Summary of

Communications Systems Engineering

WS 2021/22

Contents

1 Introduction

A *Communication System* is a system, that provides functionality in order to be able to transmit data to other communicating parties.

Communication Services implement concrete components with standardized interfaces, upon which communication systems are build.

A *Protocol* combines all those components into one entity. They define *messages* in a specific *format*, which obey *rules* for sending and receiving data. Protocols can be implemented on all layers of the ISO/OSI reference model, except layer 1 and 2, as they are responsible for the physical transportation of bits.

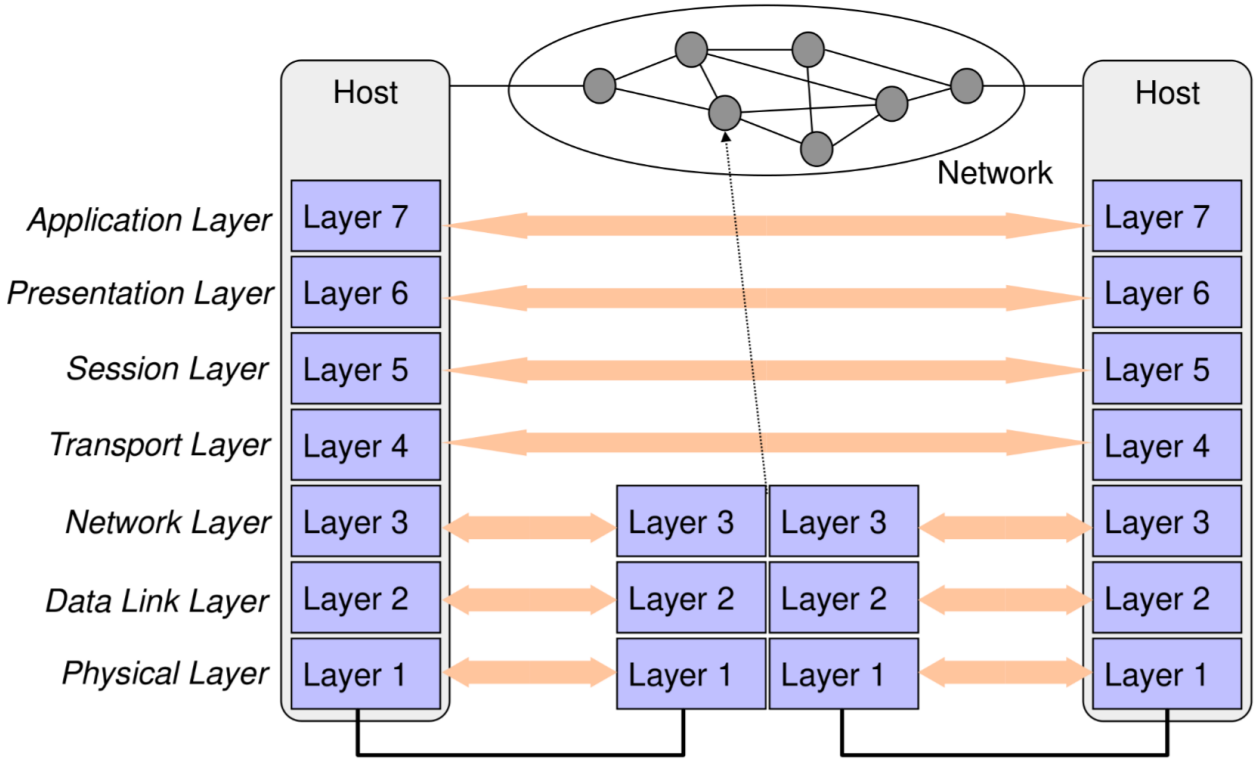

Figure 1: ISO/OSI

Units of communication are called *Packets*, which are structured, within the linux kernel, as shown in the following.

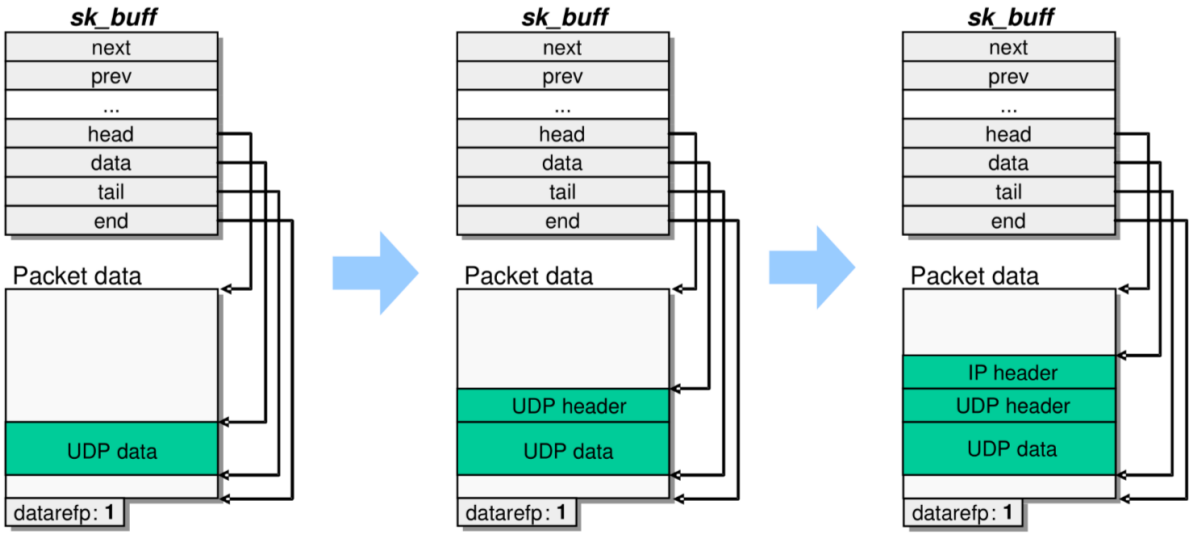

Figure 2: Packets in the Linux kernel

Each layer simply adds its header to the structure, thus minimizing copy-operations.

1.1 Clayton Tunnel Protocol

This particular malformed protocol was meant to ensure, that only one train enters a tunnel per track.

However, the protocol was not suited for signalling, that more than one train is in the tunnel, which leads to *Undefined Behavior* in cases where this happens on accident.

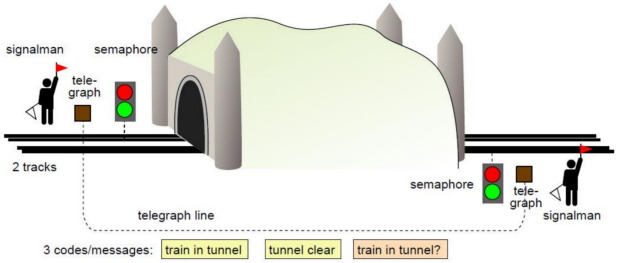

Figure 3: Depiction of the Clayton Tunnel Protocol

2 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.

2.1 C-Implemenation of Sockets

Definition: `socket(·)`

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

```
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.

```
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.

```
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(·)`

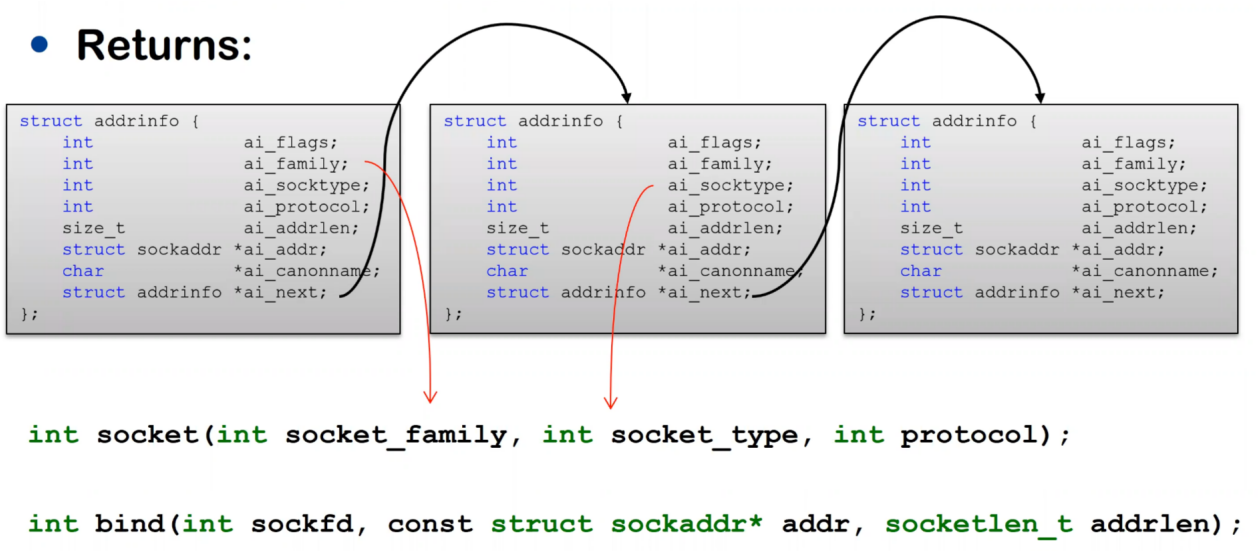

Figure 4: How to use `getaddrinfo(·)`. Note: All information used should originate in one block, not two as shown above.

2.1.1 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.

```
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(·)`.

```
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.

2.1.2 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.

```
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

2.1.3 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.

2.1.4 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`.

2.1.5 Connection Overview

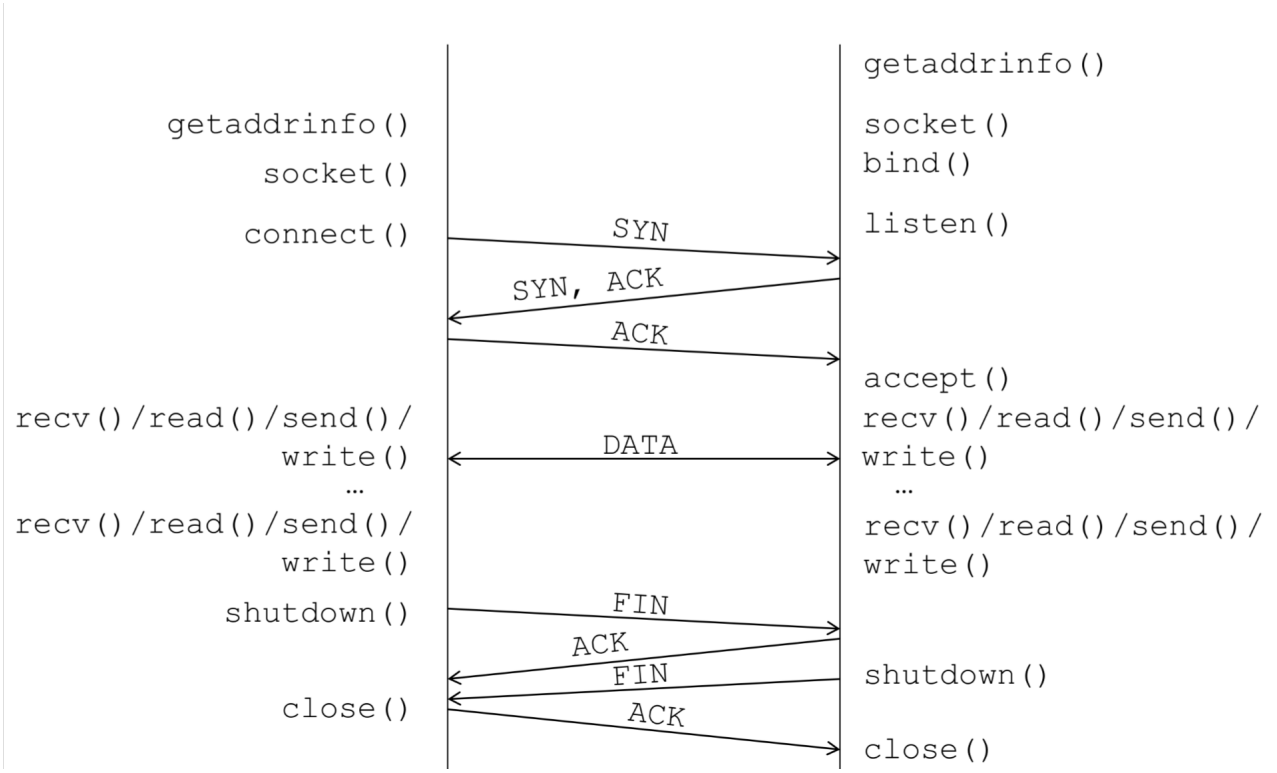

Figure 5: Overview over the course of a connection-aware socket

2.1.6 Configuring a Socket

Definition: `setsockopt(·)`

Apply options to sockets.

```
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)
- ...

2.1.7 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.

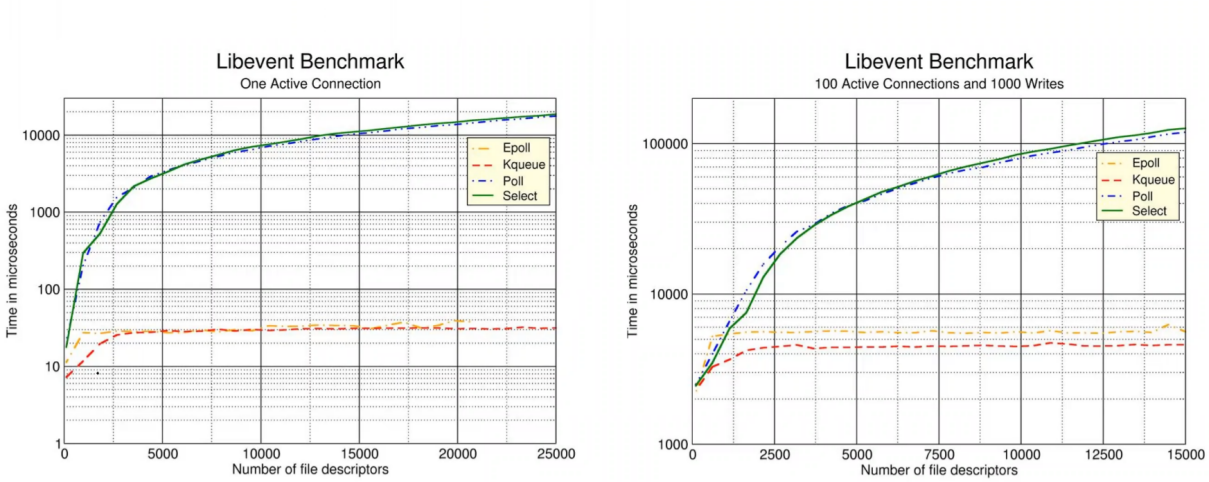

Figure 6: Performance of `epoll` vs. `select`(·)

2.2 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.

2.2.1 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`$(\cdot)$.

2.2.2 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 maximum 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 2.2.1, 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.

2.2.3 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

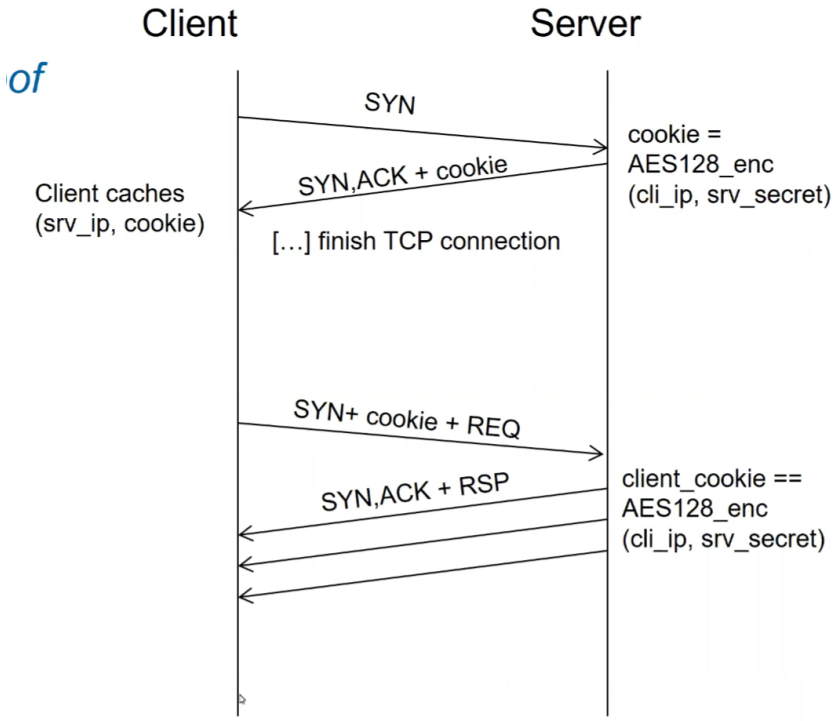

Figure 7: Using cookie as proof of validity of connection

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`$(\cdot)$ 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`.

2.2.4 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.

3 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

3.1 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.

3.2 Protocol Elements

3.2.1 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.

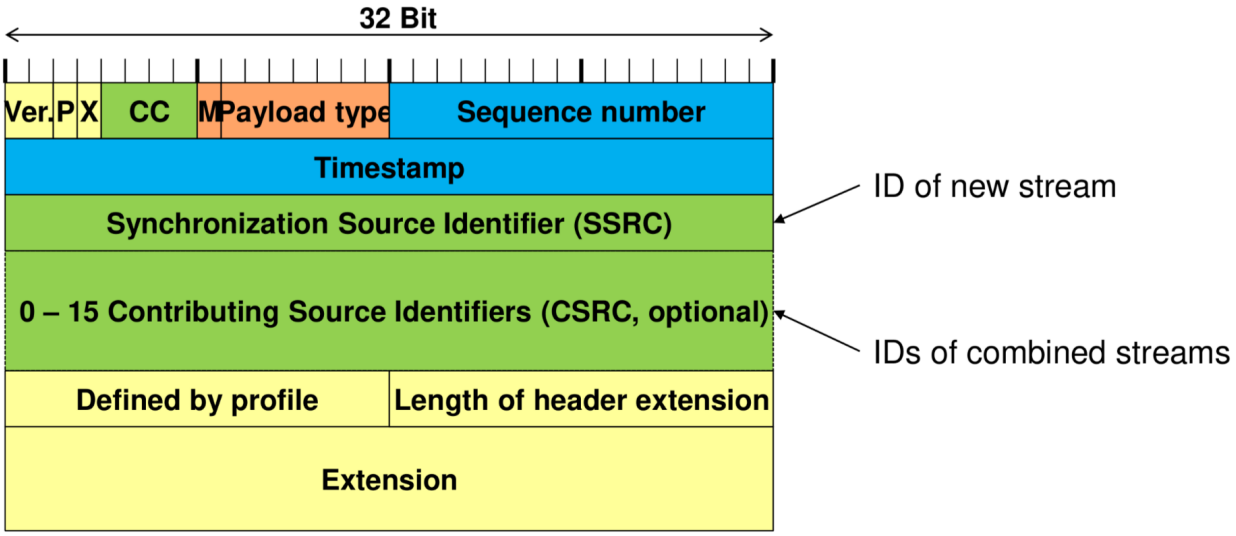

Figure 8: RTP header for grouping flows

3.2.2 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?

3.2.3 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 simultaneously.

3.2.4 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).

3.2.5 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.

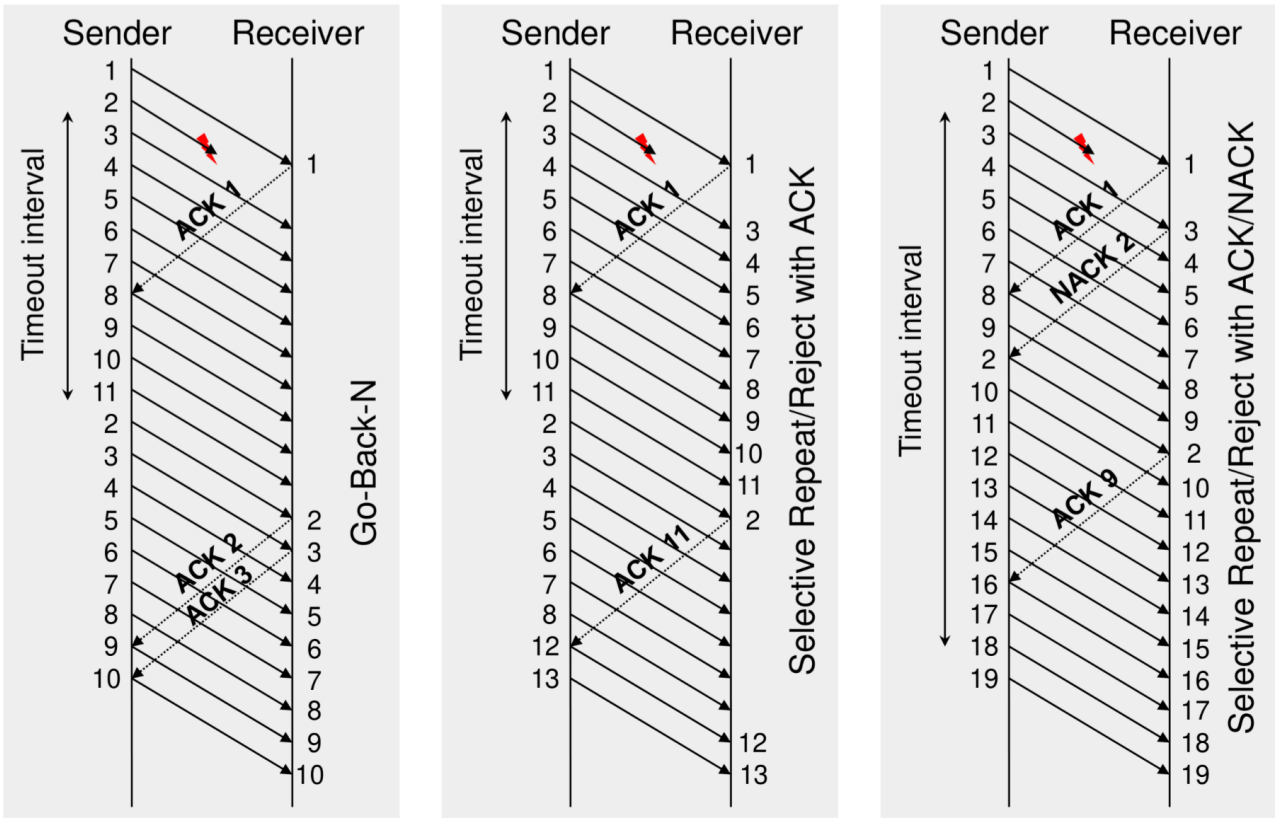

Figure 9: All introduced retransmission schemas

Definition: Go-back-N

Retransmit 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.

NACK-only strategies do not work in "normal" operation, they however work in high-throughput, high-reliability networks. This is due to the problem, that NACKs are not able to signal the reception of messages, only obviously missing packets. This way, missing of the last packet of a stream cannot be detected, as the client does not know that another packet was send, which does not generate a NACK which would signal to the server, that the last packet has gone missing.

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.

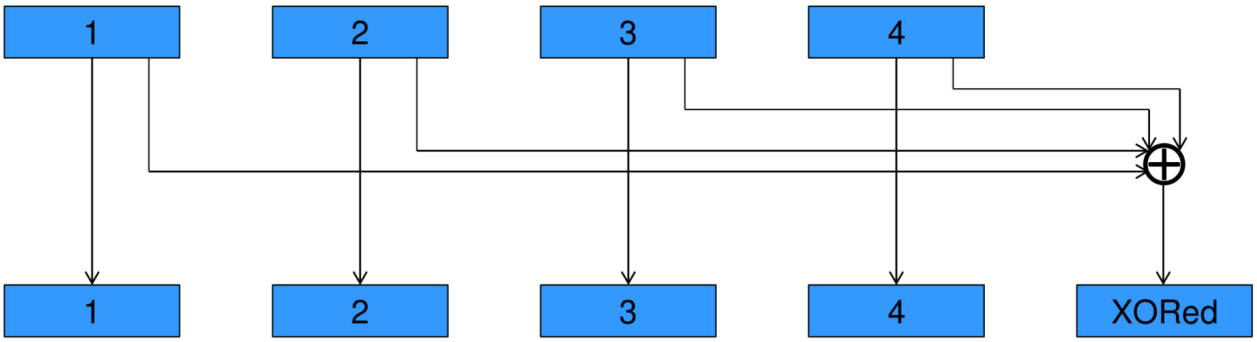

Figure 10: XOR-redundancy

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%.

3.2.6 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` → `A!9` and so on.

Definition: Differential Enconding

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

Algorithm: Lempel-Ziv-Welch Encoding (LZW)

Building a dictionary for phrases during encoding of the data.

$D = \{ \texttt{All unique symbols in the data stream} \}$

$E = \{ d_i \to e_i \mid d_i \in D, e_i \in \mathbb{N} \texttt{ unique} \}$

Start at the 1st position ($i = 1$) of the plain text $w = w_1 w_2 \cdots w_n$.

1. Encode $w_i, \ldots, w_{i+k-1}$ with longest match in E as e_j with $|e_j| = k$.
2. Add $e_j w_{i+k}$ to D and a corresponding encoding to E.
3. Move to position $i + k$.
4. Repeat until done.

We now will continue with **statistical analysis**.

Definition: Entropy

Information per symbol $I_i = -\log_2(p_i)$

Entropy for whole word $H = \sum_{i=1}^{n} p_i \cdot I_i$

with p_i is the probability of i appearing.

"How much information is carried by each symbol?"

This is used for *Huffman Code*.

Algorithm: Huffman Code

Encode a data stream in a binary tree, starting with the leafes.

Input: Data stream.

Returns: Optimal encoding w.r.t. entropy.

1. Assign each symbol of your alphabet the probability, with which it occurs in the data stream. Add mapping to a set.
2. Add two mappings $a \to p(a)$ and $b \to p(b)$ with lowest probability from your set to the tree. Remove them from the set. Left: smaller prob.
3. Add a parent node $ab \to p(a) + p(b)$.
4. Add generated mapping to set.
5. Repeat until set is empty.

Always choose *prefix free* code words.

The resulting code, which is generated by traversing the tree, can be evaluated w.r.t. entropy:

1. Compute information per symbol
2. Calculate entropy of the system, $H_{\texttt{theory}}$
3. Calculate entropy of the code like $H_{\texttt{code}} = \sum_{i=1}^{n} p_i \cdot |\texttt{code}_i|$
4. Compare the two entropies

However, Huffman code cannot guarantee the smallest code size ($H_{\texttt{code}} \approx H_{\texttt{theory}}$).

Algorithm: Arithmetic Encoding

Encode data as a value between $[0,1]_{\mathbb{R}}$

Assign each symbol a corresponding probability again. Start at position $i = 1$ of the data stream $w = w_1 w_2 \dots w_n$.

1. Divide the assigned interval (see figure below), that matches $p_{w_1 \dots w_i}$ into smaller segments, corresponding to $p_{w_1 \dots w_i} \cdot p_s$, where p_s is the probability of symbol s occuring.
2. Choose the subsegment (which is $w_1 \dots w_i \cdot s$), that matches $w_1 \dots w_{i+1}$ of your data stream
3. Increase i and continue until all is done.

Then, with $p_{w_1 w_2 \dots w_n} = p$, the data can be encoded using only $\lceil -\log_2(p) \rceil$ bits.

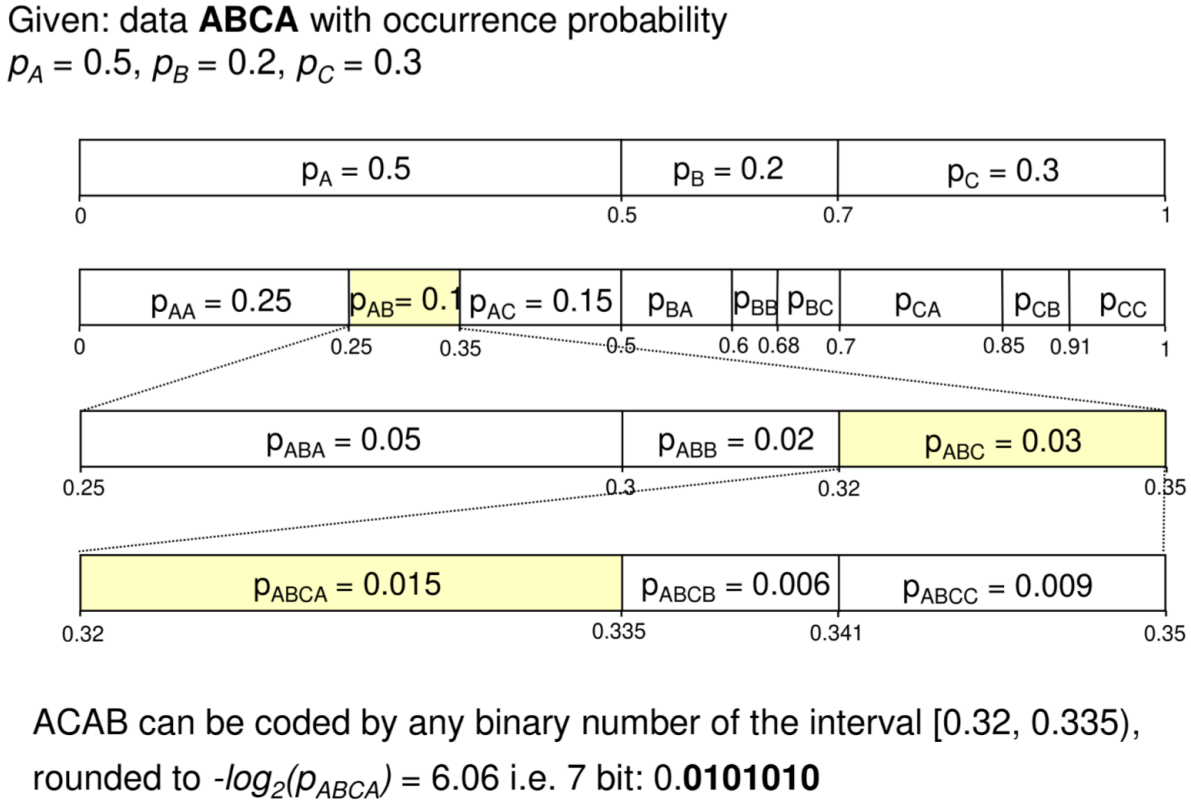

Figure 11: Example of arithmetic encoding

3.3 Protocol Design Aspects

3.3.1 Connection Type

You have to decide, if you want to design a *connectionsless* or *connection-oriented* protocol. Rule of thumb: The more controll parameters are nedded, the higher the chance you need connection oriented protocols.

If single commands will all fit in one UDP packet, would be alright as well.

UDP is also better suited for real-time and/or interactive applications, as it is much faster.

3.3.2 Signaling

Before, during and after connections, signalling must be done. This includes *status information*, *parameters of the connection*, specific to this connection and so on.

Definition: Out-of-band signalling

There are separate control and data streams. They may be implemented in two seperate protocoll "phases", or on different ports. One could even use different protocols for control and data streams.

Is able to provide QoS.

For example *SIP*.

Definition: In-band signalling

There is **no** separate control stream, but a combined data and signal stream.

For example *TCP*.

3.3.3 Maintaining Network State

Protocols may store states in the network on nodes in the path between to parties.

Hard State Maintaining State changes only with dedicated control messages.

Requires really reliable signalling. Checks, if a node is still alive has to be performed from time to time.

Example: TCP → `FIN` messages needed to terminate connections.

Soft State Maintaining Timeouts may delete a state. Must be refreshed to be kept alive.

This is almost always the approriate maintaining method!

Both state maintaining methods may implement the other one for certain things, s.t. some protocols use a mixture.

3.3.4 Control of Communication

Centralized Control

- Easy to program and debug
- Ex.: Client/Server

Decentralized Control

- More robust, no single point of failure
- Harder to implement
- Ex.: P2P

A mixed approach is *Software Defined Networking (SDN)*.

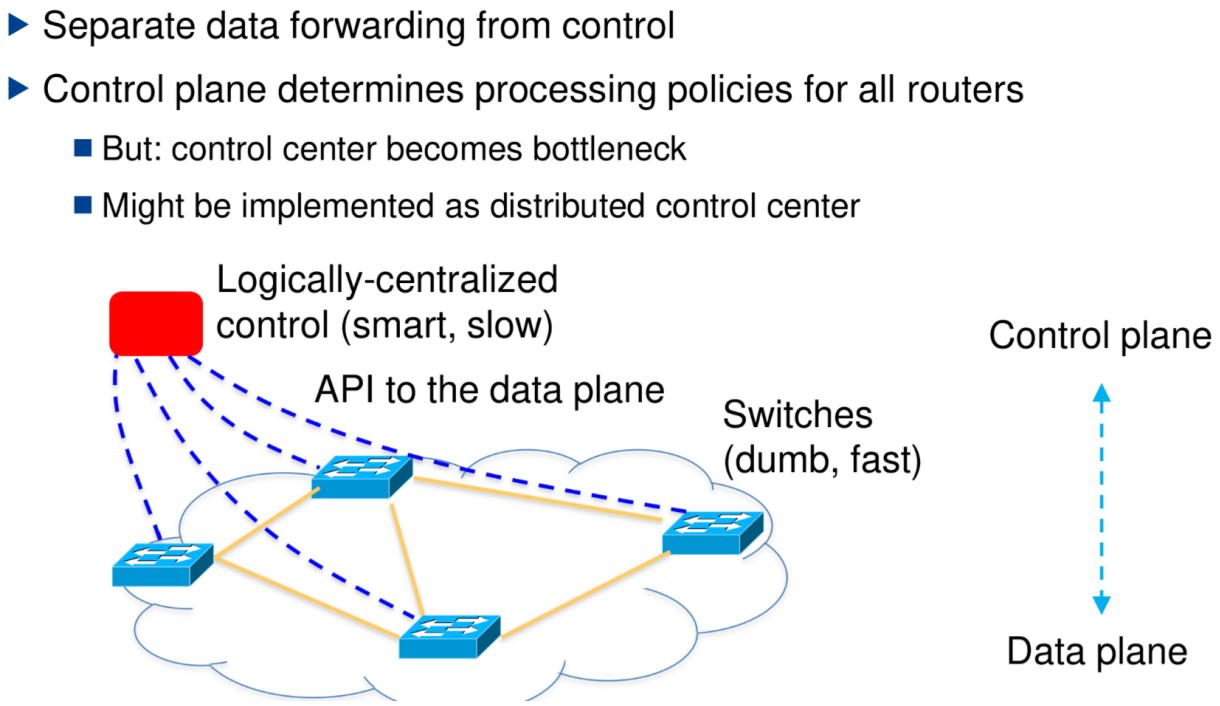

Figure 12: Software defined networking

3.3.5 Randomization of Protocols

Useful to make sequence numbers non-predictable, making it more secure.

CSMA/CD uses random waiting times on errors, to avoid consecutive collisions.

Definition: Random Early Discard

Throw away packets, **before** buffer capacity is reached.

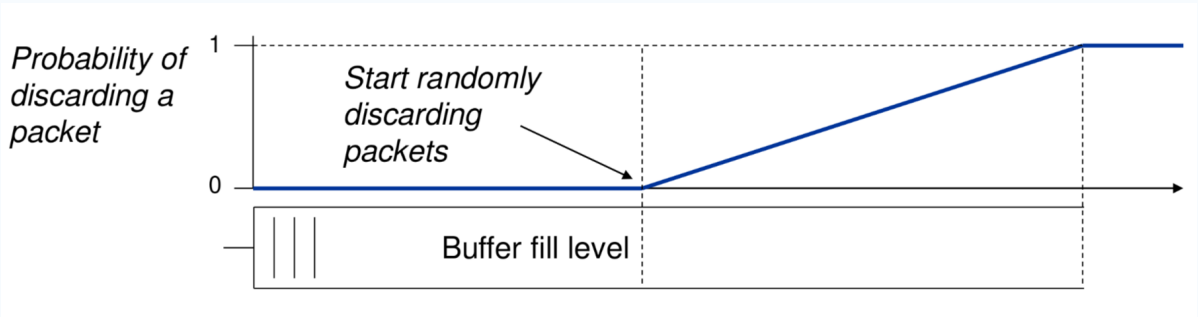

Figure 13: Random Early Discard

Zmap also makes use of this, in order to scan a complete network without being blocked by service providers, as scanning the addressspace linearly would result in a block.

3.3.6 Indirection

> "Every problem in Computer Science can be solved by another layer of indirection."

Multicast Multicast makes use of *indirection*:

- Client sends packet to address of multicast group
- Routers and middleboxes in general forward the packet to all clients, that registered for this group

Content Distribution Network Replicate content on a tree-like structure of other servers. Media servers are mirrors of a main server.

Indirection comes from the fact, that all clients use the same address to connect, but are *redirected* to the same server.

See `google.com`, there is no single Google server, there are thousands, that distribute the load among themselves.

Tool to use to achieve this: DNS.

3.4 Design of HTTP

HTTP was designed for documentation exchange between spacially far apart machines. Originally designed to work on simple `ASCII` strings, terminated by a carriage return.

3.4.1 HTTP/0.9

Operates on TCP port 80, and only supports *HTML*. Content is send directly, after a correct request was received. The connection is immediately closed after servicing a request.

Some commonly used commands are

GET Request for content
PUT Store document on server
HEAD Only request header information
POST Append content to webpage
DELETE Remove contents from server

3.4.2 HTTP/1.0

Introduces *multi-line requests* and *responses*. Response objects are not limited to HTTP anymore.

Content is send after meta information about the contents of the webpage. Otherwise basically the same as section 3.4.1.

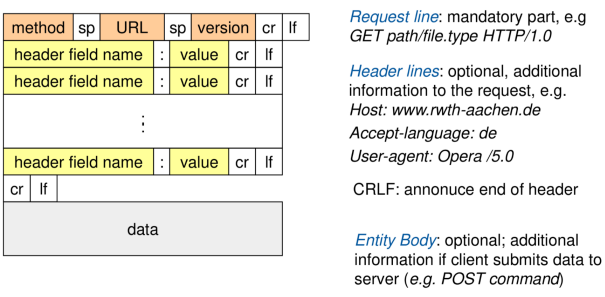

Figure 14: Request structure of HTTP 1.0

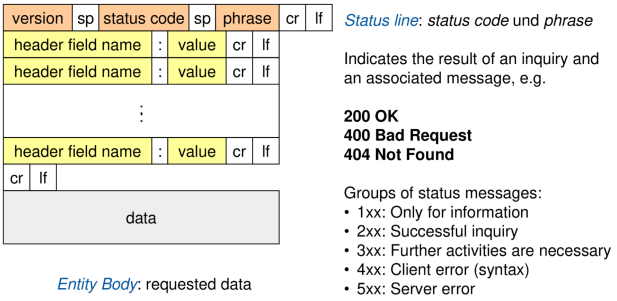

Figure 15: Response structure of HTTP 1.0

3.4.3 HTTP/1.1

In comparisson to section 3.4.2, the following things are extended:

- Request method
- Error codes
- Headers
- List of accepted file types: All types are allowed

Cookies can be used in order to make HTTP stateful. Connections are not immediatly closed upon sending out a response. Furhtermore:

- Content encodings and character sets were introduces
- Languages can be negotiated
- Caching enabled
- Cookies introduced
- ...

Dynamic Adaptive Streaming over HTTP (DASH) Method to make HTTP a "real" streaming protocol.

1. Cut down video file into several smaller chunks and store them in differnt bitrates
2. Client firstly loads file that contains names of all chunks that need to be downloaded over the course of streaming
3. Client requests chunks one-by-one, each with its own HTTP request. Bitrate can be adjusted with every request

This is **not** an **option** of HTTP, but really a use-case of it.

Head of Line Blocking Allthough HTTP/1.1 uses up to 6 different connections per connection to a webserver, websites still load pretty slowly. Solution: **Pipelining**.

This means that requests are send out directly after each other, without waiting for responses.

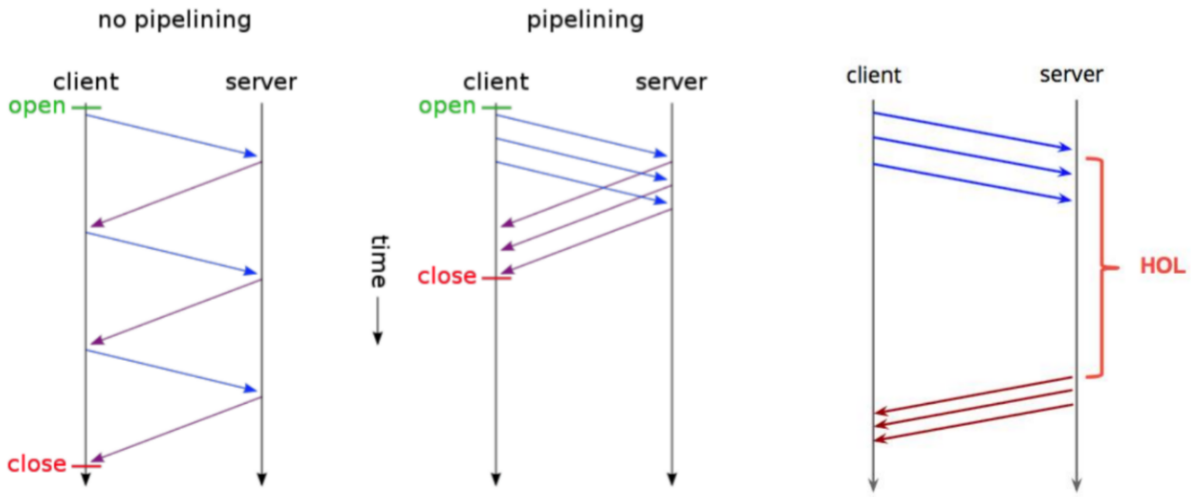

Figure 16: HoL Pipelining

However, large requests still slow down loading.

Domain Sharding Instead of keeping all resources on one webserver, distribute them accross multiple subdomains.

$$\texttt{www.example.com} \rightarrow \{\, \texttt{sub}_1, \ldots, \texttt{sub}_n \,\} \texttt{.example.com}$$

This leads to more DNS lookups, which can also slow down loading, also there are more connections to be kept alive.

If one knows what he is doing, domain sharding can increase throughput. However, most application overuse it, which leads to underused TCP connections and therefore more overhead, which in turn slows down communication again.

3.4.4 Spriting Images and Concatenating Files

Advantages

- Reduces number of overall requests

Disadvantages

- Waste of bandwidth if not all resources are needed
- If one single element is updated, the whole bundle must be acquired again
- Sprites require slow parsing
- Spritet images need more memory

$\Rightarrow$ Enjoy with care, there is not much good to it.

3.4.5 Resource Inlining

Idea: Embed resources in webpage itself.

Inline CSS, JS, even images (using $\texttt{Base}_{64}$) and audio can be embedded.

Advantages

- Reduces number of overall requests

Disadvantages

- Waste of bandwidth if not all resources are needed
- If one single element is updated, the whole bundle must be acquired again
- No reusability of assets, need to be included into every source file that use those resources, which leads to a memory penalty

3.4.6 HTTP/2

Goals were not to use as many connections and lower the *perceived* time things take to load, while retaining the high-level semenatics of HTTP/1.1. HTTP/2's smallest unit of communication now is a so called *frame*, which are send over *exactly* one connection. It also uses binary represenation instead of `ASCII`.

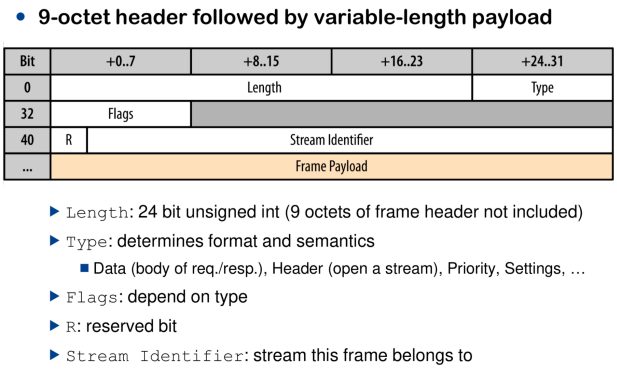

Figure 17: Frame of HTTP/2 communication

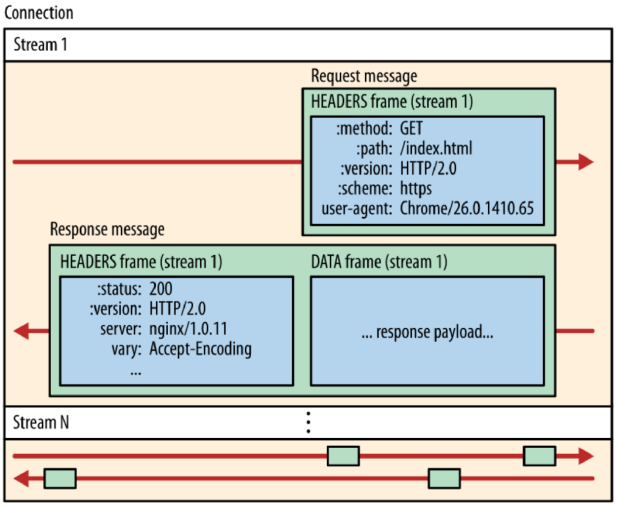

Figure 18: Layout of connection via HTTP/2

HTTP/2 also allows to specify, in which order to download content from the web server. The order is represented by a tree structure. This allows for signal priortization.

This is only a recommendation. The server is not required to follow the requested download order.

Also, a server is allowed to `PUSH` data to the client, without the client requesting it. It can be used to pre-send CSS files and similar data. The server has to announce the push, the client can deny it. Pushed data *has* to come from the same domain, such that no external data can be pushed to the client.

Is scheduled to be removed from Google services.

Flow control is supported, but has to be implemented by the application. Additional flow control is useful, as TCP only supports flow control on a per-connection basis. As HTTP/2 supports more than one stream per connection, per-stream flow control can be implemented as well. There is no implemenation specified in the standard.

Header Compression with HPACK Headers have up to 800 bytes of data, which is repetitive. They are now allowed to be compressed, using Huffman encoding and a combination of static and dynamic tables (see section 3.2.6). The static table contains codes for the 61 most common headers used in HTTP/2.

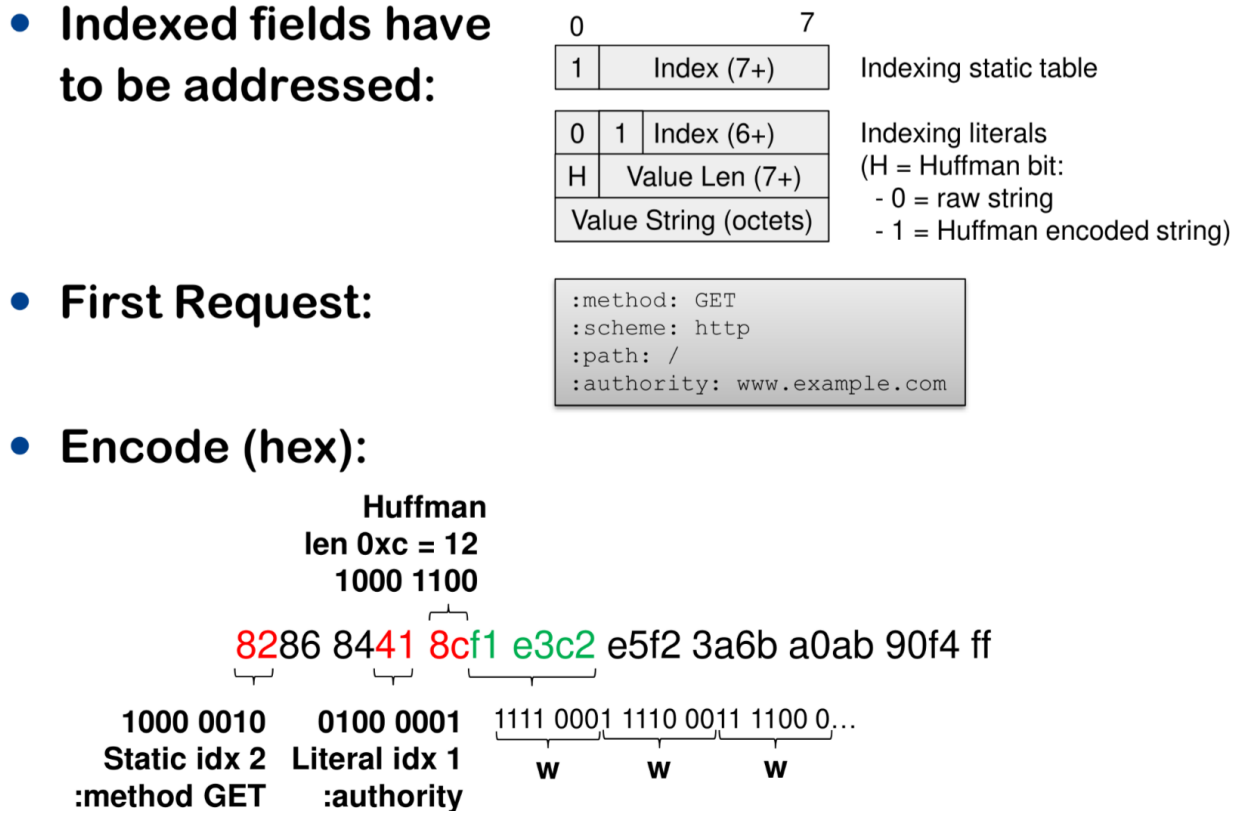

Figure 19: Huffman encoding of HTTP/2 headers

Unknown information can be learned and stored in the *dynamic* part of the table. This way, the client can send shorter requests, saving bandwidth. Things to be learned start with `0b01XX`, as this signals, that the next bits are not Huffman encoded, but just plaintext.

HTTP/2 and TLS Both together build `HTTPS`, the secure version of HTTP.

Important

HTTP/3 is the newest version available for HTTP communication.

3.5 Quick UDP Internet Connection (QUIC)

QUIC uses many already established ideas and combines them into one single protocol, whilst avoiding the huge overhead introduced by TCP.

Its main focus lies on better supporting HTTP, which it achieves by being able to map QUIC streams to HTTP/2 streams.

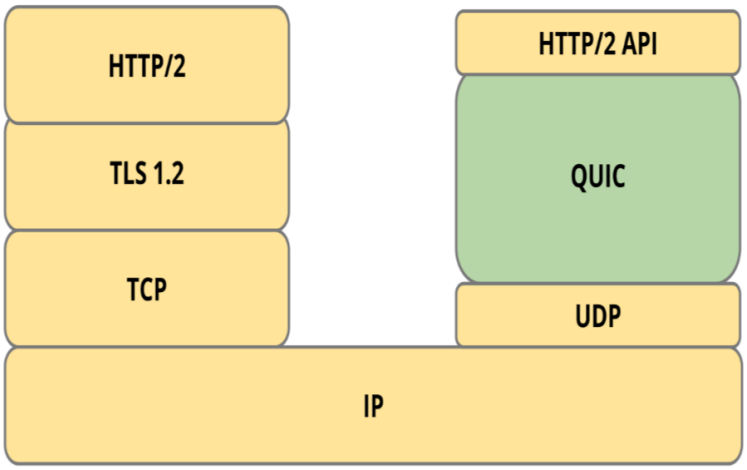

Figure 20: Comparission of structure of QUIC and HTTP

Multiplexing in QUIC Because UDP is used instead of TCP, multiplexing is smoother, as Head-of-Line blocking is not possible anymore, due to missing control mechanismns in UDP. Of course, lost packets effect the stream the packet belonged to, but not all streams anymore.

Connection Esablishment It uses the same idea as TCP fast open in section 2.2.3. Also TLS 1.3 is directly integrated in QUIC, which uses a 0-RTT design. This allows to connect to servers, that you connected to in the past, by sending an old token, which proofs your identity. Of course there is a timeout, after which a completly new connection has to be established.

Also, QUIC is not bound to the IP address of the parties, but to random 64-bit identifiers. This allows for easier mobile usage.

CHLO – Client Hellow
SNI – Server Name Identification
CERT – Certificate of Servers identity
REJ – Reject
SHLO – Server Hello
VER - Verification
SRCT – ?

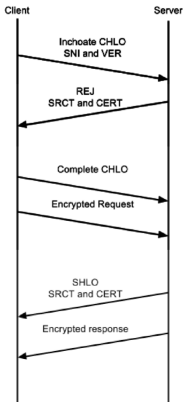

Figure 21: Connection to a server with QUIC

Subsequent connections send header information, including verification and client hello, together with encrypted request.

Congestion Control and Reliability in QUIC Is based on TCP NewReno and also implements TCP Pacing, which spreads send packages evenly across RTT. This avoids peaks, therefore lowering the risk of congesting the network. The reception of the first ACK (or NACK) is anticipated.

Selective Acknowledgements are used. Retransmissions have new sequence numbers, which increase monotonically.

Connection Migration Upon change of IP address of a client, TCP would drop the connection. QUIC is able to migrate the "new" connection into the old one, as connections are not identified by the clients ID, as explained in section 3.5.

4 Kernel Networking

A kernel is used, to abstract the complete interaction with the hardware, s.t. user space programs do not need to worry about any of that. Basic kernel tasks include, but are not limited to:

- I/O interface for user space programs
- Handle interrupts
- Handle hardware utilization
- Structuring hardware systems
- Virtualization …

No kernel ever trusts a user space program.

Definition: Kernel mode

The mode, that the kernel operates in. In kernel mode, software has full access of the hardware, and has priviledged rights.

Definition: User mode

The mode, which user applications are assigned to. No hardware access, operations are only accepted, if kernel API is used.

Kernel is not one single thread, instead utilizes threads for all common operations, which are spawned by a `init`-thread.

4.1 Interrupts and Friends

Interrupts are flags set by hardware components to get the "attention" of the CPU. If an interrupt is raised, the current operation is paused immediately, unless the current operation itself is an interrupt handler. Interrupt handlers are functions within the kernel, that perform *very* short actions, depending on what the interrupt is all about.

Interrupt sources (i.e. events, that cause an interrupt) can be system calls, external interrupt requests (by hardware), as well as exceptions.

If an interrupt is called, a context change has to be performed, which introduces some computational overhead.

4.1.1 Interrupt Vectors and Handlers

Might also be called *Interrupt Descriptor Table (IDT)*.

Interrupt vectors are used by the kernel to map a certain interrupt to the corresponding interrupt handler. This vector is a partition in the RAM, which maps interrupt sources to their respective handlers.

Interrupt handlers are loaded on boot, too. Loading them on demand would lead to significant overhead, because everytime an interrupt would be raised, the correct handler has to be fetched from the hard drive again.

Some also important terminology:

PIC Programmable Interrupt Controller
IRQ Interrupt Request

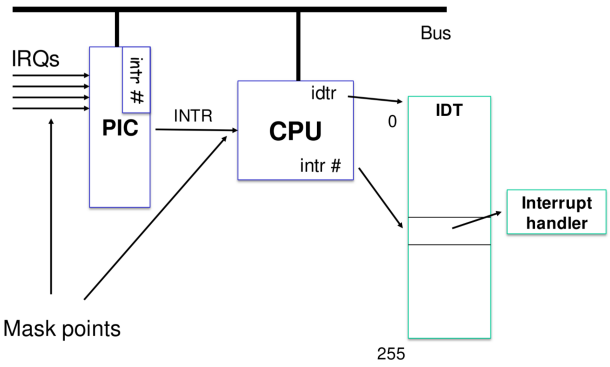

Figure 22: Conceptual scheme of how an interrupt is handled in `x86`

Interrupt handlers, under no circumstances, are not allowed to run for a long time or perform blocking operations, as the system gets unresponsive quickly. Often times, a flag is set, the actual handling of the interrupt is performed after a polling operation section 4.2.

4.1.2 Context Switching

When an interrupt is raised and the handler is called, the context, in which the previous process was running, has to be saved, in order to be able to continue the process after the interrupt was handled.

This basically means storing register values to a structure in the kernel.

In some cases, only a subset of registers is stored, as interrupts on that specific machine are not touching the other registers in any way. This saves time, because unmodifiable data is not stored.

4.1.3 Interrupt Handling in Linux

All the methods introduced in the following are executed in kernel space!

Handling of interrupts in Linux is twofold.

Top Half This is the part that is executed on the spot. Those actions include:

- Mask other interrupts, block other interrupts from firing
- Save context of prev. process
- Call the proper interrupt handler and schedule it

Bottom Half This is where all the interesting stuff is happening. This is the "real" interrupt handler, which might be time critical, but nontheless is scheduled as a normal process, maybe with a higher priority.

Important

Interrupt handlers cannot sleep. This would be way to expensive. There also is no context to restore, as interrupt handlers are not normal functions. It has no corresponding *process control block*.

4.1.4 SoftIRQs

Definition: SoftIRQ

A SoftIRQ is a process that gets scheduled with the highest possible priority. It does not interrupt the current process, but is certainely scheduled to be executed next.

SoftIRQs are **not** bound to one specific CPU. They are reentrant, s.t. data structures have to be protected with spinlocks.

Those are statically allocated at boot. A special struct is used to pass information to that SoftIRQ, which (not only) consits of:

- A pointer to the handler
- A pointer to the data that has to be passed to the SoftIRQ

These structures are used by `ksoftirqd`, the SoftIRQ daemon in the kernel, which spawns for every core of the CPU.

A special function `do_softirq`$(\cdot)$ is used in order to call all pending SoftIRQs, they can also be called in response to a finished interrupt handler. These can be run on multiple cores simultaneously and **reschedule themselves**.

It is strongly disregarded to implement SoftIRQs by yourself.

4.1.5 Tasklets

An easier to use version is the "Tasklet":

Definition: Tasklet

Build on top of SoftIRQs, but can also be created dynamically. Tasklets are assigned to one concrete CPU, and are cache-affine.

Tasklets **are** bound to a specific CPU. Tasklets are not reentrant. This means, that it does not need to protect data.

There are several functions to work with tasklets.

- `tasklet_init`$(\cdot)$
- `tasklet_disable[_nosync]`$(\cdot)$ (`*_nosync` does not wait for the tasklet to return)
- `tasklet_kill`$(\cdot)$

Note: Tasklets are run in interrupt context.

4.1.6 Work Queues

Is a subsystem of the Linux kernel. There is one dedicated *default worker thread* per CPU. You could also create your own *special worker threads*. Often times, this is not needed. However, if you need to perform much processing, defer to a worker thread in order to be able to handle incoming packets directly, when they are handed to your main thread.

Work executed in the default worker thread should ideally not block, as this prevents other tasks to be completed.

4.1.7 Overview and Comparisson

	SoftIRQ	Tasklets	Work Queues
Execution context	Deferred work runs in interrupt context.	Deferred work runs in interrupt context.	Deferred work runs in process context.
Reentrancy	Can run simultaneously on different CPUs.	Cannot run simultaneously on different CPUs. Different CPUs can run different tasklets, however.	Can run simultaneously on different CPUs.
Sleep Semantics	Cannot go to sleep.	Cannot go to sleep.	May go to sleep.
Preemption	Cannot be preempted/scheduled.	Cannot be preempted/scheduled.	May be preempted/scheduled.
Easy of use	Not easy to use.	Easy to use.	Easy to use.
When to use	If deferred work will not go to sleep and if you have crucial scalability or speed requirements.	If deferred work will not go to sleep.	If deferred work may go to sleep.

Advice: If you are not sure about performance, try from *right* to *left*

Figure 23: Comparisson of the three introduced methods

4.2 Polling

Polling is constant, periodical checking, if a certain condidtion is met or a flag is set. The polling period is critical, as too fast checking may overload the CPU, whereas too slow checking leads to not being able to handle fast occuring events.

4.3 Handling Pakets in the Kernel

A packet follows the depicted path below.

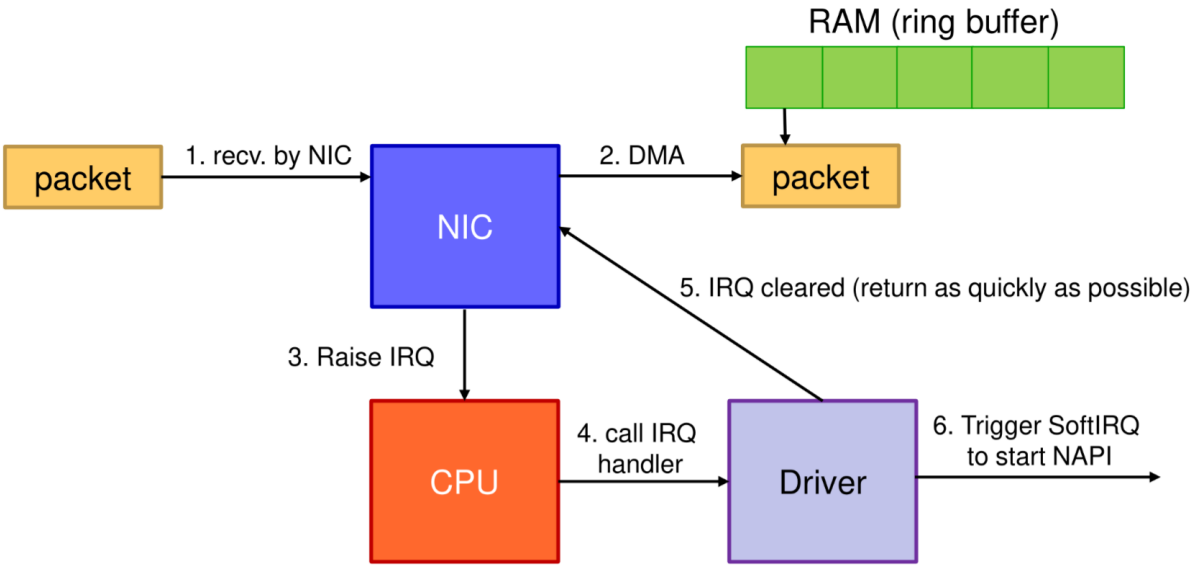

Figure 24: Processing of a packet

The SoftIRQ in (6.) is marked pending, and then is executed from the kernel daemon responsible for SoftIRQs, `ksoftirqd`. This SoftIRQ calls `net_rx_action`(·), which disables NIC interrupts, and starts a polling mechanism, see fig. 25. This is done to efficiently receive data without being interrupted.

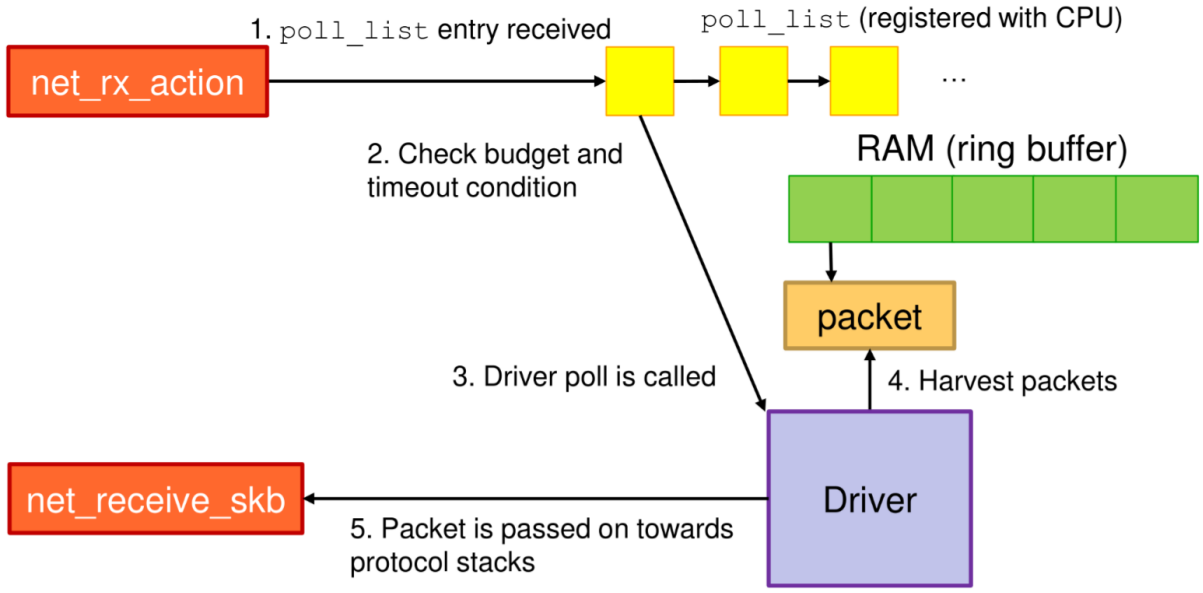

Figure 25: Inner workings of the New API

The budget in step (2.) prevents the poll to hug all system resources, as with each handled packet, the counter of the budget is decreased by one. This enables efficient handling of bursts, without

consuming the whole CPU for a long time, if there were much more packets to come.

When a concrete packet is processed, the corresponding memory in the RAM is un-mapped, s.t. the network driver cannot write to that specific location anymore. This way, packets can be processed in-place.

After a packet is processed, the memory is re-mapped.

4.4 Socket Buffers

These are the most fundamental parts of networking.

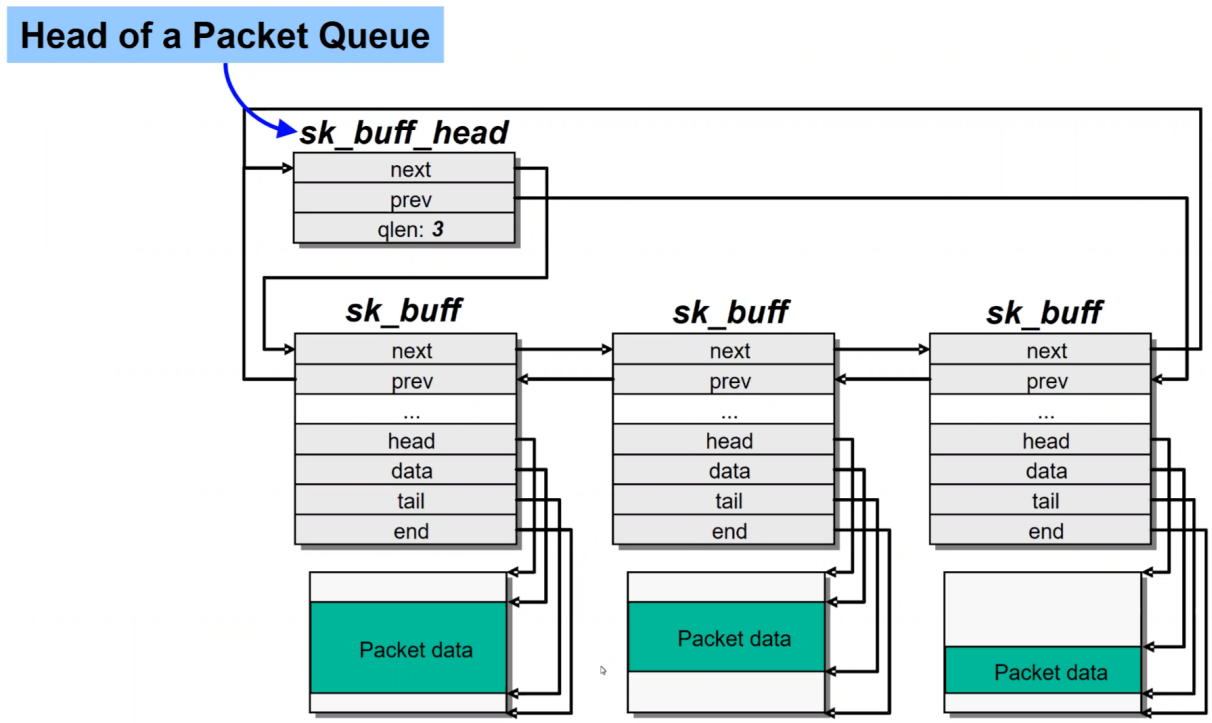

Figure 26: Depiction of usage of `sk_buff`

Each `sk_buff` is a double linked list. The position of the start of each header (TCP, UDP, IP, whatever) is stored in data fields within `sk_buff`. There are also field to store pointers to headers within headers (`*_inner_*`). Some important components:

`next` for double linked list
`prev` for double linked list
`list` to what list the buffer belongs
`sk` socket that packet belongs to
`dev` device the packet belongs to
`stamp` timestamp
`sc` security path traversed in the packet
`cb` control block, used for flags
`len` total length of packet

`data_len` lenght of payload
`user` reference counter
Header information as described in the paragraph above

4.4.1 Working on Data with `sk_buffs`

There exist blocks of memory resevered by the kernel, which are able to hold a complete packet, including header information. These are *only* used by the kernel.

Note, that the data pointer starts on top of the IP header. This is, because, the most upper layer (MAC) is not interessted in where the actual payload starts. It just needs to know, where the data starts, that needs to be send.

On reception of such a packet, the headers have information needed to disassemble the whole packet into different headers and finally the payload, by providing a field which points to the start of the data, as viewed for that protocol. For example, the IP header might have a length field, that when added to the address of the IP header, points to the start of the UDP header.

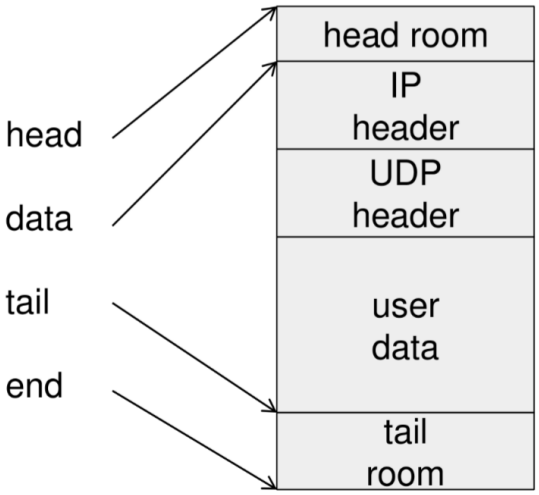

Figure 27: Data representation in an `sk_buff`

4.5 Packet Processing Workflow

`netif_receive_skb`(·) in fig. 28 is called by a function somewhere in `net_receive_skb`(·) form fig. 25.

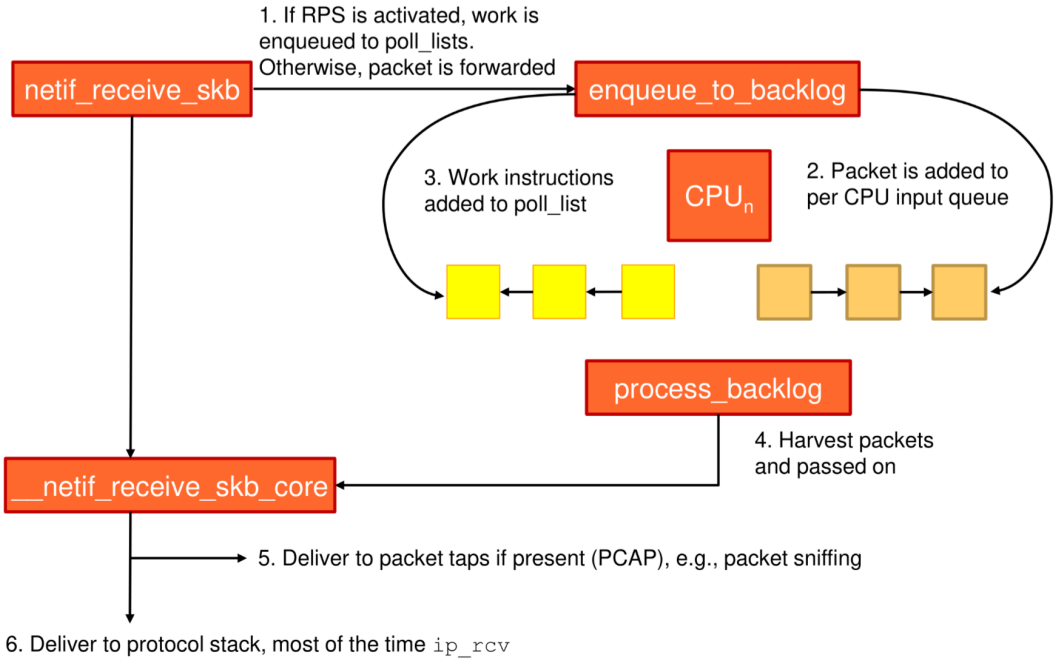

Figure 28: Packet processing workflow

4.6 Memory Management in the Kernel

`malloc`(·) is a user-space function, that deliveres a contiguous region of *virtual memory* to the user process of a size that can be selected on random by the user application. `free`(·) releases that memory, and makes the regions available to other processes again.

However, this are user-space functions. The kernel needs to be able to access the physical memory with harsh performance restrictions. Memory should be allocated without the kernel threads being required to wait (or sleep).

This leads to

4.6.1 Typical Allocation

Definition: `kmalloc`(·)

This is the kernel equivalent of `malloc`(·).

```
void *kmalloc(size_t size, int flags);
```

`flags` – Change the behaviour of the allocator

Common flags are

- `__GFP_HIGH` – Access to emergency memory pools
- `__GFP_NOFAIL` – Try allocating on failure again
- `__GFP_DMA` – Allocate DMA cabable memory
- `__GFP_ATOMIC` – Don't sleep

- `__GFP_KERNEL` – "Normal" allocation of memory

Free memory allocated by `kmalloc`(·) with `kfree`(·).

If more memory is needed than can be delivered `kmalloc`(·), try using `vmalloc`(·), which returns a virutal memory range to the kernel.

4.6.2 Slab Allocation

Pre-allocate slabs of fixes size in the kernels memory region, which then can be allocated by any other process very efficiently.

This is the preferred method of allocation for incoming IP packets.

Flags control the behaviour of the slab allocator function; some common and useful flags are:

- `SLAB_POISON` – Prefill the slab with a known value pattern
- `SLAB_RED_ZONE` – Reserve areas at the end of slabs for buffer overflow detection

Also, flags from section 4.6.1 (or variants) can be used.

Slab caches can be repurposed, if less elements of one slab size is needed and more of another.

4.6.3 Kernel Process Stacks

Each kernel thread has a **fixed size** stack, of typically 8kB on 32-bit machines (2 slabs). Thus, don't use static allocation for large buffers in the kernel, but use `kmalloc`(·) and family for large buffers.

If a kernel stack runs out of memory, `thread_info` in the threads control structure might be overwritten, which corrupts this thread for sure.

4.7 Sleeping and Locking

Locking is essential for synchronizating threads in the kernel and guarantee correct concurrency, i.e. resolving race conditions and protecting critical regions.

There are numerous sources for concurrency in Linux:

- Interrupts – interrupts current process
- SoftIRQs/Tasklets – same as for interrupts
- Kernel preemption – threads are assigned slots
- Sleeping/Synchronization – sleeping tasks are not run until awaken
- Symmetrical Multiprocessing – more than one processor executing kernel threads

There are atomic datatypes (`atomic_t`, `atomic64_t`, etc.), but those only enable **one** atomic operation in a row.

4.7.1 Deadlocks

The locking order is essential in avoiding deadlocks. Nested locks must *always* be obtained in the same order. A deadlock can only occur with at least two processes and two locked ressources.

4.7.2 Semaphores

Semaphores are initialized with a number of allowed concurrent accesses.

When **waiting** for a semaphore, it is checked if the reference counter of the semaphore is greater than 0. If so, decrease the counter and enter critical section. If not, the calling process is put to **sleep**, and awaken when the semaphore counter is greater 0 again. Sleeping is most useful, if the lock is held for a long time by a process.

When **releasing** an instance of a semaphore, simply, *atomically*, increase the semaphore counter.

4.7.3 Spinlocks

Spinlocks are basically semaphores with an initial counter value of 1.

Threads, that want to acquire control of a lock are not put to sleep, when no instance is available. Instead, it busy waits and tries to acquire the lock over and over again.

Within a spinlock, very concise operations, that are fast to execute (read: no memory operations), as other processes that are busy waiting for the lock, are blocking ressources unnecessarily long.

If you have acquired a spinlock, the scheduler is disabled!

4.7.4 Locking in User Context

Locking in user context should be performed by semaphores, as spinlocks disable the scheduler.

4.7.5 Locking User Context and SoftIRQs/Tasklets

Use `spin_lock_bh`(·) to disable SoftIRQs on that specific CPU, and then grabs the lock. Disabling hardware interrupts is also possible, but not as common. These are `spin_lock_irq[save]`(·). The `save` version is recommended, as this stores and restores interrupt context, e.g. what locks were locked before.

This way, other SoftIRQs are not able to interrupt your processing in cricical sections, if you share memory.

4.7.6 Locking between Tasklets

Only important between different tasklets, as instances of the same tasklet always run on different CPUs.

Just use normal `spin_[un]lock`(·). The `bh` version is not needed, as you already are a tasklet and none tasklet on the same CPU will interrupt you.

4.7.7 Locking between SoftIRQs

Again, instances of the same SoftIRQ are run on seperate CPUs. Thus see section 4.7.6.

4.7.8 Locking between HardIRQs and SoftIRQs/Tasklets

Use `spin_[un]lock`($\cdot$), as SoftIRQs cannot interrupt IRQs. Only use the `irq` versions, if two seperate interrupt handlers share memory.

4.8 Reader-Writer Spinlock

Read and *write* operations are treated differentely:

On read, all write operations get locked. Concurrent reads are allowed. On write, all read operations get locked. Concurrent writes are disallowed.

They are implemented by spinlocks.

5 Testing

Important

In WS 2021/22, this chapter was not relevant for the exam.

6 Discrete Event Simulation

Until now, only design and implementation of protocols were discussed. Before a protocol is deployed to the internet, it has to be thoroughly tested. For this porpose, the system, for which the protocol was designed, is simulated.

6.1 Overview

Definition: Simulation

A computer simulation is a simulation, that runs on a single computer or a network of computers to reproduce behaviour of a system. This simulation uses a discrete an minimal model of the system to simulate.

6.1.1 Systems

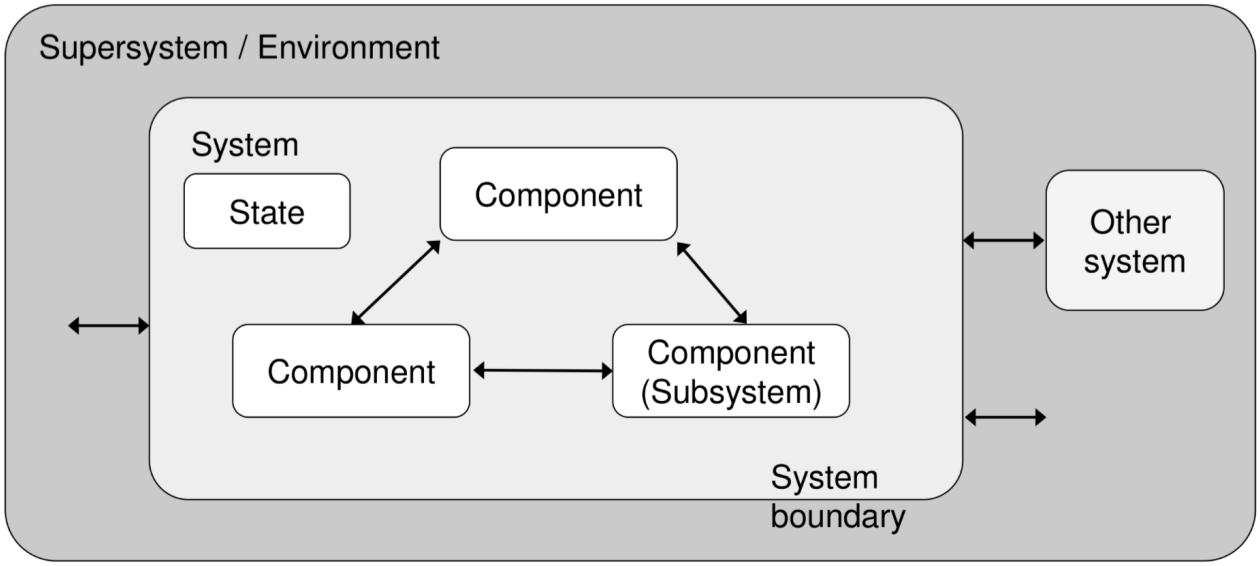

Figure 29: Depiction of a “system”

A *system state* is a set of variables with their corresponding values, which represent the state of a system at a discrete moment in time.

State of subsystems naturally are also part of the state of the whole system.

Continuous values are discretised.

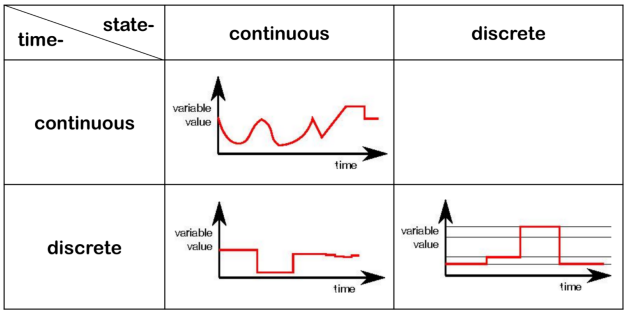

Figure 30: Possible ways a system can change state

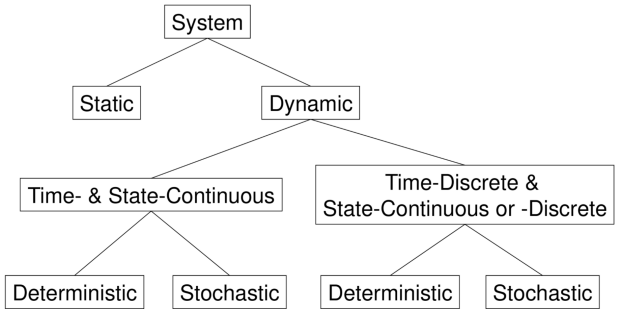

Figure 31: Possible system configurations

State changes can be *deterministic*, based on hard rules without randomness, or *stochastical*, where randomness can be used.

In simulations, stochastic, time-discrete/state-discrete state changes are used.

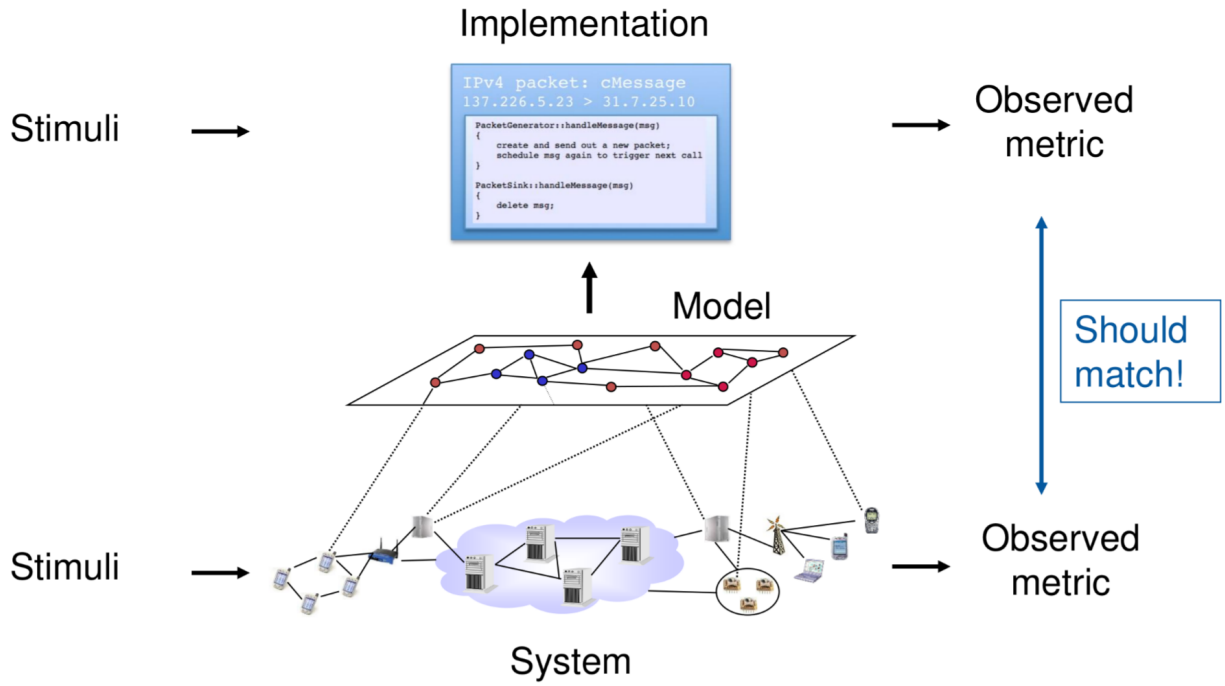

Figure 32: System to model relation

Equivalent *stimuli* have to yield the same *observed metric*.

6.2 Discrete Event Simulation

Discrete in this context means *time-discrete*, stochastic systems. This poses a problem, as stochastic systems don't necessarily yield the same results for the same input. However, as computers have no methods of generating real randomness, just use a *seeded* pseudo-random number generator.

Definition: Next-Event Time Advance Algorithm

We will simulate only the points in time, where the state of the model changes, all other times can be skipped. The time in the model is kept by a *simulation clock*. System time commonly starts at $t = 0$.

The time of occurence of the next event is known. All future events are stored in a list. This list might be pre-generated, or generated on the fly.

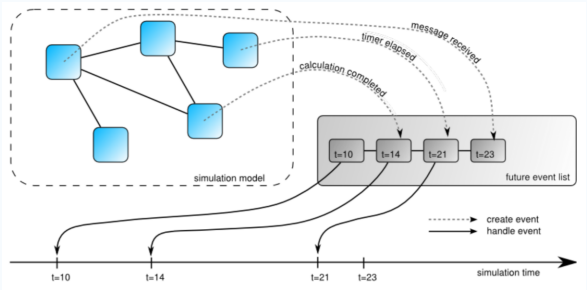

Figure 33: Depiction of the algorithm

Definition: Notions of Time

An important distinction to make is between the outside clock and the inside clock of a simulation. Notions of the times:

Inside

- Simulated time
- **Simulation time**

Outside

- **Simulation time**
- Simulation runtime
- Wall-clock time

Be careful with the term *simulation time*, as it is ambigous.

6.2.1 Framework

The protocol is executed on top of a *DES-Framework*, which exposes an API to the protocol.

Simulation Entities Objects, that are processed over the course of a run of the protocol.

Events Events can be seen as the following pseudo-`struct`:

```
struct event {
    time_t  occurance;
    fct_t   function;
    args[]  args_for_func;
};
```

where the `function` gets called upon invokation of the event at time `occurance`.

Definition: Future Event Set (FES)

The set, which holds all events, that are currently known and scheduled for future execution.

It is important to choose the correct data structure, to minimize the time overhead needed for finding the next event to process.

Simulation Clock Often *integers* are used, as there is no need to tell the computer how a real-world date format works. Maybe use the same approach as the `epoch`-time.

Init and Main Loop In the **init**-function, all objects are traversed and initialized if needed. In the **main**-function, do the following:

1. Pick next item from FES
2. Advance time to the `occurance` time of the event
3. Execute event handler `function` of event
4. Cleanup
5. Repeat, until no more events to handle

Teardown Stop the simulations `main` function. Print out statistics and so on. Cleanup.

6.3 Simulation Framework

There are some frameworks, that make running simulations easier, as they take away some hard tasks from you.

Simulation frameworks are not easy to scale. They are complex w.r.t structure and computation, if, e.g. radio propagation shall be simulated.

6.3.1 OMNeT++

Design your own model on top of this framework in `C++`. Is a "general purpose" simulation framework and not limited to communication networks.

It uses *hierarchically nested* modules. Structured like discussed above:

`init` → `event handling` → `teardown`.

Uses network *description language (NED)*. We can define *layers*, *channels* and *compound* modules. All have parameters, that describe, e.g. in case of layers, the bit error rate, speed and other defining parameters.

```
simple LinkLayer
{
  parameters:
    int bufferSize;
    double ARQtimer @unit( s);
    string address;
  gates:
    input from_upper_layer ;
    input from_lower_layer ;
    output to_upper_layer ;
    output to_lower_layer;
}
```

Figure 34: Example of *NED*

In addition to fig. 34, you have to implement event handling in `C++`.

Strenghts

- Clean separation topology definition $\leftrightarrow$ functional implementation
- Very general use-case

Weaknesses

- Networks are not simulated realistically

6.3.2 ns-3

Better suited for network protocol simulation. It allows for a mixup of simulation and real world, as it allows to integrate "real" programs to the simulation.

It implements Linux networking architecture and ISO/OSI reference models.

Applications are connected by channels. You are expected to use the protocol stack which is provided by the framework.

It is possible to use real-world nodes and challes, or simulate the whole system.

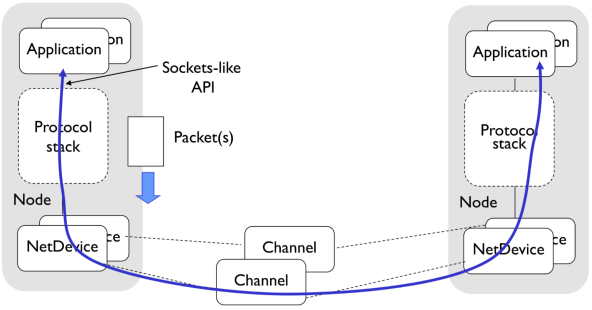

Figure 35: Data Flow Model of `ns-3`

Also follows the same structure as OMNet++. Can only be used for simulating networks.

6.4 Random Number Generators

The "stochastic" part of DES means, that there is a certain degree of randomness, that has to be taken account while simulating your protocol. It is important, to have reproducible randomness. This can be achieved by seeding a pseudo-random number generator, s.t. outputs for any input appear random but are reproducible.

Examples of random processes during network commnication:

- Transmitting over wireless link
- Pinging remote server
- ...

A simple methods of generating p.-r. numbers is the following.

Definition: Linear Congruential Generator

$$x_n = (ax_{n-1} + c) \bmod m$$

where a, c, m are fixed parameters. are fixed parameters. x_0 is the seed.

This is the easiest method to generate pseudo-random numbers.

It is also the shittiest method to do this. Don't use it!

This method is prone to starting to circle relatively quickly.

Don't implement RNGs yourself, someone else already did that for you way better than you could. Use true random numbers as seeds.

6.5 Parallel DES

There are two ways to distribute load of a simulation:

1. Inter-event –
2. Intra-event –

was tut das

was tut das

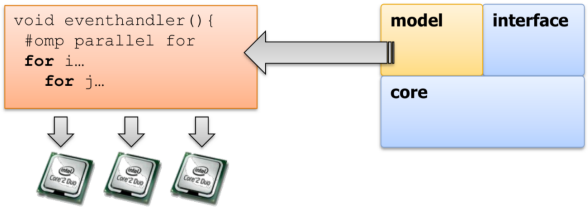

Figure 36: Intra-event parallelization

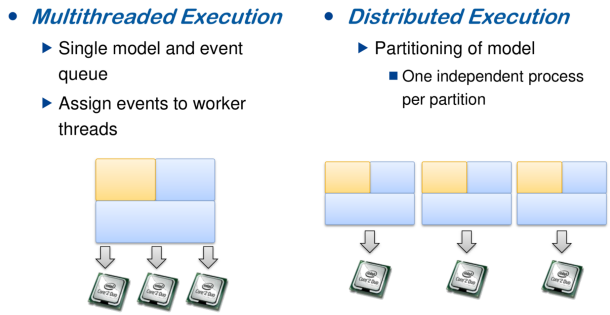

Figure 37: Inter-event parallelization

When using inter-event parallelization, synchronization of event execution is needed, as executing events in the future can block and/or change event handling in the past, as executing an event can possibly change states, that an event uses for execution.

Thus: need some way of checking if two events e_1, e_2 can be run concurrently.

6.5.1 Synchronization and Coordination Algorithms

There are two possible ways to partition events:

1. Space seperated simulation
2. Time parallel simulation

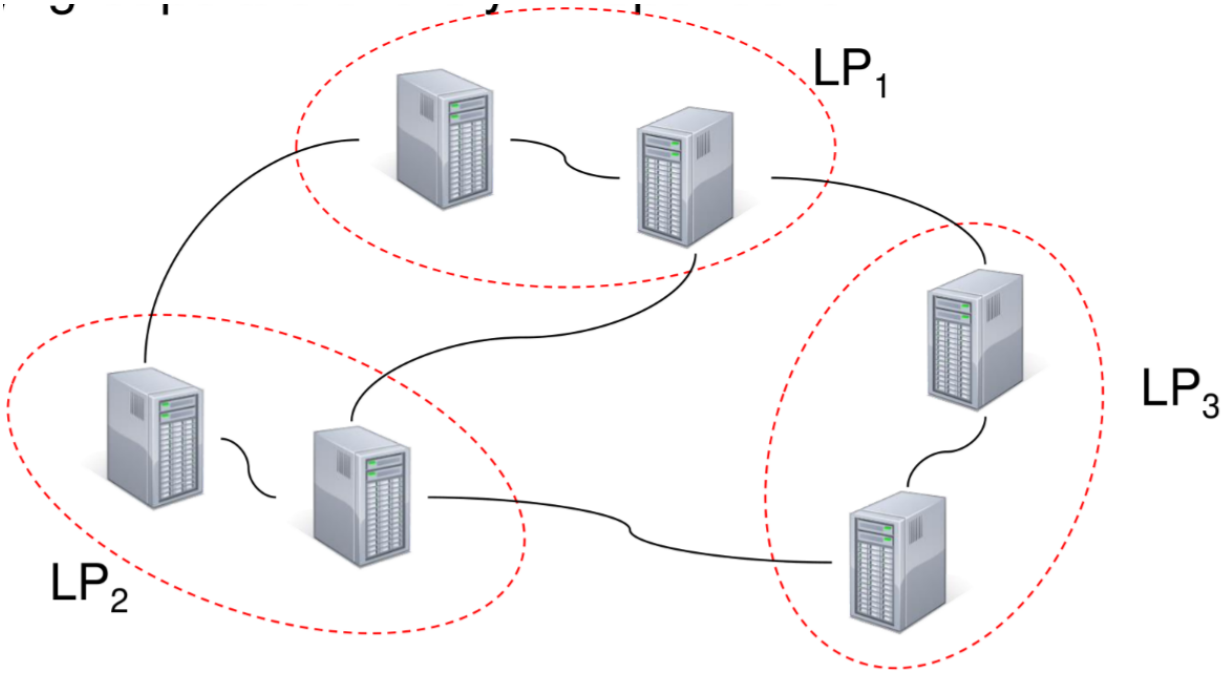

Figure 38: Space parallel simulation

Space Separated Simulations Split up the simulation into *Logical Processes*, that execute a specific partion of the simulation. If done right, this lead to very little communication overhead between the LPs, even to the degree of complete independance. Each LP has to guarantee the following:

Definition: Local Causality Constraint

Execute events in non-decreasing timestamp order.

As long as event order at the same timestamp does not matter, this guarantees correct simulations.

Conservative algorithms

- Strictly avoid causality errors
- Implement strategy to determine safety of execution of event

Optimistic algorithms

- Execute events always
- Detect errors
- Recover from errors

Conservative Algorithms Each LP holds timeinformation at runtime.

Lookahead (LA) – Delay in simulated time between LPs.
Packet e arrives at $t(e) + \texttt{LA}$.

Earliest Output Time (EOT) – Earliest simulated time, a neighbouring LP may receive a message
$t(e_{n+1}) + \texttt{LA} = \texttt{EOT}$

Earliest Input Time (EIT) – Earliest time, the LP might receive from another LP
$\min\{\texttt{EOT}_1, \ldots, \texttt{EOT}_n\}$

Deadlocks may occur. Each LP stores EOTs of neighbouring LPs. These EOTs are updated on advance of local time, and then distributed to all neighbouring LPs.

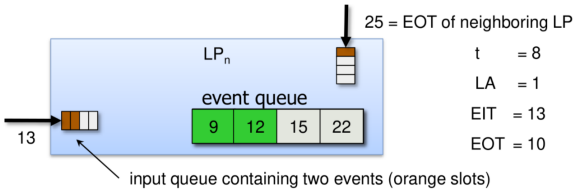

Figure 39: How to read nodes

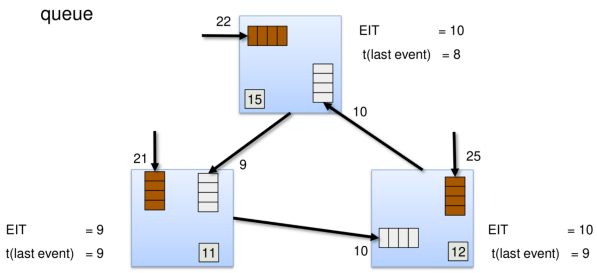

Figure 40: Depiction of a Deadlock

Each time in fig. 39 is a timestamp of earliest execution. In fig. 40, a deadlock is depicted, as every LP has to wait for something to happen (EIT is 10, 10 and 9), but each LP processes the next event at $t = 12, 15, 11$, such that no event occurs, because every node is waiting for the others to do something.

Resolve deadlocks by announcing a *null-message* upon blocking, which contains the earliest time of action for that specific LP, which then gets used to update the EOT of the next LP.

This may lead to slow incremental increase of time, if all LPs block and the next possible event lies far in the future $100s$.

Avoiding this is done by *Lower Bound on Timestamp (LBTS)*. This handles simulation in two phases:

1. Determine global barrier B: $B = \min\{t(e_{\text{next}}(P_i)) + LA(P_i)\}$
2. Parallel event simulation Process all local event with $t(e) \leq B$

Advantages

- No time creeping
- No deadlocks

Disadvantages

- Central info gathering is bottelneck
- Workload determined by most busy partition

Optimistic Algorithms As conservative algorithms often are too strict, resort to optimistic algorithms, which avoid unnecessary blocking, and is handling errors, if they occur.

To do this, optimistic algorithms save the state of all LPs regularly. On error, send a message to all LPs and roll back to correct state again.

	Conservative	Optimistic
Implementation complexity	simple	complex
Resources usage	low	high
Performance depends on	lookahead	similar workload

Figure 41: Comparisson conservative and optimistic algorithms

Overview