sole point where data must be buffered is when it is coming in from the virtual circuits. We have to do some kind of defragmentation. This might be a little bit misleading. This defragmentation is not about IP defragmentation.[4] It is a fragmentation happening within the VPN layer. This is below layer 3 (see Figure 6.1(b) on page 51).
The reason for that is that the virtual circuits, which are used within the VPN layer, are TCP based. As said in Section 3.3, TCP sessions are byte streams but tunnel devices are packet streams, as has been described in Section 4.1. This means that, if a packet which in turn consists of a number of bytes is read from the tunnel device and written to the virtual circuit, it may not arrive at the other end of the virtual circuit as the same bunch of bytes. The order is preserved but in the worst case they arrive sequentially, one byte after the other. Only in the best case they will arrive in the same bulk as they were sent.

[4] IP fragmentation is a feature of the IP protocol.

To give a real world example, assume to quickly pour a bucket of water into a pipe within one second. It can be observed that the water will flow out of the pipe after a while, but not within a single second. It will take longer. In other words, the pipe stretches the content (the water) of the bucket. And exactly that may happen when sending a bunch of bytes through a TCP session.

This means that we have to deal with packets arriving in pieces at the inbound direction.[5] But it must not be sent as those pieces to the tunnel device. The tunnel device expects to receive packets as a whole. Thus, they have to be defragmented.

Defragmentation is done using a buffer and a variable containing the current amount of bytes contained within the buffer. Initially the variable is set to 0. Obviously, every peer has its own defragmentation buffer. Data is read from the virtual circuits to the buffer being the counter variable an offset to the start address of the buffer.

As has been described in Section 6.2, OnionCat transmits raw IPv6 packets without any header, hence, defragmentation is done by detecting and decoding IPv6 headers. This has two consequences. First, it defines the minimum amount of bytes that must be in the buffer before decoding may start. Second, it defines the minimum size of the buffer.

Decoding of the IPv6 header requires the header to be complete. As described in Section 3.1, the IPv6 header has a fixed length of 40 bytes. This is the minimum amount of bytes. The IPv6 header contains a field describing the length of payload of the packet. It is 16 bits wide. Thus, the maximum length of an IPv6 packet is the length of the header plus the maximum length of its payload. This is $40 + 2^{16} = 65576$. This is the minimum size of the buffer. Actually this is not completely true. The real packet size is limited by the underlying layer 2. It is the MTU size as has been described in Section 3.2. Tunnel devices have a default MTU size of 1500. This is not changed by OnionCat for compatibility reasons. Thus, the minimum buffer size may be set to 1500. However, currently the full IP packet size is used. On normal computer systems this makes no difference but on embedded systems with limited resources this should be taken into account.

Once the buffer is filled with at least 40 bytes, the most significant four bits of the first byte are compared to 6. They contain the protocol version number and in case of IPv6 it must be 6. If this is not the case, an error occurred somewhere. The byte is dropped and the counter containing the number of bytes is decreased by one. Otherwise the payload length field of the header is examined and compared to the byte counter of the buffer. If

[5] The inbound direction is handled by the Socket Receiver thread.

the buffer currently contains at least the amount of bytes given in the payload length plus 40 (the header length), the whole packet is sent to the tunnel device. Then those bytes are discarded from the buffer and the counter is decreased by the appropriate value.

7.1.3 Interacting with the TAP Device

Everything said above within this Chapter in respect to the flow of packets relates to a tunnel device operated in TUN mode. To support TAP mode, some additional tasks are necessary. This is because the TAP mode provides and expects layer 2 information as has been described in Section 4.1. In particular, for reasons stated in Section 6.2, OnionCat has to be able to deal with the NDP protocol. NDP was described in Section 3.2.1. This includes two things. First, OnionCat needs an own MAC address that is used in the layer 2 headers. Second, it needs to maintain a list of MAC and IPv6 addresses.

The MAC address is generated at program startup. The vendor ID[6] is set to 00:00:6C,[7] the lower three bytes are copied from the lowest three bytes of the IPv6 address (which reflects the local onion-URL). As it is common for most real Ethernet switches, OnionCat maintains a list with a fixed number of entries to keep the MAC and IPv6 addresses. Additionally, every entry has a variable containing the age of an entry. After a given amount of time the entry is deleted from the list again. The time currently is set to 2 minutes. The maximum number of list entries is chosen to be 128 yet.

In TUN mode the Packet Forwarder drops all packets with a destination address that has a prefix different from that chosen for the OnionCat network (see Section 6.1). In TAP mode the Packet Forwarder behaves a little bit different. A fairly complex decision tree is necessary for deciding what to do with a frame. This decision tree is show in Figure 7.3 on page 68. The root is the gray shaded box at the upper left corner. If the destination MAC address is a unicast address not destined for OnionCat the frame is dropped. If the destination MAC address is a unicast for OnionCat and the destination IPv6 address does not have the correct prefix, that frame is dropped.[8] If the prefix is correct and the packet is not an NDP advertisement message, it is forwarded regularly. The Ethernet header is stripped off before forwarding. If it is an advertisement, an entry is made into the MAC

[6] The vendor ID is contained in the most significant three bytes of the MAC address.

[7] According to the IANA list of Ethernet numbers [19] this ID is not registered.

[8] This behavior has to be changed if routing based packet forwarding should be done. This is currently in development, hence, it is not described within this document.

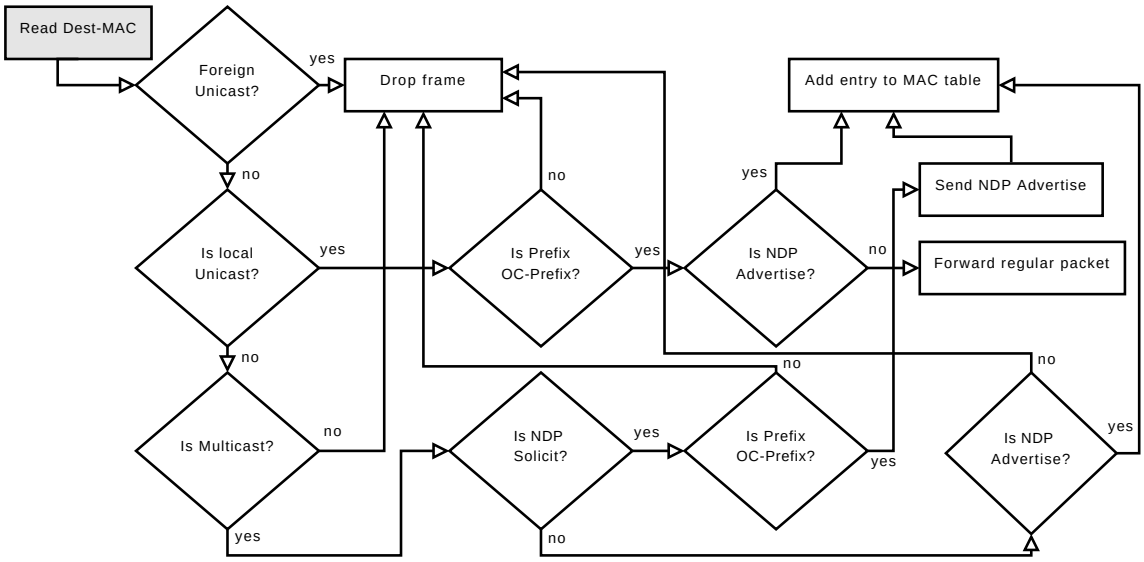


Figure 7.3: Decision tree for frame validation on TAP device.

address table. An advertisement may arrive in response to a solicitation.
If the destination MAC address is a multicast address and the frame contains an NDP solicitation for an OnionCat IPv6 address, an advertisement is sent and an entry is made into the MAC address tables. If the frame contains an advertisement this might be an unsolicited advertisement. An entry is made into the MAC address table. In all other cases the frames are dropped.

The inbound direction which is handled by the Socket Receiver thread is far less complex. First the IPv6 destination address of packets that arrived on the virtual circuits are looked up in the MAC address table. If an entry exists, an Ethernet header is prepended being the destination MAC address the one from the table entry. The source MAC address is set to the local OnionCat's MAC address which has been generated at program startup as mentioned above. If no entry exists, an NDP solicitation message for the desired IPv6 address is generated and sent to the tunnel device. The frame itself is dropped because the NDP advertisement will arrive asynchronously and cannot be waited for. Furthermore, the advertisement will be handled by the Packet Forwarder as described above.

7.1.4 Outgoing Connection Handling

The setup of outgoing connections are handled by the SOCKS Connector thread. This thread handles several I/O actions in parallel by using the select() system call. Select() is

used by other threads too, but in that case every new connection passes through a simple state machine to handle different states. The SOCKS Connector internally maintains a list which contains connections whose setups are currently in progress. Every element contains the desired destination IPv6 address, a retry counter and a variable containing the element's current state.

The thread usually blocks on a call to select(). It waits for three types of events. First, data is coming in on the internal communication pipe as shown in Figure 7.1 on page 61. The pipe is used by the Packet Forwarder to deliver new requests to the SOCKS Connector. Second, it waits that the TCP connection to the Tor proxy gets ready. Third, it waits for SOCKS responses coming in from the Tor proxy.

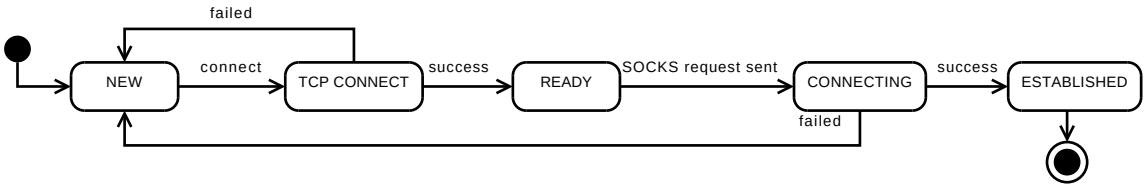


Figure 7.4: State diagram for outgoing connection setup.

Every new connection runs through the state diagram as shown in Figure 7.4. When a new request comes in on the pipe, a new element is appended to the connection list. The element's initial state is NEW. The function connect() is called in non-blocking mode and the state is changed to TCP CONNECT. The file descriptor is passed to the write set of select(). After the connection gets ready it is tested for errors as shown in Listing 4.4 on page 41. On failure it resets the state and retries from the beginning.

If the connection was successful, the state is changed to READY. A SOCKS request is sent through it. The state is further changed to CONNECTING meaning that the virtual circuit setup is in progress. The file descriptor is passed to the read set of select(). After it got ready, the response is read. In case of failure it resets the state and tries again from the beginning. In case of success, the state changes to ESTABLISHED. This is a virtual state because actually the element is remove from the list. It is a new peer ready for communication, hence, the element is inserted into the peer list. The peer list was discussed in Section 7.1.1.

7.2 Description of Software Modules

Section 7.1 covered several design issues and specific aspects worth to be mentioned because of its complexity. This Section gives a brief overview about the structure of the code.
The source code is written in C using the *GNU C Compiler* gcc. The code conforms to POSIX as far as possible and to my best knowledge. It is supported by the *GNU autotools* [46] to make porting it to other OSes more easy. Originally it was developed on *Debian Linux*, Kernel 2.6.26 on an *i386* platform, later on an *amd64* platform. The source code can be downloaded on the project's home page [13]. The source package contains several files necessary for the autotools. The source code is kept within the subdirectory src.
All preprocessor macros and includes, structure definitions, and prototypes are found in the header file ocat.h. Specifically for *Cygwin*, which is a POSIX-like environment for Windows, additional headers exist. They are found in the subdirectory cygwin within src. Cygwin does not have IPv6 support by default (see also Chapter 8). A patch exists but it does not include every definition needed for IPv6. Thus, those missing are provided within the OnionCat package. The header files are taken directly from OpenBSD and have been slightly adapted were necessary.
OnionCat needs several global variables. They are combined together in a static structure of type OcatSetup_t. The structure and some other global initialization functions are kept within ocatsetup.c. Functions regarding thread management are found in ocatthread.c. It contains wrapper functions for thread initialization and termination. The SOCKS Connector thread and all functions associated with it are found in ocatsocks.c. Functions handling NDP and the MAC table maintenance are kept in ocateth.c.
OnionCat supports logging to the console, to a logfile as well as to the syslog daemon. Those functions are found in ocatlog.c. The functions for the peer list maintenance are found in ocatpeer.c. The setup code for the tunnel device is found in ocattun.c for all UNIX-like OSes and in ocat_wintuntap.c for Windows. The code for Windows actually connects to the OpenVPN TAP driver as has been explained in Section 4.2. The file ocatroute.c contains the source code for the other threads. These are the Packet Forwarder, the Socket Receiver, the Socket Acceptor, and the Socket Cleaner.
One additional thread exists which has never been mentioned until now. That is the *Controller* thread. It was not mentioned because of its purpose mainly as a debugging facility. It is a command line interface listening on TCP port number 8066 and it provides

several commands for gathering information. As already mentioned, it was developed mainly for debugging. Its source code is found in ocatctrl.c (and is somehow ugly...). The main() function and code for program initialization and shutdown is kept in ocat.c.
There are several additional support functions surrounding the core elements mentioned above. Those are base32 encoding and decoding functions, onion-URL to IPv6 address conversion, and similar functions. They are found in the files ocatcompat.c, ocatlibe.c, and ocatv6conv.c.
The latest releases of OnionCat contain limited routing support for IPv6 as well as for IPv4. This was not discussed because it is beyond the scope of this document. The regarding functions are found in ocatipv6route.c and ocatipv4route.c.

7.3 Summary

This Chapter discussed and presented the chosen software architecture. It uses serveral different techniques to fulfill the requirements. Those are multi-threading and asynchronous non-blocking I/O.
The threads being implemented and their interaction was explained. It containes two main threads, one for the inbound one for the outbound direction. Some additional threads surround them to complete its functionality.
Three specific aspects have been discussed. First, the packet fragmentation happening on the VPN layer and the technique used to defragment it again. Second, the interaction with the tunnel device in TAP mode using the NDP protocol. Third, the state machine being used for the setup of outgoing connections.
Finally, a brief overview on modules and source files was given.

8 Application and Usage

OnionCat may be used in a wide range of different applications. It provides a kernel interface, this is a network device to which an IPv6 address is assigned. Thus it provides one of the most compatible interfaces possible. The whole background was extensively discussed in the previous Chapters.
OnionCat is based on anonymizing transport layers like Tor,[1] thus it may be used by various user groups for the same reasons as they use, for example, Tor. OnionCat is an extension of anonymizers. It adds features but it also extends the group of different users and use cases.
The most important goal of OnionCat is to enable users to transport raw IP data across an anonymizing network together with automatic IP address configuration. This has two considerations.

1. It makes it easier use.
2. It creates a single logical virtual network segment so that all users share it. Thus they are automatically connected virtually together.

The second item is achieved by every VPN, but different from any other VPN the OnionCat network is an open network. Every user can take part without any restriction or limitation in respect to network addressing. A user can decide to change his address at any time[2] and he can also leave the network again without leaving traceable footprints. Without further requirements one can use OnionCat to achieve the following use cases.

- Usage as an open anonymous network (This is described above.).
- Usage as real VPN for a privately set up closed user group.

[1] Currently it works just with Tor. Development of adapting it to I2P is in progress.
[2] This is the case only if the anonymizing transport allows this. But Tor does as well as I2P.

Both are fitted to bypass surveillance or supervised networks of any kind. In the following sections I will discuss the setup of those scenarios in detail, explain configuration and debugging issues.

8.1 The Open Anonymous Network

The default application for OnionCat is to create an anonymous VPN which is publicly accessible. It enables users to take part in the anonymous network. Once being a participant one could either use the network's services or provide ones own services or both. A prerequisite is to have a network address. The addressing method was discussed in 6.1. Thus, we first need to install and configure the anonymizer and run a hidden service. The following explanations refer to Tor as an anonymizing transport network, because OnionCat was originally developed for Tor and it is known to run stable with it.

8.1.1 Installing and Configuring Tor

To install Tor there are basically two ways.

1. Install it with a package manager.
2. Compile and install it from source.

The first solution is probably the easiest way and usually suits most users' requirements. The installation depends on the operating system and distribution one uses. It requires a package to exists for the targetted distribution which may not be the case. Then it is required to be built from source. This also gives a little bit more flexibility in fine tuning serveral build options.

To give some examples, on Debian Linux (www.debian.org) type

```
% sudo aptitude install tor
```

on FreeBSD (www.freebsd.org) the package is added with

```
% sudo pkg_add \
    ftp://ftp.freebsd.org/pub/FreeBSD/releases/amd64/7.1-RELEASE/\
        packages/security/tor-0.2.0.31.tbz
```

and on OpenBSD (www.openbsd.org) with

```
% sudo pkg_add \
    ftp://anga.funkfeuer.at/pub/OpenBSD/4.4/\
      packages/amd64/tor-0.1.2.19.tgz
```

respectively.[3] Details on package installations and package mirrors should be looked up on the operating system's or distribution's main sites.

For Windows and MacOS X you should follow the links on the download page of the Tor project [44]. For both OSes *Vidalia* (www.vidalia-project.org) is installed together with Tor. Vidalia is a configuration and control GUI for Tor which is also available for Linux.

After successful installation we need to add a *hidden service*. This is done by either editing the Tor configuration file torrc or by adding it with Vidalia. The latter results in Vidlia editing the configuration file of Tor. The configuration file is usually located in /etc/tor/torrc or /usr/local/etc/tor/torrc. On Windows it is located in C:\Documents and Settings\<user>\Vidalia\torrc.

Add the following two lines to the configuration file:

```
HiddenServiceDir /var/lib/tor/hidden_service/
HiddenServicePort 8060 127.0.0.1:8060
```

On Windows the fullpath may be omitted, for example, by just configuring HiddenServiceDir hidden_service. The directory will be created in C:\Documents and Settings\<user>. The HiddenServiceDir directive specifies the directory where to locate the private key for the hidden service. HiddenServicePort specifies that all TCP connections, which are dedicated to virtual destination port 8060 from within Tor, are forwarded to the local host (127.0.0.1) TCP port 8060. This is the port that OnionCat listens to by default, as has been said in Section 7.1.

Now start Tor, but make sure that the system clock is correct beforehand. Tor will then create a directory at the location specified by HiddenServiceDir, as well as put two files into it: private_key and hostname. The first one contains the private key associated with the local hidden service. If running a service for other users, web service for example, it

[3] Both links are valid for today, 5th of March 2009, for the stable releases FreeBSD 7.1 and OpenBSD 4.4 for amd64 machines.

is a good idea to backup this key to a safe place. It is with this key that a specific hidden service is uniquely identified. If the machine crashes and all data is lost, the hidden service can be recovered by copying the backed up key to the hidden service directory on the new machine. The key is a 1024 bit RSA key.
The file hostname contains the hostname which is used by the Tor network to lookup and connect to this hidden service. It iss the onion-URL. Look into the file.

```
% sudo cat /var/lib/tor/hidden_service/hostname
```

It contains a string like a5ccbdkubbr2jlcp.onion. Vidalia will display the hostname in the "Provided Hidden Services" field in the "Services" settings window. You will need this hostname for OnionCat setup as explained below.
Have a look at the log file to see if Tor is working. On Unix this is usually located at /var/log/daemon.log or /var/log/messages. If not have a look at the Tor configuration file torrc. If using Vidalia, then just click on "Message Log".
If Tor works correctly it will say "Tor has successfully opened a circuit. Looks like client functionality is working.". Note that Tor may need some time (a few minutes) to boot.

8.1.2 Installing OnionCat

Now, after successful installation of Tor we can run OnionCat. As far as there are no packages[4] OnionCat can be built from source. The steps are

1. Prepare build environment.
2. Build and install OnionCat.
3. Configure and test OnionCat.

On Unix-like OSes step 1 is not very difficult and it is most likely to be already setup. All that OnionCat nees is a C compiler, usually GNU gcc and the GNU make utility. On Windows we also need those two programs. It is necessary to create a POSIX-like environment beforehand. It is done with Cygwin [5] (www.cygwin.com). Using Cygwin is more difficult, hence, I will explain it in a separate Section 8.1.3. If you are going to install OnionCat on Windows read Section 8.1.3 before.

[4]The package building process can be very time consuming. There are already packages for Debian, Ubuntu, and OpenBSD, and probably more. Others are in preparation.

To check if gcc and make exist, on the shell just type gcc. If it is installed it will say "gcc: no input files". If not, the answer will be "command not found: gcc" or similar. If make is installed it will respond "make: *** No targets specified and no makefile found. Stop.". If one of those tools is missing install it with your package manager. On Debian Linux this is done with

```
% sudo aptitude install gcc
```

Now download an OnionCat source tarball from [14] www.cypherpunk.at/ocat/download/. Untar it, changed to the directory and configure it.

```
% tar xfz onioncat-0.1.12.r487.tar.gz
% cd onioncat-0.1.12.r487
% ./configure
```

Then compile and install it. The latter as root.

```
% make
% sudo make install
```

Now you should be able to run OnionCat by typing ocat. If run without any argument it will ouput version, author, compile date, and a short command usage guide, listing all available command options.

8.1.3 Installing Cygwin on Windows

To run OnionCat on Windows, create a POSIX-like environment for it. Go to the download page found at [5] www.cygwin.com and download and run the Cygwin installer. It asks some questions, but click continue until you reach the package selection menu and select "gcc: C-Compiler" and "make: GNU 'make' utility". Both are found in the "devel" section of the package selection window. Continue with the installation process until finished.
OnionCat is IPv6-based but Cygwin unfortunately does not support IPv6 at the current stage of development. But luckily there is an IPv6 patch available at win6.jp/Cygwin/ [21] by Jun-ya Kato. Download the package and extract it at the top level of Cygwin which usually is C:\cygwin. It will install several files and a new Windows library at C:\cygwin\bin\new-cygwin1.dll. Rename the existing file cygwin1.dll to something else and then rename new-cygwin1.dll to cygwin1.dll.
The next step is to install the TAP driver. This is a virtual network interface, usually called a *tunnel device*. The background on tunnel devices has been explained in the Sections 4.1

and 4.2. It connects to the kernel's routing process on one end and provides a userland interface on the other interface. Applications can connect to this interface and accept certain low level packets as routed by the kernel due to the nature of IP(v6) routing characteristic.

Go to [32] openvpn.net and download the OpenVPN Windows installer. To run OnionCat, OpenVPN itself is not necessary but the installer contains the TAP driver which was developed by the OpenVPN project. It is licensed under GPL version 2 with some additions.

After downloading, execute the installer. It will display an options menu where you can un-select everything except the TAP driver. Continue with installation.

After successful installation click on the Cygwin icon on the desktop and continue reading at Section 8.1.2.

8.1.4 Configuring OnionCat

To configure OnionCat for its primary intention as client for the open anonymous network, we need the onion-URL which is located in the hostname file in the hidden service's directory (see Section 8.1.1). When running OnionCat the first time you probably should run OnionCat in foreground to make sure that everything works correct.[5]

In the shell, run OnionCat with the command as show below. As argument put your hidden service hostname.

```
% sudo ocat -B wq52ql3uxgcxqjon.onion
```

OnionCat will produce some output. There might be errors like "select encountered error: "Interrupted system call", restarting". As long as this just happens during startup it can safely be ignored. There might also be a the warning "can't get information for user "tor": "user not found", defaulting to uid 65534" which can also be ignored.

It may fail starting with the error messages "could not open tundev /dev/*tun0*: No such file or directory" and "error opening TUN/TAP device", if OnionCat did not find a suitable tunnel device. This may be the case if either the name of it is different or if no such device exists. The tunnel device is a kernel module. The *tun* kernel module needs to be loaded in order to work. Check with lsmod (on Linux) if the module is loaded. Load it

[5] At the time of writing this document (March 8, 2009) OnionCat will fail on Windows if not run in foreground.

with modprobe tun if it is not present. After loading the module, the character device tun should appear somewhere within the /dev directory. See Section 4.1 for more details about the location of the device. Restart OnionCat again.

Now check if everything is configured correctly. Issue the command ifconfig. It lists several stanzas. One for every registered network device. One should read tun0 and have an IPv6 address assign. The appropriate line looks like "inet6 addr: fd87:d87e:eb43:b43b:a82f:74b9:8578:25cd/48 Scope:Global". If the tun0 stanza exists but has no IPv6 address assigned, OnionCat may have failed assigning the address. In some very rare cases this may happen for a currently unknow reason. In that case assign the address manually. In order to do this, first figure out your onion-URL as described in Section 8.1.1 and issue the command

```
% ocat -i wq52ql3uxgcxqjon.onion
fd87:d87e:eb43:b43b:a82f:74b9:8578:25cd
```

It returns the IPv6 address associated with the onion-URL. Now configure the address with ifconfig.

```
% sudo ifconfig tun0 add fd87:d87e:eb43:b43b:a82f:74b9:8578:25cd/48 up
```

Of course, you must supply your own IPv6 address. This is just an example. If the command fails the syntax may be different on your system. Lookup the correct syntax of ifconfig in the appropriate man page by issuing the command man ifconfig.

Now check the IPv6 routing table with the command

```
% netstat -nr6
```

The digit '6' might be omitted on some OSes. It lists all entries of the kernel's IPv6 routing table and it should contain at least one entry exactly like shown below.

```
fd87:d87e:eb43::/48      ::      U      256      0      0      tun0
```

Important are the first and the last columns. If there is no such entry which could happen in some very rare cases add the route manually.

```
% sudo route --inet6 add fd87:d87e:eb43::/48 tun0
```

The command listed above is valid for Linux. OpenBSD syntax is slight different.

```
% sudo route add -inet6 -net 3000:: \
    -prefixlen 48 fd87:d87e:eb43:365b:148d:d42e:fdc0:6b7b
```

If in doubt lookup the correct syntax in route man page.

8.2 Using the Global Anonymous Network

If everything is setup correctly as described in the previous Sections you should now be able to use the OnionCat global anonymous network. First try to ping one of the existing hidden OnionCat services. Currently there are a few services known to be permanently online which are shown in the table of Figure 8.1.

IPv6 address	Hostname
fd87:d87e:eb43:f683:64ac:73f9:61ac:9a00	dot.aio
fd87:d87e:eb43:2243:5f84:5b12:7bb5:bbc2	irc.onion.aio
fd87:d87e:eb43:f947:ad24:ec81:8abe:753e	ping.onion.aio
fd87:d87e:eb43:744:208d:5408:63a4:ac4f	mail.onion.aio

Figure 8.1: List of currently available OnionCat services.

- dot.aio[6] is a web-based service registration directory. It is intended to let OnionCat service providers register their service in order to be found by others. It is not required for a service to be registered but it enhances usability for new users. They can browse this page and lookup existing OnionCat services.

- irc.onion.aio basically is an *Internet Relay Chat*[7] (IRC) server. For a quick introduction have a look at Wikipedia at en.wikipedia.org/wiki/Internet_Relay_Chat. There is also a web-based audio stream ("OnionCat Radio") available on port 1337 at this same address and a new, web-based community plattform called "*Whose Space?*".

- ping.onion.aio currently does nothing than just respond to echo requests.

- mail.onion.aio is a combined SMTP/POP3 server. It accepts mails on port 25 for recipients of the domain onion.aio (e.g. eagle@onion.aio which is the email address of the author). Users can fetch mail using the POP3 protocol on port 110. Mailboxes need to be registered in advance. Unfortunately there is currently no automatic registration service available. Post an email to onionmail@onion.aio on this server in order to get an account. Note that this is completely anonymous as long as you do not send personal information across with your email.

[6] The term "aio" refers to *anonymous Internet overlay*.
[7] IRC is based on the protocol definition of RFC1459 [31].

Try to ping one of those hosts by issuing the ping6 command which is ping for IPv6.

```
% ping6 fd87:d87e:eb43:2243:5f84:5b12:7bb5:bbc2
PING fd87:d87e:eb43:2243:5f84:5b12:7bb5:bbc2 56 data bytes
64 bytes from fd87:d87e:...:7bb5:bbc2: icmp_seq=1 ttl=64 time=3376 ms
64 bytes from fd87:d87e:...:7bb5:bbc2: icmp_seq=2 ttl=64 time=3404 ms
64 bytes from fd87:d87e:...:7bb5:bbc2: icmp_seq=3 ttl=64 time=5358 ms
64 bytes from fd87:d87e:...:7bb5:bbc2: icmp_seq=4 ttl=64 time=4613 ms
```

After some time it will respond and list the *round trip time* (RTT) in the right most column. Be patient, Tor may need up to one minute for the first time it connects to a hidden service. After the connection is setup the RTT will be between 0.5 and 10 seconds. Currently there are many efforts within the Tor project to improve connection setup time and RTT in respect to hidden services.

If everything worked until now you can start using the network as you do with Internet with the sole exception that there is no DNS. It requires the user to use plain IP addresses instead of domain names. As long as there is no feasible DNS solution, the hostnames can be registered locally. On all Unix-like OSes this is easily done by just putting IP address hostname pairs into the file /etc/hosts. This is possible even on Windows. The file is usually located at C:\WINDOWS\SYSTEM32\drivers\etc\hosts.

8.3 Summary

This Chapter gave an overview about installation and usage of OnionCat. It included prerequisites for installation. These are build tools. Specifically for Windows, additional software is required. This is the POSIX-like environment Cygwin and the TAP driver, which is the tunnel device for Windows.
Finally, a list of available services together with a short description was presented.

9 Conclusion

This paper describes an add-on for anonymizing networks like Tor. It interferes with the IP routing process of the kernel and creates a VPN on top of anonymizing networks. Within this document the primary development ideas of OnionCat have been shown. Part I introduced the reader into details regarding cryptography, networking, and operating systems, to be able to follow the explanations of Part II. In that respectively, the issues and solutions regarding an anonymous VPN have been described.
We do address translation from internal addresses used by Tor into IPv6 addresses. This is one of the most important issues regarding this dynamic peer-to-peer VPN.
Based on those ideas, an OpenSource project namely *OnionCat* evolved. It is used as a reference implementation. Most software internals, as well as difficult parts regarding software development have been explained.
The last Chapter gave a more than brief step-by-step installation, configuration, and usage guide for OnionCat. Additionally, there are already some services available which have been presented in brief.
The OnionCat software is still in heavy development and may not work on every system without further intervention, but we managed to port it to major operating systems like Windows XP and MacOS X. Of course, it works out of the box on Linux.

During this work several new problems turned out to exists. First, this the *DNS problem*, which is still not solved sufficiently. OnionCat uses IPv6 addresses, but it would be more convenient for users if the use of hostnames would be possible. As has been said in Chapter 5, providing name resolution with the Internet DNS might leak information which is contrary to what an anonymizing network should do.
Another way would be to provide a name service within the OnionCat network. This is done yet on the hosts mail.onion.aio and irc.onion.aio (see Section 8.2), but this is ex-

perimental. It also creates the new problem of the existence of two DNS systems acting independently of each other. This in turn introduces new problems. To summarize, a project dealing with the general question "How to resolve hostnames to OnionCat IPv6 addresses?" is suggested.
OnionCat is based on Tor yet. Since November of 2008 some efforts are undertaken to adapt OnionCat to I2P [18], being another anonymizing network. I2P also provides a SOCKS interface and, obviously, it has an addressing method for its services. Although at a first glance it looks easy to adapt OnionCat to I2P, it introduces a lot of difficulties. Those are encountered when assuming a client, running Tor and I2P in parallel. In fact beside those two, other anonymizing networks exist. This right away leads to the question: "How to create an abstraction layer on top of specific anonymizers being a generic interface in respect to protocol and addressing issues?".
During software development and testing it showed up that the virtual circuits created by Tor are of very low performance. This is, a low bandwidth (2-10 kbytes) and a very high round trip time. This is a potential subject to improvement.
The MTU size of the tunnel device was discussed. It is left at the default value yet, but two interesting questions came up. First, if there is some other MTU size which fits better for this purpose. Second, if OnionCat should do IP fragmentation in order to get interoperability of tunnel devices with different MTU sizes.

For OnionCat in particular, several issues came up. As has been explained in Section 6.2, IPv6 packets are sent directly across the VPN layer. No additional header is prepended for performance reasons. But still, the IPv6 header needs some amount of data. To reduce this, header compression could be implemented. RFC2507 [8] suggests methods for compressing various headers.
When operating in TAP mode, it could be an option to provide DHCP6, in order to support better autoconfiguration.
As has been explained in Section 7.1, authentication of incoming connections is not solved sufficiently. Several ideas have been presented, being proposals for solutions.

These ideas presented within this document and the OnionCat project in particular are subject to create a new generation of the Internet, as we know it today. The user group is very small yet, but it may attract more and more people. It gives back the personal freedom which has been and still is silently restricted by authorities.

A Neighbor Discovery Messages

Figure A.1 shows a complete example of what the solicitation message of A to B looks like. Every field shown as a box has a short descriptive word in the first line, the actual value in the second line and the width of the field in bits in square brackets. One complete line of the diagram has 64 bits which are 8 bytes. Note that the IPv6 addresses are shown as a single line although they have 16 bytes. This is just to save place.
The first line contains the Ethernet header as described in Section 3.2. It has a length of 14 bytes. The lines 3 to 5 contain the IPv6 header as described in Section 3.1. Its length is 40 bytes. The lines 6 to 8 contain the solicitation message in the appropriate format of an ICMPv6 message as described above. It has a length of 32 bytes. The total length of the frame is 86 bytes.
Figure A.2 shows the neighbor advertisement which is the response to the message of Figure A.1. Basically, it has the same structure as the solicitation but it is not a broadcast/multicast message any more. It is unicasted which means it is specifically designated from the advertisement sender to the original solicitation sender.

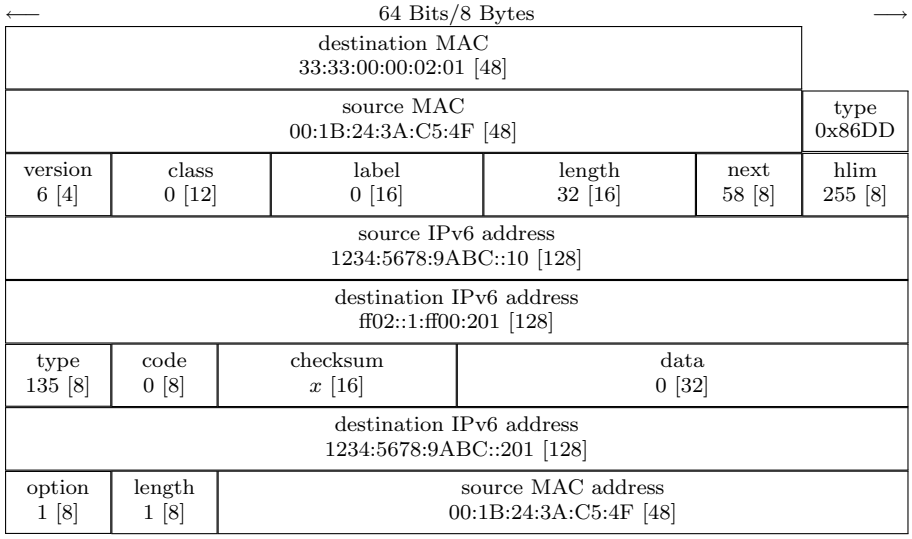


Figure A.1: Neighbor solicitation message.

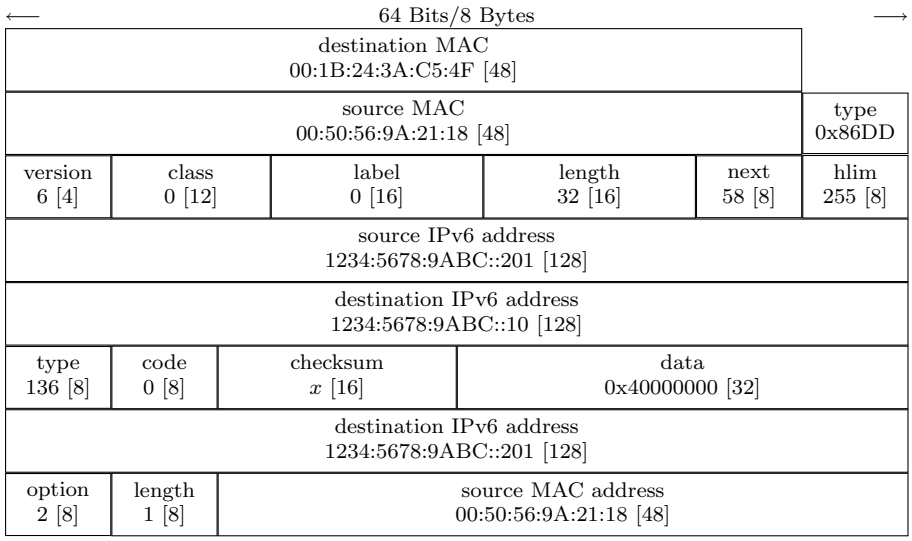


Figure A.2: Neighbor advertisement message.

B Reading from the Windows TAP Driver

```
int win_read_tun(char *buf, int n)
{
    TapData_t *tapData = &tapData_;
    DWORD len = -1, err;
    if (!ReadFile(tapData->fd, buf, n, &len, &tapData->read_overlapped))
    {
        // check if I/O is still pending
        if ((err = GetLastError()) == ERROR_IO_PENDING)
        {
            for (err = WAIT_TIMEOUT; err == WAIT_TIMEOUT;)
            {
                if ((err = WaitForSingleObject(tapData->read_event,
                                SELECT_TIMEOUT * 1000)) == WAIT_FAILED)
                    log_msg(LOG_ERR, "Error = %ld", GetLastError());
            }
            if (!GetOverlappedResult(tapData->fd, &tapData->read_overlapped,
                    &len, FALSE))
            {
                // GetOverlappedResult may fail if buffer was too small
                err = GetLastError();
                log_msg(LOG_WARNING, "Error = %ld", err);
            }
        }
    }
    return len;
}
```

Listing B.1: Non-blocking read on Windows.

Glossary

aio Anonymous Internet Overlay., 79

Asymmetric encryption Encrypting and decrypting data using a key pair of two different keys., 6

Authentication A concept to prove that a piece of information is authentic., 7

Broadcasting An addressing method used to address all hosts on a segment., 17

C A system oriented programming language., 27

Character device A kernel interface on Unix-like OSes., 28

Checksum It allows simple integrity checking of data., 20

Collision Used to refer to the case if two different inputs to a hash function result in the same output., 9

Cygwin A POSIX-like environment for Windows., 76

Data-link layer Layer 2 of the OSI model creating data structures., 11

Digital signature Message authentication with asymmetric cryptography., 9

Driver A low-level module of a Kernel., 27

Endianess The byte order of data during transmission., 16

Ethernet A major layer 2 multi-access protocol., 12

File descriptor Identifier for an I/O channel., 38

Fingerprint An identifier for a piece of data., 9

Forwarding The action of delivering packets to network interfaces., 12

Fragmentation The property of data units being split apart., 65

Frame The data unit used in layer 2., 11

Hash function A function for computing message digests., 9

Hidden Service A service being anonymously hidden within the Tor network., 43

I/O channel Allows data exchange between processes., 38

ICMP Internet control message protocol., 18

IPv6 A network protocol (of layer 3)., 12

IPv6 address A 128 bit wide address used by the IPv6 protocol., 14

Kernel The innermost part of an operating system., 26

MAC (crypto) Message authentication code with symmetric cryptography., 8

MAC (network) Media access control. An intermediate layer of layer 2., 16

MAC address The type of address used by the Ethernet protocol., 16

Message authentication A method for authenticating messages., 8

Message digest An identifier for a piece of data., 9

MTU Maximum Transfer Unit. It is a physical limitation to the maximum size of a frame., 16

Multi-access protocol A protocol based on a physical layer that allows interconnection of more than two hosts at the same time., 11

Multi-processing The capability to run several processes in parallel., 37

Multi-threading The capability to execute several threads within a process in parallel., 37

Multicasting An addressing method used to address a group of hosts on a segment., 17

NDP Neighbor discovery protocol. Used to associate layer 2 addresses to IPv6 addresses., 18

Neighbor advertisement Usually the response to a neighbor solicitation., 19

Neighbor solicitation A multicast message to find specific nodes on a segment., 18

Netmask A special notation to define the length of a prefix., 14

Network layer Layer 3 of the OSI model. Creates logical networks., 12

onion-URL An identifier for hidden services., 43

OS Operating system. The base software running on a computer system., 26

OSI model A model used in the field of networking to categorize network protocols and their dependencies., 11

Packet forwarder A thread handling packets coming in on the tunnel device., 61

Payload The actual information being carried within a data unit., 15

Peer list The list of virtual circuits and their associated IPv6 addresses., 60

Physical layer The lowest layer of the OSI model describing physical parameters., 11

Port number A layer 4 address. It is used at least by UDP and TCP., 13

Prefix The network part of an IPv6 address., 13

Process An atomic unit being executed in the userland., 36

Routing process Decision process of where to forward IP packets., 12

RSA A public key algorithm capable of doing encryption and digital signatures., 9

Segment Usually a set of nodes that are physically connected together., 11

SHA-1 A cryptographic hash function., 9

Socket A communication endpoint of an I/O channel., 27

Socket acceptor A thread accepting incoming connections., 62

Socket cleaner A thread doing housekeeping., 63

Socket receiver A thread handling packets coming in on the virtual circuits., 61

SOCKS A protocol for proxying TCP sessions., 22

SOCKS connector A thread handling setup of outgoing connections., 62

Symmetric encryption Encrypting and decrypting data with a single key., 5

System call A function provided by the OS., 27

TAP mode A tunnel device operating on layer 2, Ethernet in particular., 31

TCP A reliable layer 4 protocol., 13

TCP/IP stack Part of an OS which implements those protocols., 27

Tor An anonymizing network., 43

Transport layer Layer 4 of the OSI model. It creates data streams., 13

Transport protocol Protocols of layer 4., 21

TUN mode A tunnel device operating on layer 3., 31

Tunnel device A kernel driver being used to access the routing process., 28

Tunnel header A header prepended to tunnel device messages., 31

UDP An unreliable layer 4 protocol., 13

Unix A specific type of operating system architecture., 26

User space A different term for userland., 27

Userland The area of a computer system outside of the OS., 27

Virtual circuit A virtual connection between two hosts., 23

VPN A network based on virtual circuits., 23

VPN layer A stack of layers used to encapsulate payload., 24

Bibliography

[1] *The Base16, Base32, and Base64 Data Encodings*, July 2003. `http://www.ietf.org/rfc/rfc3548.txt`.

[2] Braden, R. T., D. A. Borman, and C. Partridge: *RFC 1071: Computing the Internet checksum*, September 1988. `ftp://ftp.internic.net/rfc/rfc1071.txt`, Updated by RFC1141. Status: UNKNOWN.

[3] Chaum, David: *Untraceable electronic mail, return addresses, and digital pseudonyms*. Communications of the ACM, 4(2), February 1981. `http://www.freehaven.net/anonbib/cache/chaum-mix.pdf`.

[4] Crawford, M.: *RFC 2464: Transmission of IPv6 packets over Ethernet networks*, December 1998. `ftp://ftp.internic.net/rfc/rfc2464.txt`, Obsoletes RFC1972. Status: PROPOSED STANDARD.

[5] *The Cygwin Project Page*, 2009. `http://www.cygwin.com/`.

[6] Deering, S. and R. Hinden: *RFC 1883: Internet Protocol, version 6 (IPv6) specification*, December 1995. `ftp://ftp.internic.net/rfc/rfc1883.txt`, Obsoleted by RFC2460. Status: PROPOSED STANDARD.

[7] Deering, S. and R. Hinden: *RFC 2460: Internet Protocol, Version 6 (IPv6) specification*, December 1998. `ftp://ftp.internic.net/rfc/rfc2460.txt`, Obsoletes RFC1883. Status: DRAFT STANDARD.

[8] Degermark, M., B. Nordgren, and S. Pink: *RFC 2507: IP header compression*, February 1999. `ftp://ftp.internic.net/rfc/rfc2507.txt`, Status: PROPOSED STANDARD.

[9] Dierks, T. and C. Allen: *RFC 2246: The TLS protocol version 1*, January 1999. `ftp://ftp.internic.net/rfc/rfc2246.txt`, Status: PROPOSED STANDARD.

[10] Diffie, Whitfield and Martin E. Hellman: *New directions in cryptography.* IEEE Transactions on Information Theory, IT-22(6):644–654, 1976. `citeseer.ist.psu.edu/diffie76new.html`.

[11] Dingledine, Roger, Nick Mathewson, and Paul Syverson: *Tor: The Second-Generation Onion Router*, 2008. `https://www.torproject.org/doc/design-paper/tor-design.pdf`.

[12] Ferguson, N. and B. Schneier: *Practical Cryptography.* Wiley, 2003.

[13] Fischer, Bernhard R.: *The OnionCat Project*, April 2009. `http://www.cypherpunk.at/onioncat/`.

[14] Fischer, Bernhard R. and Ferdinand Haselbacher: *The OnionCat Download Page*, April 2009. `http://www.cypherpunk.at/ocat/download`.

[15] *Free Haven's Selected Papers in Anonymity*, April 2009. `http://www.freehaven.net/anonbib/`.

[16] Fuller, V. and T. Li: *Rfc 4632 classless inter-domain routing (cidr): The internet address assignment and aggregation plan*, August 2006. `http://tools.ietf.org/html/rfc4632`.

[17] Hinden, R. and B. Haberman: *Unique Local IPv6 Unicast Addresses.* RFC 4193 (Proposed Standard), October 2005. `http://www.ietf.org/rfc/rfc4193.txt`.

[18] *I2P Anonymous Network*, April 2009. `http://www.i2p2.de/`.

[19] IANA: *Ethernet Numbers*, September 2008.

`http://www.iana.org/assignments/ethernet-numbers`, Last visited 2009/04/17.

[20] IANA: *Port Numbers*, April 2009.

`http://www.iana.org/assignments/port-numbers`, Last visited 2009/04/17.

[21] Kato, Jun ya: *Cygwin IPv6 Extension Patch*, 2008. `http://win6.jp/Cygwin/`.

[22] Kosiur, Dave: *Virtual Private Networks.* John Wiley & Sons, Inc., 1998.

[23] Lee, Ying Da: *SOCKS 4A: A Simple Extension to SOCKS 4 Protocol.* `http://ftp.icm.edu.pl/packages/socks/socks4/SOCKS4A.protocol`.

[24] Lee, Ying Da: *SOCKS: A protocol for TCP proxy across firewalls*. `http://ftp.icm.edu.pl/packages/socks/socks4/SOCKS4.protocol`.

[25] Lewine, Donald A.: *POSIX programmer's guide: writing portable UNIX programs with the POSIX.1 standard*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1991, ISBN 0-937175-73-0. March 1994 printing with corrections, updates, and December 1991 Appendix G.

[26] Loshin, Pete: *IPv6, Second Edition: Theory, Protocol, and Practice, 2nd Edition (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003, ISBN 1558608109.

[27] Mockapetris, P. V.: *RFC 1034: Domain names — concepts and facilities*, November 1987. `ftp://ftp.internic.net/rfc/rfc1034.txt`, Obsoletes RFC0973, RFC0882, RFC0883 See also STD0013. Updated by RFC1101, RFC1183, RFC1348, RFC1876, RFC1982, RFC2065, RFC2181, RFC2308. Status: STANDARD.

[28] *Microsoft Developer Network, Online Documentation*, 2009. `http://msdn.microsoft.com/en-us/library/`.

[29] Narten, T., E. Nordmark, and W. Simpson: *RFC 2461: Neighbor discovery for IP Version 6 (IPv6)*, December 1998. `ftp://ftp.internic.net/rfc/rfc2461.txt`, Obsoletes RFC1970. Status: DRAFT STANDARD.

[30] Nichols, Bradford, Dick Buttlar, and Jacqueline Proulx Farrell: *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996, ISBN 1-56592-115-1.

[31] Oikarinen, J. and D. Reed: *RFC 1459: Internet Relay Chat Protocol*, May 1993. `ftp://ftp.internic.net/rfc/rfc1459.txt`, Status: EXPERIMENTAL.

[32] *The OpenVPN Project*, 2009. `http://openvpn.net/`.

[33] Postel, J.: *RFC 768: User datagram protocol*, August 1980. `ftp://ftp.internic.net/rfc/rfc768.txt`, Status: STANDARD. See also STD0006.

[34] Postel, J.: *RFC 791: Internet Protocol*, September 1981. `ftp://ftp.internic.net/rfc/rfc791.txt`, Obsoletes RFC0760. See also STD0005 Status: STANDARD.

[35] Postel, J.: *RFC 793: Transmission control protocol*, September 1981. `ftp://ftp.internic.net/rfc/rfc793.txt`, See also STD0007. Status: STANDARD.

[36] Rekhter, Y., B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear: *Address Allocation for Private Internets*. RFC 1918 (Best Current Practice), February 1996. `http://www.ietf.org/rfc/rfc1918.txt`.

[37] Rieger, Gerhard: *socat - Multipurpose relay*, 2007. `http://www.dest-unreach.org/socat/`.

[38] Russinovich, Mark: *WinObj*, November 2006.

`http://technet.microsoft.com/en-us/sysinternals/bb896657.aspx`, Last visited 2009/04/14.

[39] Schneier, Bruce: *Applied Cryptography: Protocols, Algorithms, and Source Code in C, Second Edition*. Wiley, October 1995, ISBN 0471117099.

[40] Stallings, William: *Operating Systems (5th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2004, ISBN 0131479547.

[41] Stephen Hemminger: *Net:bridge*, 2009.

`http://www.linuxfoundation.org/en/Net:Bridge`.

[42] Tanenbaum, Andrew S.: *Computer Networks, Fourth Edition*. Prentice Hall PTR, August 2002, ISBN 0130661023.

[43] *The Tor Project*. `http://www.torproject.org/`.

[44] *Tor project download page*, April 2009.

`http://www.torproject.org/easy-download.html.en`.

[45] *The Single UNIX Specification*. `http://www.unix.org/online.html`, 2009.

[46] Vaughan, Gary V. and Thomas Tromey: *GNU Autoconf, Automake and Libtool*. New Riders Publishing, Thousand Oaks, CA, USA, 2000, ISBN 1-57870-190-2.

[47] Wang, Xiaoyun, Yiqun Lisa Yin, and Hongbo Yu: *Finding Collisions in the Full SHA-1*, 2005. `http://people.csail.mit.edu/yiqun/SHA1AttackProceedingVersion.pdf`.