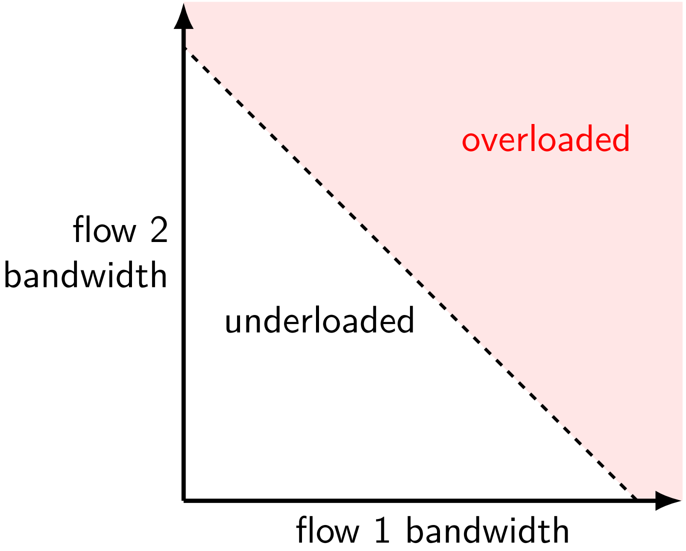
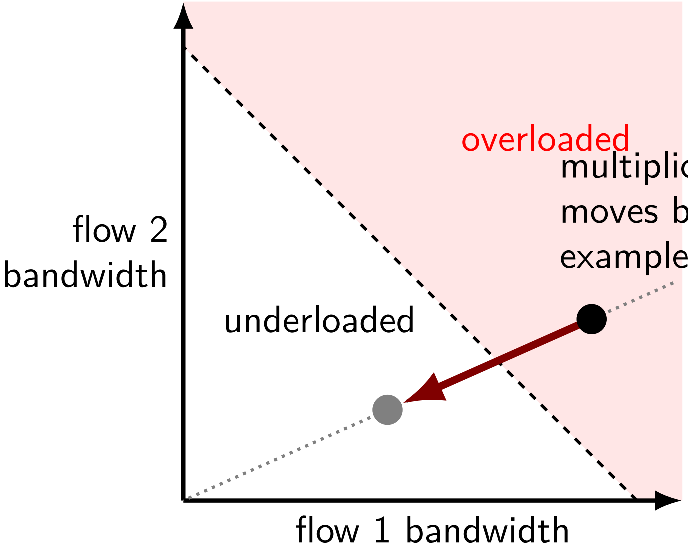
picturing sharing

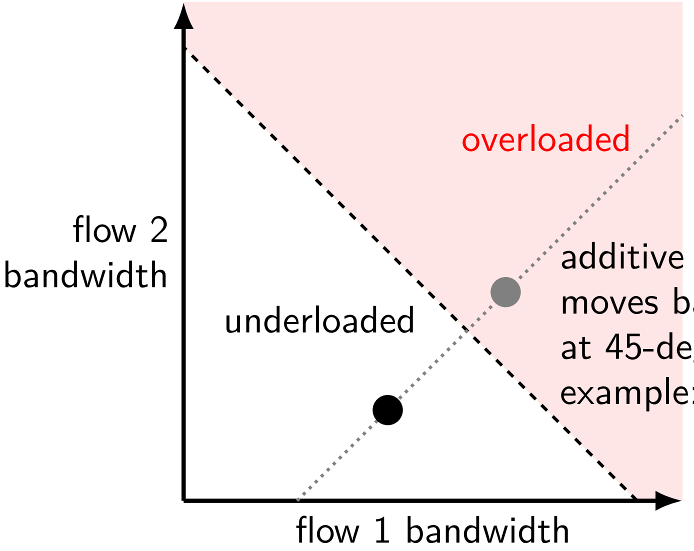


picturing sharing



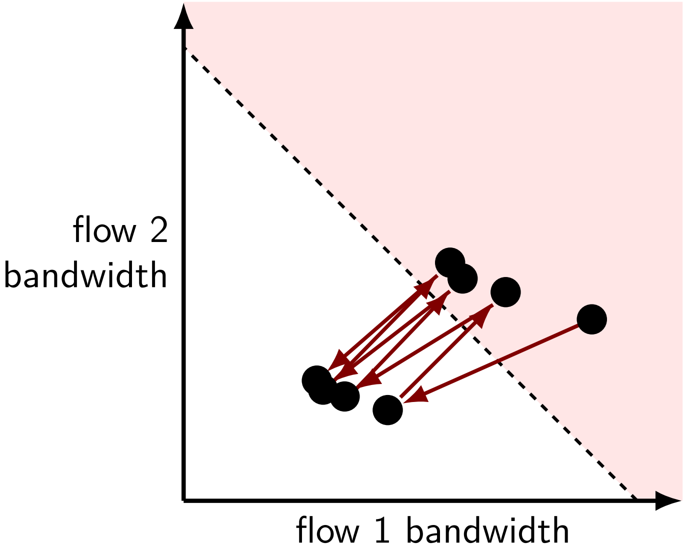
multiplicative decrease
moves bandwidths along line to origin
example: (90%, 40%) to (45%, 20%)

picturing sharing

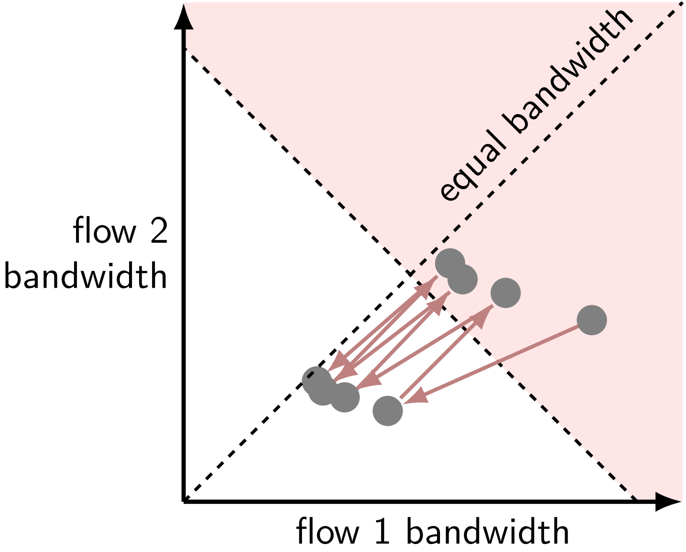


additive decrease
moves bandwidths
at 45-degree angle
example: (45%, 20%) to (55%, 30%)

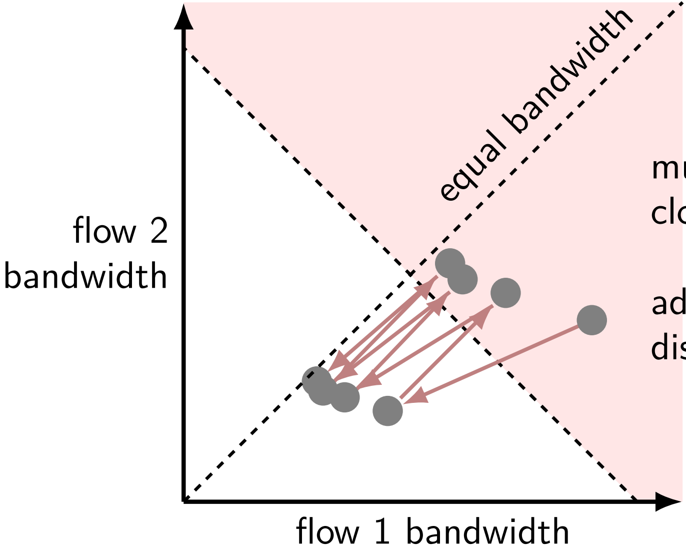
picturing sharing



picturing sharing



picturing sharing



multiplicative decrease brings closer to equal bandwidth line

additive increase keeps same distance from line some assumptions we made

...that are probably not true

both flows experience drops equally when network overloaded

same additive increase factor (for 45 degree angle) models that give numbers?

deciding on congestion control

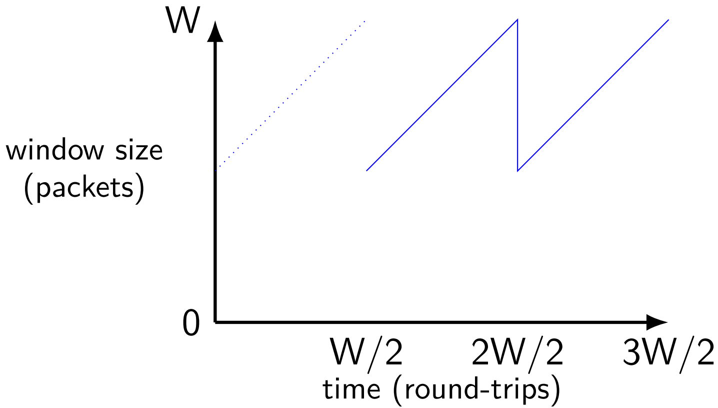
would like to do math to say how well they'll do

common approach: but gets complicated

simple example: estimating average rate of TCP-like AIMD with no congestion

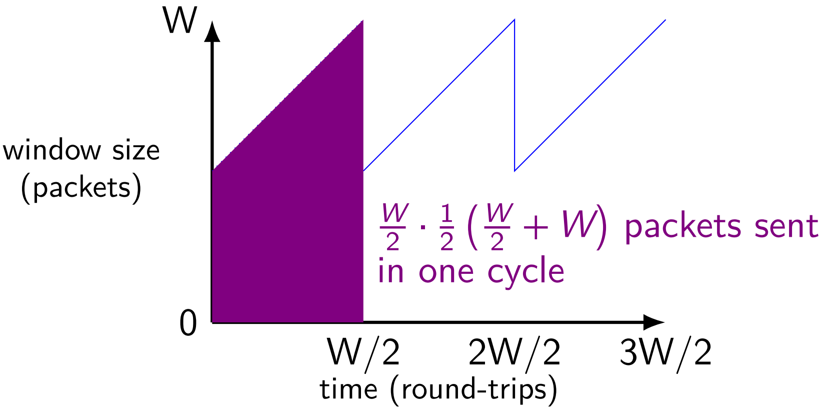
models that give numbers? (1)

figure adapted from Mathis et al, “The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm”



models that give numbers? (1)

figure adapted from Mathis et al, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm"



models that give numbers? (2)

$\frac{W}{2} \cdot \frac{1}{2}\left(\frac{W}{2} + W\right)$ per $W/2$ round trips

$\frac{3W}{4}$ packets per round trip time

packet loss should occur when window size $=$ bandwidth-delay product

at the capacity of the link(s)

so $W = \text{link BW} \cdot \text{RTT}$

$\implies$ 3/4 link BW achieved total

exercise: what things did this model miss?

some answers

- RTT/delay depends on how many packets queued
- packet loss could occur for other reasons
  - competing connections
  - network errors
- 'bursty' connection could trigger packet loss earlier
- extra packets being sent for retransmissions
- packet loss could trigger timeout/multiple decreases
- behavior of other connections sharing links
- delays in sending ACKs depending how fast receiver's CPU is

more sophisticated models?

we can add to formulas to account for other things

this is something people do, but...

most common technique is *discrete event simulation*

interlude: loss rate $\rightarrow$ transfer rate

adapted from Mathis et al, "The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm"

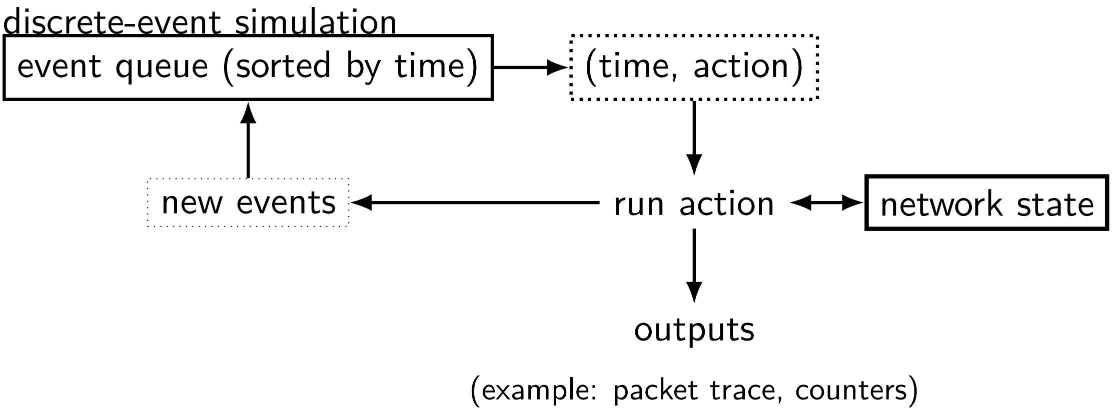
packet loss rate $p = 1$ per (number of packets sent in W/2 round trips)

$3/4 \times W \times W/2$ packets sent in W/2 round trips

$$p = \frac{3}{8}W^2$$

solving for $W = \sqrt{\frac{8}{3p}}$

average transfer rate $= \frac{3}{4}W = C \cdot \sqrt{1/p}$ (for some C)


discrete-event simulation
event queue (sorted by time)
(time, action)
new events
run action
network state
outputs
(example: packet trace, counters)

action example 1

take next packet from send queue for link X

compute whether packet is lost due to error

compute when packet is done transmitting

    schedule new event to handle next packet in queue at that time

compute reception time of packet on other end of link

    schedule new event to handle packet being received at that time action example 2

take next packet on link 0 of switch

compute next link for packet

add packet to queue for next link

schedule new events:

to dequeue from next link (if not scheduled already)

NS-3

discrete event simulator planned for AIMD assignment

written in C++

(yes, I know it's not the most familiar language)

(obvious alternative simulators aren't in better languages…)

create simulations by writing C++ programs

an NS-3 event handler

```
bool PointToPointChannel::TransmitStart(
    Ptr<const Packet> p,
    Ptr<PointToPointNetDevice> src,
    Time txTime
) {
  // ...
  uint32_t wire = src == m_link[0].m_src ? 0 : 1;

  Simulator::ScheduleWithContext(
    m_link[wire].m_dst->GetNode()->GetId(),
    txTime + m_delay,
    &PointToPointNetDevice::Receive, m_link[wire].m_dst, p->Copy());

  // Call the tx anim callback on the net device
  m_txrxPointToPoint(p, src, ...)
  return true;
}
```

an NS-3 event handler

X::Y = Y method/variable of X class

```
bool PointToPointChannel::TransmitStart(
    Ptr<const Packet> p,
    Ptr<PointToPointNetDevice> src,
    Time txTime
) {
  // ...
  uint32_t wire = src == m_link[0].m_src ? 0 : 1;

  Simulator::ScheduleWithContext(
    m_link[wire].m_dst->GetNode()->GetId(),
    txTime + m_delay,
    &PointToPointNetDevice::Receive, m_link[wire].m_dst, p->Copy());

  // Call the tx anim callback on the net device
  m_txrxPointToPoint(p, src, ...)
  return true;
}
```

an NS-3 event handler

convention: member variables with m_
(C++ member variable ∼ Java instance variable)

```
bool PointToPointChannel::TransmitS
    Ptr<const Packet> p,
    Ptr<PointToPointNetDevice> src,
    Time txTime
) {
  // ...
  uint32_t wire = src == m_link[0].m_src ? 0 : 1;

  Simulator::ScheduleWithContext(
    m_link[wire].m_dst->GetNode()->GetId(),
    txTime + m_delay,
    &PointToPointNetDevice::Receive, m_link[wire].m_dst, p->Copy());

  // Call the tx anim callback on the net device
  m_txrxPointToPoint(p, src, ...)
  return true;
}
```

an NS-3 event handler

setup future event; args:
context (for logging mostly)
time event will trigger
method to run + arguments to pass

```
bool PointToPointChannel::TransmitStart(
    Ptr<const Packet> p,
    Ptr<PointToPointNetDevice> src,
    Time txTime
) {
  // ...
  uint32_t wire = src == m_link[0].m_src ? 0 : 1;

  Simulator::ScheduleWithContext(
    m_link[wire].m_dst->GetNode()->GetId(),
    txTime + m_delay,
    &PointToPointNetDevice::Receive, m_link[wire].m_dst, p->Copy());

  // Call the tx anim callback on the net device
  m_txrxPointToPoint(p, src, ...)
  return true;
}
```

sample NS-3 simulation — setup (1)

`ns3/examples/tcp/tcp-bulk-send.cc`

```
// nodes = routers or endpoints
NodeContainer nodes;
nodes.Create(2);

// create simulated point-to-point link
    // also supported: multi-access links
PointToPointHelper pointToPoint;
pointToPoint.SetDeviceAttribute("DataRate", StringValue("500Kbps"));
pointToPoint.SetChannelAttribute("Delay", StringValue("5ms"));

// setup emulated NICs (which have queues, etc.)
NetDeviceContainer devices;
devices = pointToPoint.Install(nodes);

// setup emulated TCP/IP implementation
InternetStackHelper internet;
internet.Install(nodes);

...
```

sample NS-3 simulation — setup (2)

ns3/examples/tcp/tcp-bulk-send.cc

```
// Simulated "applications" that send/receive data
BulkSendHelper source("ns3::TcpSocketFactory", InetSocketAddress(i.GetAddress(1), port));
source.SetAttribute("MaxBytes", UintegerValue(10000000));
ApplicationContainer sourceApps = source.Install(nodes.Get(0));
sourceApps.Start(Seconds(0));
sourceApps.Stop(Seconds(10));
PacketSinkHelper sink("ns3::TcpSocketFactory", InetSocketAddress(Ipv4Address::GetAny(), p
ApplicationContainer sinkApps = sink.Install(nodes.Get(1));
sinkApps.Start(Seconds(0));
sinkApps.Stop(Seconds(10));
```

sample NS-3 simulation — setup (3)

`ns3/examples/tcp/tcp-bulk-send.cc`

```
AsciiTraceHelper ascii;

// produces text trace file of simulator events
pointToPoint.EnableAsciiAll(ascii.CreateFileStream("tcp-bulk-send.tr"));

// produces PCAP files you can open in Wireshark
pointToPoint.EnablePcapAll("tcp-bulk-send", false);
```

TCPSocketState

class to track socket state

regarldess of congestion control algorithm

includes (notably for us)

‘congesiton state’

congestion window (cwnd)

slow start threshold (ssthresh)

TCP congestion states

OPEN — normal

DISORDER — dupacks/SACKs below threshold for recovery

RECOVERY — retransmitting due to dup-ACK/simple SACK
temporarily inflated window to account for dropped packets

LOSS — retransmitting due to timeout/etc.

configurable TCP behavior

TCPRecoveryOps: handle recovery

EnterRecovery (called on start of retransmitting)

default: cwnd $\rightarrow$ ssthresh

DoRecovery (called for ACK when retransmitting)

default: cwnd $\rightarrow$ cwnd + 1 packet

ExitRecovery (called when ‘back to normal’)

default: cwnd $\rightarrow$ ssthresh

TCPCongestionOps:

IncreaseWindow (called when new segments ACKed)

GetSsThresh (called after loss)

assingment TCP changes

will customize a `MyTcpCongestionOps`

...which inherits from TCPNewReno

you can customize increase (IncreaseWindow) and decrease (GetSsThresh)

TcpNewReno

```
void
TcpNewReno::IncreaseWindow(Ptr<TcpSocketState> tcb, uint32_t segmentsAcked)
{
    NS_LOG_FUNCTION(this << tcb << segmentsAcked);

    if (tcb->m_cWnd < tcb->m_ssThresh)
    {
        segmentsAcked = SlowStart(tcb, segmentsAcked);
    }

    if (tcb->m_cWnd >= tcb->m_ssThresh)
    {
        CongestionAvoidance(tcb, segmentsAcked);
    }
}
```

aside: more advanced congestion hooks

TcpCongestionOps also has access to

- ECN (explicit congestion notification) info
- timestamps
- exactly when duplicate ACKs below threshold/ recovery/etc. occurs

used by more advanced TCP implementatoins backup slides