# Quiz 4 Recap

Dienstag, 30. Mai 2017 18:44

About Transport Layer, Congestion control and the TCP summary.

**TCP vs UDP**

TCP creates a connection and ensures ordered delivery and reliability.
UDP may lose, reorder or duplicate messages and has a limited message size, but is faster.
TCP is controlled by Congestion control from network state and by flow control from reciever state.
TCP does not keep boundaries from send() and recv().
TCP is bidirectional => add small ack package to other package for less overhead.
(TCP has urgent pointer field to skip queue)

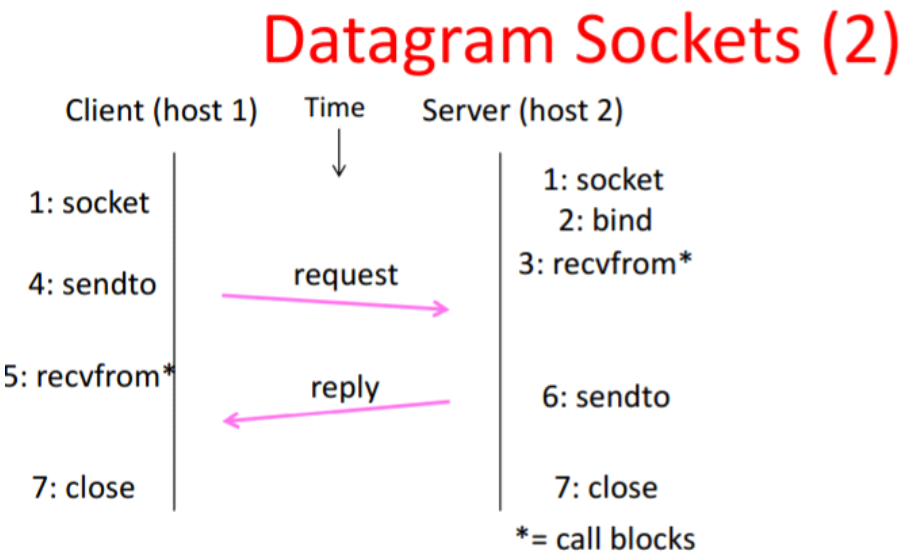
**Socket Commands**

socket - create a new communication endpoint
bind
For Streams:
- listen - announce willingness to accept connections
- accept - establish an incoming connection (passively)
- connect - actively attempt to establish a connection
- send
- recieve receive

For Datagrams:
- sendto
- receivefrom

close - release the socket

**udp**




buffers at port with message queues. Message length up to 64K (wat?)
Header contains:
- 16bit checksum.
- Source port, Dest port
- message length

The checksum is optional and covers UDP segment and IP pseudoheader. Value of zero means no checksum.
IP Pseudoheader contains source address, dest. address, protocol=17 and UDP length.

**TCP**

Should be robust against delayed duplicates.

client sends SYN(x), server replies with SYN(y) ACK(x+1)
Client replies with ACK(y+1)
If no Ack comes, syn is retransmitted.
If delayed duplicate arrives, we know it has already been here.

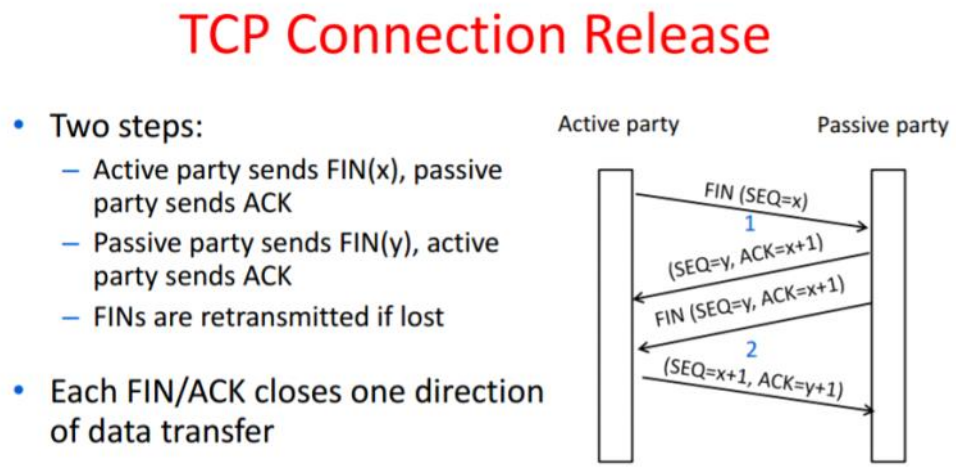
After that, the client directly sends data without waiting for a server ACK and keeps its ACK on y+1 for that next package with the same SEQ. After that, the SEQ is increased with each package.
Example:

```
Client -> SYN, SEQ=100
Client <- SYN, ACK, SEQ=700, ACK=101 <- Server
Client -> ACK = 701, SEQ=101 [50 Bytes of data]
Client -> ACK = 701 [Again, didn't receive sth from server yet], SEQ = 151
```

https://stackoverflow.com/questions/5551682/why-is-ack-1-and-not-2-in-first-tcp-request-after-connection-establishment

TCP allows simultaneous open: Both clients send a syn, acknowledge it and done.

Releasing connection:
client sends FIN, server sends ACK FIN, client sends ACK




**Sliding Windows**
Because Stop&wait ARQ only covers stop-and-wait, which is fine for LAN but not for network paths with big BD.
Allows W packets to be outstanding => can send W packets per RTT (=2D).
Gives more effiecient reliability (know when package was lost).
selective ACKS (SACK) give optionally hints for the reciever buffer state. Ack contains highest number expected next, so if a lower packet is recieved, there is an ack with the same number sent by the reciever (duplicates would be ignored instead).
ACK Clock.

Initial ARPANET used fixed sliding window size (e.g. 8 packages underway). But with more hosts in network, queues became full -> retransmissions clogged the network. => introducing congestion windows with TCP Tahoe/Reno.

**Congestion**
high traffic -> high loss -> more traffic -> more loss -> more delay
AdditiveIncreaseMultiplicativeDecrease converges over time to a fair and efficient allocation when hosts run it. MIAD, MIMD, AIAD do not.

**Feedback signals**
Packet loss, Packet delay, Router indication.
late but easy, early but guessing, early but need router support.

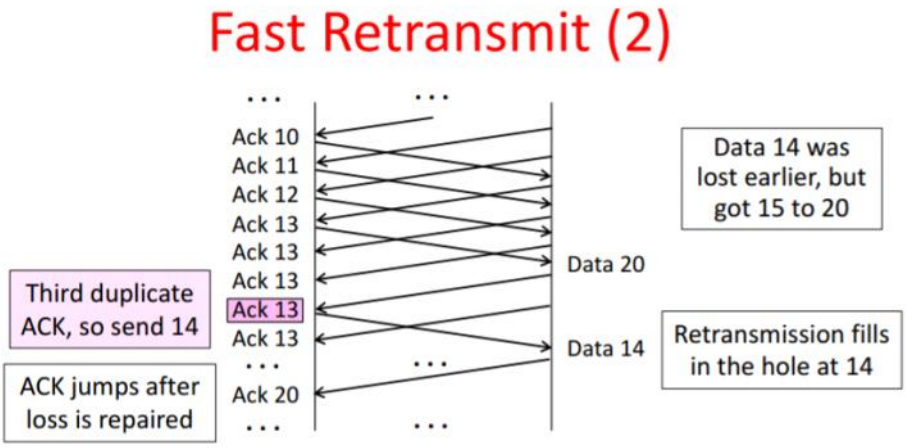
**AIMD** Example aimd

- TCP behaviors we will study:
  - ACK clocking
  - Adaptive timeout (mean and variance)
  - Slow-start
  - Fast Retransmission
  - Fast Recovery
- Together, they implement AIMD

Slow start is until a ssthresh-hold and from then on AdditiveIncrease.
Slow start: Increment the cwnd (congestion window) by 1 segment size for each ack. (doubling the packets)
After threshhold, increment only by 1 segment size every cwnd ACKs.
cwnd says how much data can be on its way at the time.
After loss, set ssthresh to half of cwnd. Halve cwnd (or completely restart, depending on whether it's Reno or Tahoe)

> When the congestion window exceeds the *ssthresh* threshold, the algorithm enters a new state, called congestion avoidance. In some implementations (*e.g.*, Linux), the initial ssthresh is large, and so the first slow start usually ends after a loss. However, ssthresh is updated at the end of each slow start, and will often affect subsequent slow starts triggered by timeouts. In congestion avoidance state as long as non-duplicate ACKs are received, the congestion window is additively increased by one MSS every round-trip time. When a packet is lost, the likelihood of duplicate ACKs being received is very high.[a]
>
> Aus <https://en.wikipedia.org/wiki/TCP_congestion_control>

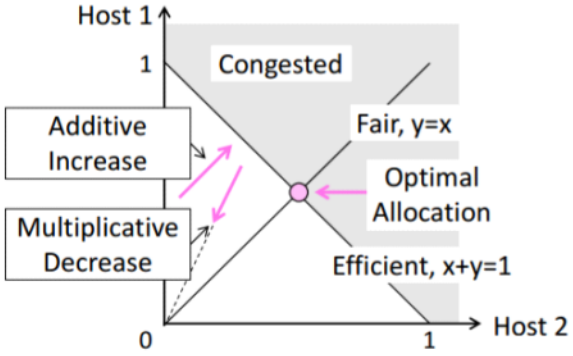
Fast retransmit: We know that some package arrived if we get an ack, but we only got ack for the highest number before the gap -> after three duplicate acks we infer that the gap was lost. ACK later jumps up after loss is repaired.




? But when the host is waiting for the ack to jump, we have a quiet time at sender and reciever and still need to MD cwnd. That seems to be because the timeount timer is reset on every send?
Fix is fast recovery.

Fast recovery:
Do fast retransmit. Then pretend further duplicate ACKs are the expected ACKs so if we sent data 14 again, then we continue with data 21 even though it has not yet been acked.

- AI and MD move the allocation




So We AI: both hosts send more. Then we MD: we decrease one host more than the other host. Thus, we get towards the middle. (if we were already to the right, we would decrese the other host more.

Adaptive Timeout is basically using the average round trip time to determine the timeout time.

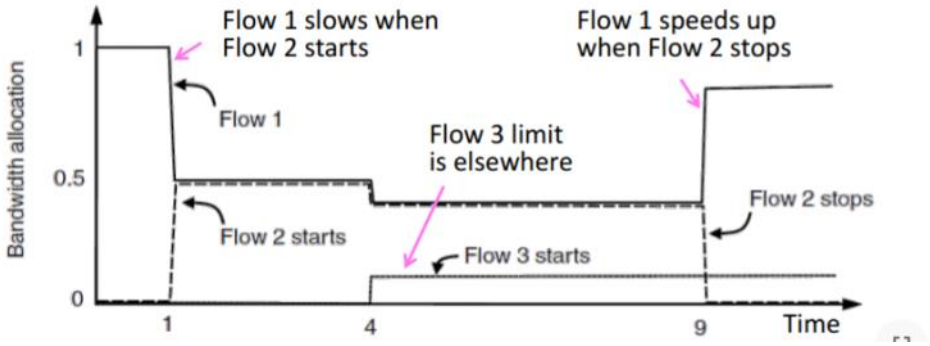
ACK clocking is part of sliding windows and useful if sender sends a burst of info. Because the next link only sends the packages when it got an ack.

## Interesting Questions

- How is MSS / MTU determined?
- What happens if UDP does not implement congestion control?
  - Do modern UDP applications need to implement congestion control?
  - What is the relationship with network neutrality?
- What if different congestion control schemes are used concurrently? What can go wrong?
- Can a malicious host obtain an unfair advantage?
- Why size would you pick for router buffers? Large or small? Which one will result in better performance if standard TCP is used?

**Max-Min fair allocation**
Maximize the minima. so every flow is equally big until one cannot grow any more. then only increase the others.

## Adapting over Time (2)




**Openloop vs Closedloop**
TCP is closed loop: it uses feedback to adjust rates.
Open would mean that it reserves bandwidth before use.
TCP is host-driven (vs network driven) because the host decides the amount it sends. And TCP is window-based instead of Rate based.

**TCP Tahoe**

## TCP Tahoe (Implementation)

- Initial slow-start (doubling) phase
  - Start with cwnd = 1 (or small value)
  - cwnd += 1 segment size per ACK
- Later Additive Increase phase
  - cwnd += 1/cwnd segments per ACK
  - Roughly adds 1 segment size per RTT
- Switching threshold (initially infinity)
  - Switch to AI when cwnd > ssthresh
  - Set ssthresh = cwnd/2 after loss
  - Begin with slow-start after timeout

So we set ssthresh to the largest that still works (because obviously cwnd was too big so doubling cwnd/2 would fail again.
Note that ssthresh is initially infinity, so we will fail at some point. Unless reciever stops us with sliding window.
*But isn't this slide reno?* Tahoe vs Reno vs SACK No. Tahoe also halves cwnd. The difference is that Tahoe then restarts at cwnd=1 MaxSegSize and Reno restarts at the new ssthresh (cwnd/2)

# Adaptive Timeout

Mittwoch, 31. Mai 2017 08:36

- TCP behaviors we will study:
  - ACK clocking
  - Adaptive timeout (mean and variance)
  - Slow-start
  - Fast Retransmission
  - Fast Recovery
- Together, they implement AIMD

***Adaptive Retransmission Based on Round-Trip Time Calculations***

It is for these reasons that TCP does not attempt to use a static, single number for its retransmission timers. Instead, TCP uses a dynamic, or *adaptive* retransmission scheme. TCP attempts to determine the approximate round-trip time between the devices, and adjusts it over time to compensate for increases or decreases in the average delay. The practical issues of how this is done are important, but are not covered in much detail in the main TCP standard. RFC 2988, *Computing TCP's Retransmission Timer*, discusses the issue extensively.

Round-trip times can "bounce" up and down, as we have seen, so we want to aim for an *average* RTT value for the connection. This average should respond to consistent movement up or down in the RTT without overreacting to a few very slow or fast acknowledgments. To allow this to happen, the RTT calculation uses a *smoothing* formula:

New RTT = ($\alpha$ * Old RTT) + ( (1-$\alpha$) * Newest RTT Measurement)

Where "$\alpha$" (alpha) is a *smoothing factor* between 0 and 1. Higher values of "$\alpha\forall$ (closer to 1) provide better smoothing and avoiding sudden changes as a result of one very fast or very slow RTT measurement. Conversely, this also slows down how quickly TCP reacts to more sustained changes in round-trip time. Lower values of alpha (closer to 0) make the RTT change more quickly in reaction to changes in measured RTT, but can cause "over-reaction" when RTTs fluctuate wildly.

***Acknowledgment Ambiguity***

Measuring the round-trip time between two devices is simple in concept: note the time that a segment is sent, note the time that an acknowledgment is received, and subtract the two. The measurement is more tricky in actual implementation, however. One of the main potential "gotchas" occurs when a segment is assumed lost and is retransmitted. The retransmitted segment carries nothing that distinguishes it from the original. When an acknowledgment is received for this segment, it's unclear as to whether this corresponds to the retransmission or the original segment. (Even though we decided the segment was lost and retransmitted it, it's possible the segment eventually got there, after taking a long time; or that the segment got their quickly but the ***acknowledgment*** took a long time!)

This is called *acknowledgment ambiguity*, and is not trivial to solve. We can't just decide to assume that an acknowledgment always goes with the oldest copy of the segment sent, because this makes the round-trip time appear too high. We also don't want to just assume an acknowledgment always goes with the latest sending of the segment, as that may artificially lower the average round-trip time.

Aus <http://www.tcpipguide.com/free/t_TCPAdaptiveRetransmissionandRetransmissionTimerCal-2.htm>

# Tahoe vs Reno vs SACK

Mittwoch, 31. Mai 2017 08:48

TCP Tahoe is the simplest one out of the four variants. It doesn't have fast recovery. At congestion avoidance phase, it treats the triple duplicate ACKs same as timeout. When timeout or triple duplicate ACKs is received, it will perform fast retransmit, reduce congestion window to 1, and enters slow-start phase.
TCP Reno differs from TCP Tahoe at congestion avoidance. When triple duplicate ACKs are received, it will halve the congestion window, perform a fast retransmit, and enters fast recovery. If a timeout event occurs, it will enter slow-start, same as TCP Tahoe. TCP Reno is effective to recover from a single packet loss, but it still suffers from performance problems when multiple packets are dropped from a window of data

Aus <http://www.roman10.net/2011/11/10/tcp-tahoe-reno-newreno-and-sacka-brief-comparison/>

So when we have a timeout, we restart the cwnd searching at 1 with slow-start.

TCP NewReno tries to improve the TCP Reno's performance when a burst of packets are lost by modifying the fast recovery algorithm. In TCP NewReno, a new data ACK is not enough to take TCP out of fast recovery to congestion avoidance. Instead it requires all the packets outstanding at the start of the fast recovery period are acknowledged.
TCP NewReno works by assuming that the packet that immediately follows the partial ACK received at fast recovery is lost, and retransmit the packet. However, this might not be true and it affects the performance of TCP. SACK TCP adds a number of SACK blocks in TCP packet, where each SACK block acknowledges a non-contiguous set of data has been received. The main difference between SACK TCP and Reno TCP implementations is in the behavior when multiple packets are dropped from one window of data. SACK sender maintains the information which packets is missed at receiver and only retransmits these packets. When all the outstanding packets at the start of fast recovery are acknowledged, SACK exits fast recovery and enters congestion avoidance.

Aus <http://www.roman10.net/2011/11/10/tcp-tahoe-reno-newreno-and-sacka-brief-comparison/>

**TCP Tahoe and Reno**[edit source]
The two algorithms were retrospectively named after the 4.3BSD operating system in which each first appeared (which were themselves named after Lake Tahoe and the nearby city of Reno, Nevada). The "Tahoe" algorithm first appeared in 4.3BSD-Tahoe (which was made to support the CCI Power 6/32 "Tahoe" minicomputer), and was made available to non-AT&T licensees as part of the "4.3BSD Networking Release 1"; this ensured its wide distribution and implementation. Improvements were made in 4.3BSD-Reno and subsequently released to the public as "Networking Release 2" and later 4.4BSD-Lite. While both consider retransmission timeout (RTO) and duplicate ACKs as packet loss events, the behavior of Tahoe and Reno differ primarily in how they react to duplicate ACKs:

- Tahoe: If three duplicate ACKs are received (i.e. four ACKs acknowledging the same packet, which are not piggybacked on data and do not change the receiver's advertised window), Tahoe performs a fast retransmit, sets the slow start threshold to half of the current congestion window, reduces the congestion window to 1 MSS, and resets to slow start state.[14]
- Reno: If three duplicate ACKs are received, Reno will perform a fast retransmit and skip the slow start phase by instead halving the congestion window (instead of setting it to 1 MSS like Tahoe), setting the slow start threshold equal to the new congestion window, and enter a phase called Fast Recovery.[14]

In both Tahoe and Reno, if an ACK times out (RTO timeout), slow start is used, and both algorithms reduce congestion window to 1 MSS.
Fast Recovery (Reno only): In this state, TCP retransmits the missing packet that was signaled by three

duplicate ACKs, and waits for an acknowledgment of the entire transmit window before returning to congestion avoidance. If there is no acknowledgment, TCP Reno experiences a timeout and enters the slow start state.

Aus <https://en.wikipedia.org/wiki/TCP_congestion_control#TCP_Tahoe_and_Reno>

**New Reno**

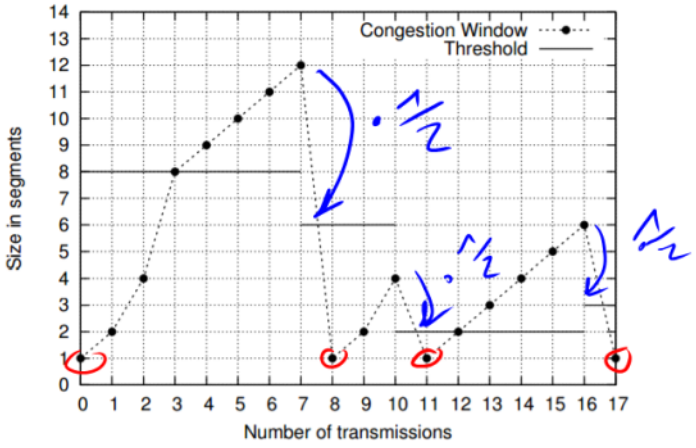
During fast recovery, for every duplicate ACK that is returned to TCP New Reno, a new unsent packet from the end of the congestion window is sent, to keep the transmit window full. For every ACK that makes partial progress in the sequence space, the sender assumes that the ACK points to a new hole, and the next packet beyond the ACKed sequence number is sent.
Because the timeout timer is reset whenever there is progress in the transmit buffer, this allows New Reno to fill large holes, or multiple holes, in the sequence space – much like TCP SACK. Because New Reno can send new packets at the end of the congestion window during fast recovery, high throughput is maintained during the hole-filling process, even when there are multiple holes, of multiple packets each.
When TCP enters fast recovery it records the highest outstanding unacknowledged packet sequence number. When this sequence number is acknowledged, TCP returns to the congestion avoidance state.
A problem occurs with New Reno when there are no packet losses but instead, packets are reordered by more than 3 packet sequence numbers. When this happens, New Reno mistakenly enters fast recovery, but when the reordered packet is delivered, ACK sequence-number progress occurs and from there until the end of fast recovery, every bit of sequence-number progress produces a duplicate and needless retransmission that is immediately ACKed.
New Reno performs as well as SACK at low packet error rates, and substantially outperforms Reno at high error rates.

Aus <https://en.wikipedia.org/wiki/TCP_congestion_control#TCP_Tahoe_and_Reno>

# Example aimd

Mittwoch, 31. Mai 2017 09:12

In the lectures, we saw two types of congestion control techniques: *Additive Increase/Multiplicative Decrease* and *Slow Start*. This question focuses on Slow Start. A TCP connection uses a threshold of 8KB for congestion control. The maximum segment size should be 1KB and the receiver's window is 16KB. After the 8th, the 11th, and the 17th transmission, timeouts are occurring, which are interpreted as network overload.




timeout happens longer time than multiple duplicate acks.
timer is reset whenever we get a package in order (with the next number. not the same)

# Recap Quiz1

Montag, 8. Mai 2017 08:56

**Transmission: Terms**
Data transmission rate: data/time
Transmission Delay: how long to put data on the cable
Propagation Delay: how long the first bit has to travel to reach the end of the cable
Bandwidth-Delay-Product: how much data is on the cable without being recieved yet at maximum.

**Nyquist Theorem**
Ein Signal mit Frequenz f kann dann exakt rekonstruiert werden, wenn es mit mindestens 2f Frequenz abgetastet wird. Anschaulich: Drehendes Rad im Stroboskop.

**Bit/Byte Stuffing**
Add Flag and Escape character. Flag tells start of frame. Esc Flag is needed if the Flag is part of the actual message. But then we'd need to escape the Esc character again. now if the esc character gets scrambled, we have a problem. So we XOR the Flag with e.g. 0x20 after escaping it so it wouldn't be a valid flag.

Bit/Byte Stuffing is actually just this to escape the flag:

Can stuff at the bit level too
- Call a flag six consecutive 1s
- On transmit, after five 1s in the data, insert a 0
- On receive, a 0 after five 1s is deleted

**Internet Checksum**
- [ ] IP-Checksum: Usually 16-bit words.

Encode: Sum all words up (4-bit words in excercise). Add remainder until no remainders left after adding. Take one's complement. => Checksum

Check: Add the checksum to the data and the results one's complement should be 0.

- [ ] **Hamming Code**

Enumerate the bits of the message-to-transmit from 1 to N. Every power-of-n-th bit is a parity bit. No parity bit includes other parity bits. We have n bits of data to actually send.

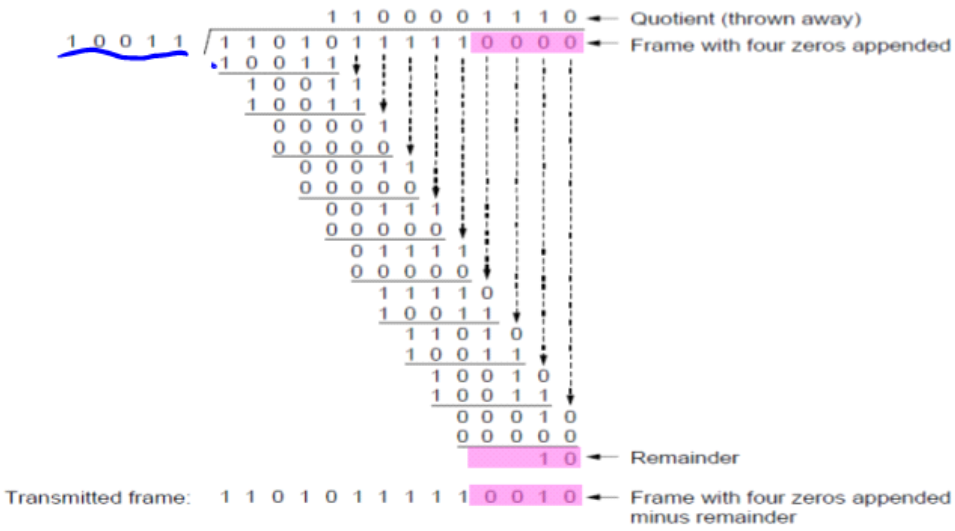
$N = n + k, \qquad N\ is\ at\ maximum = 2^k - 1$
$n = 2^k - k - 1\ at\ maximum$
If the message is shorter than this n, then just ignore the rest (set the larger bits to 0?)
The parity bits are calculated by adding their respective bits: They take all bits into account that have their i-th bit set. So the first parity bit considers the first, third, fifth and so on.

For checking, add the parity bit itself also. Order the results by their bit number, largest first. Then read it as a number to get which bit is wrong and flip it. Assumes that only one bit will be wrong.

- [ ] **CRC Checksum**

Only for recognizing random Errors, but it's easy to tamper with the data and find a way to make the checksum still match.
1. Predetermine generator polynome (in binary. highest potence of x first)
2. Take data in binary and zero-extend it on the righthand side with (r-1) zeroes. r is the length of the #bits of the generator polynome. (because the remainder is one digit smaller than the divisor)
3. Divide the extended data by the generator polynome. We are in a mod-2-arithmetic Ring so addition, subtraction are XOR. We start with the first common 1. we can usually ignore the quotient and justkeep the remainder. in the end, we set the (r-1) check bits to the remainder.

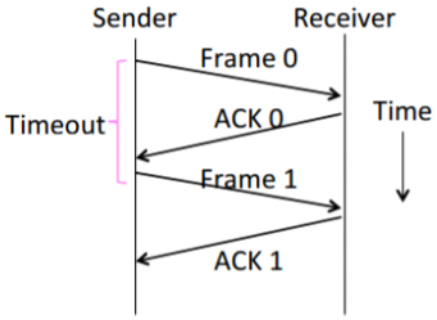
When the highest bit is 0, then we subtract 0 (XOR with 0 aka just continue) because that's how division works. (we'd usually add a digit to the quotient)




To recieve, we divide the recieved data and check whether the remainder is 0.

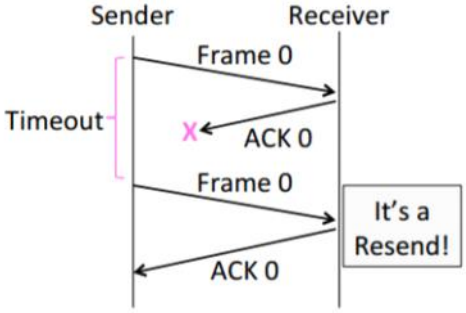
**Wired Communication: ARQ**
Automatic Repeat Request
if ACK goes lost, request again after a timeout.
In order for frames not to get confused, we add a bit to enumerate the frames.

- In the normal case:

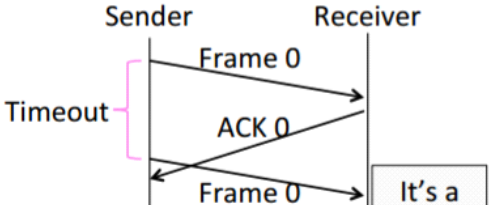



Stop-and-Wait (3)

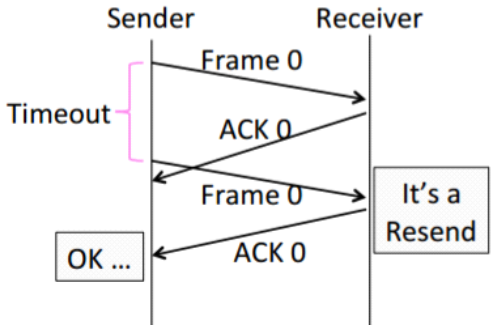
- With ACK loss:



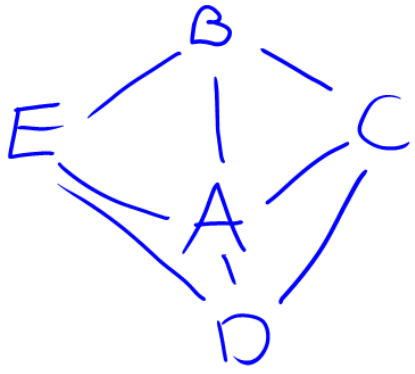

- With early timeout:



- With early timeout:




- [ ] **Wireless Communication**




b) B is sending to A
-> E and C cannot recieve because they get interference
-> A cannot recieve because it already gets a message
-> nobody can send

c) B is sending to C
-> A,B,C,E cannot recieve
-> A,B,C,D cannot send
=> E can send and D can recieve

delete the sender and its neighbours from the graph to find all possible recievers. Delete the reciever and all neighbours to find all possible senders. then check if the two are connected.

**MACA**

Terminals that think they cannot send are called exposed terminals.

If you hear a RTS but never a CTS then you are an exposed terminal and can figure this out.

If a node hears RTS but not CTS, then it might still conclude after carrier sense that the medium is used. However, it can conclude that it is an exposed node if it does not recieve CTS.

**CSMA/CA with RTS/CTS is synonymous to MACA**
Uses RTS, CTS but also ACK
Network Allocation Vector: counts down until it can send again (estimated time from virtual sensing)

- [ ] **CSMA/CA without RTS/CTS**
https://de.wikipedia.org/wiki/Carrier_Sense_Multiple_Access/Collision_Avoidance

Möchte ein Gerät Daten nach dem CSMA/CA-Verfahren versenden, so ist unter anderem folgender Ablauf möglich:

1. Zuerst wird das Medium abgehört („horcht“, „*Carrier Sense*“).
2. Ist das Medium für die Dauer eines DIFS frei, wird eine Backoffzeit aus dem Contention Window ausgewürfelt und nach Ablauf dieser gesendet.
3. Ist das Medium belegt, wird der Backoff bis zum Ablauf des Network Allocation Vectors (NAV) gestoppt, bevor er nach einem weiteren DIFS entsprechend weiter läuft.
4. Nach vollständigem Empfang des Paketes wartet der Empfänger ein SIFS, bevor das ACK gesendet wird.
5. Eine Kollision durch gleichzeitigen Ablauf des Backoffs führt zu einem ACK-Timeout – nach welchem ein EIFS gewartet wird bevor sich der gesamte Vorgang wiederholen kann ( DIFS → BO .. ).

Wird ein CTS oder RTS gehört ist das medium belegt und die Node die es hört somit am NAV zählen.

**CSMA/CD**
Contention Slot: How long to wait until we know a collision happened: Time for a signal to go there and the ACK to come back.
=> CSMA/CD has a contention slot of 2T

**Spanning Tree Algorithm (Link Layer, Switches)**
A network with LAN nodes connected by switches. Goal: create a minimal spanning tree that includes all LAN nodes.
The switch with the lowest address is the root. We grow the tree using the lowest distances from the root. (Actually, the nodes constantly choose the closest root of the multiple small trees that might exist).
Then we turn off ports for forwarding if they are not on the spanning tree. (Port = connection from LAN to LAN through switch)

**Network Layer**
Because Switches don't work across more than one link layer technology. Need to do translational stuff. IP Protocol belongs to this too.

Datagram: connectionless service (IP)
Packets contain a destination address. Router redirects this, maybe on a different path each time.

Virtual Circuits: constant connection going back and forth
Setting up circuit, transferring data, then deleting circuit. statistical sharing of Links. Packets contain a short label to identify the circuit instead of a long globally valid address. Each router has a table with existing circuits

## Datagrams vs Virtual Circuits

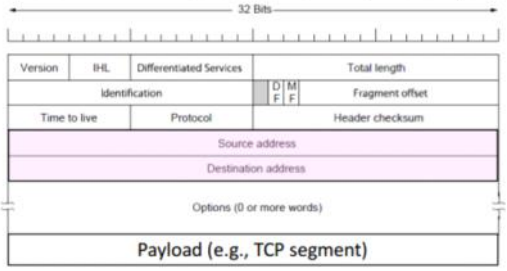
- Complementary strengths

| Issue | Datagrams | Virtual Circuits |
|---|---|---|
| Setup phase | Not needed | Required |
| Router state | Per destination | Per connection |
| Addresses | Packet carries full address | Packet carries short label |
| Routing | Per packet | Per circuit |
| Failures | Easier to mask | Difficult to mask |
| Quality of service | Difficult to add | Easier to add |

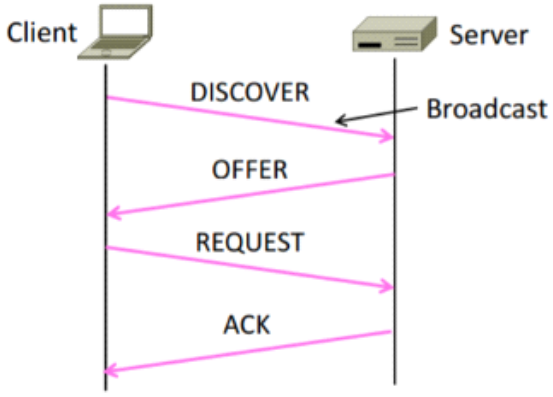
**IP, DHCP, ARP**

# IP (Internet Protocol)

- Network layer of the Internet, uses datagrams (next)
  - IPv4 carries 32 bit addresses on each packet (often 1.5 KB)



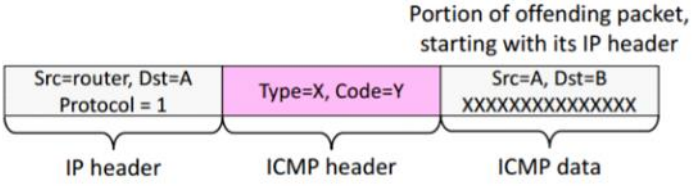

IP has the advantage that a more specific ip overrides a less specific one.
Get IP via DHCP (Dynamic Host Configuration Protocol) by broadcasting to all nodes on the network (Broadcast address 255.255.255.255)




To renew an existing lease, just use REQUEST, folowed by ACK

To get the addresses of destination links, use ARP (Address Resolution Protocol): just broadcast that the node with IP xxx.xxx.xxx.xxx should please say who it is. ARP sits on top of Link Layer

**ICMP (Internet Control Message Protocol)**
sits on top of IP. If there's a problem, returns a package back with info and discards the problematic package.




| Name | Type / Code | Usage |
|---|---|---|
| Dest. Unreachable (Net or Host) | 3 / 0 or 1 | Lack of connectivity |
| Dest. Unreachable (Fragment) | 3 / 4 | Path MTU Discovery |
| Time Exceeded (Transit) | 11 / 0 | Traceroute |
| Echo Request or Reply | 8 or 0 / 0 | Ping |

**IPv6 and NAT**
Problem: Need to map multiple shorter IPv4 internally to one external ip. => use ports.
This breaks Connectivity:

Can only send incoming packets after an outgoing connection is set up (and the table is set in the NAT)
Difficult to run a Server or P2P app behind a NAT

Breaks e.g. FTP which relies on sending the IP address and using random ports.

# Lernen Recap

Dienstag, 13. Juni 2017 09:24

**Kernel**
minimally provides basic scheduling, message passing and protection (paging)

**fork**
returns PID of the child to the parent
returns 0 to the child
wait using waitpid()
waitpid(pid, &status, 0);//0 for blocking
If not waiting, child will live longer as a zombie because maybe parent will at some point want to know its status.
waitpid returns 0 as long as there are children to wait on. use pid -1 to wait for any child.
If the parent exits before the child, the proccess will get PID 1 as a new parent, which will reap the exit code.

**exec**
never returns to the caller unless an error has returned. overwrites the whole process.

**popen**(command, type)**, pclose**
opens a process and a pipe to it. command is a /bin/sh command.
pclose waits for this proccess to terminate and returns the exit status.

**pipe**
takes as argument an array with two ints. Returns two file descriptors if successful. Usually, you then fork and each close one descriptor. fd[0] is set up for reading, fd[1] is set up for writing.

**Scheduling**
*Turnaround time* is from the first time a job is scheduled to when the job finishes.
*Waiting time for a specific job* is the time spent during which the job was not running since its queuing up until it finished.
*Average waiting time for all jobs* is the waiting times added up and divided by the number of jobs.
*Response Time* is the time it takes to react to user input.

> for RR, this is the time between the tasks turns at worst
> for EDF, this is the time between the entry time and the (deadline - execution time), because if it were larger, the job would not meet its deadline.

*preemptive Scheduling* is dispatching processes without warning, while non-preemptive scheduling waits for them to yield.

<u>RR</u>
every x ms, the next job's turn. If a job enters the queue, it will be the next to get scheduled.
Higher Turnaround-time than SJF but better response

<u>Shortest Remaining Time First</u>
If a new job enters, that is shorter, then execute that one. Else, do one job after the other, prioritizing the shortest.

<u>First Come First Served</u>

<u>Rate-monotonic scheduling</u>
Schedule periodic tasks by always running the task with the shortest period first. Period means at least every p hours has to be run.

☐ There is a feasible schedule if $\sum \frac{executiontime}{taskperiod} \leq n * \left(2^{\frac{1}{n}} - 1\right)$
assuming context switch time 0.
The period is independent of when the job is run

☐ Earliest Deadline First
Does the whole job at once. The one with the first deadline comes first.
Tasks don't have to be periodic. Complex for scheduling decisions O(n)
There is a feasible schedule if $\sum \frac{executiontime}{taskperiod} \leq 1$.
EDF is a real-time scheduling strategy with jobs that have priority.
(Contrast: RR is a dynamic scheduling strategy without priorities)j

**Real-Time Scheduling**
Usually not dynamic, because you cannot guarantee then that a correct schedule is feasible. New jobs might fuck shit up.

**Priority Inversion**
When a resource is blocked by a low priority thread, and the thread is preempted by a medium priority thread, then the high priority thread still cannot access the resource, so medium executes and high does not.

**Working Set Model**
Timer interrupts after every 5000 time units. Keep 2 bits for each page in memory. Every time the timer interrupts, shift the bits to the left and set the additional "reference bit" to 0. If page has to be evicted, check if reference bit is 0, if yes check time bits, if no evict other page. time bits say the page is still in the working set, if any of the bits is 1, which means that after ca 10000 time units being unused, the page is no more in the working set.
This is not completely accurate because if a page is referenced directly before the time hits it will live 10000 but if it is referenced directly after, it will live 15000. Improvement: smaller timestep.

**Kernel Threads vs Userlevel Threads**
Userlevel Threads are cheap to create and destroy, fast to context switch, but can block entire process.
One-to-one threads are easier to schedule and nicely handle blocking. The OS knows there what's going on.

Kernel-level threads require a context switch, involving changing processor regs that define the current memory map, permissions and the cache.

**Segmentation**
Base address + offset, but checks if offset is below some limit. Like paging, but does not cause internal fragmentation (Paging always uses 4kiB pages). suffers from external fragmentation though.
Fast Translation.
Uses:
- access same library from multiple processes
- process isolation
- interprocess communication

How:
- page fault handler duplicates page when read from child process

**other uses of address translation**
Process Isolation
Shared code segments
InterProcessCommunication
Program initialization
Efficient dynamic memory allocation

Cache management
Program debugging

**Paging: Second Chance**
approximates LRU by giving every page a bit that is set if the page was referenced. we clear the bit instead if we would remove the page and then look for another page (FIFO).

**Virtual Paging**
Last 12 bits are used for offset. first 20 are split into two 10-bit addresses. first one points to dictionary table entry, which points to page table. second points to page table entry.

So if we have 32bit addresses and the entries are 4B in size including optional additional flags, then the whole page table (if it is a direct one without dictionary) has size $\frac{2^{32}}{2^{12}} * 4B = 2^{22}B = 2^2MB = 4MB$

**optimal paging**
If known what pages will be accessed
always replace with the one that will not be used for the longest time

**Why Memory Management**
Providing processes with contiguous address space
sharing resources
minimizing fragmentation
giving or denying access to data
provide more virtual memory than actually available

**ACL** vs **Capabilities**
Access Control List (Row-wise) stores a list at each file with rights as rows and principals as columns. Not scaling well with principals.
Capabilities (column-wise) stores a list globally which contains for each principal->file the rights. Not scaling well with files.
POSIX: ACL with only three principals. owner, group, everyone.

**Directories**
Linear list (filename, block pointer) tuples
    simple, slow lookup
Hash Table with closed hashing
    collisions
    fast name lookup
    fixed size
B-Tree
    increasingly common
    Complex to maintain but scales well

**Files**
<u>Byte-sequence</u>
can be truncated, updated in place, and appended to
usually no insert

<u>Random access</u>
support read, write, seek, tell
    tell returns current index, seek gives absolute or relative to current position
Index units: for byte sequence files, offset in bytes

<u>Memory mapped files</u>
uses Virtual Addressing system to cache files. Map file content into virtual address space, set the

backing store of region to file. Can now access the file using load/store
Updates go back to file instead of swap space if memory is paged out.

**Filesystem implementations**

FAT

Linked list with blocks. Free space management with a FAT array.
The linked list itself is the Fast Access Table. lose it and you lost.
Locality: poor -> Defragmentation
Slow random access because linked list.

FFS

Fast File System. Has a fixed, asymmetric tree of blocks and a fixed bitmap for free space management.
Locality: Block groups, Reserves space

Inode contains metadata, data blocks and indirect blocks. All blocks are 4kB.
Example: 12-block file is stored in inode because 12*4=48 kB
13-block file has one indirect pointer because 13*4=52 kB is too large to fit into the inode
Total usually 12 direct blocks, one indirect, one doubly indirect and one triply indirect in inode.

NTFS

Dynamic tree. Extent granularity
Free space bitmap in file
Locality: Best fit, Defragmentation

MasterFileTable contains small files directly, else pointers.
Also contains filename and hard links.
Pointers point to data extents
attribute list can also be stored in extents, or in a second MFT entry.
File System Metadata is also held in files. MFT is at the start of the volume.

ZFS

Dynamic
COW tree (Copy on Write)
Index granularity: Block
Free space: Log-structured space map
Locality: Write anywhere, block groups (see book for details)

**Polling vs interrupts vs DMA**

polling has low latency but blocks the proccessor. So if we need to wait anyways, why not poll. Else yield and wait for an interrupt, allowing an other process to run.
DMA allows to transfer large amounts of data without involving the CPU but requires the page to be pinned (locked in RAM and always at the same place)

**Device Types**

| **Block Devices** | **Character Devices** |
|---|---|
| Deals with blocks of data<br>Looks like files -> can seek/map<br>Mountable -> Filesystem implemented above devices<br>allow random access | Unstructered I/O, byte-stream interface<br>E.g. Keyboard, Mouse |

Block devices (also called block special files) usually behave a lot like ordinary files: they are an array of bytes, and the value that is read at a given location is the value that was last written there. Data from block device can be cached in memory and read back from cache; writes can be buffered. Block devices are normally seekable (i.e. there is a notion of position inside the file which the application can change). The name “block device” comes from the fact that the corresponding hardware typically reads and writes a whole block at a time (e.g. a sector on a hard disk).

Character devices (also called character special files) behave like pipes, serial ports, etc. Writing or reading to them is an immediate action. What the driver does with the data is its own business. Writing a byte to a character device might cause it to be displayed on screen, output on a serial port, converted into a sound, ... Reading a byte from a device might cause the serial port to wait for input, might return a random byte (/dev/urandom), ... The name “character device” comes from the fact that each character is handled individually.

**Device Driver**
Hardware is interrupt driven, Applications are blocking, considerable processing in between.
e.g. TCP/IP processing, retries, file system processing, locking, etc.

Interrupt handler can't take too long (interrupts disabled during handling)
Cant change much: arbitrary system state
Process waits (in single core setting) because CPU is doing interrupt handling. We don't know until demultiplexing to which process the interrupt goes.

Solution1 : Driver Threads.
Interrupt Handler masks interrupt, does minimal processing and unblocks driver thread. Driver Thread performs all packet processing neccessary, unblocks user processes, unmasks interrupt when finished.
User process in kernel does per-process handling, copies packet to user space and returns from kernel.

Solution2 : Deferred procedure calls (DPC)
FirstLevelInterruptHandler FLIH enqueues DPC and then the next user process runs all pending DPCs when unblocked before leaving the kernel and running its own code.
This is done in most Unix versions as it does not need kernel threads and saves a context switch.
Cannot account processing time to the right process.

Solution3: Demux early, run in user space

Kernel offers driver a device. it claims it if it can handle it.
Major device number: Class of device, e.g. disk, CR-ROM, keyboard, …
Minor device number: specific device

**Deferred Procedure Calls**
are also known as 2nd level interrupt handlers, soft/slow interrupt handlers and in linux (ONLY) bottom-half handlers.

In non-linux OS: Bottom half = FLIH + SLIH, called from below
Top half = Called from user space (syscalls etc.)

NAPI switches between "each packet interrupts CPU", "CPU polls driver"

**IOMMU**
Memory is protected from malicious devices - a device can only access what has been mapped.
Device can access memory even if it does not support the whole address range
Virtual Adressing & Paging
Virtualized guest OS can use devices that do not explicitely support virtualisation.

Stand: Next: OS quizes and NET everything
To Look at:
Signals, Interrupts
File Systems (especially NTFS), Acces Control Lists

Link, Unlink, Rename

**TCP ACKs**
SEQ = x
ACK = x + length(x)
SEQ = ack
ACK = ack + length(ack)
…

- [ ] not 1-steps, but length steps.
SEQ + length always < ACK + Window Size

---

**Shadow paging**

Shadow page tables are used by the hypervisor to keep track of the state in which the guest "thinks" its page tables should be. The guest can't be allowed access to the hardware page tables because then it would essentially have control of the machine. So, the hypervisor keeps the "real" mappings (guest virtual -> host physical) in the hardware when the relevant guest is executing, and keeps a representation of the page tables that the guest thinks it's using "in the shadows," or at least that's how I like to think about it. Notice that this avoids the GVA->GPA translation step.
As far as page faults go, nothing changes from the *hardware's* point of view (remember, the hypervisor makes it so the page tables used by the hardware contain GVA->HPA mappings), a page fault will simply generate an exception and redirect to the appropriate exception handler. However, when a page fault occurs while a VM is running, this exception can be "forwarded" to the hypervisor, which can then handle it appropriately. The hypervisor must build up these shadow page tables as it sees page faults generated by the guest. When the guest writes a mapping into one of its page tables, the hypervisor won't know right away, so the shadow page tables won't instantly "be in sync" with what the guest intends. So the hypervisor will build up the shadow page tables in, e.g., the following way:

- Guest writes a mapping for VA `0xdeadbeef` into it's page tables (a location in memory), but remember, this mapping isn't being used by the hardware.
- Guest accesses `0xdeadbeef`, which causes a page fault because the real page tables haven't been updated to add the mapping
- Page fault is forwarded to hypervisor
- Hypervisor looks at guest page tables and notices they're different from shadow page tables, says "hey, I haven't created a real mapping for `0xdeadbeef` yet"
- So it updates its shadow page tables and creates a corresponding `0xdeadbeef`->HPA mapping for the hardware to use.

The previous case is called a *shadow page fault* because it is caused solely by the introduction of memory virtualization. So the handling of the page fault will stop at the hypervisor and the guest OS will have no idea that it even occurred. Note that the guest can also generate genuine page faults because of mappings it hasn't tried to create yet, and the hypervisor will forward these back up into the guest. Also realize that this entire process implies that *every* page fault that occurs while the guest is executing must cause an exit to the VMM so the shadow page tables can be kept fresh. This is expensive, and one of the reasons why hardware support was introduced for memory virtualization. (here is one quick intro to nested, or extended page tables)
A good reference for this is this book

Aus <https://stackoverflow.com/questions/9832140/what-exactly-do-shadow-page-tables-for-vmms-do>

# DV/LS Comparison

| Goal | Distance Vector | Link-State |
|---|---|---|
| Correctness | Distributed Bellman-Ford | Replicated Dijkstra |
| Efficient paths | Approx. with shortest paths | Approx. with shortest paths |
| Fair paths | Approx. with shortest paths | Approx. with shortest paths |
| Fast convergence | Slow – many exchanges | Fast – flood and compute |
| Scalability | Excellent: storage/compute | Moderate: storage/compute |

# retarded questions

Freitag, 16. Juni 2017 06:52

| ☐ Identify the seven layers of the OSI model in order from Layer 1. |
| --- |
| 7 Application Layer<br>6 Presentation Layer<br>5 Session Layer<br>4 Transport Layer<br>3 Network Layer<br>2 Data Link Layer<br>1 Physical Layer |

| Name two types of ARQ that were discussed in class |
| --- |
| Um sicher zu stellen, dass die gesendeten Pakete angekommen sind.<br>**Stop and Wait**: Nach jedem gesendeten Paket auf eine Bestätigung warten und erst dann das nächste Paket senden. Benutzt ein Bit als sequence number.<br>**Sliding Window**: Verallgemeinerung von Stop-and-Wait: Benutzt *W* Bits als sequence number. Erlaubt dadurch $2^w$ unterscheidbare Packete.<br>**Go-Back-N**: Nach N Paketen auf Bestätigung warten. Bei einem Fehler (keine Bestätigung nach bestimmter Zeit) werden die letzten N Pakete nochmals gesendet.<br>**Selective Repeat**: Bei einem Fehler werden nur die Pakete nochmals gesendet, für die es keine Bestätigung gab. |

| Difference Hub, Switch? |
| --- |
| Ein Hub arbeitet auf dem Physical Layer, ein Switch jedoch auf dem Link Layer (teilweise auch Network Layer). (Siehe Slide 91) [1]<br>Beide verbinden Netzwerkteile, jedoch broadcasted der Hub einen empfangenen Frame auf alle Ports und somit wird auch bei allen Ports Bandbreite gebraucht. Der Switch hingegen merkt sich die MAC-Adressen der an den Port angeschlossenen Geräte und kann somit einen empfangenen Frame an den richtigen Port weiterleiten |

| BGP Announcements |
| --- |
| Announce to both providers and peers. Prefer Customers when directing traffic. |

| Give two reasons why having more virtual memory than physical memory is useful |
| --- |
| • Copy-on-Write: Reuse the same spot for multiple applications if it's read-only<br>• Paging: Can page memory out to disk (e.g. Linux-Swap)<br>• Devices: Can use virtual memory to point to other hardware |

| Which hardware feature is required for preemptive multitasking, but not for cooperative multitasking? |
| --- |
| hardware timer |

| |
|---|
| Give an upper bound for the response time under earliest-deadline-first scheduling. Assume that all deadlines are met |
| Deadline - Entry Time |