TCP/IP, OSI and LAN

the basic tech that runs the (inter)net

by Blazko, mail@neveprise.de

Version: 1.0, 2001-07-11

This text is about basic technologies of networks: the TCP/IP stack, network topologies and the Socket API.

Table of contents

1.0 TCP/IP

> top of page

1.1 ISO/OSI 7 layer model

> top of page

1.1.1 OSI 7 generic

> top of page

The OSI layer model (Open Systems Interconnection) is some kind of norm for communication platform, for example networking computers. It is important to know that OSI is more a proposal rather than a classic norm. There are actually models that do not apply to this norm, such as ATM (Asynchronous Transfer Mode).
> The OSI model has been designed in the 1970's by the International Standards Organisation. <p>

The classic model describes the theoretical structure of communication systems starting from the hardware level up to the software level, without dictating specific details. To the full extend, there are seven layers distinguished:

- 1. physical layer
- 2. data link layer
- 3. network layer
- 4. transport layer
- 5. session layer
- 6. presentation layer
- 7. application layer

In the following I will try to explain the function of each layer so you can get an idea of the whole concept. Each layer provides specific functions the its upper layer, thus the degree of complexity shrinks from layer to layer.

1.1.1.1 physical layer

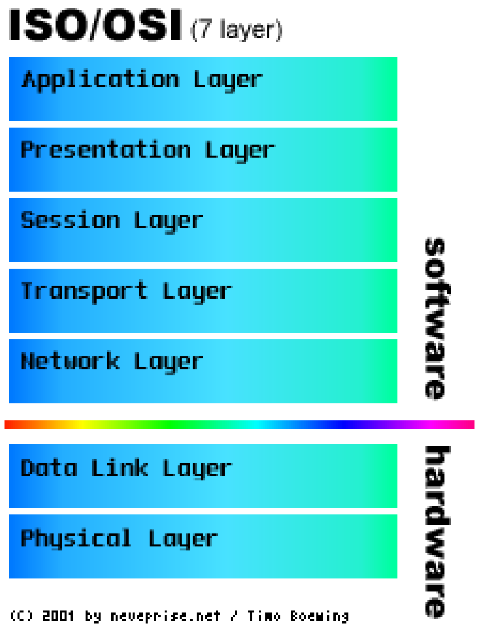
> top of page

The physical layer (bit-transmission layer) is the medium used to physically transmit data. Note that this must not be an ethernet cable, the OSI model covers also any other medium such as fiber or wireless communications.

1.1.1.2 data link layer

> top of page

The data link layer is supposed to perform basic transmission veryfication, error correction and addressing. When talking of ethernet, this is the instance handling addressing using the hardware addresses of the network adapters using MAC (Media Access Control).



picture 1a: generic OSI 7 layer model

1.1.1.3 network layer

> top of page

While layer one and two are disposed on a hardware basis, from this level forward the model is realized via software.

Besides a physical addressing is accomplished in layer two, this third layer is responsible for the logical data forwarding and address translation (e.g. MAC->ARP/RARP<-IP), e.g. using IP addresses.

1.1.1.4 transport layer

> top of page

If network communication is paket-oriented, this layer is accompanied to create pakets from streams, number then etc. and on receive to build a stream respecting the original order of each pakage.

1.1.1.5 session layer

> top of page

The session layer handles virtual point-to-point session (keep-alive). Therefore is has to maintain special contexts, so that not each connection is build - data transmitted - terminated but a connection remains active during several transmissions.

1.1.1.6 presentation layer

> top of page

Within the presentation layer, data streams are prepared and modified for the next level. That means that, for example, bytes get reordered to fit the operation system’s demands.
> Additionally, it provides send and retrievalreceive functions for the next level to allow applications the way we expect.

1.1.1.7 application layer

> top of page

This is the final layer that interacts with the user and holds the data logic. You will see in chapter 1.1.2 what you typically find in this layer.

1.1.2 OSI TCP/IP (specific)

> top of page

As seen, the ISO/OSI seven layer model is more a proposal rather than a norm. Taking TCP and IP as an example, we will further try to understand the model. Therefore it is nessessary to reduce our seven-layered model to only five steps:

- 1. physical layer (e.g. NIC, wires or WLAN)
- 2. network layer: ARP/RARP, MAC, IP, ICMP
- 3. transport + session layer: TCP or UDP
- 4. presentation layer: Berkeley Socket API
- 5. application layer: ssh/telnet, FTP, POP3, SMTP, HTTP, NFS, IMAP, NNTP...

1.1.2.1 physical layer

> top of page

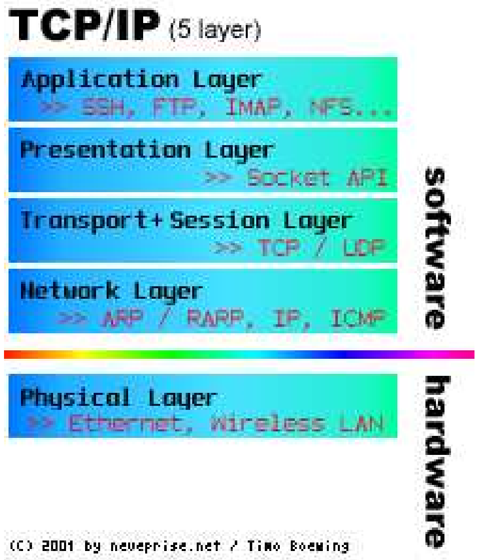
This is where the physical data transmission takes place. This can be using cable or wireless (for example ethernet, token ring infrared, WLAN [Wireless LAN], BlueTooth).
> This includes physical addressing, using ethernet this would be done with MAC (Media Access Control). <p>

1.1.2.2 network layer

> top of page

In this layer, translation between physical and logical addresses takes place. This is done using the (reverse) address resolution protocol, or short ARP/RARP.

The higher layers use the internet protocol (IP) for logical addressing purposes (networks, hosts). Nowadays, IPv4 is used which uses four bytes for addresses (32bit). We notate them in a form like "192.168.1.1". More on that later.
> Also found on this stage is the internet control message protocol (ICMP) for low level meta communication and diagnose purposes. The ping command abuses ICMP to do host echos. <p>



picture 1b: OSI model for TCP/IP (5 layers)

1.1.2.3 transport & session layer

> top of page

Mainly, this layer encapsulates two protocols: the transmission control protocol (TCP) and the user datagram protocol (UDP). The following tasks TCP and UDP have in common:

- packaging of streams into single pakets
- on receive, reordering pakets to a stream

Additionally, TCP has the following capabilities and tasks:

- holding connections (keep alive sessions)
- check if counterpart has received a paket, auto-reply (ACK)
- creation and veryfication of checksums (CRC, cyclic redundant check) to detect damaged pakets.
- request retransmission of defect pakets from partner

1.1.2.4 presentation layer

> top of page

Basically, the presentation layer provides an API (application programming interface) for the 5th layer. It offers a consistent interface for applications that do not have to care the underlying operating system and all underlying layers.
> From a developer's point of view you do not have to care if your application communicates via ethernet or wireless. <p>

One of the most prominent and most used API is the Berkeley Socket API whose sockets represent point-to-point communication endings. They support TCP+UDP/IP as well as other protocolls such as Novell's IPX or the next generation IPv6, if encorporated in the operating systems network drivers.

1.1.2.5 application layer

> top of page

On this level you will find the applications, among them ones you most probably will be familiar with: ftp, ssh, telnet and applications implementing those protocols: HTTP, NNTP, sunRPC, POP3, SMTP, IMAP etcetera.

1.2 low level services

> top of page

1.2.1 MAC

> top of page

If you are regarding an ethernet based LAN (local area network), it is build from physical connections, such as network adapters and wires. Each communication (end-) point is called a node and can also be a hub, switch, router, bridge.

Cos ethernet is not specifcally bound to any protocol, there is a requirement for direct addressing (means addressing without using IP). This is done on the hardware basis using MACP: media access control protocol.
> Every node (network device) owns a so-called MAC address consisting of 48 bits or 6 bytes. This address is hardwired into the hardware and is worldwide unique! These 48 bits are split into two parts: the first 24 identify the brand (assigned to a manufacturer) while the second 24 bits can be assigned by the manufacturer himself.
> The IEEE-SA (Institute of Electric and Electronic Engineers, Standards Organisation) assigns the manufacturer ID, all that together assures that each network device indeed owns a unique identification address.

1.2.2 ARP/RARP

> top of page

In theory, those devices could communicate, cos they could uniquely identify each other. The problem is that networks have to be freely configurable, e.g. conglomeration of network groups, free assignment of "addresses" etcetera.
> Cos devices have hardwired addresses, this seems not to be possible - but it only seems to. On a logical level there is another addressing used: IP (internet protocol). <p>

More on IP addresses later in chapter 1.2.4. For now, it should suffice to know how addresses look like. They consist from 32 bits or four bytes and are noted this way:

- 127.0.0.1
- 192.168.1.1
- 163.254.9.127

Thus, per node there are now two addresses used: the MAC and the IP address, the first burned into ROM and the second resides in the operating system’s settings. The IP can be set almost at will (under certain precautions, of course).
> The question is, how the MAC and the IP addresses work in concern. Therefore, the address resolution protocol (ARP) and its counterpart reverse address resolution protocol (RARP) have been introduced. <p>

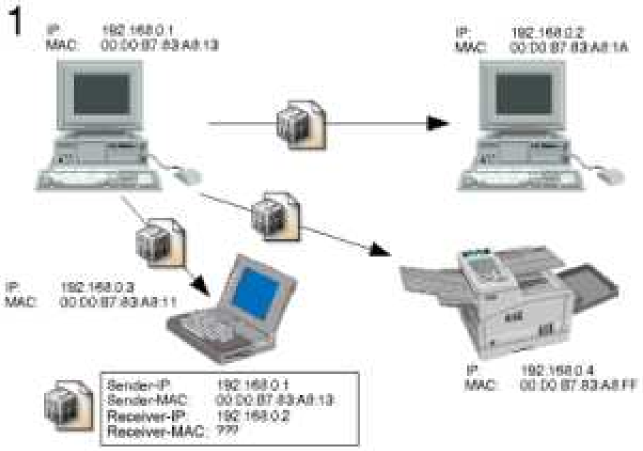
ARP resides in the lowest software level, according to our layer model this would be the network layer (layer 2). It works this way:

Given a small network consisting of - let’s say - five computers. Workstation A wants to send data to station B, whose IP address is known. But it does not know its physical address, so the problem is how to to send the data physically to computer B.
> Using ARP, station A sends a package to all computers in the net (called a broadcast, using his assigned IP but replacing the value of the last byte with "255"). This causes the operating system to send this package to all clients. The broadcast is sent with the hope that the computer with the demanded IP address will answer. This broadcasted data package contains this data:

- own MAC address (sender MAC)
- own IP ADDRESS (SENDER IP)
- MAC address of receipient (receiver MAC, empty field/unkown yet)
- IP address of receipient (receiver IP)

Cos each client computer receives this signal, each one probes if the demanded IP is assigned to itself. If the test is successful (means the computer found out that the requestet IP address is assigned to this station), the received package is taken, the missing field is filled out with the own MAC address and sent back to the caller using its - now known - MAC address.
> The original sender receives this information and is now aware that the requested computer having the IP ABC can physically be addressed using the MAC address XYZ. The following graphics shall help to visualize this process: <p>

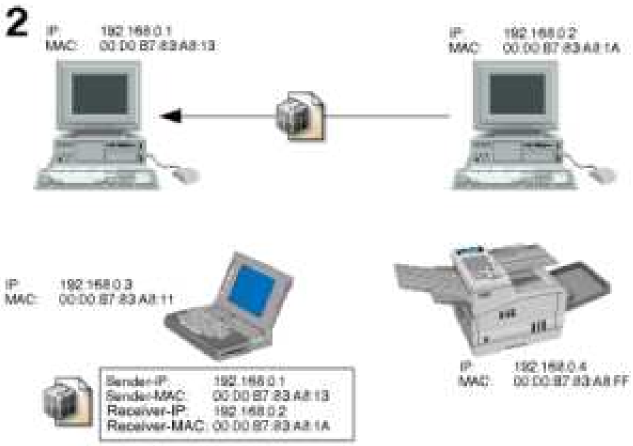
(click image to enlarge)



>arp1l.png" target="_blank">

Pic 1c: : address resolution, step 1

(click image to enlarge)



>arp2l.png" target="_blank">

Pic 1d: : address resolution, step 2

Said shortly, ARP allows to retrieve the IP assigned to a MAC address in opposite to retrieve the MAC assigned to an IP address via RARP. For this reason, each node holds an ARP table within the operating system's access. Once all nodes are known by sending ARP requests, it most porbably nessessary to send data to these workstations. Because the MAC<->IP relations are stored within that table. Thus, the system can consult this table instead of sending an ARP request per transmission.
> The RARP is very helpfull for diskless stations with no configuration regarding network settings. They can broadcast an RARP and a server will tell that workstation what IP is it assigned. <p>

1.2.3 ICMP

> top of page

The ICMP (Internet Control Message Protocol) is a low level messaging service and is primary used for diagnosing purposes. ICMP informs networking partners about:

- errors in the TCP stack
- errors in IP, MAC and ARP
- (un-) reachable network nodes (networks and hosts)
- routing errors

1.2.4 Internet Protocol (IP)

> top of page

Like discussed in chapter 1.2.2, the Internet Protocol (IP) has its purpose in logical addressing of networks and hosts - and not in physical addressing. Every TCP/IP-driven network node like a client, host, router, gateway or DNS server (Domain Name Service server) owns an IP address that uniquely identifies that device. In contrast to the MAC address, the IP is not a statical address but can be dynamically applied, regarding some specific rules (valid and allowed addresses).
> Currently, IP version four (IPv4) is the defacto standard; the successor IPv6 or IPNexGen (IPv5 existed, but only in internal labs). IPv4 uses four bytes 32 bit for the address and is written down this way:

192.168.1.9

IPv4 allows a practical bandwidth of two to four billion single addresses (in theory, you could use more, but in early days of the internet, large organisations were assigned a huge bunch of IPs that cannot be used by others).
> Regarding current explosion of network development (almost every ant on earth has got its own IP :-), it soon has become obvious that IPv4 would not last beyond the year 2010. Thus, IPv6 has been put into development to replace the predesessor step by step. <p>

IPv6 uses 128 bit addressing and theoretically allows for approx. 30 000 IP addresses per squaremeter of solid ground on earth! That is what an IPv6 address would look like:

fe80:d0:b783:a813

1.2.5 Domain Name Service (DNS)

> top of page

In the early 1980s, when the internet already grew intensively, it was hard to know all the IPs of servers. Consequently, developers were working on a solution to address networks and hosts a mnemonic way, to use easy to remind names.
> Therefore the Domain Name Service (DNS) was born, allowing to resolute URLsURIs (Uniform Resource Locator / Uniform Resource Identifier) into IP addresses and vice versa. Such URLs look this way:

ftp://diseaszed.net or http://homer.doh.lan

alternatevly you could write

http://195.20.225.17 or ftp://192.168.1.3

A complete URL consists typically of the following parts:

http://webservices.diseaszed.lan/cgi-bin/login.pl?name=blazko

Description:

>
>

part	meaning
http://	protocol identifier, e.g. http (HyperText Transfer Protocol), ftp (File Transfer Protocol)
webservices.	logical subserver (server of a branch of servers, dedicated one; commonly equals a subdirectory on one machine)
diseaszed	the server's name
.net	the top level domain (TLD), e.g. .com .net .org .de .biz .tv .info etc.
/cgi-bin/	a directory in the webserver's (the software.deamon) root filesystem, may physically reside in /var/www/webservices/cgi-bin/
?	path and program/parameter delimiter; the ? delimits a physical path to a file resource and appended parameters (if a CGI program)
login.pl	a programm / CGI script that generates output for client and / or performs some server-side tasks (here: a PERL script)

1.3 packager

> top of page

While ARP and IP more or less just handle addressing purposes, a powerful instance for error correction and (dis-) assembling streams is missing. These are the tasks of the TCP or UDP stacks.

1.3.1 Transmission Control Protocol (TCP)

> top of page

Often in the internet, there are beeing sent large chunks of data, partially exceeding one gigabyte in extreme cases. Imagine an ISO image as a file would be send through the internet in one large piece. Would a significant transmission occur that would leave the file unusable, you would have to retransmit the file, meaning to submit all successfully sent megabytes again. You know, the famous errors that happen at 99 percent...
> Working with this medium would be very disappointing. Thus, data is sent in little packages. Thus, large data chunks are split into several tiny pieces and are sent step by step. If one package arrives broken, just this package has to be retransmitted. <p>

Regarding the ISO example, you are right: there are nowadays tools and protocol extensions that allow sending data from file offsets. Even if TCP would not exists in the way we know, such tools can tell the server to transmit data from an offset of e.g. 500MB of the file of interest.

When TCP splits streams into smaller units, it gives each package a serial number bevore transmission starts. The reason is simple:
> When you send data from A to B over C, it must not mean that the next data will pass the same route, it might travel through D. This is one of the biggest strength of the internet, that several routes exists between two points. It is decentralized and if one route gets lost, another route will be taken. This comes from the origins of the internet, the ARPANet - a originally military invention - that should grant communication even in case of nuclear attacks. Even when parts of the web get destroyed, you may communicate via other channels.
> As a result that each TCP package might travel a different route, it is most probable that some packages will come later than others although beeing sent earlier.

Therefore, these serial numbers are used by the receivers TCP stack to get the packages into the correct order, allowing it to build a reasonable stream that the client application is able to use, rather than getting garbage.

Besides of this vital feature, TCP does extended error correction on packets beeing received damaged. But TCP also handles another important task: requesting resubmission of broken packages and confirmation of successfully received data; that is why TCP bears "Control" in the name.
> To sum it up, the key features of TCP:

- splitting of large data into smaller pieces
- creating of checksums and serial numbers of data packages
- sorting of received packages to the original order
- recreation of a stream
- if possible, data correction using checksums
- requesting resubmissions of damaged data packets
- confirming successfully received data

1.3.2 User Datagram Protocol (UDP)

> top of page

Like seen in the previous chapter, TCP is responsible for various complex tasks. And that is its problem: for some tasks it is just too complex, meaning it has too big latency resulting in loss of performance and more network payload. For some not-so-important tasks it is desirable to have a substitution for TCP, this is UDP. Many services just do not require e.g. receive confirmation, so the User Datagram Protocol (UDP) comes in place, for example for the Domain Name Service (DNS), responsible for resolution of hostnames into IP addresses. DNS queries happen very often, and if a DNS query fails cos UDP does no checking, the timeout of the client will most probably a requery.

2.0 Berkeley Socket API

> top of page

As simple as a network might look like so far, as comlicated are most of its internal issues in detail. This aspect must be handled within the software layer, issues to work on are e.g.: packets, addresses and their translations, checksums, sorting, byte translations according to a specific platform, TTL (time to live values), forwarding, retransmission requests, handling different protocols and versions etcetera.
> To avoid reinventing the wheel each time from the sight of an application developer, there are libraries and their (hopefully) well documented APIs (Application Programming Interface) available according to a specific programming language (C and C++ are most common). These libraries containing the binary execution code usually are shipped with the operating system of interest and are called shared objects in the UN*X world (.so files) and DLLs in the Windows world (Dynamic Linked Library, .dll files). <p>

That standard API for basic application level network developments is the Berkeley Socket API, short sockets, originating the BSD.
> The socket API is used for p2p (point to point) network conversations, mostly connection-oriented. Therefore, the socket interface offers vital functions and data structures that allow the application developer to have hisher tools making network communications.

An important note: sockets handle *only* addressing and raw data transmissions from commpartner A to B (yes, and broadcasts...), it does not implement higher level protocols like HTTP for www purposes or POP3/IMAP + SMTP for email infrastructure. The socket API only provides the very basic utilities to set higher level standard protocols or custom ones upon! However, you may choose other interfaces according to your development language that will handle these higher protocols, like many of Perl’s CPAN modules.

Having network basiscs in mind, what basic methods must we have to guarantee network transmissions? Let us sum up:

- handling addressing issues
- establishing connections with another network node
- termination a connection/session
- sending raw data streams
- receiving them
- error handling
- listening a channel to save CPU idle times (event/notify system)
- basic information infrastrucure (e.g. information on public remote host data)
- setting/getting transmission/network options

Extended information on sockets can be found in the RFCs 129 and 147.

2.1 the filehandle concept

> top of page

To understand sockets, it is important to know how files are treated (cos sockets are used as abstract files).
> When using regular files (i.e. files on the harddisk etc.), the operating system does not identify files on a filename basis. This would be very ineffective. If you are using a UN*X flavour os, please su to root and type "lsof" (list open files). You will get a list of tons of files in use by the kernel and userspace applications. When some action should be made against a file, doing it on the filename basis would not be very effective, the os must maintain a string hash.
> Instead, the filename is only used upon first file action, for example opening a file. When opened, the

file is mapped against a numeric value (commonly a 32bit integer), so the us just has an array of equal sized integer entries instead of tons of bytes using a hash. This uniquie integer id is called a handle. If you are a developer, you might know this technique already:

```
open(FILE, './neveprise.dat');
foreach my $line(<FILE>) {
   print(STDERR, $line);
}
close(FILE);
```

Although this is slightly a silly example - cos Perl has no real data typing - let us think of: the open() funtion opens the plaintext file "./neveprise.datquot; This file associated with "FILE" (a would-be integer) to uniquely identify this file resource by a number instead of the filename string. This file is read line-by-line referring to the handle symbolising the data until it is closed by its id using the close() function. But there is another handle: STDERR. This is a predefined handle identifying standard out (if this code belongs to a CGI script, print() would write each line to the webserver's logfile). If replacing STDERR by FILE, print() would write into the file opened before. Well, actually this would cause an error, cos we did not allow open() to open the file in write/truncate mode, just readonly.
> In short, a filehandle acts like some sort of pointer to the file's data. Thus, programming languages almost always require to feed file Io routines with handles instead of their names:

```
int open(const char *filename, int flafs);
```

Here a somewhat larger C-style example:

```
int   myFile = 0;
int   bytesWritten = 0;
char  msg = '';
myFile = open('/var/log/messages', 0);
if (myFile != -1) {
   msg = 'this is a message from neveprise.net';
   bytesWritten = printf(myFile, msg, strlen(msg));
   close(myFile);
} else {
   fprintf("Oops! error  %d on open() by hellomsg.c\n", errno);
}
```

Mh, using strlen() would allow a stack overflow, right? Okay, that is another story. This example just opens the messages fiel from /var/log/ and puts a simple message into it. If an error occurs, an error will be prompted. Therefore we use the fprintf() function that uses the STDOUT handle by default if another handle is omitted, thus it writes the message out to console.

2.2 sockets (abstract)

> top of page

Spoken plainly, a socket is just one of at least two communication endpoints and inherits the following properties:

- a specific protocol type (common would be TCP/IP)
- an address (if IP e.g. 192.168.1.1 [class C network])
- a given port number (e.g. 80) or service alias (e.g. http)

I think I do not have to explain addresses and protocol types, but ports and services should be worth own chapters.

2.2.1 ports

> top of page

Imagine a server uniquely to be identified by its address. Due a server may provide more than just one single service (e.g. act as webserver (http), database server, DNS server, email server etc. simultaneously) at the same time, there are specific "channels" for each service.

For better understanding a little analogy: our server is now a customer service center residing in one building. In dependence of the customers issues, these customers check the reception of interest (for example a reception for billing issues, one for technical problems, one for... and so on). The persons on the reception then guide the customer to a free office that is free and that can handle the customers desire. As you see, there is less chaos, cos persons first find the right building, then find the right reception from where they are redirected to a free office. Thus this large crows of customers is split step by step according to their desires.
> You say this is a bad example - and you are right. This has two reasons: cos English is not my natural language I could express myself hardly, and second, this was one of many bad examples that came to my mind. Please email me for a better solution :-Q <p>

However, let us regard this process by using a web browser: normally, a web browser operates using the HTTP (hypertext transfer protocol), which is by default assigned to port number 80 for requests. A request would be to send the HTML document ./htdocs/index.html to the browser for display purposes. So, this rowser sends a http request to a dedicated server and encondes the first data part of the first TCP package to use port 80.
> On the server side, you have a http deamon (httpd) running (Apache, IIS [!doh!], Tux etc.). This deamon listens on arriving data. According to our previous example, the browser would be the customer (client), while the httpd is the reception in the building (server). The TCP stack of the operating system previously read the incoming data stream and knew that port 80 is demanded. Thus it forwards this stream to the httpd. The deamon reads the query and forks a child process or creates a new thread (depending on operating system andor settings) that should handle the request. This keeps the original httpd clear for further listening the port 80 data. Again, the "reception" gets clear for the next customer and the previous customer is directed towards a free office with an employee that can "serve" the customer.
> Back on system tech, the further communication between the child process thread will be done on any free port except port 80 or any other standard or used port number. People often say that http communication always is done on the default port (80), but as you see, this is not true. The default port is only used for initial requests, after that communication of a client application and a server instance will be done above any port number greater than 1024 (cos they are reserved for widely used service listeners).
> Normally, a system has up to about 64.000 ports, but sometimes these are much too less. There are at least two solutions to this problem: for normal network services (like http, ftp, ssh, telnet, pop3, smtp, imap, nntp etc.) you must use load balancing. That means you have several identical servers that handle requests depending on their system usage. For "higher" services, there is a pssibility to expand the 64.000 ports boundaries: RPC (remote procedure call). If you are using the NFS (network file system) or CODA FS, you are already using it. You will have runnning a deamon called sunrpc that takes requests and can internally split requests, depending on internal service port numbers. In some manner, this deamon does the same like the TCP stack does with normal services like http: it looks into a stream and assigns it to the

acdording service, depending on port number. <p>

2.2.2 services

> top of page

Like said in the previous chapter, some ports are assigned/dedicated to specific services. When you are using Linux (does this also apply to BSD etc.?), you can see a list or port numbers and assigned services, Windows uses the file \windows\services. Here a small list of ports and services:
>
>

port #	service
80	HTTP: the hypertext transfer protocol, to request www files like HTML, XML, XSL, CSS, JavaScript, PNG, GIF and JPEG.
20, 21	FTP: the file transfer protocol, optimized for two-way file transport, directory browsing and access rights management. Yes, there are two ports: port 20 is used for data transport while port 21 is responsible for ftp instructions.
22	SSH: the secure shell for remote logins. SSH is a replacement of telnet and basically allows to operate on a remote system and additonally safely transfer files between systems via its scp.
37	time: the timeserver for getting time/date information from a remote system to synchronize with the own system.
53	DNS: the domain name service helps in resolving the IP address fomr a given host name and vice versa. This service is very vital to use mnemonic host names; if no DNS and static hosts.conf is available, you must use IP addresses in order to to access a system.
443	HTTPS: the http secure: used in current IPv4 if a secure data transmission is required (e.g. when transmitting personal data out into a html form), often used in webshops.

2.3 socket functions and structures

> top of page

The Berkeley Socket API offers a nice set of functions to manage network connections and communication. The following sections will show the most important ones. For more information please consult your manpages (man 2 <function>).

2.3.1 int socket()

> top of page

The socket() function creates an communication endpoint and returns a socket handle upon successful creation of a socket. In some manner, this is the network equivalent of the open() function used for files.

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

socket() expects three parameters:
>
>

parameter	meaning
domain	the communication domain, commonly is PF_INET but also PF_LOCAL or PF_UNIX for local domains
type	defines the way data is transmitted, e.g. as stream (SOCK_STREAM) or datagram (SOCK_DGRAM)
protocol	specifies the protocol family to use, e.g. TCP, IPX, Appletalk etc.

The following example creates a common socket, but note that with this function call actually *no* connection is established yet! socket() just creates the endpoint:

```
if ((s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) == SOCKET_ERROR) {
   printf("Doh! An error occurred on socket(), code %d", errno);
   exit;
}
```

2.3.2 structure sockaddr

> top of page

The structure sockaddr is used to set socket options, e.g. what kind of socket it is: a server socket or a client socket.

2.3.2.1 client socket

> top of page

A client socket is a passive socket that is created and connects to an already existing socket to perform a request. Therefore we must give sockaddr the correct settings:

```
struct hostent *hostinfo;
//retrieve IP address from host name
hostinfo = gethostbyname('neveprise.net');

//internet domain
mysockaddr.sin_family = AF_INET;
//use http (port 80)
mysockaddr.sin_port   = 80;

//read IP address from hostinfo
mysockaddr->sin_addr = *(struct in_addr *) hostinfo->h_addr;
```

2.3.2.2 server socket

> top of page

In contrast to the client socket, a server socket is created and does not connect to other sockets on own initiative. When the socket is created, it just listens on a specific port for incoming connection requests from client sockets.

```
//accept requests from the internet domain
mysockaddr.sin_family = AF_INET;
//listen on port 80 (http)
mysockaddr.sin_port   = 80;
//accepts request from all clients
mysockaddr.sin_addr.s_addr = htons(INADDR_ANY);
```

2.3.3 int bind()

> top of page

When a socket has been created, it must be fed with local system information. That means it has to be assigned the local host properties, and for more internal purposes, it receives a unique name (cos on UN*X, sockets are mapped to virtual files, thus a a temp. name for this file is generated).

```
bind(s, (struct sockaddr *) mysockaddr, strlen('localhost'));
```

The meaning of the second parameter (backlog) depends on your OS, please see man 2 listen.

2.3.4 int connect()

> top of page

If the socket created shoul be a client one, the connect() function finally establishes a connection with an existing server socket on another or local machine:

```
connect(s, (struct sockaddr *) mysockaddr, strlen('neveprise.net'));
```

connect() receives an already created socket s, the filled sockaddr structure and the length of the address.

2.3.5 int listen()

> top of page

The listen() function is for a server socket what connect() is for the client one: it must be set to an active listen mode:

```
listen(s, 0);
```

The meaning of the second parameter (backlog) depends on your OS, please see man 2 listen. If this routine succeded, the socket now listens for incoming requests on that specific port.

2.3.6 int accept()

> top of page

accept() is needed for listening server sockets only. When they receive a connection request, the server gets information on the remote socket that wants to connect. The application created the socket may query remote information and evaluate if the client may connect. If the client may connect, the server has to create a new socket on the server side that handles further conversation with the client thus the server socket will get clear for further listening.

```
newsock = accept(s, (struct sockaddr *) clientsockaddr, &size);
```

The new socket is nearly a clone of the server socket s except that it is not in the listening state. The sockaddr structure receives further socket description (remote address etc.).

2.3.7 int send()/print()

> top of page

So far, you have seen basic socket management routines. Now it is time to use the sockets for their original purpose: communication. Thus we need a function that sends any data to the conversation partner. You most probably know the print() function in C and Perl or write() in Pascal. By default, these routines purge a buffer to standard out (STDOUT) if ommiting a handle. But if you provide a filehandle (e.g. obtained from open()), the output is redirected to the file referred to by the handle:

```
#write to standard out:
print("hello!\n");
#also does:
print(STDOUT "hello!\n");
#writes buffer to a socket:
print(s "hello!\n");
```

Got it? You can use the common functions to write over a socket to another remote socket. That is the advantage of the filehandle concept. But actually, in practical use, you will get into trouble sometimes when using the normal print() function. Why? Well, there is one basic difference between any handle and a socket handle: normally, filehandles use local resources, such as STDERR, STDOUT or any local file. But a socket describes a remote target, thus there may be a large timespan between operation call and the final action (called latency).
> Thus, the socket API introduces the send() function that is some kind of a print() clones but strictly designed to be used in socket context. send() has timout implementations beeing more tolerant in time issues. Thus, data transmission via print() may fail while using send() in the same context may be successful. The reason is that print has a much tighter timeout between action release and callback. <p>

```
int bytewritten;
char *msg = new char[255];

msg = "hello world!";
bytewritten = send( s, msg, strlen(msg) +1, 0);
if (bytewritten == SOCKET_ERROR) {
   printf("Doh! Error on send(), cause %d.", errno);
}
```

The send() function in this example uses socket s to transmit the buffer msg (length of the buffer is given by strlen(msg)+1 [terminating char]) and an option 0. The option parameter is set to zero, causing to use system default settings.

2.3.8 int recv()/scan()

> top of page

Analogous to sending data, you guess that you can use the opposite function of print() to read incoming data, scan(). You are right, but you may also not wonder that there is a specialized version for sockets. This function is called recv() for receive.

```
int bytesread, maxlen = 255;
char buf = new char[maxlen];

bytesread = recv( newsock, buf, maxlen, 0);
if (bytesread > 0) {
   printf("output from client: %s\n", buf);
}
```

2.3.9 select() events

> top of page

When you use file I/O, you would call scan() upon e.g. user actions, such like opening a file in an editor. But when having a socket, you cannot just perform a simple recv(), cos you can only receive if data is expected. The socket specified in recv() is no file, it is only filled upon request or upon a sender's action.
> One way to solve this problem would be to create a loop that checks the socket for incoming data: <p>

```
int error = 0;
while (error == 0) {
   [check socket for input]
}
```

When executing this code, you will notice that your machine slows down and cpu usage will reach the 100 percent, although you actually do nothing that makes sense with your computing power. You may improove your code by doing idle calls via sleep routines, but this will also not be mor elegantly. Especially when having multiple connections (a socket array), you will have to query them all - your coding will suffer from this.
> It would be nice to have a notifying system at hand, and yes, there is one: select(). The select() function registers file listen handlers that will notify the occupying process if a file is touched by another instance. This is the way the program tail works: <p>

```
tail -f /var/log/messages
```

tail() observes the logfile "messages" for changings. But therefore it does not have to reread the whole file each time (or at least the last offset of it). Moreover, the tail process is notified of filechanges cos it registers a handler to the operating system. Cos it is the OS that handles all I/O, so when it performs changings on that resource, it finds the handler for that process and wakes it up to perform an action according to that changes.
> You know that from some abstract point of view there is no big difference between file handles and any other resource handles such as the socket. So you can register a handler for files as well as for sockets.
> You feed select() with several sockets you want to check for pending input. select() will block your application until data is available on any socket (put it into sleep mode). By using select(), you provit from very effective application external logics. However, you have to code some lines to use select(), this would extend this document too much. So far I have no article in the development section, I recommend to consult the glibc documentation, which provides excellent explainations on sockets() and other important stuff.

> top of page