Hi I am Jörg Thalheim and I will present you today, which findings I had while evaluating TCP's Initial Congestion Window under different network conditions. I will structure this talk in the following way:

- After I explained the task,
- I will give a short recap, how TCP Congestion Control works in general
- and which role the initial TCP congestion window plays in this context.
- After that I will present the test setup I have chosen
- and what throughput I have measured in this simulation.
- Finally I will conclude, what I learned
- and how applicative this findings are for bigger dynamic systems.
- The current RFC from 2002 suggestion for TCP suggest a congestion window of about 4KB of user data.
- However large-scale experiments by Google have shown that an increase to at least 10 segments improve the end-to-end latency without scarifying network stability
- as an result a new RFC was proposed
- My task was to test the throughput of TCP with different initial congestion windows under various network conditions.
- But first I will recap how TCP Congestion Control works in general.
- TCP Congestion Control mechanism are important to scale TCP/IP in the internet
- It allows to adapt the TCP window to the current network condition without overloading it
- Part of these algorithms is Slow Start
- It limits the number of bytes TCP will send before waiting for an acknowledgement
- on every acknowledged packet it will increase congestion windows size by one segment until either package loss is detected or the segment size threshold is reached.
- in case of package loss, which could be indicated by receiving a duplicate Acknowledgement 3 times, TCP will trigger a Fast Retransmit
- if the segment size threshold is reached TCP will perform Congestion Avoidance algorithm
- instead of exponential grows the window will increase additive
- In practice there several TCP congestion algorithms with different emphasis

- In this graph from wireshark, we see a single HTTP download over a 10Mbit link using different TCP congestion avoidance algorithm.
- Reno:
- a classical TCP congestion algorithm, half the window on reset
- Cubic:
- Current standard in Linux, less aggressive and optimized for long pipes
- Westwood:
- newer version of Reno, very efficient
- Vegas:
- Focus on wireless networks and network delay
- tries to detect early congestion based on RTT
- what they have in common is the TCP slow start part at the benign of a connection
- Google claims that most web response sizes already finishes within the TCP slow start.
- So that TCP cannot build up the full possible window and use all the available bandwidth
- we see that a lot the web objects are below 20kb, which is less then the average what have seen as a stable TCP congestion window for 10mbit/s link with a rather lower Round Trip time
- on modern internet 100mbit is not uncommon any more and also the round trip time is usually higher.
- to speed up TCP anyway a technique used by browser vendors,
- is to open multiple TCP connections at once
- and request resources in parallel
- However this has the disadvantage,
- that now multiple connection compete for the same bandwidth
- and we the overhead of TCP and TCP slow start adds up
- a second problem are the so called Elephant Flow
- these are long and continuous TCP flows,
- they get an unfair amount of the available bandwidth compared to short Requests, because their congestion window has grown to the maximum
- If we look at the math,
- we get the throughput of TCP by multiplying the congestion window size by the MTU divided by the round trip time

- under the assumption that no segment get lost during this round trip
- If we take the initial window size as input size and transfer a file of the size S, we get the following equation for the overall transfer time.
- in this equation, we see that in theory if we increase the initial window size, we get a smaller transfer time
- To test this this assumption I choose the following setup
- The evaluation was done by using Mininet
- This is a network virtualisation framework
- It allows processes to run separate networks in arbitrary topologies
- Because of missing data/resources, I was choosing a rather simple setup:
- the assumption I made by this model is, that the uplink, which connects a client to the internet is almost always the bottleneck
- I created with mininet 2 network namespaces, which are connected over a software bridge
- In one container I was running curl, which act as my HTTP client
- In the other namespace the web server Nginx was running, serving static files
- I made multiple measurements with various parameter:
- set the initial window size using the route attribute `initcwnd`
- On the linked website I have uploaded all graphs
- in the following, will show some representative graphs
- limit the forwarding delay and bandwidth with traffic control
- and use different files sizes for HTTP
- on X-Axis: are the initial Window sizes depicted
- on Y-Axis: we see the Transfer-Time relative to an initial window of 3
- from up to down: the request size increase
- left to right: the delay increases
- we see bigger gains if the latency gets higher
- this is because we save more RTTs on this network
- the gain decreases with bigger file sizes
- this is because the TCP window can fully build up with bigger file sizes
- the gain ranges from no improvement to 60%
- for higher bandwith we also see bigger speed ups for bigger file sizes