TECHNISCHE
UNIVERSITÄT
DARMSTADT

Introduction to RAW-sockets

Jens Heuschkel, Tobias Hofmann, Thorsten Hollstein, Joel Kuepper

16.05.2017

Technical Report No. TUD-CS-2017-0111
Technische Universität Darmstadt

Telecooperation Report No. TR-19,
The Technical Reports Series of the TK Research Division, TU Darmstadt
ISSN 1864-0516
http://www.tk.informatik.tu-darmstadt.de/de/publications/

Introduction to RAW-sockets

by
Heuschkel, Jens
Hofmann, Tobias
Hollstein, Thorsten
Kuepper, Joel

May 17, 2017

Abstract

This document is intended to give an introduction into the programming with RAW-sockets and the related PACKET-sockets. RAW-sockets are an additional type of Internet socket available in addition to the well known DATAGRAM- and STREAM-sockets. They do allow the user to see and manipulate the information used for transmitting the data instead of hiding these details, like it is the case with the usually used STREAM- or DATAGRAM sockets. To give the reader an introduction into the subject we will first give an overview about the different APIs provided by Windows, Linux and Unix (FreeBSD, Mac OS X) and additional libraries that can be used OS-independent. In the next section we show general problems that have to be addressed by the programmer when working with RAW-sockets. We will then provide an introduction into the steps necessary to use the APIs or libraries, which functionality the different concepts provide to the programmer and what they provide to simplify using RAW and PACKET-sockets. This section includes examples of how to use the different functions provided by the APIs. Finally in the additional material we will give some complete examples that show the concepts and can be used as a basis to write own programs. The examples are programmed in C++ and we assume that the reader has basic programming skills and networking knowledge to be able to understand the listings and content of this document.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

First of all we want to define the basic wordings and concepts used. Then the reader is given a short overview about the possibilities and restrictions the available APIs impose on the user.
Internet sockets are the common way to perform network communication implemented in most operating systems. They are usually provided by a socket API and are based upon the same principles as reading and writing a file. A program can get a socket via a function provided by the operating system. This function then returns a socket descriptor, usually a simple integer, similar to the ones provided by most operating systems for Read- and Write-Operations on files. This socket descriptor then can be used to write or read data from the socket. The data written to the socket is encapsulated by the operating system by adding headers and trailers and then the complete packet is sent via the network interface over the network to the target host. The data that has been received from the socket is presented to the program without the headers and trailers, only the user data is presented to the user. The sockets that work this way are the DATAGRAM- or STREAM-sockets provided via the Berkeley socket API. The user is unaware of the communication between the lower layers. All network communication steps, like for example the connection establishment, are taken care of without knowledge of the user. The user is only responsible for creating the socket and then providing the data that he wants to send to the correct function.

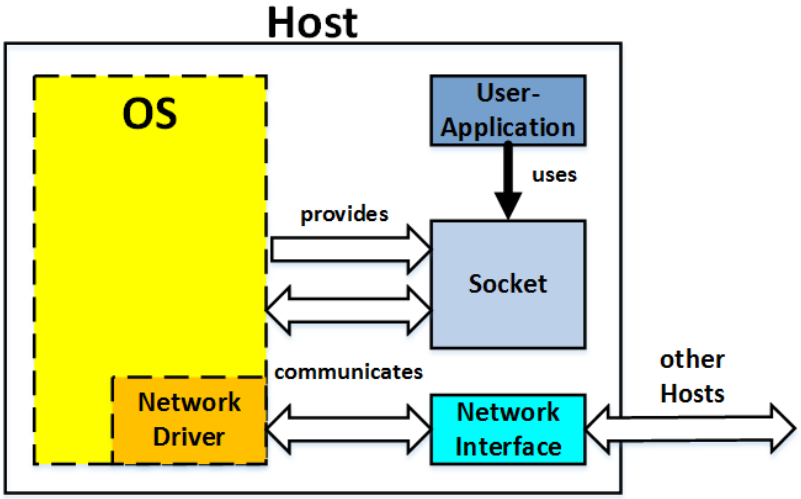

Figure 1: Socket provided by the operating system

The only parameters the user has to set are:

- The source socket address, a combination of IP address and port number can be set usually via the `bind()` function or defined directly with one of the `send()` functions.
- By choosing either the DATAGRAM- or STREAM-socket we can choose if we want to have a separate datagrams or a byte stream. This also chooses the Transport Layer Protocol that is being used (UDP or TCP).

If we want to gain access to the data of the lower layers we have several possibilities: RAW-sockets, PACKET-sockets, network drivers and Data Link layer APIs. The programming of Network Drivers will not be discussed further in this work, since we want to have a look at portable solutions that work for different operating systems. What can be accessed by the APIs we will present is shown in figure 2 on the following page.

With these APIs it is possible for an application to change and access the fields of the Network layers that are used for sending the data. This might be seen as a break with the traditional layering model, since we can influence the service the lower layers provide.

1.1 RAW-sockets

RAW-sockets are part of the standard Berkeley sockets and the socket API that is based upon it [1]. They are another option in addition to the already mentioned DATAGRAM- or STREAM-sockets to create data packets with the socket API [1]. In addition to simply sending data and defining address information the RAW-socket allows the user to access and manipulate the header and trailer information of the lower layers, more specifically with RAW-sockets to the Network and Transport layer (layer 3 and 4 of the OSI model). Since RAW-sockets are part of the Internet socket API they can only be used to generate and receive IP-Packets.

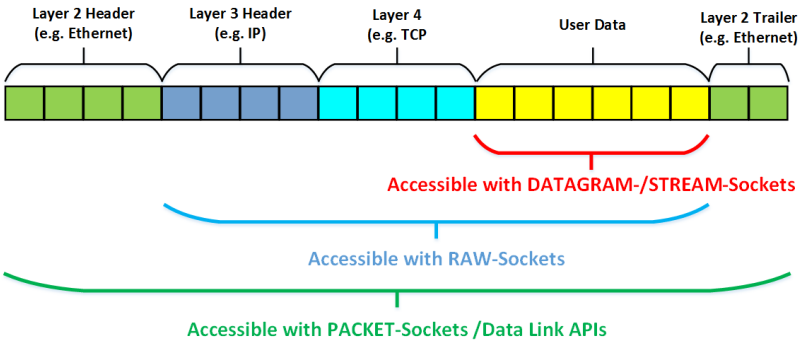

Figure 2: Overview over the layers and access possibilities

The biggest problem with RAW-sockets (also the PACKET-sockets we discuss later) is that there is no uniform API for using RAW-sockets under different operating systems:

- The provided APIs differ in regard to the used byte order. Depending on the operating system the fields of the packets have to be filled with data in network- or host-byte-order.
- Differences also exists in the types of packets and protocols that can be created using the RAW-socket API.
- The usage and functionality of the APIs is also different for each operating system.
- Some operating systems do not allow certain packet types to be received using the RAW-socket API.
- There are also differences in the definitions and paths of the necessary header files provided by each operating system.
- The required access levels for using the RAW-socket API can also differ. However most operating systems require root or admin access permissions to use them.

The user has to keep these things in mind if he wants to use the RAW-socket API, especially if he plans to use them across different operating systems.

1.2 PACKET-sockets and Data Link Layer APIs

If the user also wants to access the header and trailer of the Data Link layer then a Data Link Layer API has to be used. Under Linux (and Unix as well) this a functionality is provided by the operating system as the so called PACKET-socket and can be seen as a special form of RAW-sockets, so the limitations and problems of the previous section also apply to them [1]. But since this is not a standardized functionality there might be a different API provided by the operating system to perform these operations. For example under BSD the so called BSD Packet Filter (BPF) not the PACKET-socket API provides access to the Data Link Layer [2]. In general all of these APIs usually allow access to the whole packet from the Data Link Layer (OSI Layer 2), the Network Layer (OSI Layer 3) up to the Transport layer (OSI Layer 4). Not all operating systems provide a simple interface to access the Data Link Layer. As an alternative we introduce a library that can be used to provide this functionality.

2 Implementations for different Operating Systems

First we want to give a short overview over some of the APIs and libraries that are available for RAW-socket and Data Link Layer network programming. In general we want to split the available libraries into two parts, the APIs provided by the operating systems and independent libraries that are available for multiple operating systems.

2.1 Windows

Windows as a operating system is pretty restricted when it comes to RAW-socket, PACKET-socket and Data Link Layer programming. In general it is difficult to get the programs running under Windows and the available options are pretty limited. For that reason in general 3rd party libraries are recommended if the user wants to do network programming on Windows systems and still write portable code.

2.1.1 Winsock-API

Winsock, currently available in version 2, is the Windows implementation of the Berkeley socket. Winsock is based on the implementation of sockets on Linux and BSD and allows both sending and receiving of RAW-Packets, but adds additional functionality.

Nevertheless, in the latest operating system versions (Windows XP and newer), there are the some restrictions the programmer has to consider according to the MSDN-library [3] for developers:

- Like in most operating system the Winsock-API also need admin access permits to work properly for RAW-sockets.
- A call to the bind function with a RAW-socket for the TCP protocol is not allowed, in fact TCP data cannot be sent over a RAW-socket at all.
- UDP datagrams with an invalid source address cannot be sent over RAW-sockets.
- The IP source address for any outgoing UDP datagram must exist on a network interface or the datagram is dropped. This change was made to limit the ability of malicious code to create distributed denial-of-service attacks and to limit the ability to send spoofed packets (TCP/IP packets with a forged source IP address).
- Especially receiving is much slower with RAW-sockets than using a customized driver based on the lower network layers, since all traffic is received (this consideration is the same for any other operating system).

2.1.2 winpcap

The ability of Winsock to also manipulate the Data Link Layer was removed in the current versions. Instead Windows does require to program a driver with the required functionality. With Network Driver Interface Specification (NDIS) version 6.40 (currently) the Operating system provides an API to do this if needed. This is now the preferred way to write own applications that want to access the Data Link Layer on Windows. Since want to only look at APIs and libraries provided by the operating system, we want to point to winpcap and libnet at this point as an alternative to the Winsock-API. Winpcap is the windows port of libpcap and allows generation and receiving of packets from the Data Link Layer upwards. Libnet is a network programming library available for many operating systems that allow easy sending of RAW-packets. All further details of winpcap and libnet can be found under their sections, if there are differences using the windows versions of the libraries they will be pointed out accordingly [4].

2.2 Linux

Linux is one of the operating systems for which it is easy to do RAW-socket and Data Link Layer Programming. It provides the APIs to do both, but the kernel has to be compiled with the option to support the options.

2.2.1 RAW-sockets

The RAW-socket is included in the socket-API as we already discussed. It allows both sending and receiving of RAW-packets, however the following items still have to be observed:

- Due to essential nature of header information for networking functionality and security using RAW-sockets requires root access permissions [5].
- The ports of the network layer are not endpoints anymore since RAW-sockets work on the layers below. Filtering based on ports has to be done manually [5].

- The `bind()` and `connect()` functions are no longer necessary [5]. `bind()` and textttconnect() can still be used to the define source address and target address to be entered automatically by the kernel [5]. Additionally a raw socket can be bound to a network device using `SO_BINDTODEVICE`.
- The `listen()` and `accept()` functions are without function, since the client-Server-Semantic is no longer present [5]. When we use RAW-sockets we are sending unconnected packets [5].
- IP-headers of RAW sockets can be manually created by the programmer if the option `IP_HDRINCL` is enabled [1]. This way, Raw sockets allow a programmer to implement new IP based protocols. If `IP_HDRINCL` is not enabled the IP header will be generated automatically.
- When a RAW socket is created any IP based protocol can be specified [1]. This results in a socket only receiving messages of the type of the specified protocol.
- If a programmer does not want to specify a protocol when creating a RAW-socket he can also use the `IPPROTO_RAW` protocol (which implies that the headers will be created manually) [1]. This way he can send any IP based protocol. However this socket is not able to receive any IP packets, to receive all IP based packets a PACKET-socket has to be used.

2.2.2 PACKET-sockets

PACKET-sockets are a special type of RAW-sockets that allow access to the fields of the Data Link Layer. To use them there are some definitions required that not part of the socket-API, but are provided by the Linux Kernel. So while they are integrated in the socket-API of Linux, they are dependent on our operating system [5].
Similar restrictions than for the RAW-sockets have to be considered:

- Similar to RAW-sockets, due to essential nature of header information for networking functionality and security reasons using PACKET-sockets also requires root access permission [5].
- The ports of the Network layer are not endpoints anymore, since RAW-sockets work on the layers below [5]. Filtering based on ports has to be done manually.
- The `bind()` and `connect()` functions are no longer necessary [5]. `bind()` can still be used to define the interface over which the Data Link Layer packet should be sent [5]. `connect()` has no function anymore [5].
- The `listen()` and `accept()` functions are no use at all since the Client-Server-Semantic is no longer present [5]. When we use RAW-sockets we are sending unconnected packets.

2.3 Unix (FreeBSD, Mac OS X)

Like Linux under Unix operating systems there are also RAW-sockets provided with the Unix kernel. RAW-sockets are however more restricted than under Linux. Access to the Data Link Layer is possible in Unix via the Berkeley Packet Filters (BPF) provided by the operating system.

2.3.1 RAW-sockets

RAW-sockets are included in the socket-API in Unix like they are in Linux. In Unix there are different header files available and needed than in Linux, that makes some considerations necessary to keep code portable between these two operating systems [1]. Also there are some differences to the functionality.
Similar to Linux there are some restrictions to take into consideration:

- It is not possible to read packets for anything that has a handler (like TCP or UDP), but it is possible to read packets for other protocols like ICMP [6].
- For BSD and its ports it could be that the Packets are modified by the operating system before sending. For example with the release 10 the IP length is modified to the actual size of the IP header regardless of what is set by the programmer.
- Due to essential nature of header information for networking functionality and security reasons using RAW-sockets requires root access permission [5].
- The ports of the Network layer are not endpoints anymore, since RAW-sockets work on the layers below [5]. Filtering based on ports has to be done manually.
- The `bind()` and `connect()` functions are no longer necessary [5]. `bind()` and `connect()` can still be used to define source address and target address to be entered automatically by the kernel [5].

- The `listen()` and `accept()` functions are no use at all since the Client-Server-Semantic is no longer present [5]. When we use RAW-sockets we are sending unconnected packets.

2.3.2 Berkeley Packet Filter (BPF)

The Berkeley Packet Filter is the API provided by Unix operating systems to allow the user access to the Data Link Layer. It is compiled into the kernel of Unix-like hosts and allows access to all packets received at the Network Interface Controller (NIC) [7]. It has a built in filtering mechanism that allows the user to filter the received traffic for the packets he is interested in [7]. Since the RAW-socket implementation does not support receiving of packets that have a handler (like TCP or UDP) the Berkeley Packet Filter has to be used if the user wants to receive these packets [7].

2.4 OS independent

Other than the OS dependent APIs and the standard APIs provided by the standard Berkey socket API there are also universal packet capturing and creation libraries. Of those we want to introduce two libraries, namely **libpcap** and **libnet** which are available for many operating systems (e.g. Windows, OS X, BSD, Linux) and therefore can make programming with RAW-sockets more portable. The libraries complement each other **pcap** is usually used to receive the packets while **libnet** provides easy mechanisms to send packets.

2.4.1 pcap

Pcap is an open library for packet capture. It allows the user to receive and filter all packets received at the network interface from the Data Link Layer upwards. It is available across different platforms, for Unix and Linux systems there exists libpcap (library for packet capture), the library for Windows is called Winpcap. It also supports injection of layer 2 packets, but the use of this functionality is very limited and not very comfortable. The use of the library might require admin or root permissions to work, depending on the operating system. One of the most well known programs that uses pcap is Wireshark.

2.4.2 libnet

Libnet is a library for constructing and sending packets from the Data Link Layer upwards. It is intended as the counterpart for pcap. What it does provide is a simple, modular interface for packet construction and injection. It gives programmers many helpful functions to make the construction of own packets very easy. Currently over 30+ protocols are supported by the library and it is available for many operating systems.

3 Programming with the APIs

3.1 General

We start by discussing some general topics that have to be considered for the development with any of the following APIs. Other than the topics presented here, it should be noted that the programmer also has to observe the particularities of the network protocols he wants to use. If these are not observed, like for example the correct order in that protocol information has to be exchanged, it cannot be ensured that we are able to communicate with the other host since the results than can be very different based on the implementation of the the target host.

3.1.1 Byte Order

In network communication the byte-order, also known as endianness, is essential for setting and interpreting the fields correctly. It comes into play whenever a value does need more than 1 byte of storage space. Whenever this is the case we have to translate between network-byte-order and host-byte-order. The network-byte-order and host-byte-order may may be different depending on the used protocol and operating system. For the IP-protocol the network-byte-order is big-endian, so we have to translate whenever the byte-order of the host is not big-endian [5]. So we have two cases:

- When we send network packets all data that needs more than 1 byte it has to be translated from host-byte-order to network-byte-order.
- When we receive data on a socket all data that needs more than 1 byte it has to be translated from network-byte-order to host-byte-order.

For the transformations used for the IP-protocol the Berkeley socket API provides a set of standard functions which are available in the header file `<arpa/inet.h>`:

Function	Description
`uint32_t htonl(uint32_t` **`hostlong`**`)`	host-to-network
`uint16_t htons(uint16_t` **`hostshort`**`)`	host-to-network
`uint32_t ntohl(uint32_t` **`netlong`**`)`	network-to-host
`uint16_t ntohs(uint16_t` **`netshort`**`)`	network-to-host

Table 1: Byte-Order Transformation Functions [8]

With **Linux** we can be sure that the data receive/sent over a socket always is in network-byte-order [5].

For **Unix** it can be the case that some fields of the header are in host-byte-order depending of the release (for example BSD) [5].

3.1.2 Checksum

Many protocols used for network communication use a checksum to be able to detect transmission errors. There are different ways to calculate a checksum. If we choose to generate the packet headers completely by our self we have to calculate and set the applicable header fields manually. There are some points to take into consideration if we do this:

- Which algorithm has to be used to compute the checksum.
- Which fields of the header, sometimes even of multiple headers, have to be included in the calculation of the checksum [5]. Of course the user data might have to be included in the calculation as well.
- If we want to calculate the checksum all header information already has to be set with the correct data.
- Do we have to calculate the checksum or can it be calculated automatically by the network driver/hardware (for example the Ethernet checksum), the operating system or a predefined function call [5].

As a possible code example for the calculation of a checksum here is the listing for computing the Internet-checksum which was provided with the RFC 1071 which can be found in the appendix.

3.1.3 Type-Casting

One of the basic functionalities of the C programming language is that it supports Type-Casting. The ability to cast binary data to a data structure `struct` in C is essential to make working with header data easier for the programmer. To perform type casting we simply create a pointer to the desired structure:

```
struct header* protocolHeader;
```

Then we do need a pointer to the memory space where the binary data of the packet is stored, in our examples mostly a *char*-Array with a fixed *SIZE*[5]:

```
char buffer[SIZE];
```

Finally we can cast the binary data to our **struct** and afterwards are able to access the header fields[5]:

```
protocolHeader = (struct header*) buffer;
```

When accessing the header-field we still have to take care of the byte order like mentioned before. For the majority of the well known protocols there are predefined structs available in the header files included in the libraries. Some of the available headers are shown in the table below, but it does only show a part of the available header structures. More headers may be available depending on the operating system and the library.

Linux Under Linux typically the following header files are available:

Header-File	Description
`<netinet/ip.h>`	Defines macros, variables, and structures for IP.
`<netinet/ip_icmp.h>`	Defines macros, variables, and structures for ICMP.
`<netinet/udp.h>`	Defines macros, variables, and structures for UDP.
`<netinet/tcp.h>`	Defines macros, variables, and structures for TCP.
`<netns/idp.h>`	Defines IPX packet headers.
`<netns/sp.h>`	Defines SPX packet header.
`<ssl.h>`	Defines SSL prototypes, macros, variables, and structures.

Table 2: C-Header-Files for Network Headers [8]

3.1.4 Header-Positions

Another issue that has to be taken into consideration is the position of the packet headers in the binary data. The following has to be taken into consideration:

- Like it was shown in the beginning the header of a lower layer always encapsulate the header of the higher layers and the data.
- The header of the lowest layer can always be found at the beginning of the binary data packet. It might be the case that not all fields are present, for example this is the case for the Ethernet-Protocol in the Data Link layer. For the Ethernet-Protocol the preamble, start of frame delimiter and the frame checksum are removed by the network driver since they do belong more to the Physical Layer (OSI Layer 1)[5].
- If we type cast binary data to structs we have to add the length of the last header to the data pointer to get the beginning of the next header [5]:

  ```
  next_header = (struct header*) (buffer + sizeof(struct header));
  ```

- To find the correct start position might be a bit more difficult than a simple add operation as shown here if the last header has a variable length with optional fields (like for example the IP-header or the TCP-header). In that case we must first get the fixed part of the header, then access the length field and finally compute the length of the optional header. Only after doing this we are able to find the start of the next header. If we are interested in the content of the optional fields we also have to access the header field containing the type of optional fields and cast it to the correct type accordingly.

3.2 RAW-sockets

The first API discussed here is the RAW-socket. It is available on Linux and Unix Systems. As already mentioned the PACKET-sockets can be seen as a specialized form of RAW-sockets and just allow access a lower layer then the normal RAW-socket. RAW-sockets allow access to the Network (OSI layer 4) and Internet (OSI layer 3) layer of the network stack [5]. The use of RAW-sockets is limited to processes with an effective user ID of 0 or the `CAP_NET_RAW` capability since they require **root**-access permissions [1]. In the following we will start by giving a generic overview over the functions available and necessary to create and work with RAW-sockets. After that we show the structure of different protocol level write and read operations.

3.2.1 `socket()`

Both the read and write to a RAW-socket require the socket to be created first. For the creation of a socket the same function as for normal sockets is used. It is available in the `<netinet/in.h>` header and has the following form [8]:

```
int socket(int family, int type, int protocol)
```

The following parameters have to be defined:

- **`family`** expects a constant value that describes the used address family [8]. The following values are defined in `<sys/socket.h>`:

Constant	Description
`AF_LOCAL`	Local communication
`AF_UNIX`	Unix domain sockets
`AF_INET`	IP version 4
`AF_INET6`	IP version 6
`AF_IPX`	Novell IPX
`AF_NETLINK`	Kernel user interface device
`AF_X25`	Reserved for X.25 project
`AF_AX25`	Amateur Radio AX.25
`AF_APPLETALK`	Appletalk DDP
`AF_PACKET`	Low level packet interface
`AF_ALG`	Interface to kernel crypto API

Table 3: Address family constants provided in `<sys/socket.h>` [8]

The function only passes errors originating from the network to the user when the socket is connected. In that case only `EMSGSIZE` and `EPROTO` are passed for compatibility. If the `IP_RECVERR` flag is enabled, all network errors are saved in the error queue. It returns a non-negative socket-descriptor if it was created successful and -1 if an error occurred during creation of the socket. Additionally the variable `errno` defined in `<errno.h>` is set and can have the following values [8]:

Flag	Description
`EACCES`	User tried to send to a broadcast address without having the broadcast flag set on the socket.
`EFAULT`	An invalid memory address was supplied.
`EINVAL`	Invalid argument.
`EMSGSIZE`	Packet too big. Either Path MTU Discovery is enabled or the packet size exceeds the maximum allowed IPv4 packet size of 64KB.
`EOPNOTSUPP`	Invalid flag has been passed to a socket call.
`EPERM`	The user doesn't have permission to open raw sockets.
`EPROTO`	An ICMP error has arrived reporting a parameter problem.

Table 4: Errno flags for `socket()` as defined in `<errno.h>` [8]

Users have to be aware that setting the **`family`** of course also influences which **`protocol`** can be chosen later. The usual option to use with RAW-sockets is the `AF_INET` to send and receive IP version 4 packets.

- **type** defines the socket type. The following values are defined in <sys/socket.h>:

Constant	Description
SOCK_STREAM	Stream (connection) socket
SOCK_DGRAM	Datagram (connection-less) socket
SOCK_RAW	RAW socket
SOCK_RDM	Reliably-delivered message
SOCK_SEQPACKET	Sequential packet socket
SOCK_PACKET	Linux specific way of getting packets at the dev level.

Table 5: The socket types are defined in <sys/socket.h>[8]

Since we want to work with RAW-sockets we only use the SOCK_RAW constant in the following sections.
From Linux kernel 2.6.27, there is a second purpose for the type argument, it additionally allows to modify the behavior of the socket by including the following options with bit-wise OR [8]:

  - SOCK_NONBLOCK: Sets the O_NONBLOCK file status flag on the new open socket descriptor. Using this changes the socket to a non-blocking one.
  - SOCK_CLOEXEC: Sets the close-on-exec flag for the new socket descriptor. Useful if the program creates child processes which use exec() to run another program. In that case this flag will prevent the other program from using this socket descriptor.

- **protocol** defines like the name implies the protocol of the packets that can be sent and received with the socket [8]. The protocol numbers are defined by the IANA (Internet Assigned Numbers Authority) and a complete list can be found on their website. The table **??** shows the constants for protocols which are defined in the <netinet/in.h> header file.

  Again, users have to be aware of which **family** was chosen for the first option, since only protocols of that family can be chosen as **protocol**. So with our usual AF_INET option selected we can only use IP-based protocols [8].
  Also it should already be noted here that the IPPROTO_RAW constant in for most operating system (depends on the Linux/Unix distribution) does imply that the socket also expects an IP header to be manually created by the user [8]. If we chose this constant we have layer 3 write access as a result. The regular way to do this is to use another constant and then to change the socket options with setsockopt() as described in section 3.2.3 on the next page.

  In addition to these constants we could also use constants that are defined for layer 2 (PACKET-sockets) to access their protocol information. These constants are operating system dependent and therefore code that uses them cannot be ported as easy as the general constants from the previous table. As an example, table 48 in the appendix shows the constants in Linux for Ethernet protocols which are defined in the <linux/if_ether.h> header.

Unix The socket() function is POSIX and 4.4BSD conform [8]. The SOCK_NONBLOCK and SOCK_CLOEXEC flags are Linux-specific [8]. The function is generally portable to BSD systems, but some systems require the <sys/types.h> header to work [8]. The manifest constants might also be different under some BSD versions.

3.2.2 setsockopt()

The setsockopt() function like the name implies can be used to change the options that are selected for the socket. The function can manipulate the options for different protocol levels such as IP or TCP, but also for the sockets level API by setting the level to SOL_socket. The function which is defined in the <sys/socket.h> header has the following form [8]:

int setsockopt(*int* **sockfd**, *int* **level**, *int* **optname**, *const void ** **optval**, *socklen_t* **optlen**)

The following parameter can be set [8]:

- **sockfd** - Specifies the socket for which the options should be set.
- **level** - The protocol level of the option we want to set.

- **optname** - The name of the option we want to set. Together with the **optval** and **optlen** it is passed uninterpreted to the protocol module for processing.
- **optval** - The buffer for the option we want to specify, usually an Integer. It should then be non-zero to enable a Boolean option and zero to disable it.
- **optlen** - The length of the buffer that **optval** points to in bytes.

We want to only give an short overview over the most interesting options for us, the socket level and the IP protocol level, which can be set with the `setsockfunction()` function. Additional information about the use and meaning of the options can be found in the programmer manuals for the protocols. They exist for all protocols available for the respective operating system or distribution.

For the socket level API that can be accessed with `SOL_socket` as the level, the options in table 49 in the appendix are supported and already included in the `<sys/socket.h>` header.

For the IP level API that can be accessed with `IPPROTO_IP` as the level, table 50 in this appendix shows the following options supported and already included in the `<netinet/ip.h>` header.

The function returns zero on success and -1 on an error [8]. Additionally the variable `errno` defined in `<errno.h>` is set and can have the following values:

Flag	Description
EBADF	The argument sockfd is not a valid descriptor.
EFAULT	The address pointed to by optval is not in a valid part of the process address space.
EINVAL	optlen invalid or invalid optval.
ENOPROTOOPT	The option is unknown at the level indicated.
ENOTSOCK	The argument sockfd is a file, not a socket.

Table 6: Errno flags for `setsockopt()` as defined in `<errno.h>` [8]

Unix The function is POSIX and 4.4BSD conform [8]. According to the POSIX-standard the use of `setsockopt()` does only require to include `<sys/socket.h>` header, so under Linux it can be used by only including this single header [8]. To use it under Unix also require the `<sys/types.h>` header to be included [8]. For portability it is generally recommended to include both.

3.2.3 `getsockopt()`

For retrieving an specified option of the socket the `getsockopt()` function defined in the `<sys/socket.h>` header can be used [8]:

int getsockopt(*int* **sockfd**, *int* **level**, *int* **optname**, *void ** **optval**, *socklen_t ** **optlen**)

The following parameter have to be set [8]:

- **sockfd** - Specifies the socket for which the options should be retrieved.
- **level** - The protocol level of the option we want to retrieve.
- **optname** - The name of the option we want to retrieve. Together with the **optval** and **optlen** it is passed uninterpreted to the protocol module for processing.
- **optval** - The buffer for the option we want to retrieve.
- **optlen** - The length of the buffer that **optval** points to in bytes. Initially it should contain the length of the buffer, after the call it indicates the actual size of the value returned. In case no value is supplied or returned it might return `NULL`.

The function returns zero on success and -1 on an error [8]. Additionally the variable `errno` defined in `<errno.h>` is set and can have the following values:

Flag	Description
EBADF	The argument sockfd is not a valid descriptor.
EFAULT	The address pointed to by optval or optlen is not in a valid part of the process address space.
EINVAL	optlen invalid or invalid optval.
ENOPROTOOPT	The option is unknown at the level indicated.
ENOTSOCK	The argument sockfd is a file, not a socket.

Table 7: Errno flags for getsockopt() as defined in <errno.h> [8]

Unix The function is POSIX and 4.4BSD conform [8]. According to the POSIX-standard the use of getsockopt() does only require to include <sys/socket.h> header, so under Linux it can be used by only including this single header [8]. To use it under Unix also require the <sys/types.h> header to be included [8]. For portability it is generally recommended to include both.

3.2.4 bind()

After creating a socket like discussed in the previous section 3.2.1 on page 15, we can bind the created socket to a specific address. Traditionally this is also called *assigning a name to a socket*. For RAW-sockets and PACKET-sockets this is optional, but we use it to define the source address of our packets with it and also define from which network-interface we want to read packets. Other socket types might require to be bound to a specific address before they can be used. For binding the socket to an IP-address we can use the bind() function which is defined in the <sys/socket.h> header and has the following form [8]:

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

The following parameters can be set [8]:

- **sockfd** - Specifies the socket for which the address should be set.
- **addr** - The address structure containing the address to be set.
- **addrlen** - The length of the address structure in bytes.

The function returns zero on success and -1 on an error [8]. Additionally the variable errno defined in <errno.h> is set and can have the following values:

Flag	Description
EACCES	The address is protected, and the user is not the superuser.
EADDRINUSE	The given address is already in use.
EADDRINUSE	All port numbers in the port range are currently in use.
EBADF	sockfd is not a valid file descriptor.
EINVAL	The socket is already bound to an address.
EINVAL	addrlen is wrong, or addr is not a valid address for this socket's domain.
ENOTSOCK	The file descriptor sockfd does not refer to a socket.

Table 8: Errno flags for bind() as defined in <errno.h> [8]

Unix According to the POSIX-standard the use of bind() does only require to include <sys/socket.h> header, so under Linux it can be used by only including this single header [8]. To use it under Unix also requires the <sys/types.h> header to be included [8]. For portability it is generally recommended to include both. For a UNIX domain (AF_UNIX) the following errno are additionally defined [8]:

Flag	Description
EADDRNOTAVAILA	Nonexistent interface was requested or the requested address was not local.
ENAMETOOLONG	addr is too long.
ENOMEM	Insufficient kernel memory was available.

Table 9: Errno flags for UNIX-bind() as defined in <errno.h> [8]

3.2.5 `getsockname()`

To get the currently defined name of a socket the `getsocketname()` function can be used. It is defined in the `<sys/socket.h>` and has the following form [8]:

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

The following parameters can be set [8]:

- **`sockfd`** - Specifies the socket for which the address should be retrieved.
- **`addr`** - The address structure for the returned address that is set for the socket. The returned address is truncated if the buffer is too small.
- **`addrlen`** - The length of the address structure in bytes. It should be initialized to the size of the buffer pointed to by **`addr`**. Contains the actual size of the socket address after return.

The function returns zero on success and -1 on an error [8]. Additionally the variable `errno` defined in `<errno.h>` is set and can have the following values:

Flag	Description
`EBADF`	The argument sockfd is not a valid file descriptor.
`EFAULT`	The addr argument points to memory not in a valid part of the process address space.
`EINVAL`	addrlen is invalid.
`ENOBUFS`	Insufficient resources were available in the system to perform the operation.
`ENOTSOCK`	The file descriptor sockfd does not refer to a socket.

Table 10: Errno flags for `getsocketname()` as defined in `<errno.h>` [8]

Unix The function is compliant to POSIX and 4.4BSD [8]. In some distributions the third argument is in reality a pointer to an `int`[8]. This could still be the case in some distributions.

3.2.6 `connect()`

This function can be used to initiates a connection to a specific destination host and is required for the `write()`, `send()`, `read()` and `recv()` functions. For connection based protocols like `SOCK_STREAM` this function also will try to connect to the host on the other side [8]. For Datagram based protocols only the default destination is defined with this function [8]. To use this function the header `<sys/socket.h>` (for the function) has to be included. The function is called like this [8]:

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

With the following parameters that can be set [8]:

- **`sockfd`** - Specifies the socket for which the address should be set.
- **`addr`** - The function specifies a *`struct sockaddr *`* with this parameter that contains the socket address family and the protocol address for that family. With this the address to connect to can be defined.
- **`addrlen`** - Defines how long the address that is passed to the function is.

The function returns zero on success and -1 on an error [8]. Additionally the variable `errno` defined in `<errno.h>` is set and can have the values sown in table 51 in the appendix.

Unix The `connect()` function might require the `<sys/types.h>` header for some BSD systems [8]. Also the third argument might be an `int` depending on the system version [8].

3.2.7 Read

We can access the socket with three different function to retrieve data from it. For the `read()` and `recv()` usually the source address is defined by connecting the socket to a specific host. This can be done by using the `connect()` function as described in section 3.2.6.

read() The read function works identical to the `read()` on a file. As already said the the socket can be connected to a specific host first by using the connect function as described in section 3.2.6 on the preceding page. For the `read()` function the definitions can be found in the `<unistd.h>` header. The function has the following form [8]:

```
int read(int fd, char * Buff, int NumBytes)
```

The following parameters have to be set [8]:

- **fd** - Takes a file descriptor from which the data should be read, for RAW-sockets we can use the socket descriptor instead.
- **Buff** - Defines the memory space for the binary data we want to read from the socket.
- **NumBytes** - How many bytes should be read from the socket, usually initialized with the size of our memory space.

A usual call of the read function for our purposes would look like this:

```
ssize_t = read(socket, packet, sizeof(packet));
```

On success the number of bytes is returned and on an error -1 [8]. Additionally the variable `errno` defined in `<errno.h>` is set and can have the following values:

Flag	Description
EAGAIN	The file descriptor fd refers to a file other than a socket and has been marked non-blocking (O_NONBLOCK) and the read would block.
EWOULDBLOCK	The file descriptor fd refers to a socket and has been marked non-blocking (O_NONBLOCK) and the read would block.
EBADF	fd is not a valid file descriptor or is not open for reading.
EFAULT	buf is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	fd is attached to an object which is unsuitable for reading or the file was opened with the O_DIRECT flag.
EIO	I/O error.

Table 11: Errno flags for `read()` as defined in `<errno.h>` [8]

The possible return values and error flags can be found in section 13 on the next page.

recv() The `recv(` function is another possibility to to retrieve data from a socket and does not require an address to be defined as well. As already said the the socket can be connected to a specific host first by using the `connect()` function as described in section 3.2.6 on the preceding page.To use the `recv()` function the headers `<sys/types.h>` (for the data types) and `<sys/socket.h>` (for the function) have to be included. The function is called like this [8]:

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

The `recv()` function normally is used on a connected socket (after connecting with the `connect()` function), but also works with RAW-sockets. Similar parameters to the read function have to be set [8]:

- **sockfd** - Specifies the socket from which data should be read.
- **buf** - Defines the memory space for the binary data we want to read from the socket.
- **len** - How many bytes should be read from the socket, usually initialized with the size of our memory space.
- **flags** - The flags that can be set for the function are listed in table 12 on the next page.

recvfrom() The `recvfrom()` function also allows us to define the address of a host we want to receive data from. To use this function the headers `<sys/types.h>` (for the data types) and `<sys/socket.h>` (for the function) have to be included. The function is called like this [8]:

ssize_t recvfrom(*int* **sockfd**, *void* * **buf**, *size_t* **len**, *int* **flags**, *struct sockaddr* * **src_addr**, *socklen_t* * **addrlen**)

The parameters are identical to the `recv()` function, it only has two additional parameters [8]:

- **sockfd** - Specifies the socket from which data should be read.
- **buf** - Defines the memory space for the binary data we want to read from the socket.
- **len** - How many bytes should be read from the socket, usually initialized with the size of our memory space.
- **src_addr** - The function returns a *struct sockaddr* * with this parameter that contains the socket address family and the protocol address for that family. The address is filled in by the called protocol and contains the source address of the packet received. The address will be truncated in case it is too long for the provided buffer. The parameter can be set to `NULL`, then it will not be filled by the protocol.
- **addrlen** - The length of the address. In case the **src_addr** parameter was too small for the returned address an address length greater than the one provided to the function will be returned.
- **flags** - The flags that can be set for the function are listed in table 12.

The possible return values and error flags can be found in section 13.

flags and errors for the `recv()`- and `recvfrom()`-functions Here are the possible flags for the `revc()` and `recvfrom()` functions:

Flag	Description
`MSG_DONTWAIT`	Enables non-blocking operation, if the operation would block the call fails with the error `EAGAIN` or `EWOULDBLOCK`.
`MSG_ERRQUEUE`	Specifies that queued errors should be received from the socket error queue, the buffer has to be supplied by the user.
`MSG_OOB`	This flag requests receipt of out-of-band data that would not be received in the normal data stream.
`MSG_PEEK`	This flag causes the receive operation to return data from the beginning of the receive queue without removing that data from the queue.
`MSG_TRUNC`	For raw (`AF_PACKET`), Internet datagram, netlink and UNIX datagram: return the real length of the packet or datagram even when it was longer than the passed buffer.
`MSG_WAITALL`	This flag requests that the operation block until the full request is satisfied. However the call may still return less data than requested if a signal is caught, an error or disconnect occurs or the next data to be received is of a different type than that returned.

Table 12: Flags for `recv()` and `recvfrom()` defined in `<sys/socket.h>` [8]

The functions `recv` and `recvfrom()` returns the count of bytes read on success and -1 on an error [8]. Additionally the variable `errno` defined in `<errno.h>` is set and can have the values shown in the following table:

Flag	Description
`EAGAIN`	The socket is marked non-blocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received.
`EWOULDBLOCK`	The socket is marked non-blocking and the receive operation would block, or a receive timeout had been set and the timeout expired before data was received.
`EBADF`	The argument sockfd is an invalid descriptor.
`EFAULT`	The receive buffer pointer(s) point outside the processes address space.
`EINTR`	The receive was interrupted by delivery of a signal before any data were available.
`EINVAL`	Invalid argument passed.
`ENOTSOCK`	The argument sockfd does not refer to a socket.

Table 13: Errno flags for `recv()` and `recvfrom()` as defined in `<errno.h>` [8]

Unix The `read()` function is POSIX and 4.3BSD conform and should be available as described [8]. The `recv()` and `recvfrom()` are also conform to POSIX, but only the `MSG_OOB`, `MSG_PEEK` and `MSG_WAITALL` flags are described in the standard [8]. It is also conform to 4.4BSD, but there might be differences in the data types depending on the library used in the system [8].

3.2.8 Write

We can send data over the socket with three different functions. The `write()` and `send()` functions we discuss in the beginning both require a destination address to be defined first. This can be done by using the `connect()` function described section 3.2.6 on page 19. The last function we discuss, `sendto()`, has the option to define the destination address, but it can also be set by using the `connect()` function. For all functions we must take the buffer constraints into consideration, since otherwise the write connection will be closed before all data is transferred completely.

`write()` Identical to how we use a the `read()` function to get data from a socket descriptor instead of a file descriptor, we can also use the `write()` function that is usually used for data output to a file for sending data over a socket. As already mentioned to use `write()` with a socket a valid destination address has to be provided to the socket first for it to work. That can be done by using the `connect()` function which is discussed in section 3.2.6 on page 19. The `write()` function is defined in `<unistd.h>` and has the following from [8]:

ssize_t write(*int* **fd**, *const void ** **buf**, *size_t* **count**)

The parameters are:

- **`fd`** - In our case specifies the socket to which we want to write data.
- **`buf`** - Defines the memory space for the binary data we want to write.
- **`count`** - How many bytes should be written from the buffer to the socket. If there is not enough free space on the physical medium used, less bytes might be written. Also if the process is interrupted by a signal, there also might be less bytes written than specified.

The function returns the amount of bytes written on success and -1 on an error [8]. Additionally the variable *errno* defined in `<errno.h>` is set and can have the following values:

Flag	Description
`EAGAIN`	The file descriptor refers to a file other than a socket and has been marked non-blocking, and the write would block.
`EWOULDBLOCK`	The file descriptor fd refers to a socket and has been marked non-blocking (`O_NONBLOCK`), and the write would block.
`EBADF`	fd is not a valid file descriptor or is not open for writing.
`EDESTADDRREQ`	fd refers to a datagram socket for which a peer address has not been set using `connect()`.
`EFAULT`	buf is outside your accessible address space.
`EINTR`	The call was interrupted by a signal before any data was written.
`EINVAL`	fd is attached to an object which is unsuitable for writing or the file was opened with the `O_DIRECT` flag and either the address specified in buf, the value specified in count or the current file offset is not suitably aligned.
`EIO`	A low-level I/O error occurred while modifying the inode.
`ENOSPC`	The device containing the file referred to by fd has no room for the data.
`EPIPE`	fd is connected to a pipe or socket whose reading end is closed.

Table 14: Errno flags for `write()` as defined in `<errno.h>` [8]

`send()` The `send()` function is another function to send data over a socket. Like `write()` it does require that a destination address is specified first before it can be used, since it does require the socket to be in a connected state [8]. This can be achieved by using the `connect()` function discussed in section 3.2.6 on page 19. There is no indication of a failure implicitly shown when using `send()`, only locally detected errors are indicated by returning -1 [8]. If `send()` is used to transmit messages, then it will block (if not set otherwise) if a message does not fit in the buffer [8]. To use this function the headers `<sys/types.h>` (for the data types) and `<sys/socket.h>` (for

the function) have to be included. The function is called like this [8]:

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

With the following parameters that can be set [8]:

- **sockfd** - Specifies the socket over which the data should be sent.
- **buf** - Defines the memory space for the binary data we want to send over the socket.
- **len** - How many bytes should be read from the memory space given by the parameter buffer.
- **flags** - The flags that can be set for the function are listed in table 15 on the next page

Like already mentioned, the function only indicates local errors by setting the return value to -1 on an error and returning the number of characters (bytes) sent on success [8]. Additionally the variable `errno` defined in `<errno.h>` is set and can have the values described in the table 16 on the following page. Other errors might be reported by the used protocol modules.

sendto() In comparison to the possibilities to send data we discussed so far, the `sendto()` function allows us to define a specific address to which the data should be sent without having to call `connect()` first to set the destination address [8]. It should be noted that if the `sendto()` function is used on a connection-mode socket the additional address information is ignored and an `EISCONN` error might be returned in `errno`, if they are not set to `NULL` and zero respectively [8]. This is the case since for such a socket the destination is already implied by the connection type. To use this function the headers `<sys/types.h>` (for the data types) and `<sys/socket.h>` (for the function) have to be included. The function is called like this [8]:

```
ssize_t sendto(int sockfd, void * buf, size_t len, int flags, struct sockaddr *
dest_addr, socklen_t  addrlen)
```

With the following parameters that can be set [8]:

- **sockfd** - Specifies the socket over which the data should be sent.
- **buf** - Defines the memory space for the binary data we want to send over the socket.
- **len** - How many bytes should be read from the memory space given by the parameter buffer.
- **dest_addr** - The function defines a *struct sockaddr ** with this parameter that contains the socket address family and the protocol address for that family. The parameter can be set to `NULL`, then it will not be filled in.
- **addrlen** - The length of the address provided to the socket.
- **flags** - The flags that can be set for the function are listed in table 15 on the next page.

Like already mentioned, the function only indicates local errors by setting the return value to -1 on an error and returning the number of characters sent on success [8]. Additionally the variable `errno` defined in `<errno.h>` is set and can have the values described in the table 16 on the following page. Other errors might be reported by the used protocol modules.

Flags and errnos for the `send()` and `sendto()` functions The following flags can be set for both the `send()` and `sendto()` functions:

Flag	Description
MSG_CONFIRM	Only valid on SOCK_DGRAM and SOCK_RAW. Tell the layer 2 that you got a successful reply from the other side.
MSG_DONTROUTE	Don't use a gateway to send out the packet, only send to hosts on directly connected networks.
MSG_DONTWAIT	Enables non-blocking operation, if the operation would block EAGAIN or EWOULDBLOCK is returned.
MSG_EOR	Terminates a record (when this notion is supported).
MSG_MORE	The caller has more data to send. This flag is used with UDP/TCP sockets.
MSG_NOSIGNAL	Requests not to send SIGPIPE on errors on stream oriented sockets when the other end breaks the connection. The EPIPE error is still returned.
MSG_OOB	Sends out-of-band data on sockets that support this notion, the underlying protocol must also support out-of-band data.

Table 15: Flags for send() and sendto() as defined in <errno.h> [8]

The following errno constants are defined in <errno.h> for both the send() and sendto() functions:

Flag	Description
EAGAIN	The socket is marked non-blocking and the requested operation would block.
EWOULDBLOCK	The socket is marked non-blocking and the requested operation would block.
EBADF	An invalid descriptor was specified.
ECONNRESET	Connection reset by peer.
EDESTADDRREQ	The socket is not connection-mode, and no peer address is set.
EFAULT	An invalid user space address was specified for an argument.
EINTR	A signal occurred before any data was transmitted.
EINVAL	Invalid argument passed.
EISCONN	The connection-mode socket was connected already but a recipient was specified.
EMSGSIZE	The socket type requires that message be sent atomically, and the size of the message to be sent made this impossible.
ENOBUFS	The output queue for a network interface was full. This generally indicates that the interface has stopped sending, but may be caused by transient congestion.
ENOMEM	No memory available.
ENOTCONN	The socket is not connected, and no target has been given.
ENOTSOCK	The argument sockfd is not a socket.
EOPNOTSUPP	Some bit in the flags argument is inappropriate for the socket type.
EPIPE	The local end has been shut down on a connection oriented socket. In this case the process will also receive a SIGPIPE unless MSG_NOSIGNAL is set.

Table 16: Errno flags for send() and sendto() as defined in <errno.h> [8]

Unix The write() function is POSIX and 4.3BSD compliant, so it should work in BSD based systems without any changes [8]. The send() and sendto() are also conform to POSIX, but only the MSG_OOB, MSG_PEEK and MSG_WAITALL flags are described in the standard [8]. It is also conform to 4.4BSD, but there might be differences in the data types depending on the library used in the system [8].

3.2.9 close()

After we are finished with our network communication, we can close the socket like we would do it for a file. The close() function is defined in the <unistd.h> and looks like this [8]:

```
int close(int fd)
```

The only parameter is the socket descriptor that should be closed [8]. All locks held by the associated process are released as well [8]. The function returns zero on success and -1 on an error [8]. Additionally the variable errno defined in <errno.h> is set and can have the following values:

Flag	Description
EBADF	fd is not a valid file descriptor.
EINTR	The close() call was interrupted by a signal.
EIO	An I/O error occurred.

Table 17: Errno flags for close() as defined in <errno.h> [8]

Unix Since the close() function is part of the POSIX-standard there is no difference to the call between Linux-distributions and Unix-systems [8].

3.2.10 inet_ntop()

inet_ntop() is a helper function to convert from a network address to a human readable character string. It currently supports IP version 4 and version 6 addresses [8]. The function is defined in the <arpa/inet.h> and looks like this [8]:
const char * inet_ntop(*int* **af**, *const void* * **src**, *char* *__**dst**, *socklen_t* **size**)

With the following parameters that can be set [8]:

- **af** - The address family we want to convert from. Currently only AF_INET for IP version 4 and AF_INET6 for IP version 6 addresses are supported.
- **src** - The network address structure we want to convert.
- **dst** - A pointer to a buffer for the resulting character string.
- **size** - The size of the buffer.

On success the dst parameter will contain the human readable form of the address [8]. If an error occurs NULL is returned [8]. Additionally the variable errno defined in <errno.h> is set and can have the following values:

Flag	Description
EAFNOSUPPORT	The parameter **af** was not a valid address family.
ENOSPC	The converted address string would exceed the size given by size.

Table 18: Errno flags for ntop() as defined in <errno.h> [8]

Unix The function is a part of the POSIX standard and should work as described on all Linux and Unix systems [8].

3.2.11 inet_pton()

inet_pton() is a helper function to convert from a human readable character string to a network address. It currently supports IP version 4 and version 6 addresses [8]. The function is defined in the <arpa/inet.h> and looks like this [8]:
int inet_ntop(*int* **af**, *const char* * **src**, *void* *__**dst**)

With the following parameters that can be set [8]:

- **af** - The address family we want to convert from. Currently only AF_INET for IP version 4 and AF_INET6 for IP version 6 addresses are supported.
- **src** - The character string we want to convert. For the AF_INET address family the preferred format is the dotted-decimal format, "ddd.ddd.ddd.ddd". For the AF_INET6 address family the preferred format is the hexadecimal format ”x:x:x:x:x:x:x:x”, but the usual abbreviation forms are also supported.
- **dst** - A pointer to the resulting network address structure.

On success the dst parameter will contain the network address structure and the return value is set to 1 [8]. If the **src** parameter did not contain a valid address string 0 is returned [8]. If an error occurred -1 is returned and additionally the variable errno defined in <errno.h> is set and can have the following values:

Flag	Description
EAFNOSUPPORT	The parameter **af** was not a valid address family.

Table 19: Errno flags for pton() as defined in <errno.h> [8]

Unix The function is a part of the POSIX standard and should work as described on all Linux and Unix systems [8].

3.2.12 Data Types

There are several structures that we use for all system calls and functions we use when working with network functions. Here we want to show these structures and show a way to initialize them. The first stucts we want to introduce are the sockaddr structs that are use to hold a layer 3 address and are defined in the <netinet/in.h> header. The struct sockaddr is the basic definition of an address that many of the functions discussed so far take as a parameter. It is possible to cast a pointer to one of these structures into each other without any harm [9].

See listing **??** in the appendix.

The struct sockaddr_in is the struct we use to hold an IP version 4 address. Note the sin_zero field, for which it sometimes recommended is that it should be set to zero, even though it is not even mentioned in the Linux programmer's manual [9]. We recommend to initialize the whole struct with 0 using the memset() function. The other fields are usual information for IP version 4 addresses.

Listing 1: sockaddr in cpp

```
1
2  struct sockaddr_in {
3      short            sin_family;   // e.g. AF_INET, AF_INET6
4      unsigned short   sin_port;     // e.g. htons(3490)
5      struct in_addr   sin_addr;     // see struct in_addr, below
6      char             sin_zero[8];  // zero this if you want to
7  };
```

The struct sockaddr_in is the struct we use to hold an IP version 6 address. Similar to the struct for the IP version 4 address it contains all the required fields for the protocol information.

Listing 2: sockaddr in6 cpp

```
1  struct sockaddr_in6 {
2      u_int16_t       sin6_family;   // address family, AF_INET6
3      u_int16_t       sin6_port;     // port number, Network Byte Order
4      u_int32_t       sin6_flowinfo; // IPv6 flow information
5      struct in6_addr sin6_addr;     // IPv6 address
6      u_int32_t       sin6_scope_id; // Scope ID
7  };
```

The following example adapted from [9] shows exemplary how to init an IP version 4 address:

Listing 3: sockaddr init cpp

```
1  // the struct for the socket-address
2  struct sockaddr_in ip4addr;
3
4  // init the address family to internet addresses
5  ip4addr.sin_family = AF_INET;
6
7  // fill out the port
8  ip4addr.sin_port = htons(3490);
9
10 // fill in the ethernet address
11 inet_pton(AF_INET, "10.0.0.1", &ip4addr.sin_addr);
```

There is also a structure we can use to define a layer 2 address, for example a MAC-address or an index for one of the network interfaces of the host. It is available in the `<linux/if_packet.h>` header.

Listing 4: sockaddr ll cpp

```
struct sockaddr_ll {
    unsigned short sll_family;   // family is always AF_PACKET
    unsigned short sll_protocol; // physical layer protocol
    int sll_ifindex;    // interface number
    unsigned short sll_hatype;  // header type
    unsigned char sll_pkttype;   // packet type
    unsigned char sll_halen;   //  length of the address
    unsigned char sll_addr[8];   // physical layer address
};
```

Many `ioctl()` calls we use to get network device information return an `ifreq` structure [10]. We usually specify which device to affect by setting `ifr_name` to the name of the interface [10]. It is defined in the `<net/if.h>` header.

Listing 5: ifreq cpp

```
 struct ifreq {
    char ifr_name[IFNAMSIZ]; /* Interface name */
    union {
        struct sockaddr ifr_addr;
        struct sockaddr ifr_dstaddr;
        struct sockaddr ifr_broadaddr;
        struct sockaddr ifr_netmask;
        struct sockaddr ifr_hwaddr;
        short           ifr_flags;
        int             ifr_ifindex;
        int             ifr_metric;
        int             ifr_mtu;
        struct ifmap    ifr_map;
        char            ifr_slave[IFNAMSIZ];
        char            ifr_newname[IFNAMSIZ];
        char           *ifr_data;
        };
};
```

One example is the use of the `ifreq` for getting the device index of an network interface as shown in the following code:

Listing 6: ifreq init cpp

```
// struct for the interface definition
struct ifreq ifr;

// the name of the device we want to use
char * device = "wlan0";

// set the structs to zero
bzero(&ifr , sizeof(ifr));

// define the name of the interface we want to set
strncpy((char *)ifr.ifr_name ,device , IFNAMSIZ);

// get device index
ioctl(sockfd , SIOCGIFINDEX , &ifr)
```

3.2.13 Layer 4

Now we can start with describing the different layers we can access and write with the RAW-sockets. We start by discussing working with RAW-sockets for Transport layer (OSI layer 4) access. The basic operation is generally very similar for all layers we want to work with. For example the basic call to create a RAW-socket for IP looks like this [5]:

```
sockfd = socket(AF_INET, SOCK_RAW, int  protocol )
```

In this example the `AF_INET` specifies IP as the protocol, the `SOCK_RAW` defines the socket type as a RAW-Socket and the *protocol* can be used to specify any layer 4 protocol we want to use, so for example `IPPROTO_IP` is not supported as the *protocol* since it is a dummy layer 3 protocol. Note that we can only send and receive a **single protocol** using this, it is not possible to receive data for all protocols with a RAW-socket [1]. We also have to be aware that the kernel still also receives all protocol data and will act accordingly [1]. If we do not want the kernel to also react to the packets we have to take additional steps to prevent this.

Unix RAW-sockets work as described in Linux distributions. For Unix based systems there are some limitations we have to observe.

- The Write-commands work for layer 3 and 4 work for Unix as well.
- The Read-commands for layer 3 and 4 only work with some limitations. TCP and UDP packets cannot be received with RAW-sockets, they are only delivered to the kernel [6]. Copies of ICMP packets are delivered to the RAW-socket as well as to the kernel [6]. All IGMP packets are delivered to the RAW-socket, as well as any other protocol packets [6].
- Unix does not support the PACKET-socket as described for layer 2 later. Instead Unix has the Berkeley Packet Filter (BPF) interface that can be used for layer 2 communication [2]. We will give a short introduction in a later section.
- Some of the functions that are used in this section are requiring additional headers to work under Unix. The additional headers are noted in the last section were the functions were introduced.

• Other than these limitations we can work with the RAW-socket under Unix in the same way as in Linux [8].

Read The structure of the complete Read operation for a basic layer 4 RAW-socket is shown in figure 3 on the next page.

- We first have to create the socket as it was discussed in section 3.2.1 on page 15. Like mentioned, if we only want to create layer 4 headers later we can choose any protocol family we want to receive data for except the `IPPROTO_RAW` [1]. Also even if we just read data to the buffer it makes sense to also reserve buffer space and initialize it with zero to have defined data in the buffer.
- As an optional step we can define the interface we want to receive the network data from by defining the source address using the `bind()` function.
- Additionally we can, as an option, use the `connect()` function to define a standard source we want to receive data from or we want to connect to in case of connection-oriented protocols [8].
- Then we can read binary data from the socket with one of the function described in section 3.2.7 on page 19. We can either read data using the `read()` or `recv()` functions. As an alternative we can use the `revcfrom()` function which also returns the source address of the packet we did receive. It is recommended to check the return value to know how many bytes were received to be able to react accordingly.
- Optionally we can `close()` the socket now if we received all the data we want to receive.
- Then we can type-cast the binary data onto our headers as it is described in section . We have to be aware that the Buffer of a RAW-socket includes the headers as well as the data . The included headers start at the Network layer (OSI layer 3) [1]. Since we do not want to use the information of the Network layer at this point, we have to add the length of its header to the data pointer to get the Transport layer (OSI layer 4) header information [5]. When doing this we have to be aware that there can be variable network layer headers as described in section 3.1.4 on page 14. Depending on the protocol and possibly additional variable headers we can compute the start positions of the next header (or headers) and type-cast them into structures.

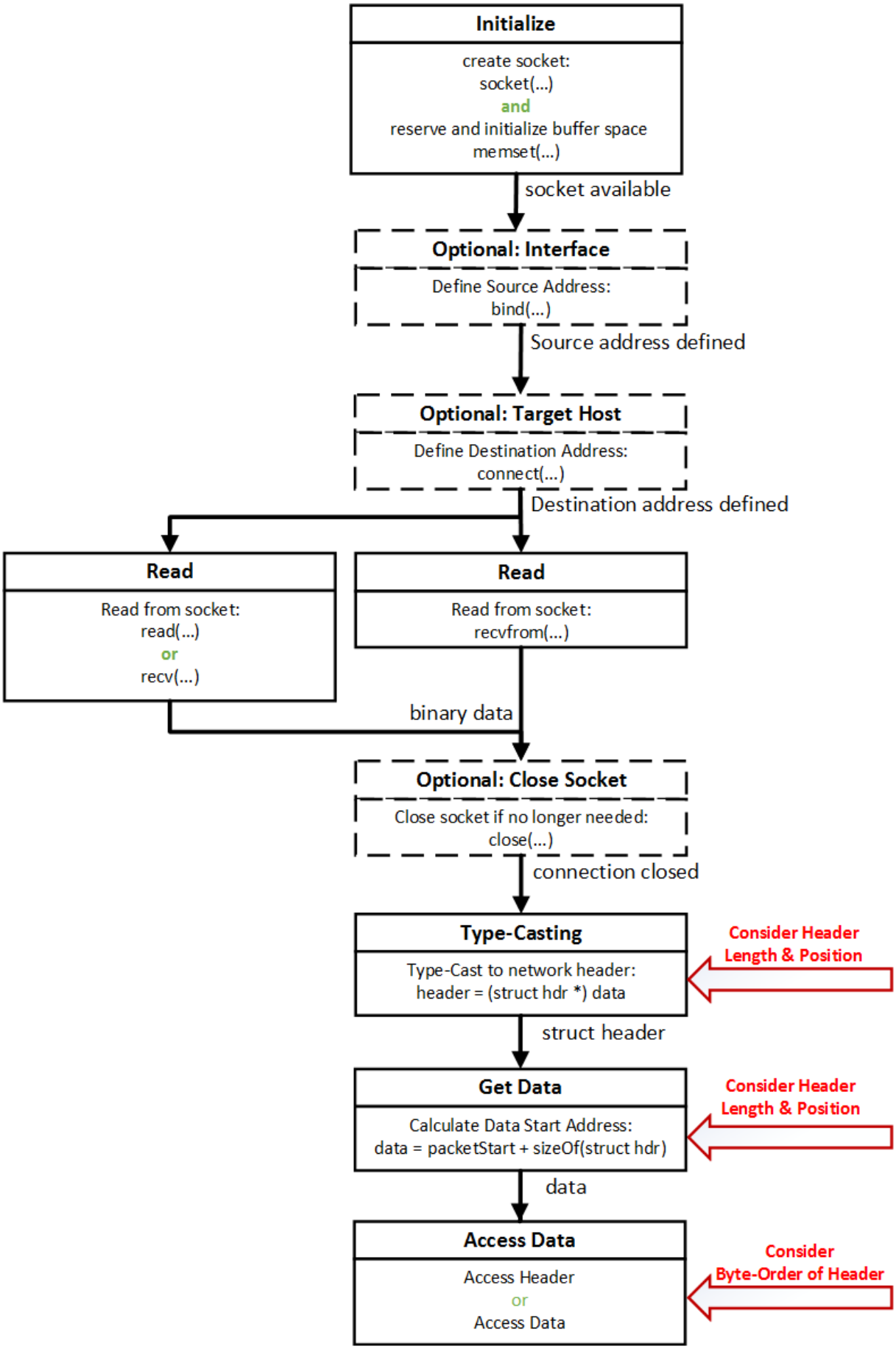

Figure 3: Structure of a RAW-socket layer 4 Read Operation

- After all headers are handled the remainder of the packet is the user data and can also be extracted.
- Then we can access the extracted headers or data. When accessing the headers we have to observer the byte-order, as discussed in the section 3.1.1 on page 13.

As the general programming paradigms suggest, the user should handle possible errors that might occur in the steps. Possible error values are presented in the introductions of the available functions. This is even more important when working with RAW-sockets, since working with them is very error prone.

Write When we use the layer 4 Write access, the operating system will take over most function required to send the packet. We only have to start to create our packet at layer 4 and supply the required destination address to the API to send the packet. All layer 3 headers will be filled in by the operating system accordingly [1].

- We first have to create the socket as it was discussed in section 3.2.1 on page 15. Like mentioned if we only want to create layer 4 headers, we can choose any protocol family except the `IPPROTO_RAW`[1]. Also we reserve buffer space for the packet we want to create and initialize it with zero to have defined data in the buffer [5].
- As an optional step we can define the interface we want to use to send the network data from by defining the source address using the `bind()` function.
- Then we can type-cast the buffer to our layer 4 headers like it is described in section 3.3.7 on page 44. We have to take into consideration variable header lengths if we want to use optional fields and compute the start of the next header accordingly as described in section 3.1.4 on page 14. The remaining buffer can be used for the user data we might want to pass along.
- As the next step we set the data of our header as required. We have to consider the byte-order in case of fields that span multiple bytes and change the byte-order according to the requirements of the protocol we want to use [5]. Additional information about this can be found in section 3.1.1 on page 13.
- If the protocol does use a checksum to protect the header from changes we also have to compute the checksum for the layer 4 ourselves [5]. Only for the layers below, the operating system will take care of computing the checksum. We have to see in the documentation which kind of checksum is required by the protocol we want to use. Additional information about checksum computation can be found in the section 3.1.2 on page 13.
- In the next step we have the choice to use one of two possibilities. We can use the `sendto()` function, which allows us to pass a destination address along with the data. Optionally we can as an option use the `connect()` function to define a standard source we want to send data to or want to connect to in case of connection-oriented protocols. If we define the destination with `connect()` the destination address of `sento()` might be left empty, then the already defined address is used [8]. The other possibility is to use `write()` or `send()` which both require the socket to be in a connected state, so we have to call `connect()` first to use one of these functions [8]. All functions have the same result, the data is being sent. The result these functions return is the number of bytes that were sent. It is recommended to check the return value and match it with the length of the data that should have been sent to make sure the function was not interrupted during the call and all data has been sent as intended [8].
- Optionally we can `close()` the socket now if we already sent all the data we want to transmit.

3.2.14 Layer 3

Using RAW-sockets to also access the layer 3 of a network packet is very similar to the layer 4 access we discussed in the previous section. The most significant differences are that we need to set a different option to get write access to the complete layer 3 header. We also now have to take the structure and lengths of multiple headers into consideration. For example the basic call to create a raw-socket for IP could look like this [5]:

```
sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_RAW)
```

In this example the `AF_INET` specifies IP as the protocol, the `SOCK_RAW` defines the socket type as a RAW-Socket and the `IPPROTO_RAW` specifies that we are interested in sending any type of protocol with this socket. Note that we can only send any protocol with this, it is not possible to receive any data with this socket [1]. If we want to still receive packets we could specify a single protocol in the last parameter of the `socket()` call and then have to set the `IP_HDRINCL` with the `setsockopt()` function to be able to write and receive with the same socket. We also have to be aware that the kernel still also receives all protocol data and will act accordingly [1]. If we do not want the kernel to also react to the packets we have to take additional steps to prevent this.

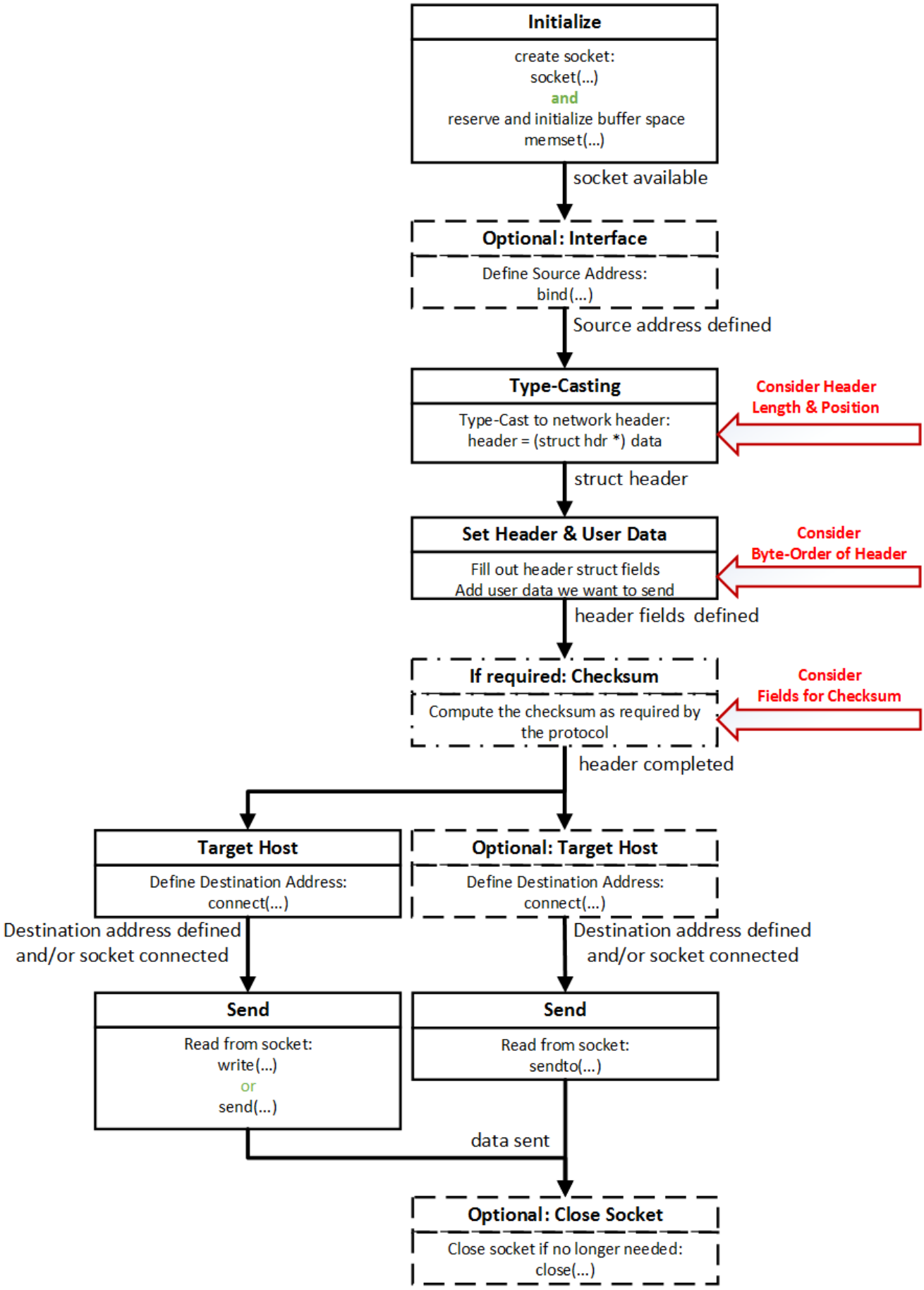

Figure 4: Structure of a RAW-socket layer 4 Write Operation

Read For the Read operation the structure and commands are the same as for the layer 4 read access. As already discussed we simply ignored the layer 3 information before since we did not want to use it. Nevertheless the same command already gave us the possibility to read-out the header information. Also if we also want to have write access to the layer 3 headers for the write operation later it is a possible simplification to create a socket and then set the `IP_HDRINCL` flag as already discussed. This is not necessary for the Read operation we want to discuss first only for the Write later. Details on this option will follow in the layer 3 Write section. It should be noted again that receiving data of all IP-protocols is not possible with with RAW-sockets, only the data for a single protocol can be received at a time [1]. A possible solution for this are the PACKET-sockets discussed in the next section.

The structure of the complete Read operation for a basic layer 3 RAW-socket is shown in figure 5 on the next page.

As already mentioned there are only slight differences due to the handling of multiple layers in comparison to the layer 4 read.

- We first have to create the socket as it was discussed in section 3.2.1 on page 15. Like mentioned if we want to create layer 3 headers we can additionally set the `IP_HDRINCL` flag with the `setsockopt()` function. Also even if we just read data to the buffer it makes sense to also reserve buffer space and initialize it with zero to have defined data in the buffer [5].
- As an optional step we can define the interface we want to receive the network data from by defining the source address using the `bind()` function.
- Additionally we can as an option use the `connect()` function to define a standard source we want to receive data from or we want to connect to in case of connection-oriented protocols.
- Then we can read binary data from the socket with one of the function described in section 3.2.7 on page 19. We can either read data using the `read()` or `recv()` functions. As an alternative we can use the `revcfrom()` function which also returns the source address of the packet we received. It is recommended to check the return value to know how many bytes were received to be able to react accordingly [8].
- Optionally we can `close()` the socket now, if we received all the data we want to receive.
- Then we can type-cast the binary data into our headers like it is described in section 3.3.7 on page 44. We have to be aware that the buffer of a RAW-socket includes the headers as well as the data [1]. The included headers start at the network layer (OSI layer 3). Since we want to use the information of the network layer at this point, we can typecast the header onto a struct to get access to all the fields. We still have to add the length of its header to the data pointer to get the Transport layer (OSI layer 4) header information. When doing this we have to be aware that there can be variable network layer headers as described in section 3.1.4 on page 14. Also depending on which functions we used for reading data from the socket, we might want to filter from which source and/or for which target address we want to receive data from. Depending on the protocol and possibly additional variable headers we can compute the start positions of the next header (or headers) and type-cast them into structs.
- After all headers are handled the remainder of the packet is the user data that can also be extracted.
- Then we can access the extracted headers or data. When accessing the headers we have to observer the byte-order, as discussed in the section 3.1.1 on page 13.

Write Like already noted, to get write access to the layer 3, we have to either create an `IPPROTO_RAW` socket when choosing our protocol for the `socket()` function or set the `IP_HDRINCL` with the `setsockopt()` to deactivate automatic IP-header generation by the operating system and be able to manually fill the fields [5]. It should be noted here that `IPPROTO_RAW` socket does not include the `IP_HDRINCL` flag for all operating systems since this is not defined in the POSIX-Standard [11]. So if you want to be sure that the code works regardless of the distribution used, it is recommended to set the **protocol** option to the protocol that can be found in the layer 3 protocol header and use `setsockopt()` to tell the OS that we want to generate the header manually [11].

The Structure of the complete Write operation for a basic layer 3 RAW-socket is shown in figure 6 on page 34.

- We first have to create the socket as it was discussed in section 3.2.1 on page 15. Like mentioned we have to use either `IPPROTO_RAW` when choosing our protocol for the `socket()` function, or set the `IP_HDRINCL` with the `setsockopt()` function to deactive automatic IP-header generation as already discussed [5]. Also we reserve buffer space and initialize it with zero to have definded data in the buffer [5].

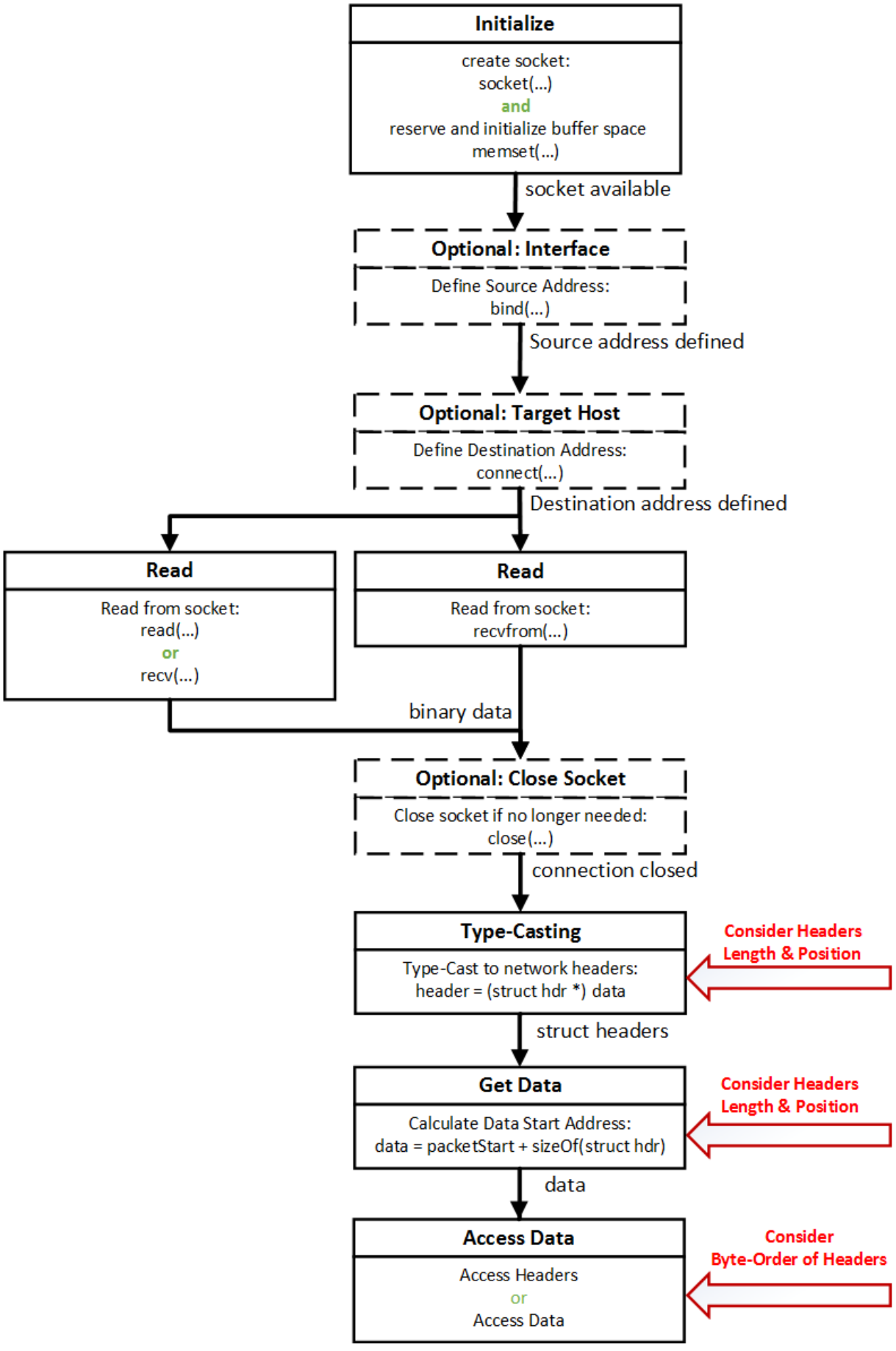

Figure 5: Structure of a RAW-socket layer 3 Read Operation

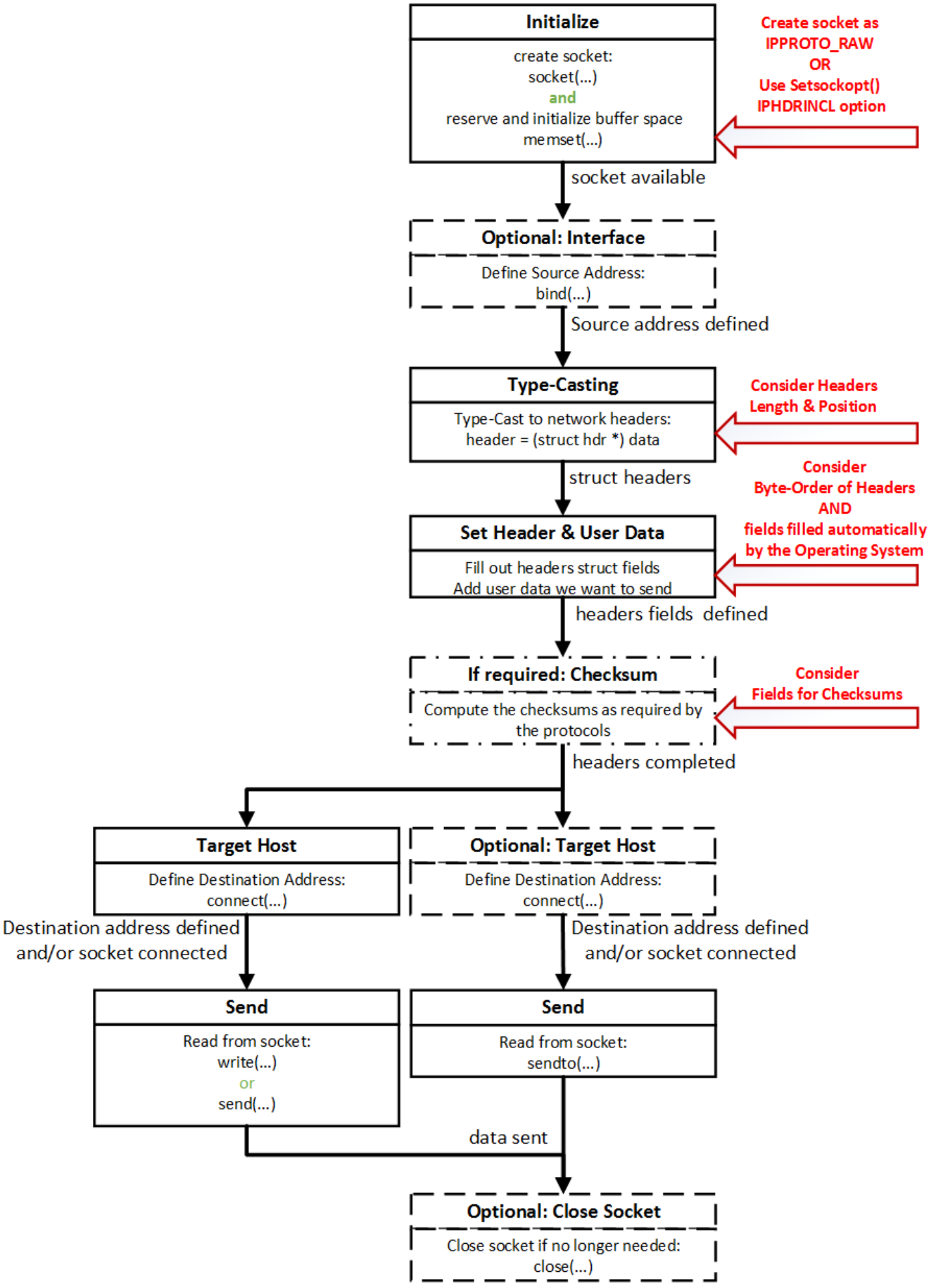

Figure 6: Structure of a RAW-socket layer 3 Write Operation

- As an optional step we can define the interface we want to use to send the network data with by defining the source address using the `bind()` function.
- Then we can type-cast the buffer to our headers like it is described in section 3.3.7 on page 44. The included headers start at the Network layer (OSI layer 3). We have to take into consideration variable header lengths, if we want to optional fields and compute the start of the next header accordingly as described in section 3.1.4 on page 14. The remaining buffer is reserved for the user data we might want to pass along.
- As the next step we set the data of our headers as required. However, we do not need to fill the checksum and total length of the IP-packets, since the operating system will fill in the values automatically [1]. Optionally we can also fill in zero in the fields source address and packet ID [1]. If we do this then the operating system also will fill in the values automatically. When we fill in the values of the header fields we have to consider the byte-order in case of fields that span multiple bytes and change the byte-order according to the requirements of the protocol we want to use. Additional information about this can be found in section 3.1.1 on page 13.
- If the protocol does use a checksum to protect the header from changes, we also have to compute the checksum for the layer 4 ourselves [5]. Only for the layers below, the operating system will take care of computing them. For the IP-Protocol for example the Operating System will always fill in the value for the checksum. We have to see in the documentation which kind of checksum is required by the protocol we want to use. Additional information about computing the checksum can be found in the section 3.1.2 on page 13.
- In the next step we have the choice to use one of two possibilities. We can use the `sendto()` function, which allows us to pass a destination address along with the data. Optionally we can as an option use the `connect()` function to define a standard source we want to send data to, or want to connect to in case of connection-oriented protocols. If we define the destination with `connect()` the destination address of `sento()` might be left empty, then the already defined address is used. The other possibility is to use `write()` or `send()` which both require the socket to be in a connected state, so we have to call `connect()` first to use one of these functions [8]. All functions have the same result, the data is being sent. It is recommended to check the return value and match it with the length of the data that should have been sent to make sure the function was not interrupted during the call and all data has been sent as intended [8].
- Optionally we can `close()` the socket now, if we already sent all the data we want to transmit.

3.2.15 Layer 2 - PACKET-sockets

In addition to the layers we discussed so far, it is also possible under Linux to access the Data-Link-layer with RAW-sockets. In earlier versions there was a special type of sockets for this, the **PACKET_socket**[5]. Nowadays the use of packet sockets is no longer recommended, instead we can use a **normal RAW_socket** with a different protocol supplied to the `socket()` function, like shown in section 3.2.1 on page 15[5]. The possible socket types we can use are the `SOCK_RAW` to get access to the whole packet including the Data-Link-layer header or the `SOCK_DGRAM` which operates on the Data-Link-layer packet without the headers [12]. The protocols for Ethernet we can use in Linux are shown in the table 48 on page 81. One common option might be the `ETH_P_ALL` protocol constant which results in all incoming and outgoing Ethernet packets to be accessed with the RAW-socket. A typical call looks like this, note the `htons` function used to change the byte-order of the supplied protocol constant since it is a multi-byte constant [5]:

```
s = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
```

With this all headers down to the Data-Link-layer can now be accessed and all data received on all interfaces can be received. However as already mentioned, parts of the Data-Link-layer header might not be accessible, for example for the Ethernet frames the CRC (Cyclic Redundancy Check) checksum, the Preamble and the Start-of-Frame Delimiter are not accessible since these fields are more part of the Physical-layer and therefore handled by the network driver [5]. Also with PACKET-socket the `connect()` function has lost its purpose, only the `bind()` function still makes sense to define the network interface [5]. We have to be aware that the kernel still also receives all protocol data and will act accordingly [1].

Promiscuous Mode The promiscuous mode is a special mode of a network card in which all network traffic of an interface is received, in contrast to the usual case where only the traffic addressed to the interface is received [5]. PACKET-sockets work similar to the promiscuous mode, but receive all traffic for all interfaces of the host, but can be configured to only receive the traffic on a single interface by using the `bind()` function [5]. If it is necessary to activate the promiscuous mode for some reason it can be done by setting the `IFF_PROMISC` using the `ioctl()` systemcall [5].

MAC-Address Since we now work on the layer 2 we also have to work with the addresses of the layer, for example the MAC-addresses used for example in Ethernet. The MAC-address of the interface we want to use as a source can be accessed with a call of the `ioctl` function. We demonstrate the call in the following listing. Note that the listing does not include any protection against errors.

Listing 7: mac cpp

```
1  struct ifreq ifr;  // struct for the interface information
2  char * eth = "wlan0"; // name of the interface
3
4  // copy the name to the struct
5  memcpy(ifr.ifr_name,if_name,IFNAMSIZ);
6
7  //create a socket descriptor
8  int sockfd = socket( AF_PACKET , SOCK_RAW , htons(ETH_P_ALL)) ;
9
10 //get the information
11 ioctl(sockfd,SIOCGIFHWADDR,&ifr)
12
13 //extract the mac
14 const unsigned char* mac = (unsigned char*) ifr.ifr_hwaddr.sa_data;
```

Read Except for the creation of the packet socket, the missing `connect()` statements and the handling of more layers, the layer 2 read is not much different from the other read operations and share the same basic structure. The Structure of the complete Read operation for a basic layer 2 PACKET-socket is shown in figure 7 on the next page.

For a read of the layer 2 the general structure we discussed until now is the same, only some steps are different due to different initialization we need and the handling of Data-Link-layer data:

- We first have to create the socket as it was discussed in section 3.2.1 on page 15. As already discussed the possible socket types we can use are the `SOCK_RAW` or the `SOCK_DGRAM`. The protocols for Ethernet we can

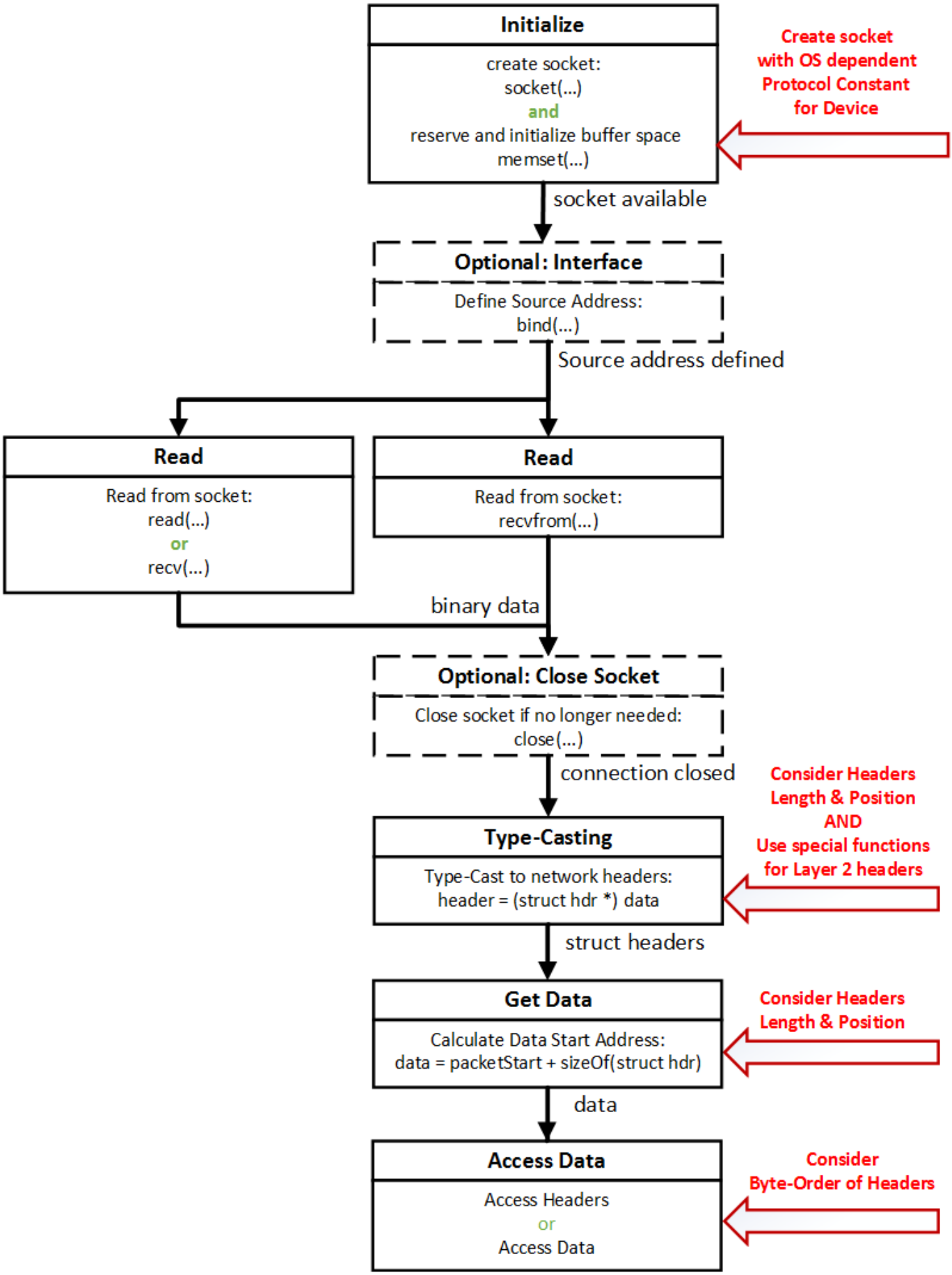

Figure 7: Structure of a PACKET-socket layer 2 Read Operation

use in Linux are shown in the table 48 on page 81. Since they could be multi-byte constants we have to use functions to change the byte-order as discussed in the section 3.1.1 on page 13. Also even if we just read data to the buffer it makes sense to also reserve buffer space and initialize it with zero to have defined data in the buffer [5].

- As an optional step we can define the interface we want to receive the network data from by defining the source address using the `bind()` function.
- Then we can read binary data from the socket with one of the function described in section 3.2.7 on page 19. We can either read data using the `read()`, `recv()` functions, or we can use the `revcfrom()` function which also returns the source address of the packet we did receive. It is recommended to check the return value to know how many bytes were received to be able to adapt accordingly [8].
- Optionally we can `close()` the socket now, if we received all the data we want to receive.
- Then we can type-cast the binary data into our headers like it is described in section . Since we did start at the Data-Link-layer, there could be different layer 3 protocols included in the frame. This is why we have to access the correct field in the header to find out what is encapsulated in the frame and which header structure we have to handle. When doing this we have to be aware that there can be variable network layer headers as described in section 3.1.4 on page 14. Also depending on which functions we used for reading data from the socket we need to filter from which source and/or for which target address we want to receive data from. Depending on the protocol and possibly additional variable headers we can compute the start positions of the next header (or headers) and type-cast them into structs.
- After all headers are handled the remainder of the packet is the user data that can also be extracted.
- Then we can access the extracted headers or data. When accessing the headers we have to observer the byte-order, as discussed in the section 3.1.1 on page 13.

Write The Write on a PACKET-socket differs from from the Write on a RAW-socket in some important parts. The IP-header fields that were filled by the operating system when using a RAW-socket have to be filled in manually when using a PACKET-socket. The only layer where the operating system still helps the user is on the Data Link Layer, were some fields are automatically generated [5]. That is the case for example when creating Ethernet frames, then the Padding field of the frame, the Preamble, the Start-of-Frame Delimiter and the CRC checksum [5]. These fields are still generated automatically by the network driver.

The Structure of the complete Write operation for a basic layer 2 PACKET-socket is shown in figure 8 on the following page.

- We first have to create the socket as it was discussed in section 3.2.1 on page 15. As already discussed the possible socket types we can use are the `SOCK_RAW` or the `SOCK_DGRAM`. The protocols for Ethernet we can use in Linux are shown in the table 48 on page 81. Since they could be multi-byte constants we have to use functions to change the byte-order as discussed in the section 3.1.1 on page 13. Also we reserve buffer space and initialize it with zero to have defined data in the buffer.
- Even if packets sent over PACKET-sockets have to be generated completely manual it is still required for the send functions to define a socket-address. The necessary struct *sockaddr_ll* can be found in the header `<linux/if_packet.h>`. The only part of the struct we have to fill in is the network interface `sll_ifindex`[5]. Now to find the network interface, in Linux usually denoted as f.e. *eth1* we can use the so called *Netdevice*-Interface [10]. It describes the `ioctl()`-system calls that can be used to access different properties of network interfaces in Linux, for example the callcode `SIOCGIFINDEX` has to be used to get the desired information. The access can be done on all socket types with the data structure `ifreq` that is defined in `<linux/net/if.h>`. The complete call we have to use looks like this [5]:
  ```
  ioctl(socketfd, SIOCGIFINDEX, &ifr)
  ```

  The `socketfd` again is our socket descriptor, the `SIOCGIFINDEX` is the parameter to retrieve the information we need and `ifr` is the `ifreq` that after the call has the information we need. With the information we can later call the `bind()` or `sendto()`.
- Then we can type-cast the buffer to our headers like it is described in section . The included headers start at the Data-Link-layer (OSI layer 2). It is necessary to include or define additional headers to have structs for the Data-Link-layer header. We have to take into consideration variable header lengths, if we want to optional fields and compute the start of the next header accordingly as described in section 3.1.4 on page 14. The remaining buffer is reserved for the user data we might want to pass along.

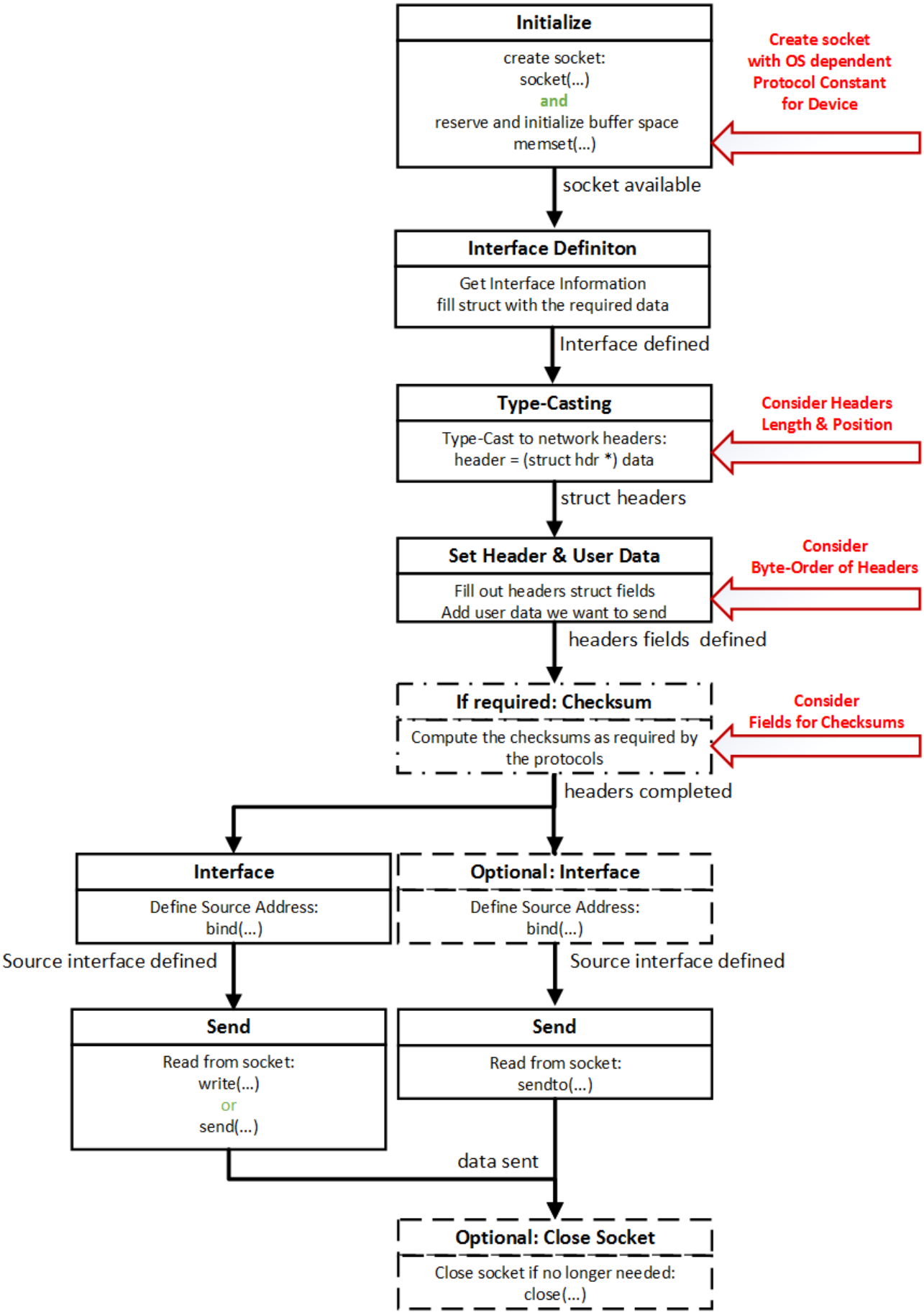

Figure 8: Structure of a PACKET-socket layer 2 Write Operation

- As the next step we set the data of our headers as required. We have to fill in all fields since the operating system does not automatically create fields for PACKET-sockets [5]. When we fill in the values of the header fields have to consider the byte-order in case of fields that span multiple bytes and change the byte-order according to the requirements of the protocol we want to use. Additional information about this can be found in section 3.1.1 on page 13.
- If the protocol does use a checksum to protect the header from changes, we have to compute them and set the fields accordingly [5]. We have to see in the documentation which kind of checksum is required by the protocol we want to use. Additional information about computing a checksum can be found in the section 3.1.2 on page 13.
- In the next step we have the choice to use one of two possibilities. We can use the `sendto()` function, which allows us to pass the socket address of the interface we created along with the data. Optionally we can as an option use the `bind()` function to define a standard socket address for the interface we want to send data with. If we define the destination with `bind()` the destination address of `sento()` might be left empty, then the already defined address is used [8]. The other possibility is to use `write()` or `send()` which both require a call of `bind()` first. All functions have the same result, the data is being sent. It is recommended to check the return value and match it with the length of the data that should have been sent to make sure the function was not interrupted during the call and all data has been sent as intended [8].
- Optionally we can `close()` the socket now, if we already sent all the data we want to transmit.

3.3 Berkeley Packet Filter (BPF)

The Berkeley Packet Filter (BPF) provides us with the same functionality we have introduced for the PACKET-socket in Linux, it provides full access to the data link layer for read and write [2]. So as it is the case for the PACKET-socket we can receive any packet received by any interface regardless if it is meant for our host or not [2]. Additionally to these functions the BPF also includes an advanced filtering mechanism that we can use to filter incoming packets according to different criteria [2]. We will only give a short overview over this function since our main focus is not to give an introduction on how to program a network sniffer, but how to send and receive data including the protocol headers.

To use the BPF we have to use a distribution that has the `device bpf` in its kernel [7]. Only than can we create a BPF device. The BPF appears as a special character device `/dev/bpf`. To open it we have to use the well known `open()` function which is defined in `<fnctl.h>` and looks like this:

int open(*const char ** **pathname**, *int* **flags**)

We have to set the following parameters:

- **pathname** takes the path of a file. In our case the path has to be `/dev/bpf`**n** where **n** is the number of the device depending on how many other programs use a BPF [7].
- **flags** takes a flag as the name implies, the most useful for us is the `O_RDWR` flag that gives us read and write access.

• After we have done this we have to associate the BPF device to a network interface [7]. This works similar to the `bind()` function used for the RAW- or PACKET-socket, but is realized with a `ioctl()` call with the flag `BIOCSETIF`. The completed command looks like this, note the `ifreq` structure:

Listing 8: bpf cpp

```
1  // define the name of the interface
2  const char* interface = "fxp0";
3
4  // create the struct for the interface
5  struct ifreq bound_if;
6
7  // copy the name into the struct
8  strcpy(bound_if.ifr_name, interface);
9
10 // associate the bpf to the interface
11 ioctl( bpf, BIOCSETIF, &bound_if )
```

Afterwards we just have to set the BPF to immediate mode, since a `read()` would otherwise block until the kernel buffer becomes full or a timeout occurs [2]. We also have to get the buffer size, since the BPF might return us more than one packet at a time [7]. The code for this is shown in the listing below:

Listing 9: bpf2 cpp

```
// initialize buffer length
int buf_len = 1;

// activate immediate mode
ioctl( bpf, BIOCIMMEDIATE, &buf_len )

// request buffer length
ioctl( bpf, BIOCGBLEN, &buf_len )
```

After this we can read and write to the BPF like we did it before. Note that we only can use the `read()` and `write()` functions since the BPF is handled like a file. Before we introduce these operations we want to show some BPF specialties and give an overview over some preferences that can be set for the BPD subsystem.

3.3.1 BPF Header

The BPF device adds an additional header with information about the received packet [2]. One of the headers shown in the following listing is appended to each packet, `bpf_xhdr` is used by default, `bpf_hdr` is used only when the timestamp format is set to certain values using the `ioctl()` function [2].

Listing 10: bpf structs cpp

```
struct bpf_xhdr {
    struct bpf_ts    bh_tstamp;     /* time stamp */
    uint32_t      bh_caplen;     /* length of captured portion */
    uint32_t      bh_datalen;    /* original length of packet */
     u_short      bh_hdrlen;     /* length of bpf header (this struct
                           plus alignment padding) */
};

struct bpf_hdr {
    struct timeval   bh_tstamp;     /* time stamp */
    uint32_t      bh_caplen;     /* length of captured portion */
    uint32_t      bh_datalen;    /* original length of packet */
    u_short          bh_hdrlen;     /* length of bpf header (this struct
                         plus alignment padding) */
};
```

The `bh_hdrlen` is used for padding between the header and the link layer protocol, additionally each packet is padded so that it starts on a word [2]. A macro `BPF_WORDALIGN()` is defined in the header that allows us to compute the header-position correctly when we compute the header-positions.

3.3.2 Buffer Modes

BPF devices can deliver packet data to the applications in two different modes, **Buffered read mode** and **Zero-copy mode** [2]. The mode can be set by using the `ioctl()` function using the `BIOCSETBUFMODE` flag. The default mode is the **Buffered read mode** which can also be set using the `BPF_BUFFMODE_BUFFER` flag. In this mode the packet data can be accessed by explicitly call the `read()` function. A fixed buffer size is used for all internal buffers as well as for the `read()` function which can be queried using the `BIOOCGBLEN` flag for the `ioctl()` call and set by using the `BIOCSBLEN` flag in the `iotl()` call [2]. Note that packets longer than this buffer size are truncated. The other mode is the **Zero-copy buffer mode** that can be set using the `ioctl()` call and the `BPF_BUFMODE_ZEROCOPY` flag. In this mode the user process registers two equal sized buffers using the `BIOSETZBUF` flag with the `ioctl` function in which the packet data is directly stored [2]. The user process than has to use atomic operations to check if it can read data from a buffer and than return it to the kernel for storing the next data as fast as possible [2]. See the Unix MAN-pages for detailed information how to use this mode.

3.3.3 IOCTLS

The following flags can be used to change the behavior of the BPF device. All constants are defined in the `<bpf.h>` header, except for the `BIOGETIF` and `BIOSETIF` which also require the `<sys/socket.h>` and `<net/if.h>` headers. They can be set using the `ioctl()` function for any open BPF file, with the type indicated as the third argument of the function. See table 52 in the appendix.

3.3.4 SYSCTL Variables

A set of sysctl-Variables for controlling the behavior of the BPF subsystem exist [2]:

Variable	Description
net.bpf.optimize_writers	Turning this option on makes new BPF users to be attached to write-only interface list until program explicitly specifies read filter via pcap_set_filter().
net.bpf.stats	Binary interface for retrieving general statistics.
net.bpf.zerocopy_enable	Permits zero-copy to be used with net BPF readers.
net.bpf.maxinsns	Maximum number of instructions that BPF program can contain.
net.bpf.maxbufsize	Maximum buffer size to allocate for packets buffer.
net.bpf.bufsize	Default buffer size to allocate for packets buffer.

Table 20: ioctl() flags defined in <bpf.h> [2]

3.3.5 Filter Maschine

An additional capability we have is the Filter machine we want to just show here. It allows to filter the incoming packets for specific packet types. To do this we can define an array of instructions which is executed for every received packet and which does consists of an accumulator, index register, scratch memory store, and implicit program counter [2]. The following listing shows the struct:

Listing 11: bpf structs filter cpp

```
1 struct bpf_insn {
2    u_short code;
3    u_char  jt;
4    u_char  jf;
5    u_long k;
6 };
```

The k field is used in different ways by different instructions, and the jt and jf fields are used as offsets by the branch instructions [2]. There are eight classes of instructions like for example jump and value copy [2]. Various other mode and operator bits are inserted into the class to give the actual instructions [2]. Further information about the filters can be found in the Unix manpages.

3.3.6 Read

The read of a packet using an BPF device is similar to the same operation in a packet socket. The basic structure is shown in figure 9 on the next page.

For a read operation with the BPF we have to do the following:

- We first have to create the BPF device as discussed in the beginning of this section. Also even if we just read data to the buffer it makes sense to also reserve buffer space and initialize it with zero to have defined data in the buffer.
- We then have to attach the BPF device to a interface as discussed in the beginning of this chapter to be able to receive frames [7]. We use a ioctl() call to achieve this.
- Then we can set options, like the BIOCIMMEDIATE mode which was discussed earlier, using ioctl() calls [7]. We also could set a filter to only receive certain packets we are interested in [2].
- As the next step we retrieve the buffer length from the kernel to know how many bytes are received per packet [7]. That is necessary since the received packet also includes a BPF header containing information about the packet received as discussed in the earlier sections.
- Then we can read binary data from the socket with the read function described in section 3.2.7 on page 19. It is recommended to check the return value to know how many bytes were received to be able to adapt accordingly [7]. If we want to read multiple packets from the BPF device we have to use the BPF_WORDALIGN()-macro to align the pointer for the next header correctly [7].
- Optionally we can close() the BPF device now, if we received all the data we want to receive.

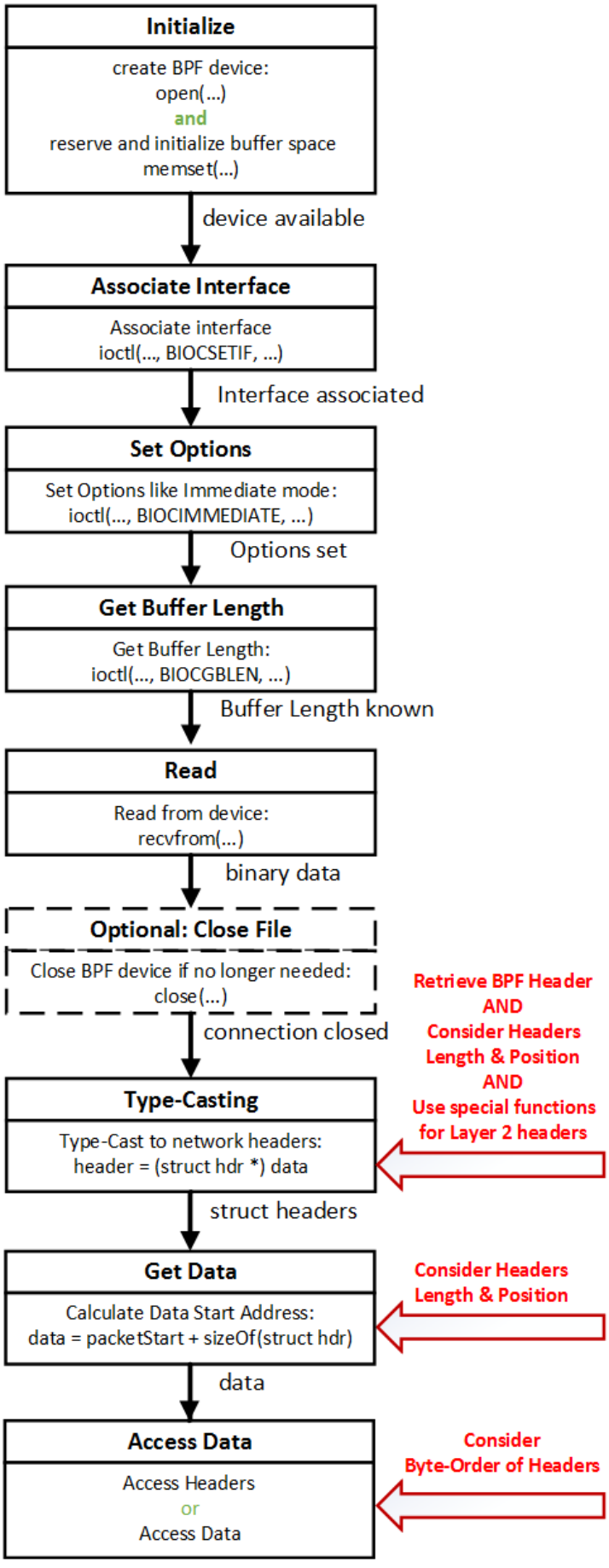

Figure 9: Structure of a BPF device layer 2 Read Operation

- Then we can type-cast the binary data into our headers like it is described in section . We have to take into consideration that there is an additional BPF header attached to the received packet which has to be retrieved first [7]. Since we did start at the Data-Link-layer, there also could be different layer 3 protocols included in the frame. This is why we have to access the correct field in the header to find out what is encapsulated in the frame and which header structure we have to handle. When doing this we have to be aware that there can be variable network layer headers as described in section 3.1.4 on page 14. Also depending on which functions we used for reading data from the socket we need to filter from which source and/or for which target address we want to receive data from. Depending on the protocol and possibly additional variable headers we can compute the start positions of the next header (or headers) and type-cast them into structs.
- After all headers are handled the remainder of the packet is the user data that can also be extracted.
- Then we can access the extracted headers or data. When accessing the headers we have to observer the byte-order, as discussed in the section 3.1.1 on page 13.

3.3.7 Write

As figure 9 on the previous page shows the basic structure, the write operation of a packet using an BPF device is also similar to the same operation in a packet socket.

- We first have to create the BPF device as discussed in the beginning of this section [7]. Also even if we just read data to the buffer it makes sense to also reserve buffer space and initialize it with zero to have defined data in the buffer.
- We then have to attach the BPF device to a interface as discussed in the beginning of this chapter to be able to receive frames [7]. We use a `ioctl()` call to achieve this.
- Then we can set options, like the `BIOCIMMEDIATE` mode which was discussed earlier, using `ioctl()` calls. It also is possible to set the `BIOCGHDRCMPLT` option, which select if the operating system should auto fill in the data link headers[2].
- Then we can type-cast the buffer to our headers like it is described in section . The included headers start at the Data-Link-layer (OSI layer 2). It is necessary to include or define additional headers to have structs for the Data-Link-layer header. We have to take into consideration variable header lengths if we want to optional fields and compute the start of the next header accordingly as described in section 3.1.4 on page 14. The remaining buffer is reserved for the user data we might want to pass along.
- As the next step we set the data of our headers as required. We can have the kernel auto fill the data link layer addresses if we used the `BIOCGHDRCMPLT` option [2]. When we fill in the values of the header fields have to consider the byte-order in case of fields that span multiple bytes and change the byte-order according to the requirements of the protocol we want to use. Additional information about this can be found in section 3.1.1 on page 13.
- If the protocol does usea checksum to protect the header from changes, we have to compute them and set the fields accordingly [5]. We have to see in the documentation which kind of checksum is required by the protocol we want to use. Additional information about computing a checksum can be found in the section 3.1.2 on page 13.
- In the next step we send the data using the `write()` function as discussed earlier.
- Optionally we can `close()` the BPF device now, if we already sent all the data we want to transmit.

3.4 Winsock-API

As mentioned in section 2.1.1 on page 10 the Winsock-API does not allow any user interaction by itself, which would justify the usage of RAW-sockets. The rolled out Winsock-API is designed to be used with usual sockets using predefined protocols. Indeed, there is the possibility to use `SOCKET_RAW`, but it is mostly used for diagnostics. The limits, as also shown above are limiting to internet procotols `IP`, `UDP` and `SCTP`.

To write own protocols using Windows, or to receive packtes, which are netiher known or specified, there is the need to used libraries (npcap, libnet) to be able to support the Windows infrastructure. Otherwise the *invalid* or more *falsely marked as invalid* packtes will be dropped by driver and OS.

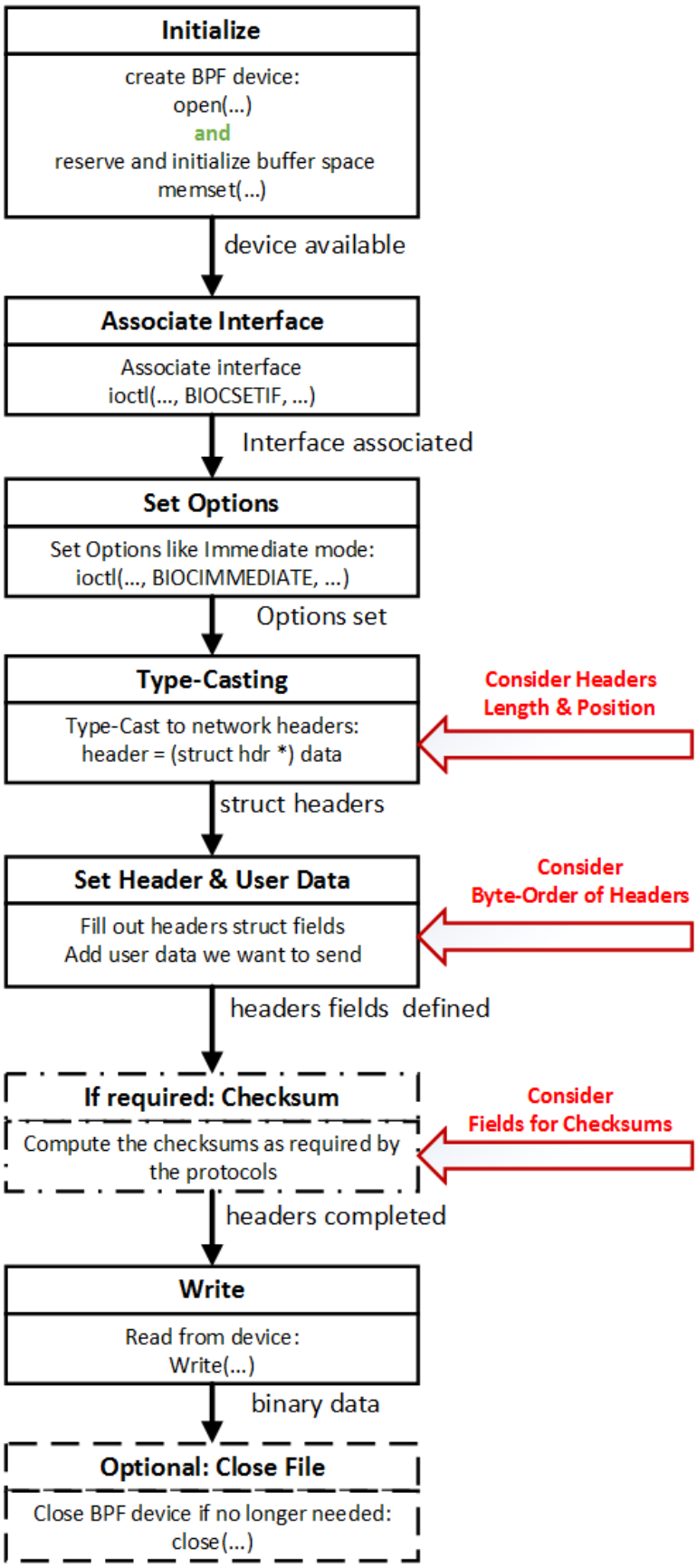

Figure 10: Structure of a BPF device layer 2 Write Operation

3.4.1 Preparations and Usage

If anyway there is no possibility to use the libraries, sockets of type `SOCK_RAW` can be created, despite being restricked as mentioned. A call to the socket function with *address family* set to `AF_INET` or `AF_IP6` and *type* to `SOCK_RAW` will return a raw ip socket. For the netork layer one of the mentioned rescricted protocols need to be set in *protocol* parameter. The following listing provides an extended example of this socket-function call. It also gives valid example values for the *type*, *family* and *protocol*. Complete descreption of the socket-function is found at msdn: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms740506(v=vs.85).aspx`

See listing 13 in the appendix.

4 Programming with the libraries

This chapter will cover the two libraries that provide packet injection and capturing mechanisms for the most common operating systems. First, libnet, which is used for the packet injection, is introduced then we continue with the packet capture library, called pcap.

4.1 Libnet

Libnet is a library, mainly written in C, that defines an high-level portable interface for low-level network packet construction and injection. When libnet was designed the goal was to abstract out the pedantic architecture-specific details of low-level network packet tasks and provide some kind of platform-independent standard. It is under the BSD Licence and offers the ability to create and manipulate network packets from the link layer upwards. For each network layer libnet supports the manipulation of a set of protocols which are listed in figure 11. Libnet only supports the creation and sending of packets. It does not provide any functionalities to receive these packets. For latter purpose, one has to rely on an additional library called pcap.

The list of supported Operating Platforms is unfortunately no longer maintained. The latest state was that libnet supports the following OS [13]:

- Linux
- Windows
- FreeBSD
- OS X
- Solaris

The current version of libnet is 1.2. It is important to note that code from version 1.0.x will not work anymore as the syntax was simplified with version 1.1. Libnet was originally maintained on packetfactory.net by Mike D. Schiffman until 2004. However this page does not exist anymore and the author is also unreachable.

Since 2009 Sam Roberts maintains libnet on GitHub. As of today (December, 2016) the latest release is already four years ago but is still actively used [14].

4.1.1 Preparations

Libnet can be downloaded from the official libnet GitHub repository [14]. Alternatively, the currently latest version 1.2 is also available at sourceforge [15]. To install libnet, go to the directory, unzip it and run the following commands:

1. `./configure`
2. `make`
3. `make install` (might require root permissions)

Now <libnet.h>can be included in a C program. In order to successfully compile the program, run:

1. `gcc` -Wall -g ‘*libnet-config –defines*‘ -c **foo.c**
2. `gcc` -Wall **foo.o** -o **output** ‘*libnet-config –libs*‘

4.1.2 Process of packet creation

In order to build and inject a network packet in libnet, there is a standard order of four operations:

1. **Library initialization**
2. **Packet Building**
3. **Packet Writing**
4. (Optional) **Packet destruction**

Library initialization The first step is to initialize the libnet library and to set up the environment. Doing so, the programmer receives a so called *libnet context*. This context maintains the state for the entire session which tracks all memory usage and packet construction. The *libnet context* is also often required as parameter for several functions such as packet building or injection.

Note that, the libnet library initialization can only be successfully executed with root permissions.