Network Programming Laboratory
Winter 2018/2019

Session 1: Wireshark and Sockets API UDP Echo Server and Client

Janne Riihijärvi

RWTH Aachen University - Institute for Networked Systems

Contents

Introduction

Welcome to the first Network Programming lab session. We hope you will enjoy it and learn things you consider useful. Please spend some time studying this text carefully before coming to the lab. This preparation is helpful to solve the tasks and ensure a productive lab time.

This session will deal with three topics. In the first part we conduct some C programming exercises to brush up on the basics of the language and to become familiar with the programming environment. In the second part you will learn how to use some common networking tools under Linux, like Wireshark to capture and analyze the transmitted packages. These skills will be very useful later on in debugging programs using the network. Then you will program a client that reads strings from the keyboard and sends them over a UDP connection to another program (on your own or another computer). Then it should wait for a return message and display it on the screen. For the other side, program a server that waits for incoming messages and returns them to the sender.

1 The Sockets API

1.1 The Sockets Concept

To facilitate the programming of networked applications, the abstraction of sockets was introduced (because they were first used in the BSD version of UNIX, they are sometimes called BSD sockets). The sockets are a unified interface, over which an application can exchange data with the network. This interface looks the same for all protocols. The choice of the protocol (e.g. TCP or UDP) is specified using three parameters: *family*, *type* and *protocol*.

The Sockets API follows the UNIX paradigm to map all objects, which can be accessed to read or write, to files, so that they can be processed with the normal file I/O operations (like `read` and `write`). Sending and receiving in communications fits well to that approach. The communication endpoints are the sockets. Sending data thus means writing it to the socket (like to a file).

In the first three sessions of this lab course we will only work with TCP and UDP sockets. They both belong to the protocol family `PF_INET`, but have a different type: the type `SOCK_STREAM` specifies TCP, the type `SOCK_DGRAM` UDP.

It is also possible to directly access the IP layer (layer 3) by using raw sockets (type `SOCK_RAW`). Linux also allows an elegant method to directly access a network device (layer 2) using sockets of protocol family `PF_PACKET`.

1.2 Address Structures

Messages must be addressed somehow to reach a destination, and this addressing is mapped to a socket. To specify an address of any kind (generic for all protocols), the following address structure is provided:[1]

```
typedef unsigned short sa_family_t;
struct sockaddr {
```

[1]NOTE: All code is written in C language and is assumed to be compiled with gcc.

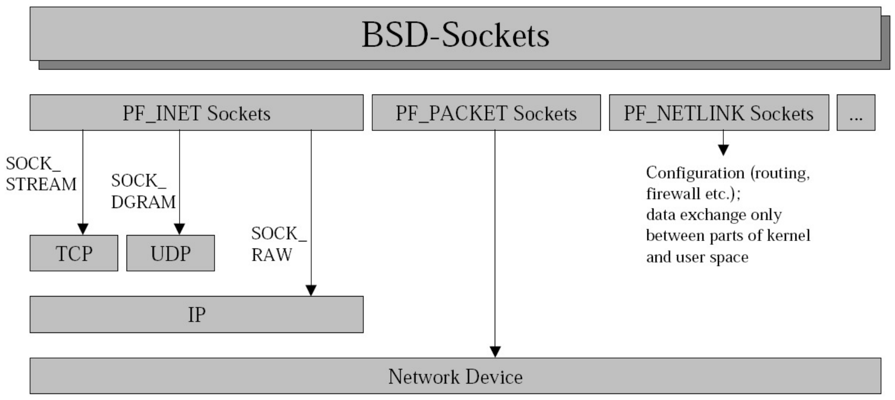


Figure 1: Structure of sockets support in Linux.

```
  sa_family_t sa_family;
  char        sa_data[14];
};
```

The element `sa_family` specifies the address family, e.g. `AF_INET` for the family of Internet protocols (`PF_INET`). The data area `sa_data[14]` is filled differently by different protocols. For IPv4 addresses the following structure `sockaddr_in` is defined:

```
struct in_addr { ___u32 s_addr; };
struct sockaddr_in {
  sa_family_t        sin_family; // Address family AF_INET
  unsigned short int sin_port;   // Port number
  struct in_addr     sin_addr;   // IPv4 address
  // Pad to the size of struct socaddr
  unsigned char      sin_zero[sizeof(struct sockaddr)
                          - sizeof(sa_family_t)
                          - sizeof(uint_16_t)
                          - sizeof(struct in_addr)];
};
```

The protocol specific address structure is of course used for addressing, but when calling functions of the sockets API, their parameter is of type `struct sockaddr*`. Therefore, a pointer to the protocol specific address structure must be typecast to fit.

1.3 Functions for Connection Control

Sockets are represented at the API through socket descriptors, which can be handled like normal file descriptors (reading and writing). But because for example the establishment of a communication connection differs from opening a file, additional system calls are available for sockets.

Figure 2 shows which system calls are to be used and in which order. It distinguishes between the client role (left) and the server role (right). This

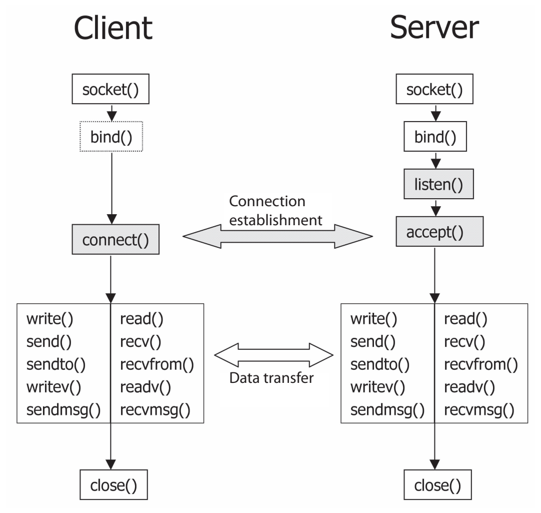


Figure 2: System calls of the sockets API. Gray shaded calls are not needed for connectionless protocols like UDP. Adapted from *Linux Netzwerkarchitektur*.

distinction does not refer to data transfer, but simply to the connection establishment: the program initiating the connection is the client, while a server is passively listening for that.

1.3.1 `socket()`

A new socket is created using the `socket()` call. The protocol (e.g. TCP or UDP) must be specified by the parameters `family`, `type` and `protocol`. Returned is a socket descriptor, that will be a parameter in future calls like `bind()`, `connect()` or `close()`. If an error occurs, -1 is returned. The error code is then contained in `errno` and can be output for example by calling `perror()`.

```
#include <sys/types.h>  // for things like AF_INET, SOCK_DGRAM
#include <sys/socket.h> // for socket()

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

1.3.2 `close()`

The normal UNIX system call `close()` is used to close a socket. Its sole parameter is the socket descriptor. `close()` returns zero on success, `-1` if an error occured.

```
int close(int s);
```

1.3.3 `bind()`

A newly created socket doesn't have any mapping to a local address or port number. `bind()` provides such a mapping. For a client's local address it doesn't have to be called, a port number is then selected automatically (ephemeral port number). For a server address the port number must be specified, because otherwise the client couldn't address the server.

```
int bind(int s, struct sockaddr* local_addr, int addr_len);
```

1.3.4 `listen()`

The system call `listen()` prepares a socket (on the server side) for incoming connections. The operating system will queue incoming connection requests into a socket queue. `listen()` is only used for connection oriented socket types (e.g. `SOCK_STREAM`). Parameters are the socket descriptor and the maximum number of connection requests to be queued (normally 5).

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

1.3.5 `accept()`

The system call `accept()` dequeues the next connection request from the socket queue that was initiated using `listen()`. If the queue is empty when `accept()` is called, the process is blocked. It will be scheduled again by the operating system when a new connection request arrives.

After writing the source address of the connection request (client address and port number) into the address structure pointed to by `peer_addr`, the operating system creates a new socket for this connection and returns its socket descriptor as return value of `accept()`. In case of an error, -1 is returned. The server thus has one listening socket (for incoming connection requests) and probably several connected sockets, one for each connection.

```
int accept(int s, struct sockaddr* peer_addr, int* addr_len);
```

1.3.6 `connect()`

Before a client can exchange data with a server using a connection oriented protocol (like TCP), it must create a socket (using the `socket()` call) and then establish a connection to the server using the `connect()` call. Later, when exchanging data, the server address doesn't need to be specified again.

For sockets for connectionless datagram services (e.g. UDP), the `connect()` call doesn't have to be used. For them, data can be sent directly (by specifying the server address).

```
int connect(int s, struct sockaddr* server_addr, int addr_len);
```

1.4 Functions for Data Transfer

Sending Data

To send data over a socket, there are among others the following system calls (for the header files to include consult the man-pages):

- `size_t write(int s, const void* buffer, size_t len);`
- `int send(int s, const void* buffer, size_t len, int flags);`

- `int sendto(int s, const void* buffer, size_t len, int flags, const struct sockaddr* to_addr, socklen_t to_addr_len);`

The parameter `void* buffer` is the start address of a buffer that contains the data to be sent. The next parameter `size_t len` specifies the number of bytes to be sent starting from the buffer beginning. `flags` specify the transfer control—if you want to know what that can do, please refer to the man page (e.g. `man sendto`). All three functions return the number of bytes that have been sent.

While `write()` and `send()` can only be called if the socket is in the connected state, `sendto()` can always be called (because it has the destination address as parameter).

Receiving Data

Analog to the sending functions described above, the following receive functions exist:

- `size_t read(int s, void* buffer, size_t len);`
- `int recv(int s, void* buffer, size_t len, int flags);`
- `int recvfrom(int s, void* buffer, size_t len, int flags, struct sockaddr* source_addr, socklen_t* source_addr_len);`

All three functions return the number of bytes that have been read. If there is nothing to read, the process is by default blocked by these system calls (can be changed using `flags`). It gets scheduled again by the operating system when there is something to read.

Additionally, the functions `readv()`, `writev()`, `sendmsg()` and `recvmsg()` are available for data transfer. They can be useful for example when messages consist of several parts.

1.5 Functions for Byte Ordering

Not all processors save the single bytes of larger types in the same order. The two ways to do it are called *little-endian* and *big-endian*. Intel processors are normally little-endian, while Sun processors are normally big-endian.

To correctly exchange data between those machines, a byte ordering has been specified. It is called the *network byte order*—in contrast to the *host byte order* (for the TCP/IP protocol family the network byte order has been defined as big-endian, but this is more of general interest). Your code should of course be able to run on all machines. Therefore several functions have been defined to map between host and network byte order. Their implementation on different machines can be different (on some machines they will do nothing), but you don't have to care about that. It is important to use the functions to make your code portable. The functions are (again consult the man pages for headers to be included):

- 32-bit host $\rightarrow$ net: `unsigned long htonl(unsigned long hostlong);`
- 16-bit host $\rightarrow$ net: `unsigned short htons(unsigned short hostshort);`

- 32-bit net → host: `unsigned long ntohl(unsigned long netlong);`
- 16-bit net → host: `unsigned short ntohs(unsigned short netshort);`

The functions for 32-bit integers are used for example for IPv4 addresses, while the functions for 16-bit integers are for example used for port numbers.

1.6 Functions to Handle Internet Addresses

Several funtions are provided to handle IP addresses and DNS names. IP addresses are normally given in the dotted decimal notation as a string (like `134.130.222.125`), but they are stored and processed as 32-bit unsigned integers. The following functions do the mapping:

- ASCII → numerical:
  `int inet_aton(const char* cp, struct in_addr* inp);`
- numerical → ASCII:
  `char* inet_ntoa(struct in_addr in);`
- `unsigned long int inet_addr(const char* cp);`

Instead of `inet_ntoa()` and `inet_aton()` the following two functions can be used. They have the advantage of supporting several address families (e.g. `AF_INET` and `AF_INET6`). Therefore it is often recommended to use them:

- `int inet_pton(int af, const char* src, void* dst);`
- `const char* inet_ntop(int af, const void* src, char* dst, size_t cnt);`

To get the IP address of your communication peer from your connected socket, the function `getpeername()` is available. With `gethostname()` you can get the DNS name (not the IP address) of your own computer. If you know the name of the communication peer, you can get its address using `gethostbyname()`.

2 Wireshark

2.1 `wireshark`—New Visual Packet Sniffing

Wireshark is an upgraded version of the classical tool Ethereal for capturing and analyzing network traffic. It becomes now the world's most popular network protocol analyzer. A short summary on use of Wireshark is provided through L2P. You will be using Wireshark to capture/understand the transmitted packages in a typical socket scenario.

3 Programming Task

3.1 Recommended Development Environment

We recommend using `eclipse` as the editor and the environment for debugging. The compiler is `gcc`, but `eclipse` knows how to call the compiler directly from the user interface. If you dislike `eclipse`, you can also use other editors or IDEs like for example `gedit` and compile your code manually.

3.2 UDP Data Transfer

As a step towards the echo server and client, we recommend you to start with simplified programs: the client reads data from the keyboard and transmits it to the server, while the server is waiting for it and just puts it on the screen.

If you have trouble getting started, you might find the following tips useful:

Server

- Creates a socket and binds the local address to it (choose port number).
- For a UDP server you don't have to use `listen()` and `accept()` like for a TCP server; you can directly call `recvfrom()`.

Client

- Creates a socket.
- Can directly send data using `sendto()` (with the correct server address and port number).

It is normally convenient to test network applications (client and server) on your own computer. To do that you can address messages to the loopback interface (`lo`) with IP-address 127.0.0.1. You can then also watch the packets using for example `Wireshark` for that interface.

3.3 Echo Server and Client

Now extend the previous step to the echo client and server. The client should read a string from the keyboard, transmit it to the server, then wait for the reply string and put it on the screen. The server should receive the string and echo it.

4 In the Lab

4.1 Getting familiar with the environment

- In the beginning of the lab you will have some time to brush up on C basics and become familiar with Eclipse by going through basic C programming exercises. These will be circulated in the beginning of the lab session.

4.2 Wireshark

- Start by capturing network traffic generated by at least two different applications (you can use, for example, a web browser and the `traceroute` tools for this purpose).

- Learn how to filter the traffic displayed only to show traffic originating from your computer with a specific protocol. Try to see how `ping` and `traceroute` work using this functionality.
- Use the TCP connection tracing functionality to see how a web browser establishes a connection, downloads a webpage, and closes down the connection again. Understanding this process will be very important in the coming sessions.

4.3 Programming

- Yes, the echo client and server, please.

5 Useful Networking Tools and Commands

This is a small but powerful selection of a wide variety of available tools. Understanding these commands is important for network programming applications. For the syntax please refer to the manual pages (`man`). To use most of the tools and commands, you will need root privileges.

5.1 `ifconfig`—Device Administration

For configuration of a network device under Linux the command `ifconfig` is used. It mainly provides activating, deactivating, and configuring of a device and its physical adapters. Protocol specific parameters like address and netmask as well as interface specific parameters like I/O port and interrupt can be set. To use a network device, it must be activated with `ifconfig` (and the adapter must be known by the kernel). The command does not store the information permanently. Upon reboot it is lost.

5.2 `route`—Route Selection

The `route` command is used to show and manipulate the IP routing table (static routing). Dynamic routes are set by routing protocol (e.g. Routing Information Protocol, RIP). `route` knows exactly two options, one to set static routes and one to remove them. Packets with addresses for which no directly fitting entry is found in the routing table are sent using the default entry.

5.3 `ping`—Testing Reachability

`ping` is used to test if a host is reachable over the network. `ping` sends an "ICMP Echo Request" packet to the specified computer and expects an "ICMP Echo Reply" back. If `ping` is complaining about an "unknown host", then most probably the name service is not correctly configured—try pinging the IP address instead of the hostname then. If no echo is returned however, you cannot say the other computer is not reachable, because a firewall may be configured not to respond to ping. Sometimes this is done to look invisible to denial-of-service (DOS) attackers who use ping to watch a machine and launch an attack when its presence is detected.

5.4 `traceroute`—Packet Tracing

As the name of this program indicates, it provides two interesting pieces of information:

1. Do your packets reach their destination?
2. If not, where are they stopped?

Therefore, `traceroute` is often the next step if a computer is not reachable with `ping`. `traceroute` works by sending UDP packets with a small time to live (TTL) value towards the destination. Every host that is passed is counting the TTL down. If the TTL reaches zero, an ICMP "time exceeded" is replied. The TTL is then increased by one. Because the UDP packets are not intended to be processed, an unlikely port is chosen (can be configured). `traceroute` continues, until either ICMP "port unreachable" is returned (destination reached) or the TTL has reached its maximum (can be configured). If there is no answer within a certain duration, a "*" is printed for that probe.

An interesting option is to use traceroute with the IP source routing option. With strict source routing the exact path for the packet is specified, with loose source routing a list of IP addresses can be given that must be passed.

5.5 `host`—Name Service

`host` is a simple utility to perform DNS lookups. It converts names to IP addresses and vice versa.

5.6 `netstat`—Network Status

`netstat` prints information about the Linux networking subsystem. It can print network connections, routing tables, interface statistics, masquerade connections and multicast memberships.

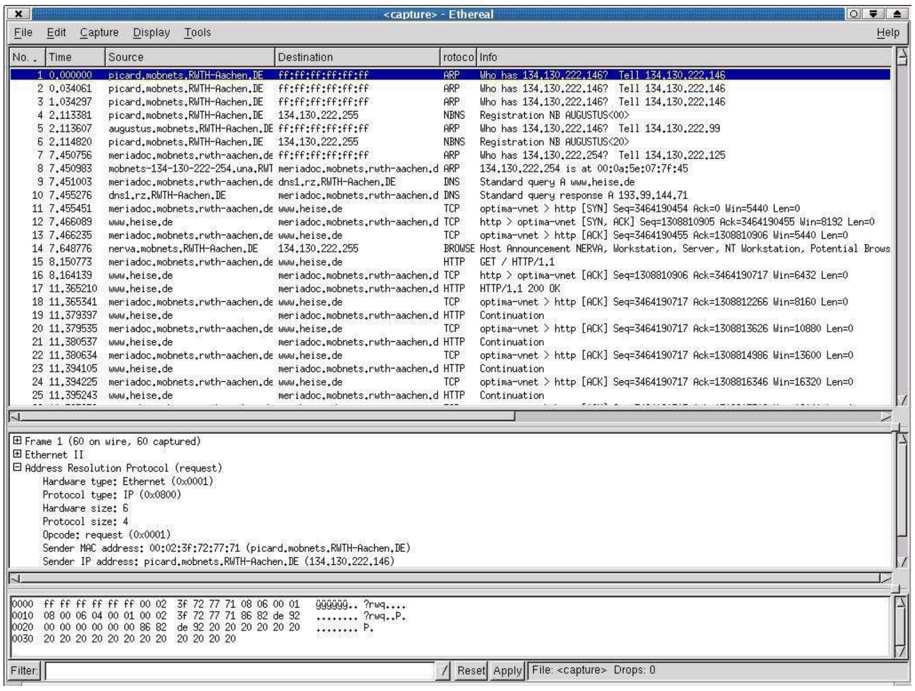
5.7 `tcpdump`—Packet Sniffing

`tcpdump` prints out the headers of packets on a network interface. The headers can be filtered by specifying a boolean expression (e.g. to only output traffic of a special protocol). `tcpdump` operates by putting the network interface card into promiscuous mode, so that every packet going across the wire is captured. Normally interface cards only capture link level frames addressed to the particular interface or to the broadcast address.

5.8 `ethereal`—Visual Packet Sniffing

This program provides similar functionality to `tcpdump`, but shows the information graphically. In comparison to `tcpdump` and `ntop` (see 5.9) it is normally used for detailed analysis of one datastream. It is useful for example if you want to reassemble and monitor data of a complete TCP conversation (like transporting emails)[2]. However, there has been no active development on Ethereal since the tool has been replaced by its fork "Wireshark".

[2]On the command line similar information can be extracted from `tcpdump` data using the tool `tcpflow`.

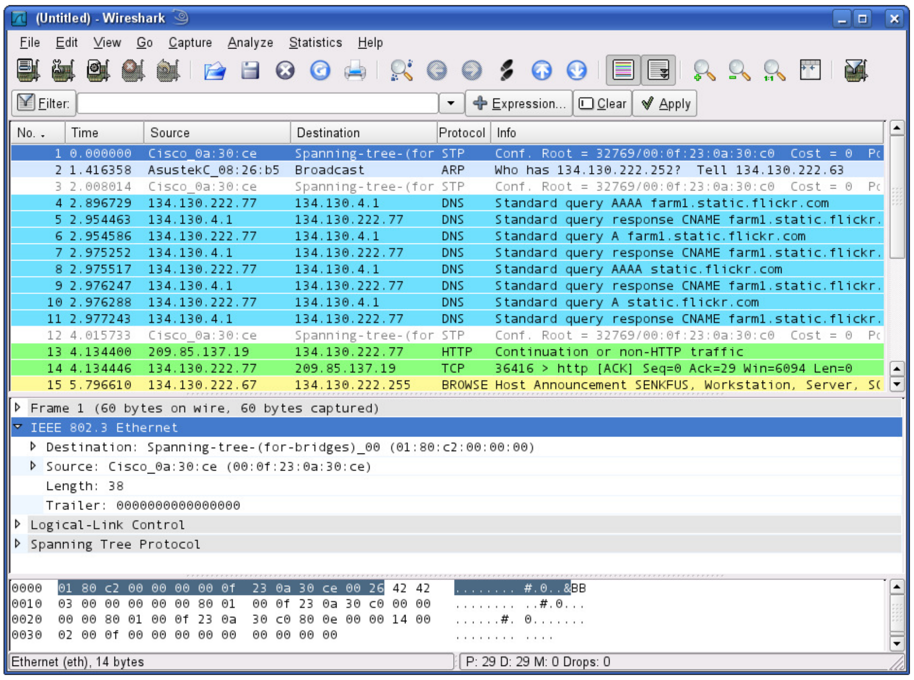


```
No. . Time      Source                         Destination                    rotoco Info
  1 0.000000  picard.mobnets.RWTH-Aachen.DE  ff:ff:ff:ff:ff:ff              ARP    Who has 134.130.222.146?  Tell 134.130.222.146
  2 0.034061  picard.mobnets.RWTH-Aachen.DE  ff:ff:ff:ff:ff:ff              ARP    Who has 134.130.222.146?  Tell 134.130.222.146
  3 1.034297  picard.mobnets.RWTH-Aachen.DE  ff:ff:ff:ff:ff:ff              ARP    Who has 134.130.222.146?  Tell 134.130.222.146
  4 2.113381  picard.mobnets.RWTH-Aachen.DE  134.130.222.255                NBNS   Registration NB AUGUSTUS<00>
  5 2.113607  augustus.mobnets.RWTH-Aachen.DE ff:ff:ff:ff:ff:ff             ARP    Who has 134.130.222.146?  Tell 134.130.222.99
  6 2.114820  picard.mobnets.RWTH-Aachen.DE  134.130.222.255                NBNS   Registration NB AUGUSTUS<20>
  7 7.450756  meriadoc.mobnets.rwth-aachen.de ff:ff:ff:ff:ff:ff             ARP    Who has 134.130.222.254?  Tell 134.130.222.125
  8 7.450983  mobnets-134-130-222-254.una.RWT meriadoc.mobnets.rwth-aachen.d ARP    134.130.222.254 is at 00:0a:5e:07:7f:45
  9 7.451003  meriadoc.mobnets.rwth-aachen.de dns1.rz.RWTH-Aachen.DE        DNS    Standard query A www.heise.de
 10 7.455276  dns1.rz.RWTH-Aachen.DE         meriadoc.mobnets.rwth-aachen.d DNS    Standard query response A 193.99.144.71
 11 7.455451  meriadoc.mobnets.rwth-aachen.de www.heise.de                  TCP    optima-vnet > http [SYN] Seq=3464190454 Ack=0 Win=5440 Len=0
 12 7.466089  www.heise.de                   meriadoc.mobnets.rwth-aachen.d TCP    http > optima-vnet [SYN, ACK] Seq=1308810905 Ack=3464190455 Win=8192 Len=0
 13 7.466235  meriadoc.mobnets.rwth-aachen.de www.heise.de                  TCP    optima-vnet > http [ACK] Seq=3464190455 Ack=1308810906 Win=5440 Len=0
 14 7.648776  nerva.mobnets.RWTH-Aachen.DE   134.130.222.255                BROWSE Host Announcement NERVA, Workstation, Server, NT Workstation, Potential Brows
 15 8.150773  meriadoc.mobnets.rwth-aachen.de www.heise.de                  HTTP   GET / HTTP/1.1
 16 8.164139  www.heise.de                   meriadoc.mobnets.rwth-aachen.d TCP    http > optima-vnet [ACK] Seq=1308810906 Ack=3464190717 Win=6432 Len=0
 17 11.365210 www.heise.de                   meriadoc.mobnets.rwth-aachen.d HTTP   HTTP/1.1 200 OK
 18 11.365341 meriadoc.mobnets.rwth-aachen.de www.heise.de                  TCP    optima-vnet > http [ACK] Seq=3464190717 Ack=1308812266 Win=8160 Len=0
 19 11.379397 www.heise.de                   meriadoc.mobnets.rwth-aachen.d HTTP   Continuation
 20 11.379535 meriadoc.mobnets.rwth-aachen.de www.heise.de                  TCP    optima-vnet > http [ACK] Seq=3464190717 Ack=1308813626 Win=10880 Len=0
 21 11.380537 www.heise.de                   meriadoc.mobnets.rwth-aachen.d HTTP   Continuation
 22 11.380634 meriadoc.mobnets.rwth-aachen.de www.heise.de                  TCP    optima-vnet > http [ACK] Seq=3464190717 Ack=1308814986 Win=13600 Len=0
 23 11.394105 www.heise.de                   meriadoc.mobnets.rwth-aachen.d HTTP   Continuation
 24 11.394225 meriadoc.mobnets.rwth-aachen.de www.heise.de                  TCP    optima-vnet > http [ACK] Seq=3464190717 Ack=1308816346 Win=16320 Len=0
 25 11.395243 www.heise.de                   meriadoc.mobnets.rwth-aachen.d HTTP   Continuation
```

```
⊞ Frame 1 (60 on wire, 60 captured)
⊞ Ethernet II
⊟ Address Resolution Protocol (request)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (0x0001)
    Sender MAC address: 00:02:3f:72:77:71 (picard.mobnets.RWTH-Aachen.DE)
    Sender IP address: picard.mobnets.RWTH-Aachen.DE (134.130.222.146)
```

```
0000  ff ff ff ff ff ff 00 02  3f 72 77 71 08 06 00 01   ÿÿÿÿÿÿ.. ?rwq....
0010  08 00 06 04 00 01 00 02  3f 72 77 71 86 82 de 92   ........ ?rwq..P.
0020  00 00 00 00 00 00 86 82  de 92 20 20 20 20 20 20   ........ P.
0030  20 20 20 20 20 20 20 20  20 20 20 20
```

Figure 3: Packet sniffing with `ethereal`.



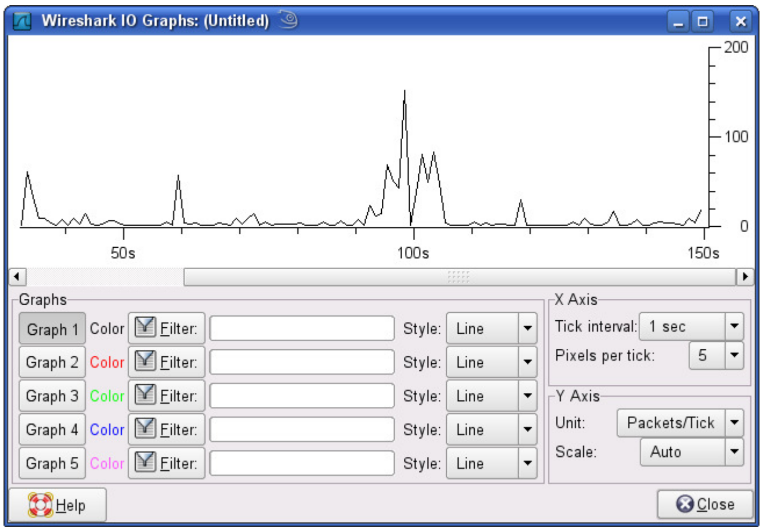
(Untitled) - Wireshark
File Edit View Go Capture Analyze Statistics Help
Filter: Expression... Clear Apply
Ethernet (eth), 14 bytes
P: 29 D: 29 M: 0 Drops: 0

```
No. . Time      Source             Destination          Protocol Info
    1 0.000000  Cisco_0a:30:ce     Spanning-tree-(for   STP      Conf. Root = 32769/00:0f:23:0a:30:c0  Cost = 0  Pc
    2 1.416358  AsustekC_08:26:b5  Broadcast            ARP      Who has 134.130.222.252?  Tell 134.130.222.63
    3 2.008014  Cisco_0a:30:ce     Spanning-tree-(for   STP      Conf. Root = 32769/00:0f:23:0a:30:c0  Cost = 0  Pc
    4 2.896729  134.130.222.77     134.130.4.1          DNS      Standard query AAAA farm1.static.flickr.com
    5 2.954463  134.130.4.1        134.130.222.77       DNS      Standard query response CNAME farm1.static.flickr.
    6 2.954586  134.130.222.77     134.130.4.1          DNS      Standard query A farm1.static.flickr.com
    7 2.975252  134.130.4.1        134.130.222.77       DNS      Standard query response CNAME farm1.static.flickr.
    8 2.975517  134.130.222.77     134.130.4.1          DNS      Standard query AAAA static.flickr.com
    9 2.976247  134.130.4.1        134.130.222.77       DNS      Standard query response CNAME farm1.static.flickr.
   10 2.976288  134.130.222.77     134.130.4.1          DNS      Standard query A static.flickr.com
   11 2.977243  134.130.4.1        134.130.222.77       DNS      Standard query response CNAME farm1.static.flickr.
   12 4.015733  Cisco_0a:30:ce     Spanning-tree-(for   STP      Conf. Root = 32769/00:0f:23:0a:30:c0  Cost = 0  Pc
   13 4.134400  209.85.137.19      134.130.222.77       HTTP     Continuation or non-HTTP traffic
   14 4.134446  134.130.222.77     209.85.137.19        TCP      36416 > http [ACK] Seq=0 Ack=29 Win=6094 Len=0
   15 5.796610  134.130.222.67     134.130.222.255      BROWSE   Host Announcement SENKFUS, Workstation, Server, S(
```

```
▷ Frame 1 (60 bytes on wire, 60 bytes captured)
▽ IEEE 802.3 Ethernet
  ▷ Destination: Spanning-tree-(for-bridges)_00 (01:80:c2:00:00:00)
  ▷ Source: Cisco_0a:30:ce (00:0f:23:0a:30:ce)
    Length: 38
    Trailer: 0000000000000000
▷ Logical-Link Control
▷ Spanning Tree Protocol
```

```
0000  01 80 c2 00 00 00 00 0f  23 0a 30 ce 00 26 42 42   ........ #.0..&BB
0010  03 00 00 00 00 00 80 01  00 0f 23 0a 30 c0 00 00   ........ ..#.0...
0020  00 00 80 01 00 0f 23 0a  30 c0 80 0e 00 00 14 00   ......#. 0.......
0030  02 00 0f 00 00 00 00 00  00 00 00 00               ........ ....
```

Figure 4: Packet sniffing with `wireshark`.



Graph 1 Color Filter: Style: Line
Graph 2 Color Filter: Style: Line
Graph 3 Color Filter: Style: Line
Graph 4 Color Filter: Style: Line
Graph 5 Color Filter: Style: Line

Tick interval: 1 sec
Pixels per tick: 5
Unit: Packets/Tick
Scale: Auto

Figure 5: A throughput plot using `wireshark`.

5.9 `ntop`—Traffic Statistics

`ntop` is a traffic measurement and monitoring tool. It was originally programmed as a simple tool able to report the top network users (hence the name `ntop`) to quickly identify those hosts that were currently using most of the available network resources. Then it evolved into a more powerful tool. `ntop` normally works browser-based, i.e. it provides its information in the form of web pages.
exameple:

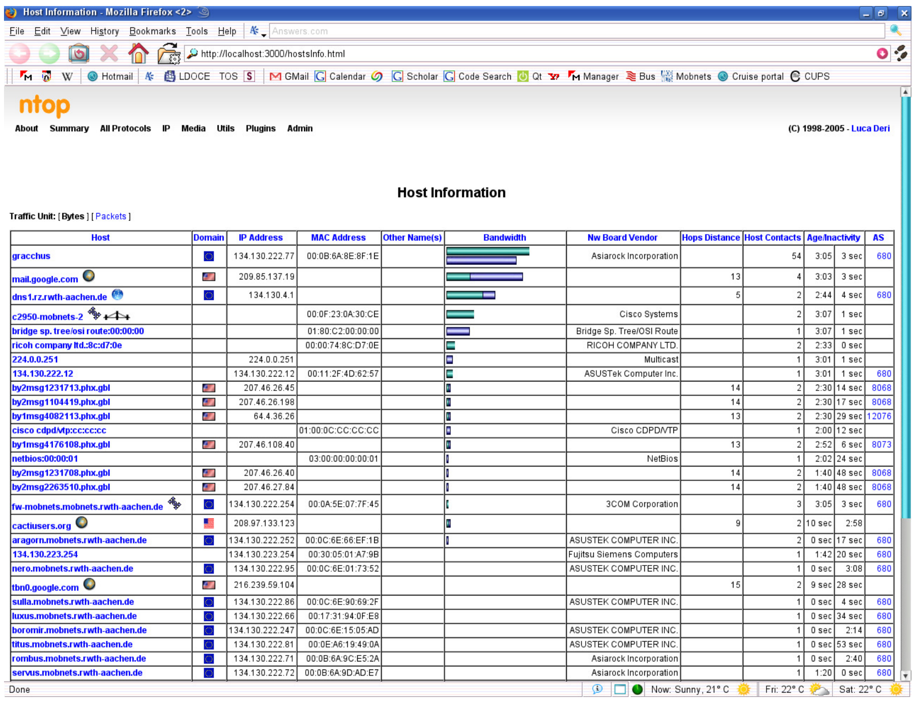
```
ntop -w 3000
```

In your web brower, put `http://localhost:3000`

5.10 `nmap`—Port Scanner

`nmap` is able to scan large networks to determine which hosts are up and what services they are offering. `nmap` supports a large number of scanning techniques (see the man page). It also offers a number of advanced features as for example remote OS detection via TCP/IP fingerprinting.

The result of running `nmap` is usually a list of interesting ports on the machines being scanned. `nmap` always gives the port's "well known" service name, number, state, and protocol. The state is either *open*, *filtered*, or *unfiltered*. Open means that the target machine will `accept()` connections on that port. Filtered means that a firewall, filter, or other network obstacle is covering the port and preventing `nmap` from determining whether the port is open. Unfiltered means that the port is known by `nmap` to be closed and no firewall/filter seems to be interfering with its attempts to determine this. Unfiltered ports are the common case and are only shown when most of the scanned ports are in the filtered state.


Host Information - Mozilla Firefox <2>
File Edit View History Bookmarks Tools Help
Answers.com
http://localhost:3000/hostsInfo.html
W Hotmail LDOCE TOS GMail Calendar Scholar Code Search Qt Manager Bus Mobnets Cruise portal CUPS
ntop
About Summary All Protocols IP Media Utils Plugins Admin
(C) 1998-2005 - Luca Deri
Done
Now: Sunny, 21° C
Fri: 22° C
Sat: 22° C

Host Information

Traffic Unit: [**Bytes**] [Packets]

Host	Domain	IP Address	MAC Address	Other Name(s)	Bandwidth	Nw Board Vendor	Hops Distance	Host Contacts	Age/Inactivity		AS
gracchus	🇪🇺	134.130.222.77	00:0B:6A:8E:8F:1E		▬	Asiarock Incorporation		54	3:05	3 sec	680
mail.google.com	🇺🇸	209.85.137.19			▬		13	4	3:03	3 sec	
dns1.rz.rwth-aachen.de	🇪🇺	134.130.4.1			▬		5	2	2:44	4 sec	680
c2950-mobnets-2			00:0F:23:0A:30:CE		▬	Cisco Systems		2	3:07	1 sec	
bridge sp. tree/osi route:00:00:00			01:80:C2:00:00:00		▬	Bridge Sp. Tree/OSI Route		1	3:07	1 sec	
ricoh company ltd.:8c:d7:0e			00:00:74:8C:D7:0E		▬	RICOH COMPANY LTD.		2	2:33	0 sec	
224.0.0.251		224.0.0.251			▬	Multicast		1	3:01	1 sec	
134.130.222.12		134.130.222.12	00:11:2F:4D:62:57		▬	ASUSTek Computer Inc.		1	3:01	1 sec	680
by2msg1231713.phx.gbl	🇺🇸	207.46.26.45			▬		14	2	2:30	14 sec	8068
by2msg1104419.phx.gbl	🇺🇸	207.46.26.198			▬		14	2	2:30	17 sec	8068
by1msg4082113.phx.gbl	🇺🇸	64.4.36.26			▬		13	2	2:30	29 sec	12076
cisco cdpd/vtp:cc:cc:cc			01:00:0C:CC:CC:CC		▬	Cisco CDPD/VTP		1	2:00	12 sec	
by1msg4176108.phx.gbl	🇺🇸	207.46.108.40			▬		13	2	2:52	6 sec	8073
netbios:00:00:01			03:00:00:00:00:01		▬	NetBios		1	2:02	24 sec	
by2msg1231708.phx.gbl	🇺🇸	207.46.26.40			▬		14	2	1:40	48 sec	8068
by2msg2263510.phx.gbl	🇺🇸	207.46.27.84			▬		14	2	1:40	48 sec	8068
fw-mobnets.mobnets.rwth-aachen.de	🇪🇺	134.130.222.254	00:0A:5E:07:7F:45		▬	3COM Corporation		3	3:05	3 sec	680
cactiusers.org	🇺🇸	208.97.133.123			▬		9	2	10 sec	2:58	
aragorn.mobnets.rwth-aachen.de	🇪🇺	134.130.222.252	00:0C:6E:66:EF:1B		▬	ASUSTEK COMPUTER INC.		2	0 sec	17 sec	680
134.130.223.254		134.130.223.254	00:30:05:01:A7:9B			Fujitsu Siemens Computers		1	1:42	20 sec	680
nero.mobnets.rwth-aachen.de	🇪🇺	134.130.222.95	00:0C:6E:01:73:52			ASUSTEK COMPUTER INC.		1	0 sec	3:08	680
tbn0.google.com	🇺🇸	216.239.59.104					15	2	9 sec	28 sec	
sulla.mobnets.rwth-aachen.de	🇪🇺	134.130.222.86	00:0C:6E:90:69:2F			ASUSTEK COMPUTER INC.		1	0 sec	4 sec	680
luxus.mobnets.rwth-aachen.de	🇪🇺	134.130.222.66	00:17:31:94:0F:E8					1	0 sec	34 sec	680
boromir.mobnets.rwth-aachen.de	🇪🇺	134.130.222.247	00:0C:6E:15:05:AD			ASUSTEK COMPUTER INC.		1	0 sec	2:14	680
titus.mobnets.rwth-aachen.de	🇪🇺	134.130.222.81	00:0E:A6:19:49:0A			ASUSTEK COMPUTER INC.		1	0 sec	53 sec	680
rombus.mobnets.rwth-aachen.de	🇪🇺	134.130.222.71	00:0B:6A:9C:E5:2A			Asiarock Incorporation		1	0 sec	2:40	680
servus.mobnets.rwth-aachen.de	🇪🇺	134.130.222.72	00:0B:6A:9D:AD:E7			Asiarock Incorporation		1	1:20	0 sec	680

Figure 6: Traffic analysis with `ntop`.

A Resources for Further Information

- Regarding the tools and commands you have to use in this session, you will most probably often rely on the man pages.

- The Internet is, of course, often a helpful resource. An interesting tutorial is for example *Beej's Guide to Network Programming: Using Internet Sockets*, which is available at ⟨*http://www.ecst.csuchico.edu/~beej/guide/net/*⟩.

If you want to consult books, the following hints might be useful:

- The best book on network programming in C is probably *Unix Network Programming* by R. Stevens. The first of two volumes—the relevant one—is called *The Sockets Networking API*.

- Most of this session description is based on *Linux Netzwerkarchitektur* by Wehrle *et al*. The English edition covering the Linux kernel 2.6 is also available under the title *Linux Network Architecture*.

- Also very interesting and more general in scope, are *TCP/IP Illustrated* Volume 1 *The Protocols* and Volume 2 *The Implementation* by R. Stevens.

- For the first three sessions of this course, *Linux Socket Programming by Example* by W. Gay can be recommended.