Communications Systems Engineering

Moritz Hüllmann

March 5, 2022

1 Network Programming

There are five socket types, but only three are relevant.

- `SOCK_RAW`
- `SOCK_STREAM`
- `SOCK_DGRAM`
- `SOCK_RDM`
- `SOCK_SEQPACKET`

Combined with **Protocol Family**, communication is defined completely. Focus is on **IPv4/v6**. Relevant protocol families

- `PF_INET` – IPv4
- `PF_INET6` – IPv6

Sockets give access to **transport layer**. One may also use `AF` instead of `PF` prefixes.

1.1 C-Implemenation of Sockets

Definition: `socket(·)`

This function creates a new reference to a socket in the OS.

```c
int socket(int socket_family, int socket_type, int protocol);
```

`protocol` – optional, if there is only one possibility. Set to `0` if not wanted

Returns: 0 on success, -1 on error

Definition: `bind(·)`

This socket binds a socket to a specific address and port.

```c
int bind(int sock, const struct sockaddr* addr, socklen_t addrlen);
```

`sock` – a file descriptor, i.e. the return value of `socket(·)`

Returns: 0 on success, -1 on error

`bind(·)` – commonly used by servers. `strcut sockaddr* addr` has all information regarding IPv4/v6.

One has to be careful regarding byte order. Sockets usualy require **network byte order**.

`htonl(·)` Host to network long
`ntohl(·)` Network to host long
… and many more

Definition: `getaddrinfo(·)`

This function gets all possible based on the information passed to it. Can be used to determine IP addresses, ports and so on.

```c
int getaddrinfo(const char *node, const char *service, const struct addrinfo *hints,
    struct addrinfo **res);
```

`hints` – used to tell the function what is should do specifically.

Returns: 0 on success, -1 on error. On success, `res` contains a pointer to a linked list of whatever was requested

This function should be used everytime, one implements server/client applications, as it can handle DNS resolution, handle both IPv4 and v6 and so on.

Suggested call order:

1. `getaddrinfo(·)`
2. `socket(·)`
3. `bind(·)`

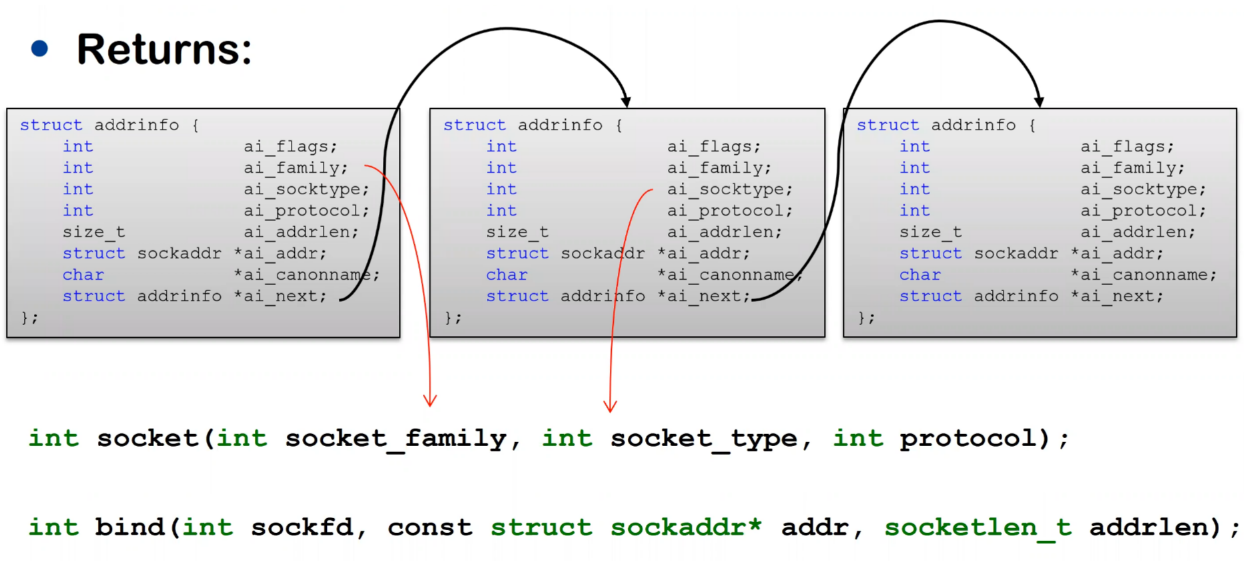

Figure 1: How to use `getaddrinfo(·)`. Note: All information used should originate in one block, not two as shown above.

1.1.1 Server-side programming

Up until now, only communication setup was discussed, we need to communicate now.

Definition: `listen(·)`

This functions listens for incoming connection requests.

```c
int listen(int sockfd, int backlog);
```

`sockfd` – Socket to listen on.

`backlog` – Queue length for pending requests.

Note: This is only valid with `SOCK_STREAM` and `SOCK_SEQPACKET`.

If the queue is full, the server will not answer. `listen(·)` is a blocking function.

Definition: `accept(·)`

This function is used to accept requests received by `listen(·)`.

```c
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

`addr` – holds the address that should be accepted

`addrlen` – length of `addr`. One can pass a structure that is big enough for IPv4 and IPv6.

Returns: File descriptor for new socket, that handles the accepted connection from now on.

1.1.2 Client-side programming

Definition: `connect(·)`

This function connects the socket to a server. It may perform a TCP handshake and similar things in the process.

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

`addr` – address to connect to

`addrlen` – length of the address

Returns: 0 on success, -1 on failure

1.1.3 Sending and receiving

One can *write* to and *read* from sockets, with the following functions:

- `write(·)` and `read(·)`
- `send(·)` and `recv(·)` – TCP
- `sendto(·)` and `recvfrom(·)` – UDP

Definition: Stream

A stream is like reading from a file. TCP just decides to cut the stream at certain points and transmits each fragments as packets. We do not know the context of a single packet.

Definition: Datagram

Messages transmitted via datagrams are not cut up by the kernel, they are send as is. If the message is too large, there either is an error, or the package gets fragmented. Each datagram is self-contained, so every received packet can be seen as a unit.

Definition: `send(·)/write(·)`

These functions can be used to write to a socket. As in Linux everything is a file, one can use `write(·)` instead of `send(·)`.

```
ssize_t send(int sockfd, const void* buf, size_t len, int flags);
ssize_t write(int fd, const void* buf, size_t count);
```

`flags` – instruct kernel to handle send request in a certain way

`buf` – Buffer with data to send

Returns: Bytes actually written. -1 on error, non-negative otherwise **Note:** `write(·)` is equals to `send(·)` with `flags` $= 0$

Definition: `recv(·)/read(·)`

These functions are designed to read a certain amount of bytes from a socket (or file).

```
ssize_t recv(int sockfd, void* buf, size_t len, int flags);
ssize_t read(int fd, void* buf, size_t count);
```

Parameters are analogous to `send(·)`.

Actually using the return value of the reading functions is really important. It may be unknown, how many bytes the applications is going to receive in a single read operation. If $retval \neq lenght$, then not enough was read.

Definition: `sendto(·)` for connection-less sockets

This function sends a certain amount of bytes from a buffer to the specified address and port.

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

`dest_addr` – Address and port to send to

Note: There is no equivalent `write` function.

Note: Parameters up to `flags` same as for `send(·)`. Exclusively for sending datagrams.

Definition: `recvfrom(·)` for connection-less sockets

This functions receives a certain amount of bytes from a given address.

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

Note: `src_addr` will be replaced with the address, that really was received from. `addrlen` will then hold the correct length of the structure.

Note: Parameters up to `flags` same as `recv(·)`.

Again, if it is not known which version of IP is going to be used, use `struct sockaddr_storage`, to allow for both versions.

1.1.4 Closing connections

There are several methods for closing a connection, each to be used in a different scenario.

Definition: `close(·)`

Drops all pakets queued for receiving and sending, and shuts down everything else related to the socket. Also frees all previously allocated space.

```
int close(int sockfd);
```

Returns: 0 on success, -1 on error

For more conservative shutdown operations, the following function can be used.

Definition: `shutdown(·)`

More conservative `close(·)` variant.

```
int shutdown(int sockfd, int flags);
```

`flags` – 0: no further receives, 1: no further sends, 2: both

Note: Return values same as for `close(·)`

Note: Sends `FIN` and/or `FIN_ACK`, other party might get `RST` (reset) when trying to read from the shut down socket

Note: This allows all queued data to be send.

`shutdown(·)` cannot stand alone. `close(·)` must be called after a call to `shutdown(·)`, as it releases all hugged `structs`.

1.1.5 Connection overview

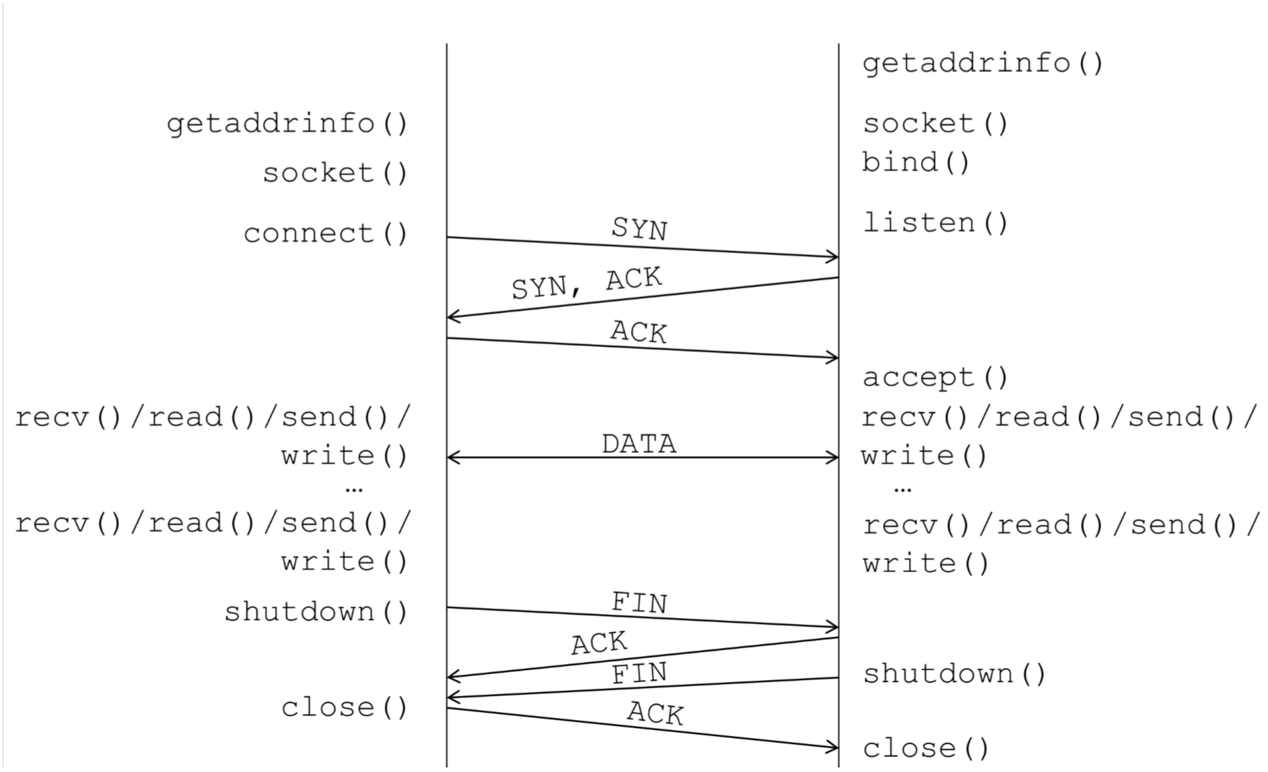

Figure 2: Overview over the course of a connection-aware socket

1.1.6 Configuring a socket

Definition: `setsockopt(·)`

Apply options to sockets.

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

`level` – basicalli ISO/OSI level the option should be applied to

Returns: 0 on success, -1 on error

Getting options set for a sockets uses a functions with more or less the same signature as `setsockopt(·)`, but is called `getsockopt`.

`I/O`-controls can also be used to set and read properties of a socket. Commonly:

- Timestamp of last received packet
- How many bytes are unsend (TCP)
- How many bytes are unread (TCP)
- ...

1.1.7 Non-blocking sockets

There are two methods that one could use: `select`(·) and `epoll`.

Non-blocking sockets using `select(·)` All network calls are blocking by default. This is not optimal in a server setting, as this introduces great overhead and occupies many ressources of the server, that could be used elsewhere. **Threads** are not an option for highly scalable systems, as they introduce great computation overhead. `fcntl`(·) can be used to set flags to sockets, specically `O_NONBLOCK`. Callbacks are called, to notify a program, if a socket is write-/readable. Polling is not an option, this is busy waiting.

Definition: `select`(·)

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct
    timeval *timeout);
```

`nfds` – highest file descriptor number in all sets + 1

`readfds` – list of sockets to monitor readability for

`writefds` – list of sockets to monitor writeability for

`exceptfds` – list of sockets to monitor exceptions for

`timeout` – maxium wait time

Returns: 0 on timeout, -1 on error, else number of fd's with event `read/write/excetpfds` is then set to the sockets with events. Manual checking which sockets have events is required.

Non-blocking sockets using `epoll` It knows two modes:

1. level triggered, which is just as `select`(·)
2. and edge triggered.

Edge triggered `epoll` informs on every *change* of states for read and write queues, as well as excepts.

If there are 5 new bytes received, calling `epoll` returns that new data is readable. However, if just 1 byte is read and `epoll` called again, it will block, as there is no **new** data available.

An `epoll` file descriptor is attached to the sockets queue, directly in the kernel. Each `epoll` *fd* has a queue of its own, and notifies for every socket listed there.

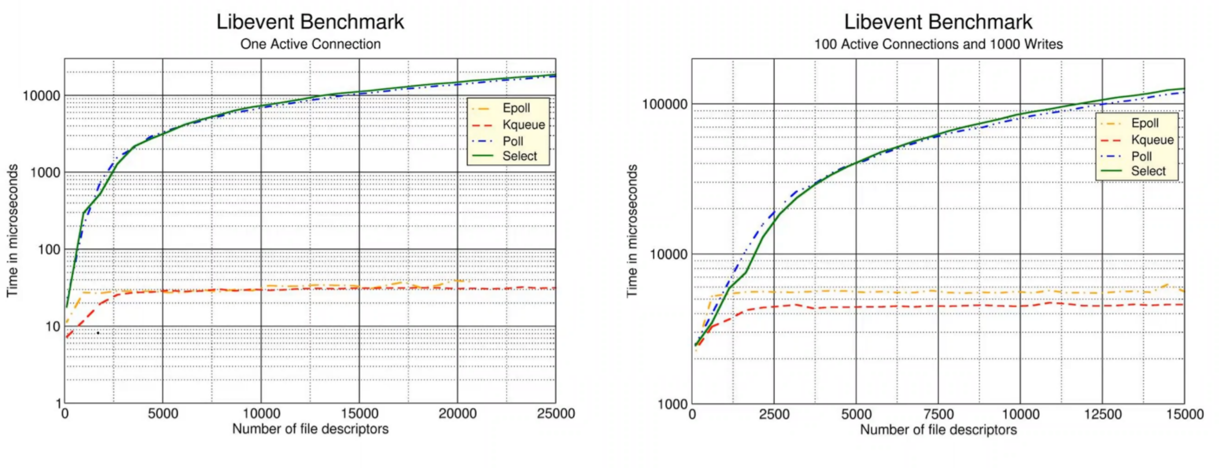

Figure 3: Performance of `epoll` vs. `select`(·)

1.2 TCP options and packetization

TCP needs to decide important questions:

1. When to wait for more data from user space
2. When to send queued data

If not done so, every `send`(·) results in a packet. If only one byte is send, that would result in packages with 40 bytes of overhead, but only one byte of payload.

1.2.1 Nagle's Algorithm

Algorithm: Nagle's Algorithm

This algorithm is default TCP socket behaviour.

Small chunks are accumulated and send, when previous data was ACKed.

```
if (there is new data to send) {
    if (window size >= max(segment size) && available data >= max(segment size)) {
        complete max(segment size) and send now;
        queue remaining data;
    }
    else if (exists(data in flight waiting to be ACKed)) {
        queue data and send when ACK is received;
    }
    else {
        send data immediately;
    }
}
```

Note: This algorithm is not suited for every application, as it may wait, when an immediate send might be necessary.

Disable with `TCP_NODELAY` with `setsockopt`$(\cdot)$.

1.2.2 Delayed ACKs

Basic idea: when data is received, we will want to send data as a response. Piggy back ACKs for previous data on data you want to send. The delay is $\leq 0.5s$. At least every second packet with MSS is ACKed immediately.

This also is TCP default behaviour. However, don't combine with section 1.2.1, this may lead to unnecessary waiting times of up to 0.5 s.

Disable with `TCP_QUICKACK` with `setsockopt`$(\cdot)$.

Also possible: packetize TCP yourself, using `TCP_CORK`, which works best with `TCP_NODELAY`. It will still send out MSS packets immediately. Allows for application level flushing.

Can also be achieved with `send`$(\cdot)$ and its `flags`, by adding `MSG_MORE` to it.

1.2.3 TCP fast open

Normal TCP has 1 RTT delay due to 3-way handshake.

This option adds the TCP request to TCP's `SYN` message. Leads to 5-7% speed-up.

Problems:
Don't process request before handshake is compelete: risk to security, if handshake would not be completed. Possible DoS scenarios:

- Resource exhaustion attack – Leave connection half open with `SYN`-flood
- Reflection attack – Spoof live IP addresses, so those get spamed

Also problem with duplicated data. This also makes the above attacks easier.

Avoiding those problems:
What to do to tackle the problems

- Keep verified hosts on a *secure* whitelist
- Only trust peers that completed a handshake – this needs a proof
- Application must tolerated duplicated `SYN` data

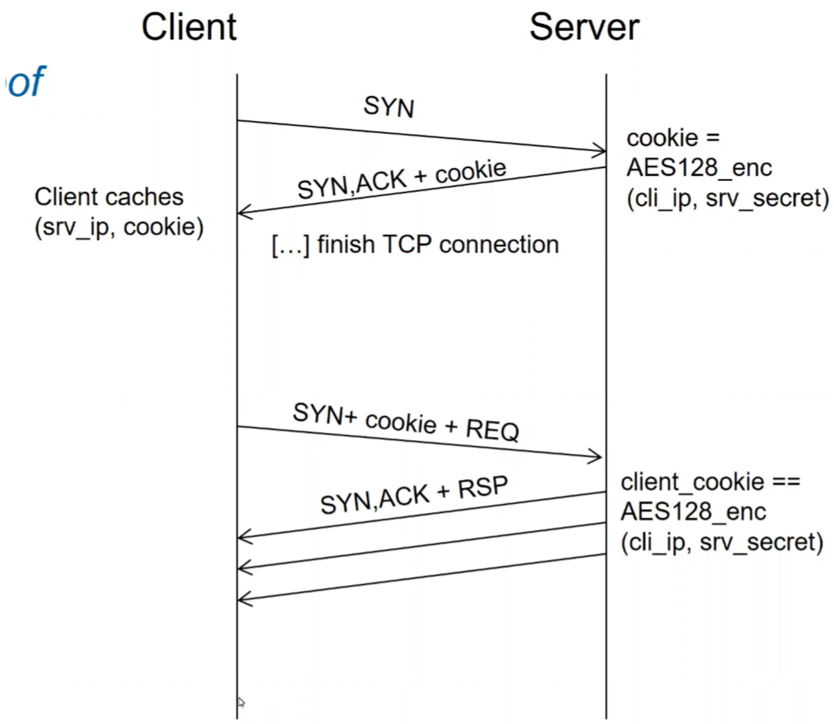

Figure 4: Using cookie as proof of validity of connection

Cookies are not valid forever. If cookie is invalid continue with normal TCP handshake.

Cookies may not reach destination. Can be dropped by middleboxes, also timeouts can occur. Cookies can be stolen → Rediraction attacks Cookies are dependent on the IP address of the client → change in networks invalidates the cookie, so mobile usage is not optimal

TCP fast open is not used today, as it leads to more problems than it solves. It also has significant problems with middleboxes.

Activating TCP fast open on servers using `setsockopts`(·) with `TCP_FASTOPEN` together with a maximum count of connections.

Activating TCP fast open for clients by piggybacking data to the connect call. Also need to use `MSG_FASTOPEN`.

1.2.4 Multipass TCP

Enable multiple connections between clients and servers. This enables for some backup paths, if one connection suddenly fails. However, TCP is *single-path* only, meaning there can only be one connection per socket. This leads to poor performance for mobile users, if, for example, if you change from WiFi to 4G networks.

How to use multi-path TCP:

1. Send options `MP_CAPABLE` with `SYN` message to signal multi-path capability.
2. If servers sends the same option back, both parties know, that the other party supports
3. Send specific `JOIN` with another `SYN` message to server, signalling to which connection to join the incoming one to.

Again, there are problems with middle boxes. Middleboxes (esp. NAT) may **change** the following field in the IP header.

- IP source address
- IP destination address
- Source port
- Destination port
- Sequence number
- ACK number

They especially can remove the `MP_CAPABLE` flag, which leads to unsuccessful connection establishment $\rightarrow$ normal TCP connection established.

Also, non-sequential ACKs (when only looking at one path) may be dropped.

Middleboxes may also change all other possible fields, but those changes are not common.

2 Design Patterns

This chapter describes, how to design networking protocols properly, such that they can be extended and worked with nicely.

General design principles are

Simplicity – Modular structure, with each module implementing a specific task
Modularity – Tool for the above
Well-formedness – Respecting system boundaries as memory capacity, defined state after errors, adapt to changes within limits
Robustness – Protocol can always be exectuted
Consistency – avoiding deadlocks, endless loops without progress

These lead to the 10 rules of design.

Definition: 10 rules of design

There are 10 rules, that every desing process regarding protocols must follow.

1. Define the problem well
2. Define the service first
3. Design external functionality first, then internal
4. Keep it simple
5. Do not connect what's independent
6. Don't impose irrelevant restrictions
7. Build high level prototyp and validate first
8. Implement, evaluate and **optimize** the design
9. Check equivalence of prototyp and implementation
10. Don't skip 1-7

2.1 Layering

Layering describes a modular structure of a protocol, each layer being responsible for a distinct task.

Advantages

- Smaller subproblems to handle on each layer
- Implementation as modules
- Exchangeable modules
- Reusable modules

Disadvantages

- Information is hidden → performance loss
- Redundantly implemented functionality on different layers

Cross-layer communication can help with redundancy, but is not common, as it is hard to change.

2.2 Protocol elements

2.2.1 Addressing communication parties

All communicating parties need to be identifiable within the network. Therefore, a unique identifier is needed for every party. You have to keep in mind address size, to be safe in the future. It may be possible to introduce an option to extend the address space.

Typically, single parties are identified, but it may also be useful to group mutiple data flows into one connection.

RTP flow IDs The RTP (*Real-time Transport Protocol*) is capable of grouping flows.

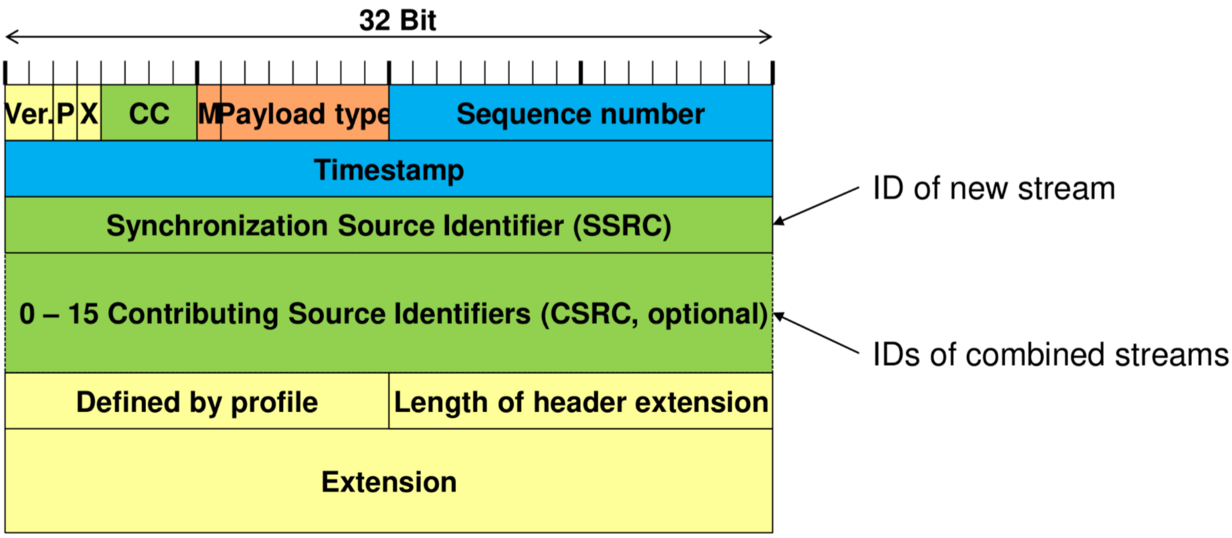

Figure 5: RTP header for grouping flows

2.2.2 Sequence Control

This is necessary in order to be able to asure, that packets are received in the correct order or in which order to process incoming packets. This commonly uses *sequence number*. Again, consider the maximum length of sequence number fields in your protocol header. What to do when you run out of sequence numbers to use?

2.2.3 Flow control

Adapt transmit speed to clients abilities. Popular approaches are

- Stop-and-wait – end2end
- Sliding window – hop2hop

General advice: Orient around RTT and only consider bandwidth reservation if resources are scalable w.r.t. number of communication parties.

Example: Transmission Rate Control (*TRC*). It uses the idea of applying flow control on multiple protocol layers simulaneously.

2.2.4 Access and congestion control

One need to consider how to avoid network overload, and what to do, if overloading is not avoidable.

In local networks, *medium access control* is used, in global networks *congestion control*.

Might be necessary to violate layering approach, as is done for TCPs *Explicit Congestion Control*.

Congestion control can be done on application layer by scaling down resolution of content (bitrate, lower resolution of video, etc.). May change based on messages from the client. Alternatively use *interarrival time* between packets and packet loss ratio (UDP only).

2.2.5 Error control

There often is the need to detect and correct corrupted packets in order to minimize packet loss.

General types of error correction methods are:

Automatic Repeat Request *ARQ* On error, rerequest the packet. ACK or ACK/NACK approaches

Forward Error Correction *FEC* Add redundancy to packets, use sophisticated methods such as CRC to correct them

TCP uses ARQ, but emulates a NACK with two consecutive ACKs for the same packet.

Selective ACKs can be used to acknowledge ranges of sequence numbers → more efficient acknowledging

Retransmission schemas for ARQ There are three main methods used for retransmission.

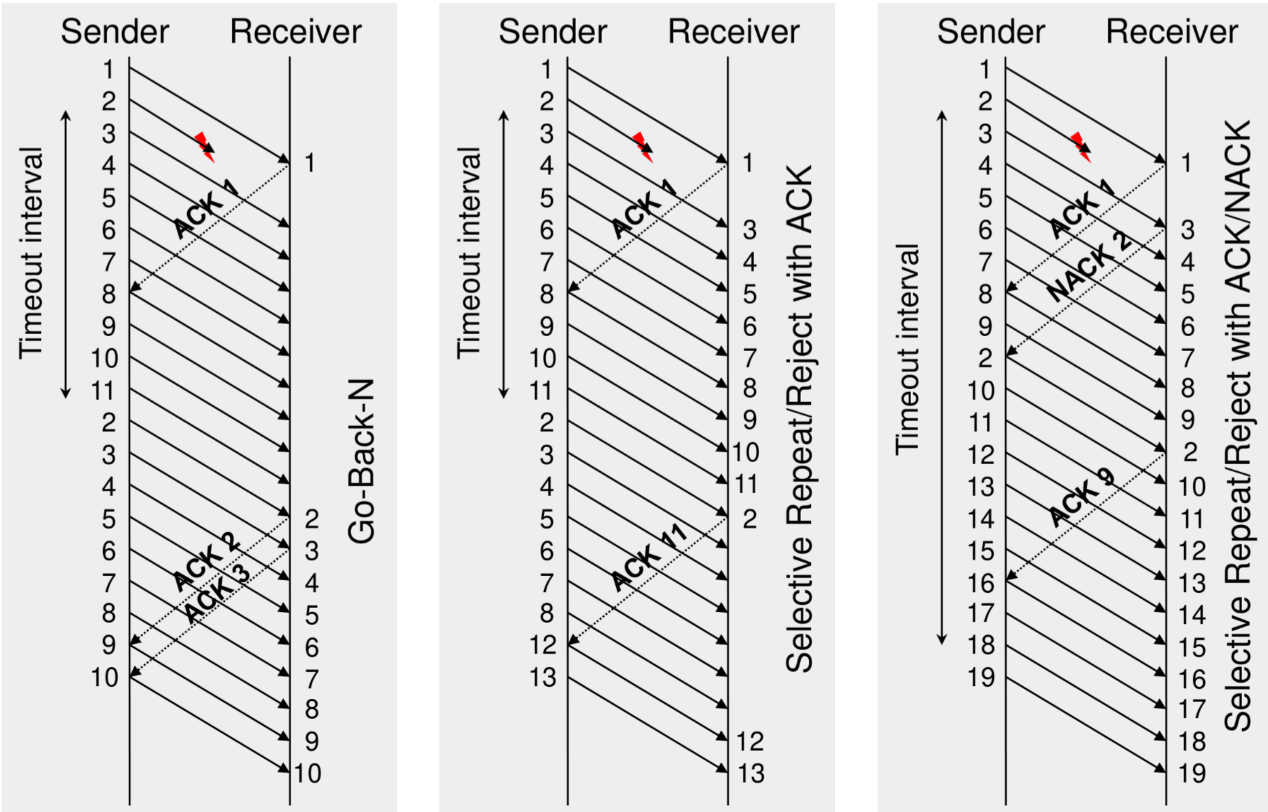

Figure 6: All introduced retransmission schemas

Definition: Go-back-N

Retransmitt all non-ACKed packages after a timeout occurs.

Definition: Selective Repeat/Reject with ACK

Commulative acknowledgement for all packages up to, in this case 11. 2 gets resend, as a timeout for the ACK for 2 occurs on the server side. All packets 3 through 11 are buffered, but partially retransmitted by the server nontheless.

Definition: Selective Repeat/Reject with ACK and NACK

Basically the same as selective repeat/reject with ACK, but NACK all missing packets upon receiving the next packet.

Use stop-and-wait only in cases with low RTT or where error rates are really low.

Use go-back-n if the receiver is very limitted.

Use any selective method in any other case.

Forward error correction schemas Send all packets n times, to be able to correct corrupted packets.

Redundancy per packet, i.e. with Hamming-Codes can also be used.

Also guessing what the lost packet contents were can be done, it smaller errors in some packets is not too bad. May be done by interpolating past packets.

Definition: XOR-redundancy

Combine n packets into one XORed packet and send it as well. XOR-redundancy is capable of restoring one packet. However the clients ability to process packets in a fastly manner must be considered, as only **exactly** one packet can be reconstructed.

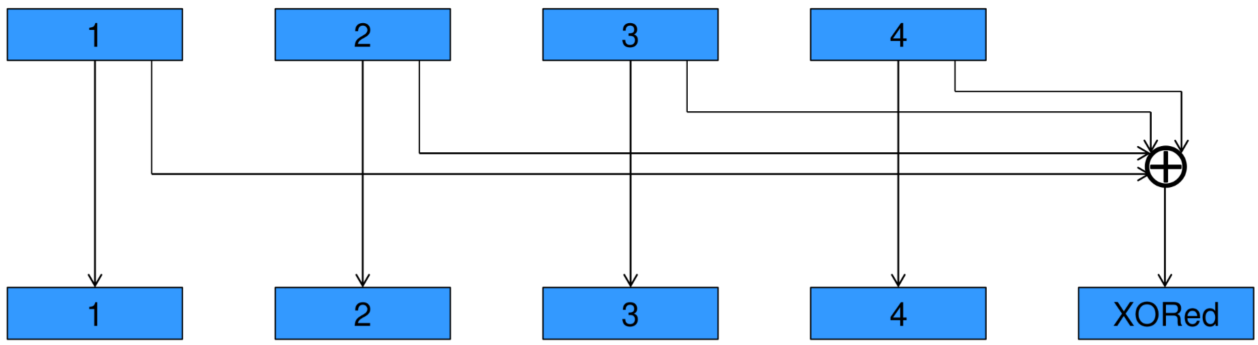

Figure 7: XOR-redundancy

Interleaving redundancy packets and original packets can lead to the ability of restoring n sequenced packets.

Definition: Hamming correction

Insert parity bits at positions that are labeled with a power of 2. Start counting with 1. This blows up the original message by $\log_2(|\texttt{message}|)$. A parity bit n is calculated by adding all positions, that in their binary representation, have a 1 in position $\log_2(n)$.

Let D be the hamming distance of two strings. Then a t errors can be corrected, if $D \geq 2t+1$, and t errors can be detected, if $D \geq t+1$.

Limitation: Hamming distance of 3 required, to be able to correct one error.

Important

Be aware, that **BCH** codes do not seem to be relevant for the exam and are therefore not included in detail here.

You don't have to be able to calculate BCH codes!

Definition: Bose, Chaudhuri, Hocquenghem Code (*BCH*)

Let

- *block length* be $n = 2^m - 1$
- *number of parity bits* be $p = n - k \leq m \cdot t$
- *minimum distance* be $d_{min} \geq 2t + 1$,

with $t < 2^m - 1$ and $m \geq 3$. Then t is the number of errors that can be corrected.

This is not a complete defintion. BCH codes are a family of codes, where each instance can be specified by giving a tuple.

BCH functions more or less like CRC.

Definition: Reed-Solomon Code

Able to deal with burst errors, basically a non-binary BCH code. It generates less overhead then BCH codes. Let

- *block length* be $n = q - 1$
- *number of parity bits* be $n - k = 2 \cdot t$
- *minimum distance* be $d_{min} = 2t + 1$,

where q is any power of any prime p, then t errors can be corrected.

Ignoring checksums might be a valid way to minimize paket loss, if application can tolerate errors.

Refector makes use of this approach, with an opt-in mechanism. It is used on the end hosts. Analogy: Postmen can handle wrongfully addressed letters to a certain degree. Refector does the same, and can therefore handle minimally corrupted headers. The "correct" application is found by choosing the one with the minimal hamming distance to the received header. There are some field in the IP header, that can be ignored in the process, such as *Version*. The protocol build on UDP. It reduces packet loss up to 25%.

2.2.6 Encodings

The focus here lies mainly on compression.

There are two variants: Lossy and loss-less compression.

Lossy compression is highly recommended for video streams, as small pixel errors are not recognizable by the human eye.

Lossless compression Again, two main methods:

1. remove redundancy: compress sequences of the same symbol
2. statistical analysis: statistically construct the optimal compression scheme

Definition: Run-length encoding *RLE*

Basically compress all sequences `AAAAAAAAA` $\rightarrow$ `A!9` and so on.

Definition: Differential Enconding

Instead of storing the values themselves, encode the distance to the nearest other datapoints.