Aggressive TCP Slow Start

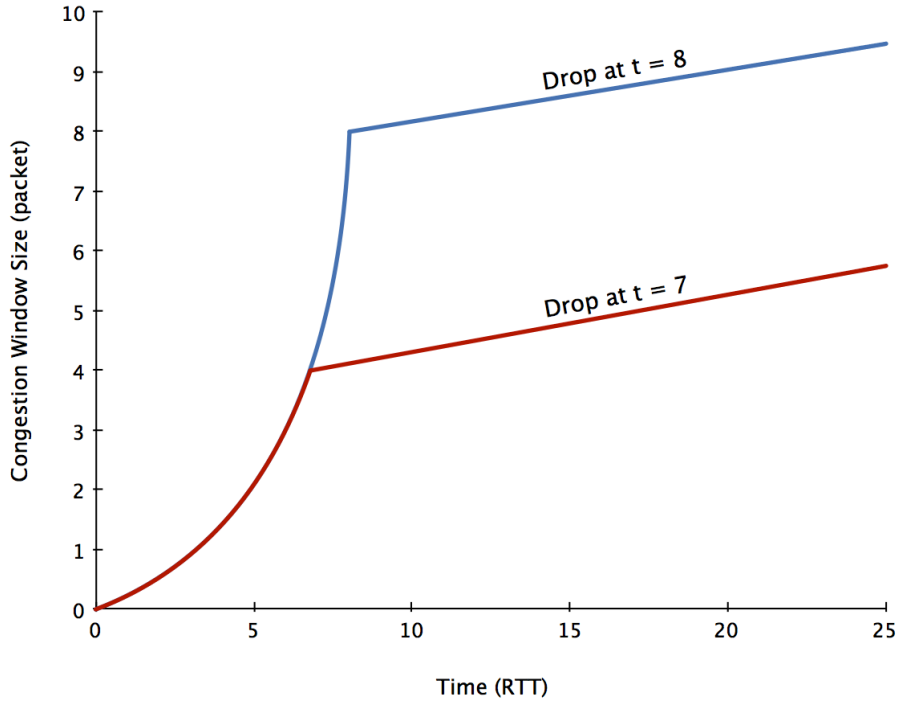
Soonho Kong, Qinsi Wang, and Gabriel Weisz

TCP Slow Start

- During normal operation, TCP achieves good bandwidth utilization through a congestion control mechanism that uses an Additive Increase, Multiplicative Decrease policy
  - This means that the transfer rate goes up until a packet is dropped, at which point it is halved
- TCP Slow Start is meant to help the connection ramp up to the peak transfer rate.
  - TCP Slow Start increases the transfer rate exponentially until a packet is dropped

Project Motivation

- TCP Slow Start generally works well
- But …



- If a packet is dropped even one round trip early, the transfer might never hit peak speed.

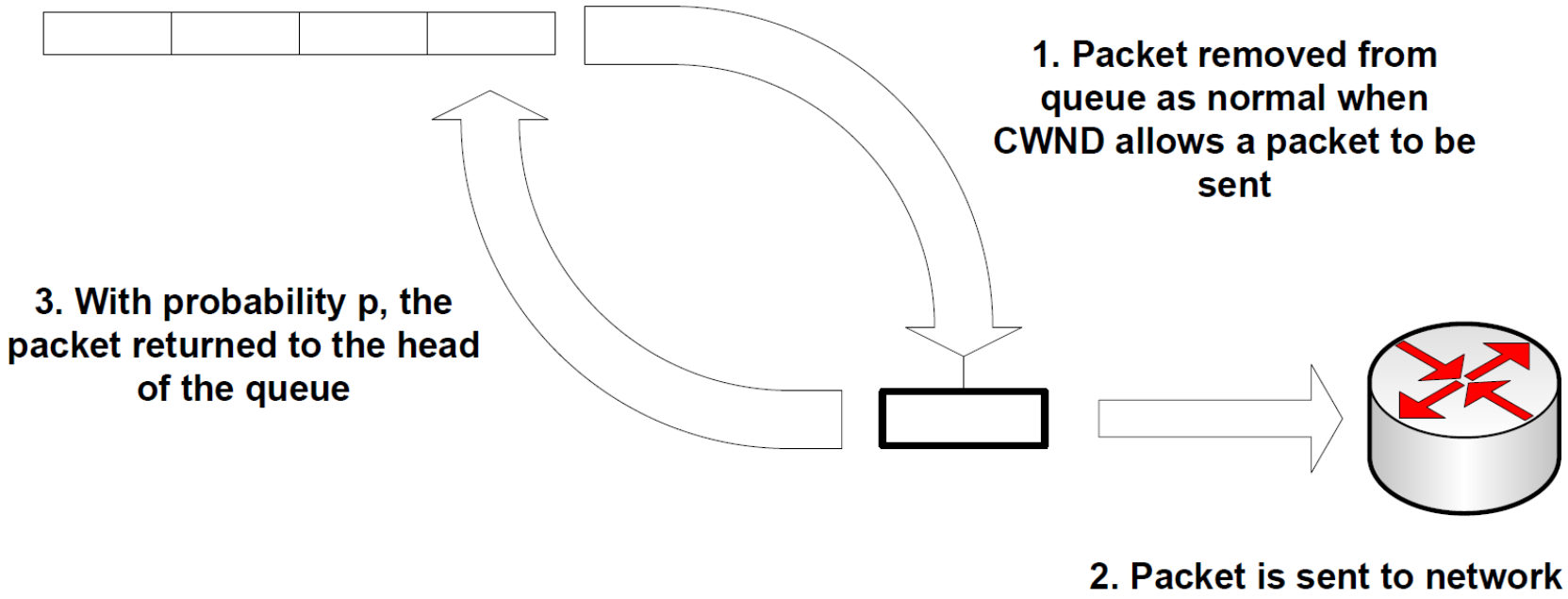
Reasons for dropped packets

- The connection might have exceeded the connection bandwidth (the design is based upon this case)
- Some other connection might be hogging the bandwidth (so why should this one suffer?)
- The connection might be wireless (which drops packets due to the medium, not just due to congestion control)

The idea

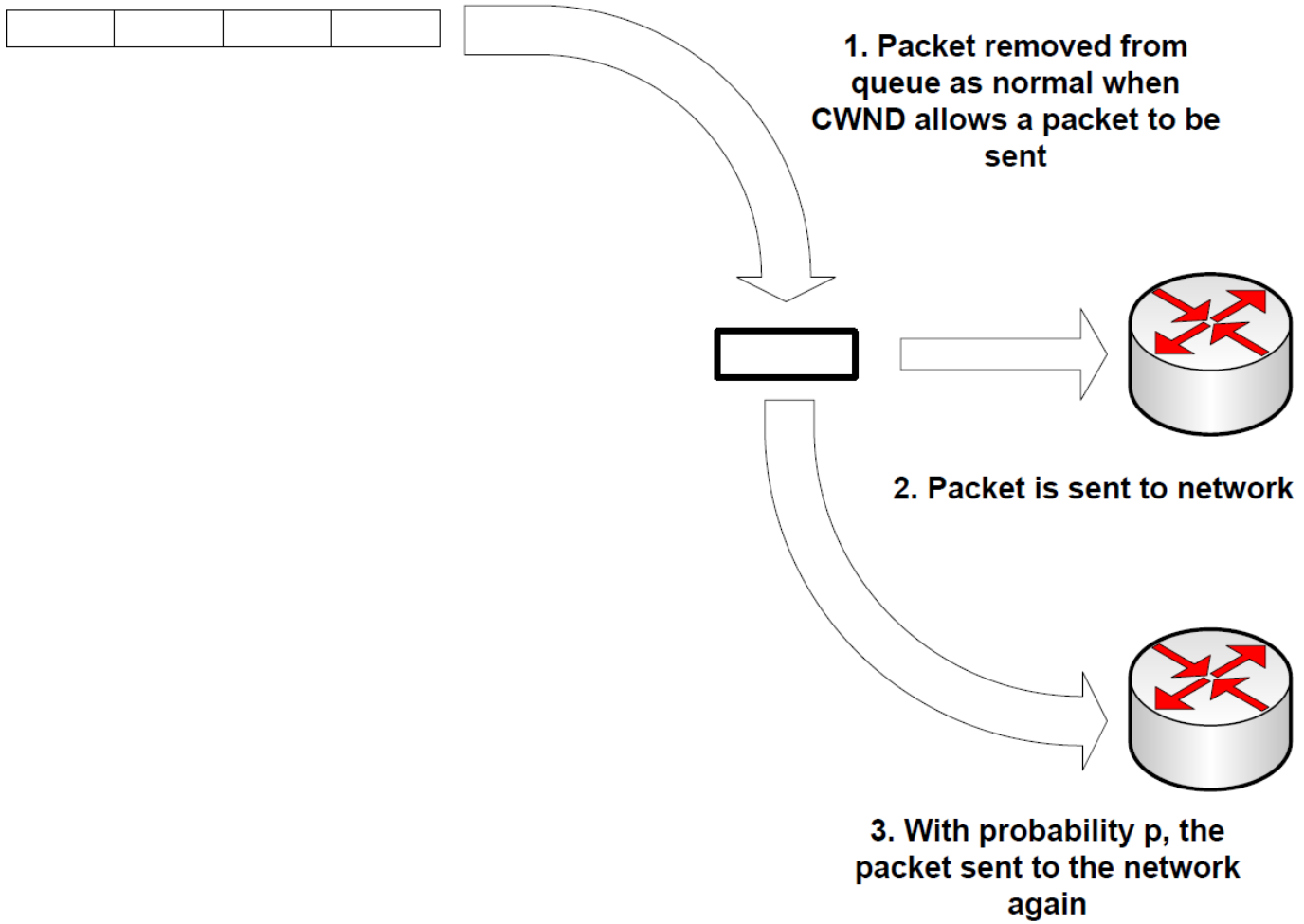
- We'd like the connection to have an opportunity to stay in slow start even if a packet was dropped, just in case it was not due to the connection reaching peak bandwidth
- What about just randomly sending some packets twice?
  - Use same sequence #, data, everything, so it does not matter which one gets through
- But we're duplicating packets!
  - Who cares if it works?
- Make sure that we do not negatively impact other clients

The implementation (Mechanism A)



- Connection never has more than # packets allowed by congestion window outstanding

The implementation (Mechanism B)



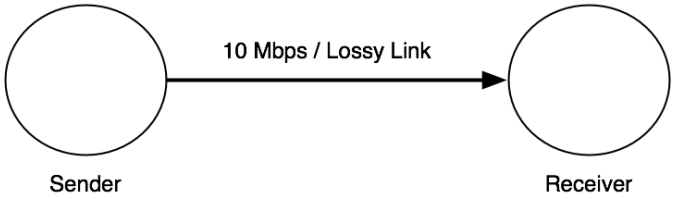
- This mechanism does have more packets than are specified by the congestion window, but still has that many unique packets

More details

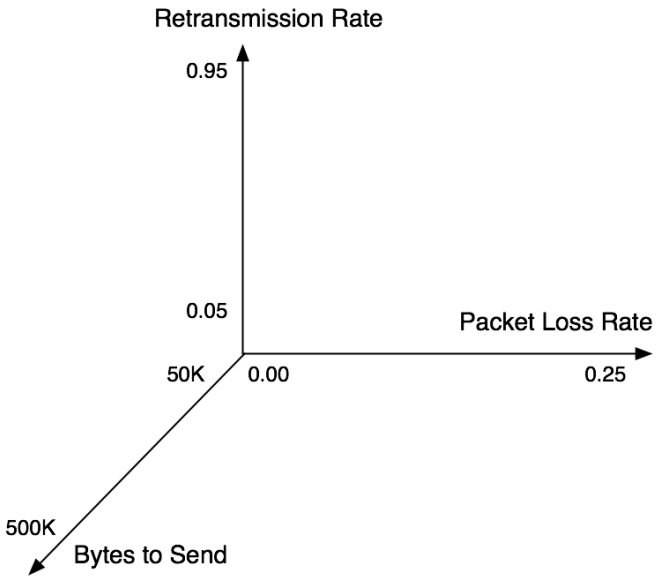
- Perhaps, instead of always using the same retransmit probability, scale the retransmit probability by the current congestion window size
  - It is really bad to drop packets …

Experiments:

- 1st Exp: Which mechanism works best?
  - Method A with Static Retransmission
  - Method B with Static Retransmission
  - Method A with Dynamic Retransmission
  - Method B with Dynamic Retransmission

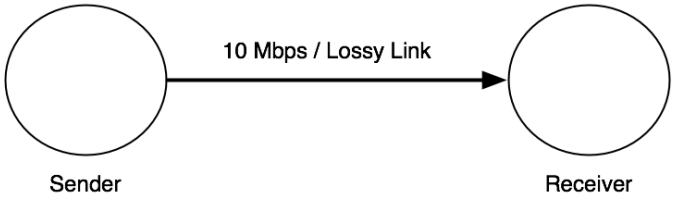


Network Configuration

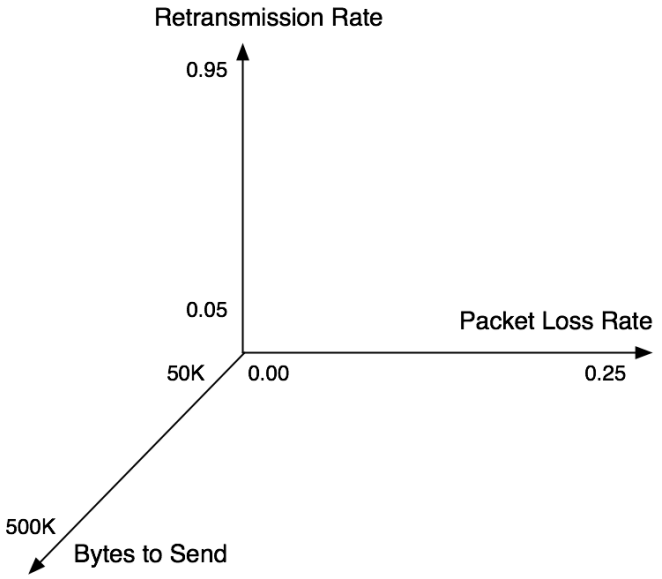


Experiments:

- 2nd Exp: How does it compare to regular TCP?
  - Normal TCP Sender
  - Best Retransmission Method

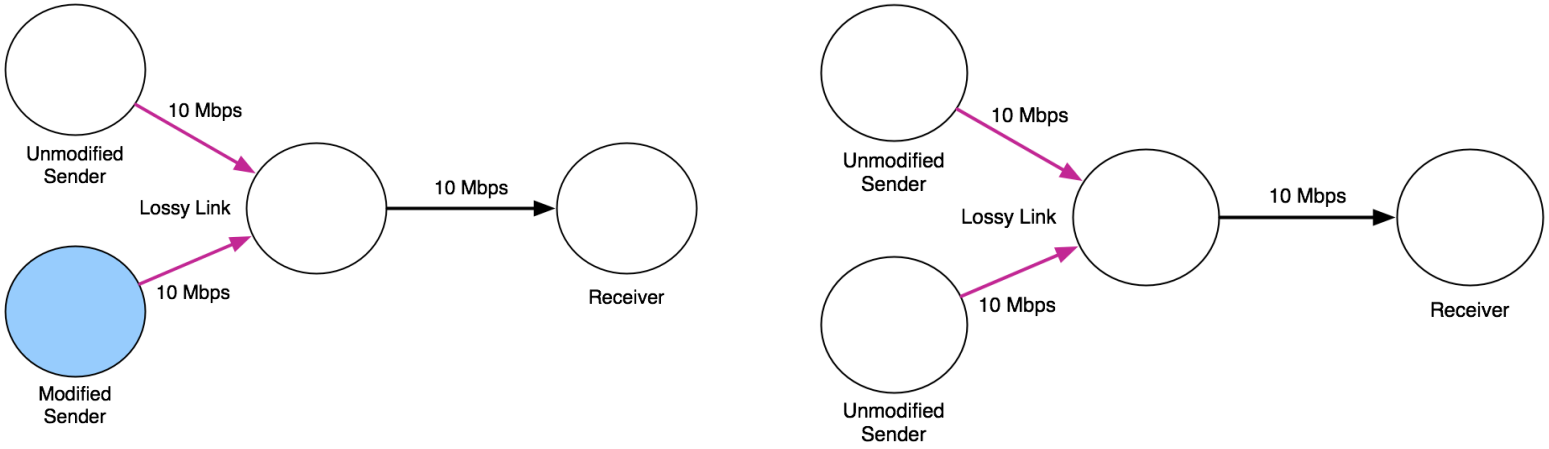


Network Configuration



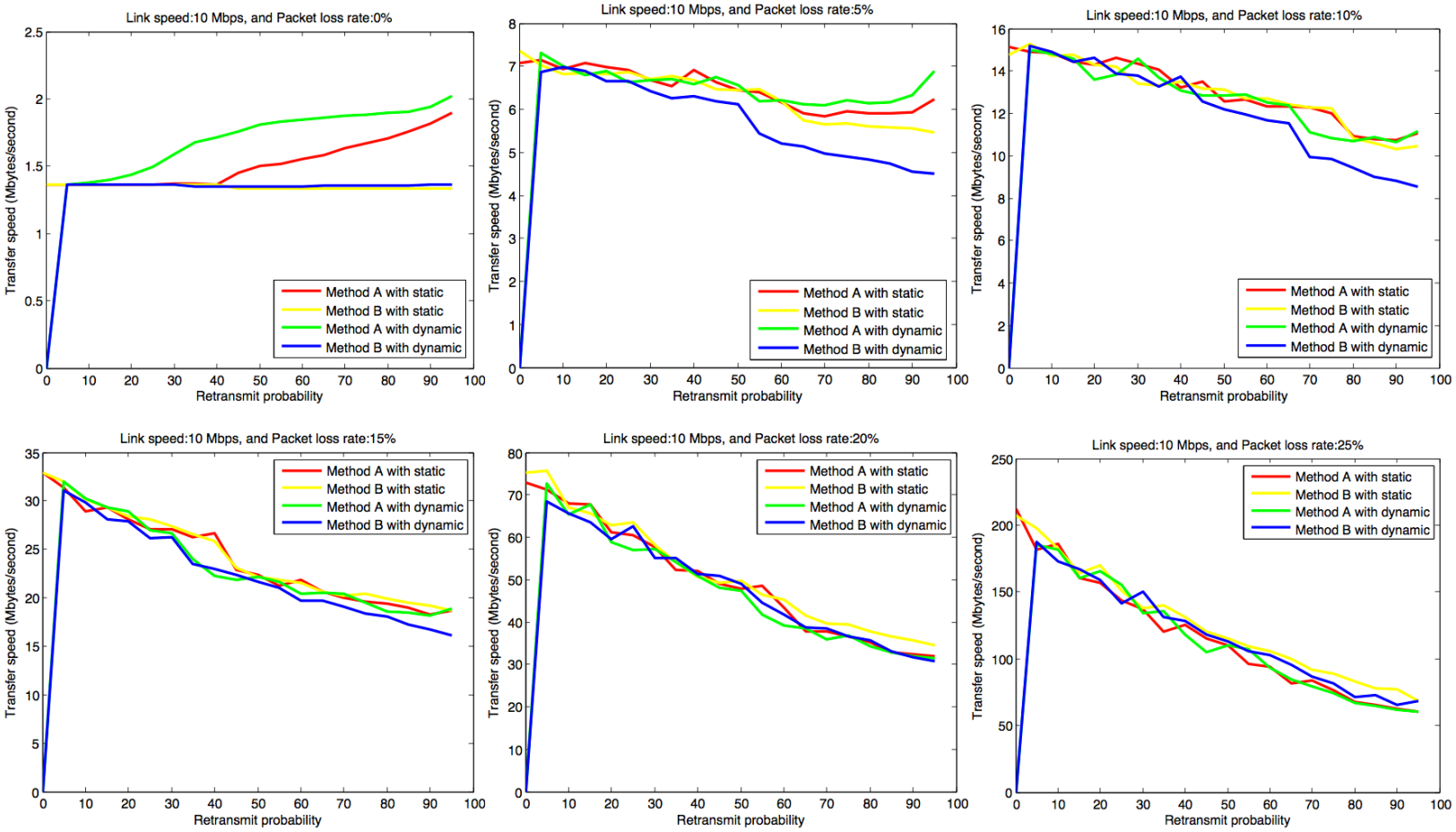
Experiments:

- 3rd Exp: What's the overhead? Does it break existing clients?
  - Two unmodified TCP sender
  - **One modified** and One unmodified TCP sender



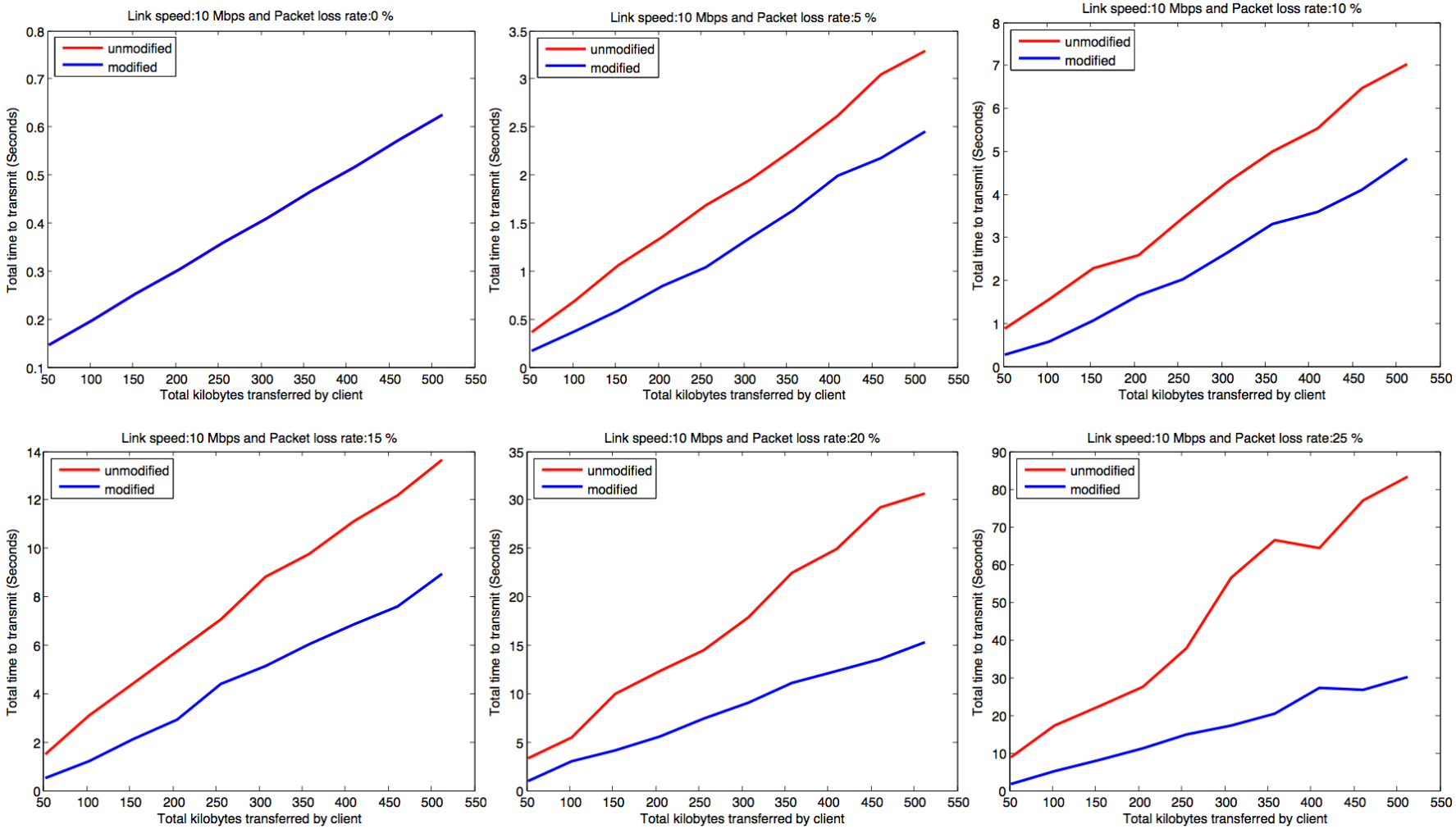
Network Configuration

Results: Which mechanism works best?



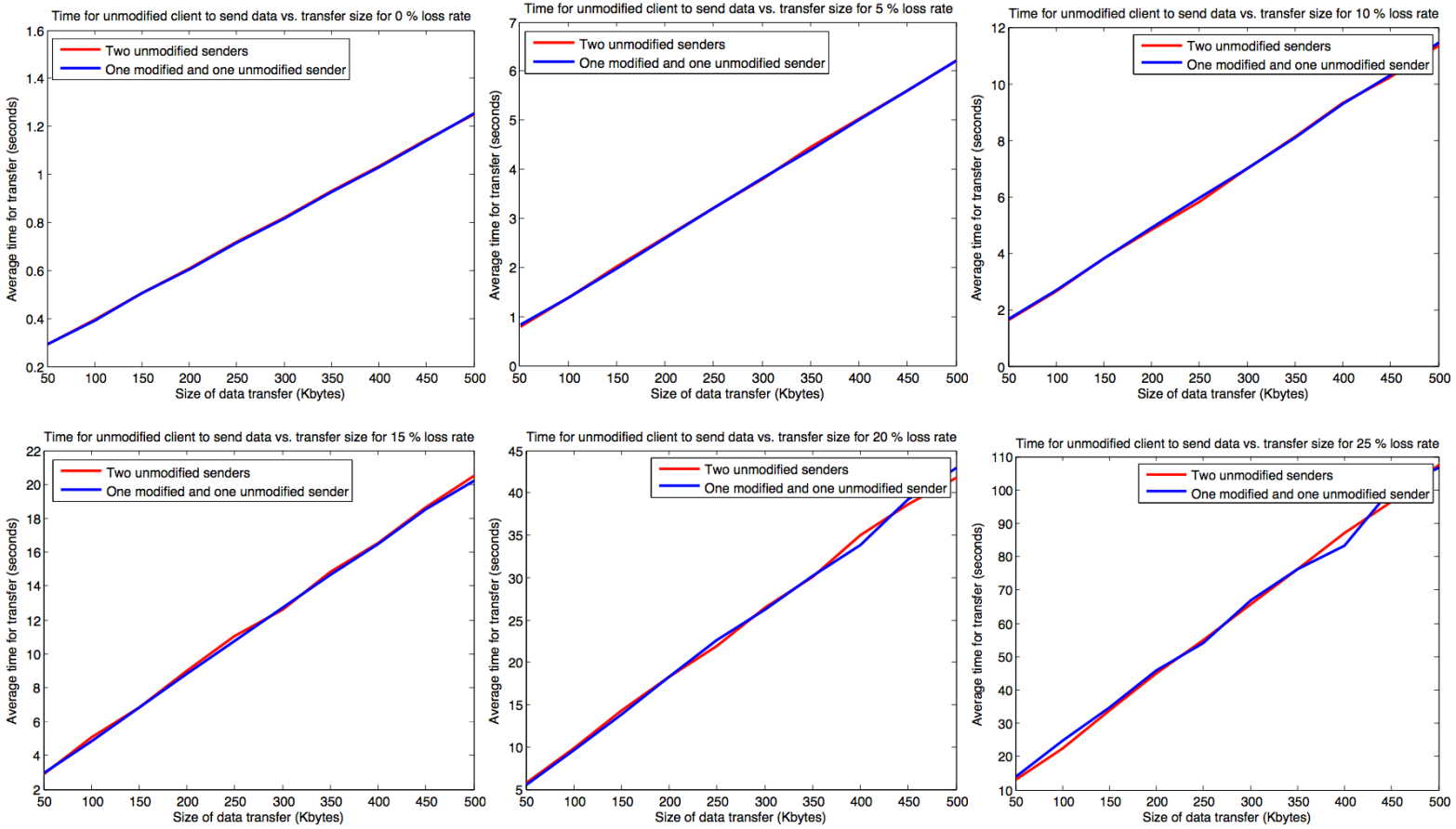
Dynamic Method B almost always works best

Results: How does it compare to regular TCP?



As expected, this works better than regular TCP

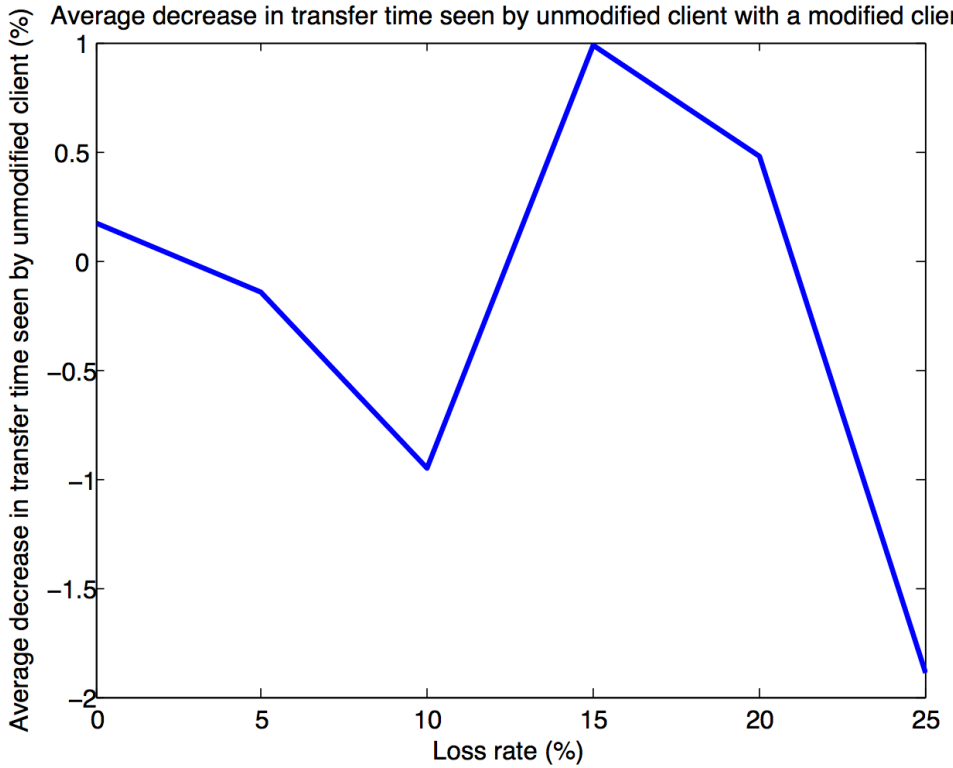
Results: What's the overhead?



Not only does it work, but it does not starve the standard client

Not only does it not starve the standard client, but the standard client may even see better performance! (more to follow)

Performance Comparison Summary



- Unmodified client does not suffer too much
- In some cases, it even improves slightly
- We think this is due to the modified sender finishing earlier

Conclusions

- It works!
- Method B with dynamic retransmit rate works best (base retransmit rate 95%)
- Modified client sees better performance
- Unmodified client does not suffer, and may even see better performance