TCP Tuning

Domenico Vicinanza
DANTE, Cambridge, UK
domenico.vicinanza@dante.net

EGI Technical Forum 2013, Madrid, Spain

TCP

- Transmission Control Protocol (TCP)
- One of the original core protocols of the Internet protocol suite (IP)
- >90% of the internet traffic
- Transport layer
- Delivery of a stream of bytes between
  - programs running on computers
  - connected to a local area network, intranet or the public Internet.
- TCP communication is:
  - Connection oriented
  - Reliable
  - Ordered
  - Error-checked
- Web browsers, mail servers, file transfer programs use TCP

Connection-Oriented

- A connection is established before any user data is transferred.
- If the connection cannot be established the user program is notified.
- If the connection is ever interrupted the user program(s) is notified.

Reliable

- TCP uses a sequence number to identify each byte of data.
- Sequence number identifies the order of the bytes sent
- Data can be reconstructed in order regardless:
  - Fragmentation
  - Disordering
  - Packet loss

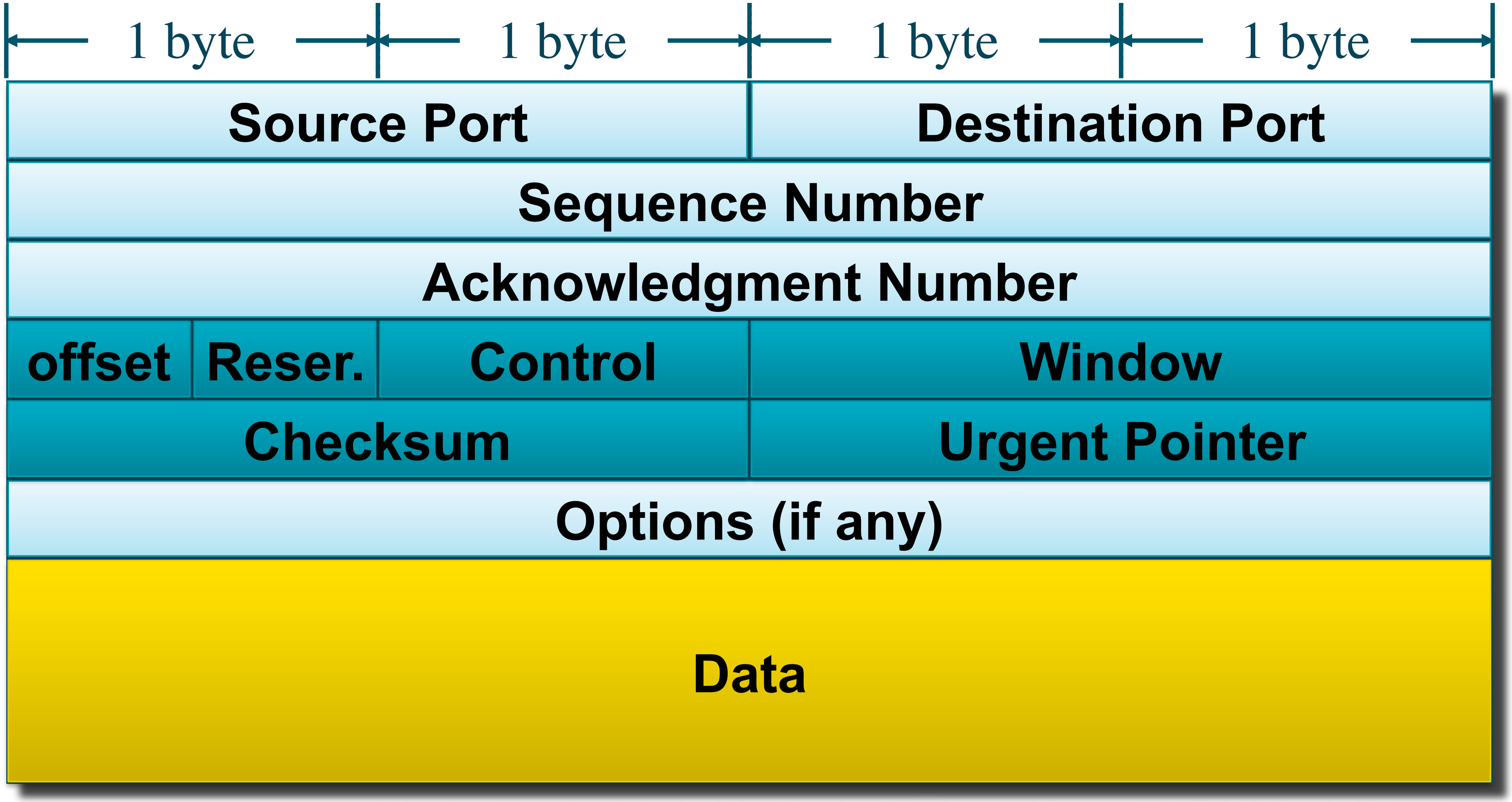
  that may occur during transmission.
- For every payload byte transmitted, the sequence number is incremented.

TCP Segments

- The block of data that TCP asks IP to deliver is called a *TCP segment*.

- Each segment contains:
  - Data
  - Control information

TCP Segment Format



Client Starts

- A client starts by sending a SYN segment with the following information:
  - Client's ISN (generated pseudo-randomly)
  - Maximum Receive Window for client.
  - Optionally (but usually) MSS (largest datagram accepted).

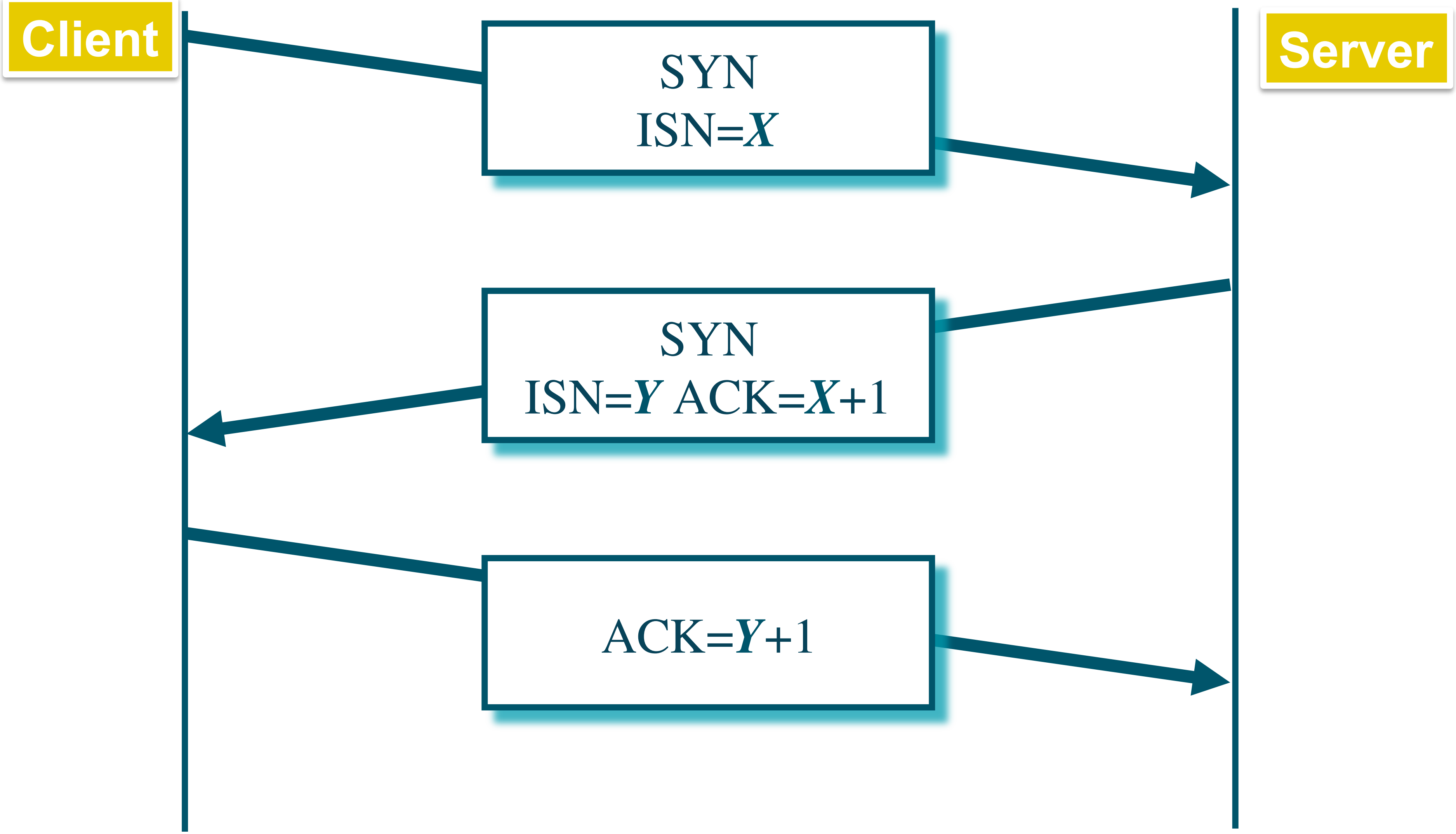
Server Response

- When a waiting server sees a new connection request, the server sends back a SYN segment with:
  - Server’s ISN (generated pseudo-randomly)
  - Request Number is Client ISN+1
  - Maximum Receive Window for server.
  - Optionally (but usually) MSS

Connection established!

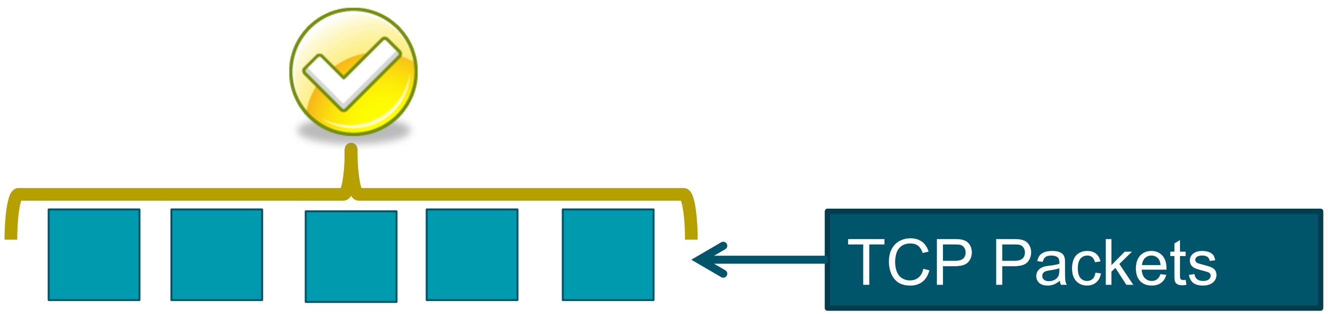
- When the Server’ s SYN is received, the client sends back an ACK with:
  - Acknowledgment Number is Server’s ISN+1

In blocks:



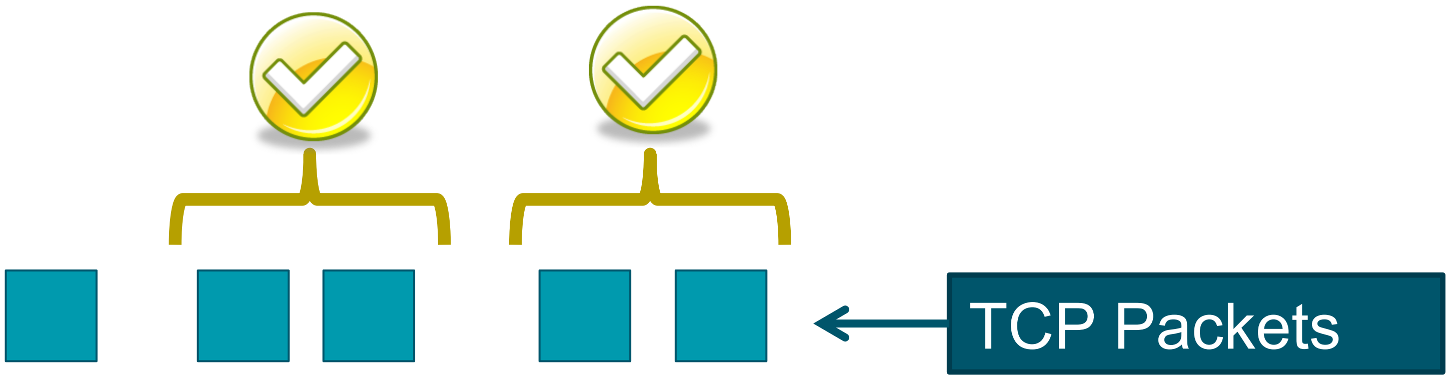
Cumulative acknowledgement

- Cumulative acknowledgment:
  - The receiver sends an acknowledgment when it has received **all** data preceding the acknowledged sequence number.
- Inefficient when packets are lost.
- Example:
  - *10,000 bytes are sent in 10 different TCP packets and*
  - *the first packet is lost during transmission.*
  - *The receiver cannot say that it received bytes 1,000 to 9,999 successfully*
  - *Thus the sender may then have to resend all 10,000 bytes.*



Selective acknowledgment

- Selective acknowledgment (SACK) option is defined in RFC 2018
- Acknowledge discontinuous blocks of packets received correctly
- The acknowledgement can specify a number of SACK blocks
- In the previous example above:
  - *The receiver would send SACK with sequence numbers 1000 and 9999.*
  - *The sender thus retransmits only the first packet, bytes 0 to 999.*



Buffering

- TCP works by:
  - buffering data at sender and receiver

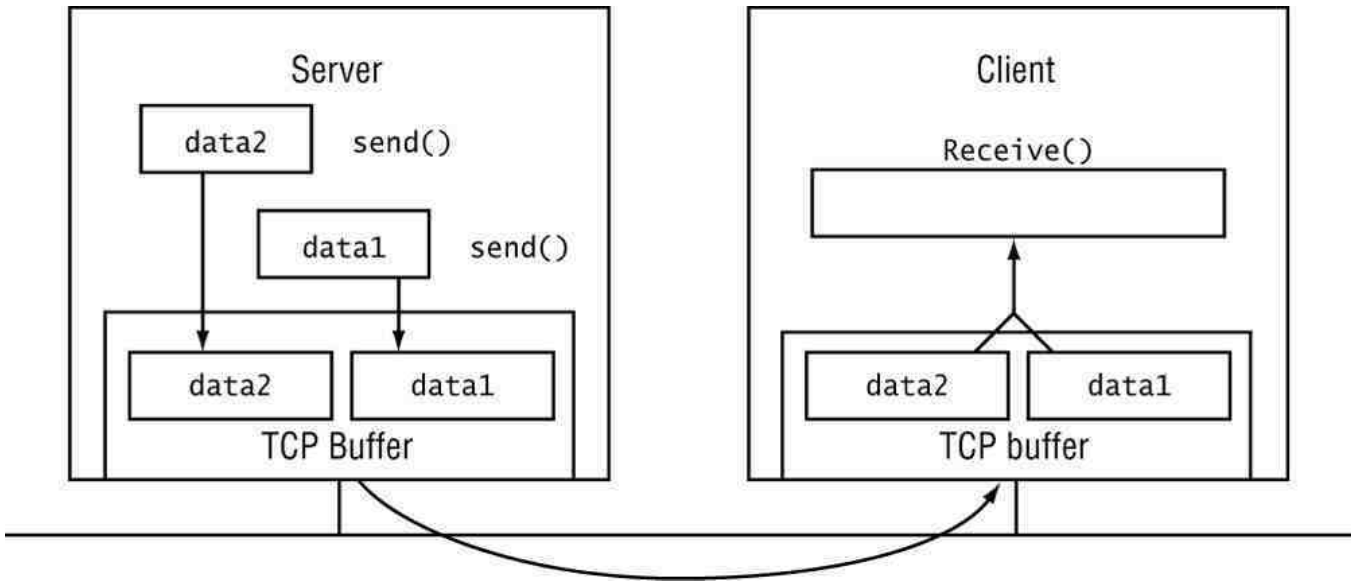


Image source: http://codeidol.com/img/csharp-network/f0502_0.jpg

Dynamic transmission tuning

- Data to send are temporarily stored in a send buffer
  - where it stays until the data is ACK'd.
- The TCP layer won't accept data from the application unless there is buffer space.
- Both the client and server announce how much buffer space remains
  - Window field in a TCP segment, with every ACK
  - TCP can know when it is time to send a datagram.

Flow control

- Limits the sender rate to guarantee reliable delivery.
- Avoid flooding
- The receiver continually hints the sender on how much data can be received
- When the receiving host buffer fills
  - the next ack contains a 0 in the window size
  - this stop transfer and allow the data in the buffer to be processed.

TCP Tuning

- Adjust the network congestion avoidance parameters for TCP
- Typically used over high-bandwidth, high-latency networks
  - Long-haul links (Long Fat Networks)
  - Intercontinental circuits
- Well-tuned networks can perform up to many times faster

Tuning buffers

- Most operating systems limit the amount of system memory that can be used by a TCP connection.
- Maximum TCP Buffer (Memory) space.
- Default max values are typically too small for network measurement and troubleshooting purposes.
- Linux (as many OSes) supports separate send and receive buffer limits
- Buffer limits can be adjusted by
  - The user
  - The application
  - Other mechanisms
- within the maximum memory limits above.

BDP Bandwidth Delay Product

- BDP=Bandwidth * Latency
- Number of bytes in flight to fill path
- Max number of un-acknowledged packets on the wire
- Max number of simultaneous bits in transit between the transmitter and the receiver.
- High performance networks have very large BDPs.

TCP receive buffer

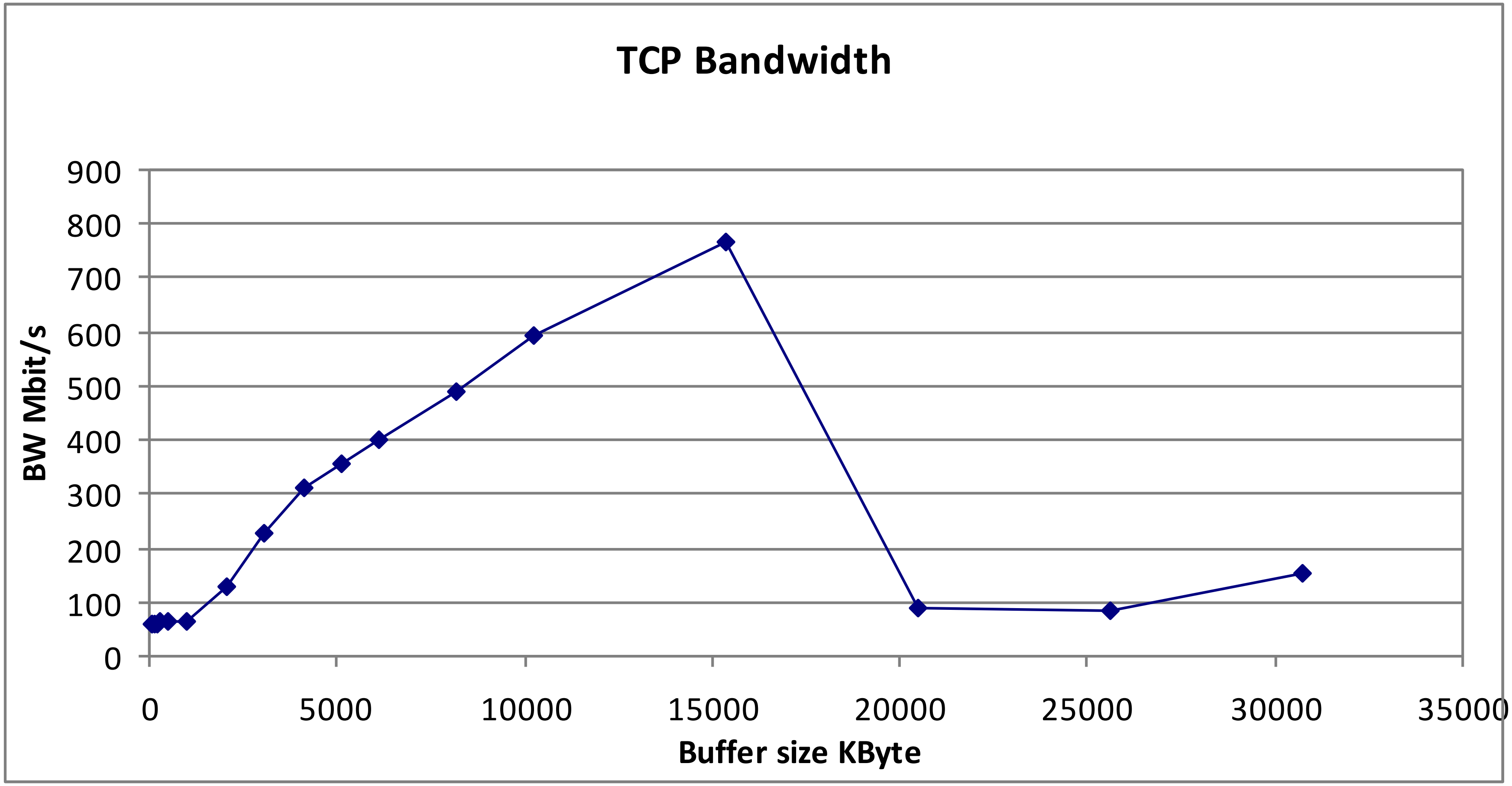
- Amount of data that a computer can store without acknowledging the sender.
- It can limit throughput
  - even if there is no packet loss in the network!
- TCP transmits data up to the buffer size before waiting for the ack
- Therefore the full bandwidth of the network may not always get used.

$$Throughput \leq TCP\ Receive\ buffer / RTT$$

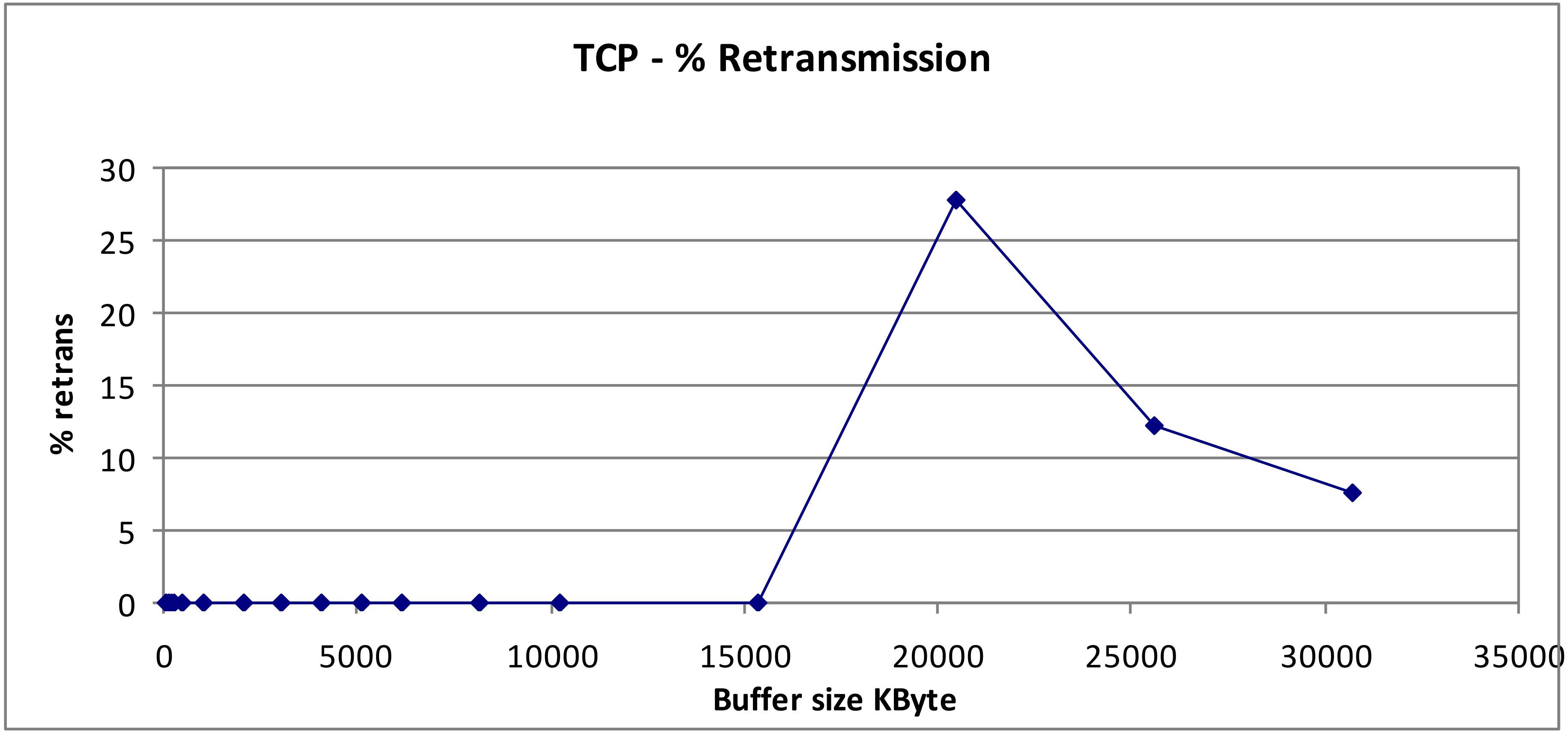
Optimising buffers

- TCP receiver and sender buffers needs tuning
- They should be ideally equal to BDP to achieve maximum throughput
- The sending side should also allocate the same amount of memory
- After data has been sent on the network
  - the sending side must hold it in memory until it has been ack'd
  - If the receiver is far away, acks will take a long time to arrive.
  - If the send memory is small, it can saturate and block transmission.

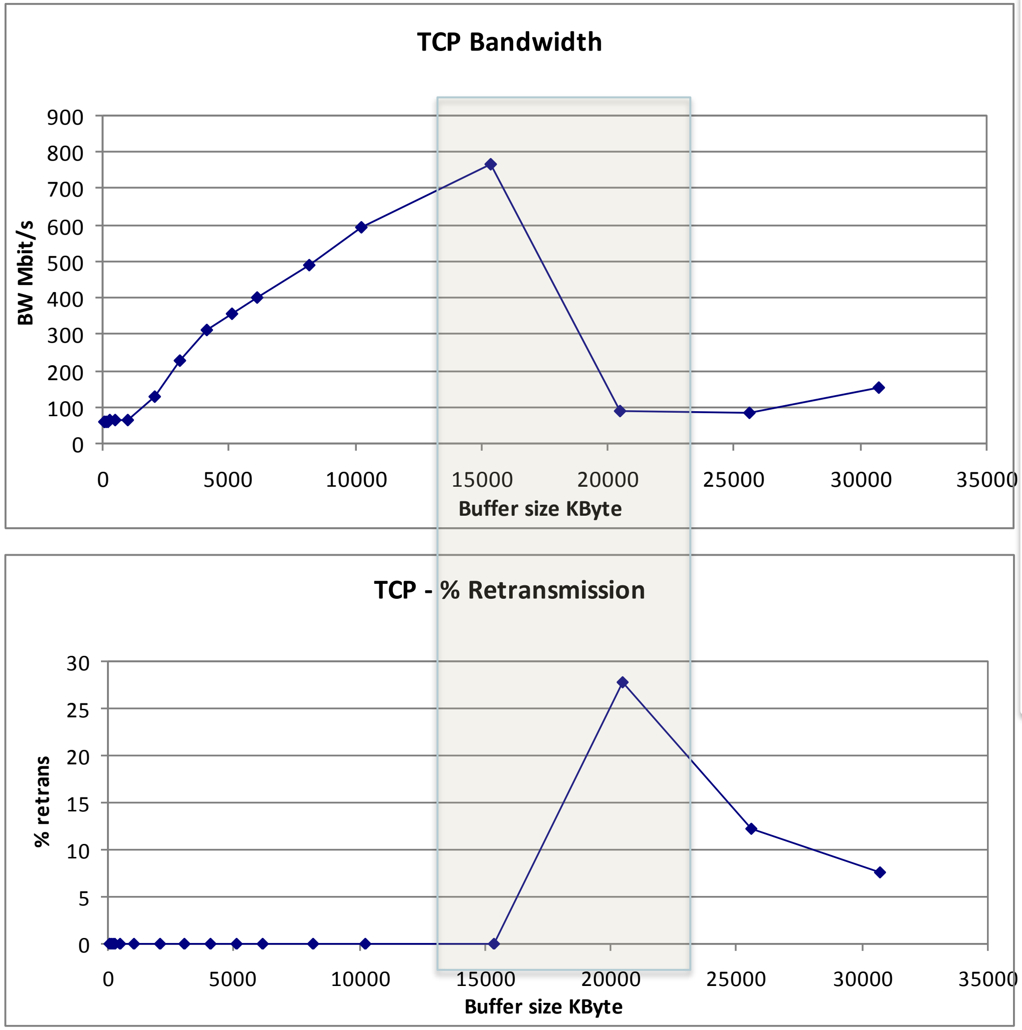
Madrid-Mumbai Bandwidth



Madrid-Mumbai Retransmission



BDP as optimal buffer parameter



Bandwidth increases with buffer size until it reaches BDP

RTT ~168ms
Bandwidth limit to 1GE interface
➔ ~20 Mbytes.

Checking send and receive buffers

- To check the current value type either:

```
$ sysctl net.core.rmem_max
net.core.rmem_max = 65535
$ sysctl net.core.wmem_max
net.core.wmem_max = 65535
```

- or

```
$ cat /proc/sys/net/core/rmem_max
65535
$ cat /proc/sys/net/core/wmem_max
65535
```

Setting send and receive buffers

- To change those value simply type:

```
sysctl -w net.core.rmem_max=33554432
sysctl -w net.core.wmem_max=33554432
```

- In this example the value 32MByte has been chosen:

  32 x 1024 x 1024 = 33554432 Byte

Autotuning buffers

- Automatically tunes the TCP receive window size for each individual connection
- Based on BDP and rate at which the application reads data from the connection
- Linux autotuning TCP buffer limits can be also tuned
- Arrays of three values:
  - minimum, initial and maximum buffer size.
- Used to:
  - Set the bounds on autotuning
  - Balance memory usage while under memory stress.
- Controls on the actual memory usage (not just TCP window size)
  - So it includes memory used by the socket data structures
- The maximum values have to be larger than the BDP
- Example: for a BDP of the order of 20MB, we can chose 32MB

Check and set autotuning buffers

- To check the TCP autotuning buffers we can use sysctl:

```
$ sysctl net.ipv4.tcp_rmem
4096    87380   65535
$ sysctl net.ipv4.tcp_wmem
4096    87380   65535
```

- It is best to set it to some optimal value for typical small flows.
- Excessively large initial buffer waste memory and can even hurt performance.
- To set them:

```
$ sysctl -w net.ipv4.tcp_rmem="4096    87380
33554432"
$ sysctl -w net.ipv4.tcp_wmem="4096    87380
33554432"
```

Checking and enabling autotuning

- TCP autotuning is normally enabled by default.
- To check type:

```
$ sysctl net.ipv4.tcp_moderate_rcvbuf
1
```

or

```
$ cat /proc/sys/net/ipv4/tcp_moderate_rcvbuf
1
```

- If the parameter *tcp_moderate_rcvbuf* is present and has value 1 then autotuning is enabled.
- With autotuning, the receiver buffer size (and TCP window size) is dynamically updated (autotuned) for each connection
- If not enabled, it is possible to enabled it by typing:

```
$ sysctl -w net.ipv4.tcp_moderate_rcvbuf=1
```

Interface queue length

- Improvement at NIC driver level
- Increase the size of the interface queue. To do this, run the following command.

```
$ ifconfig eth0 txqueuelen 1000
```

- TXQueueLen: max size of packets that can be buffered on the egress queue of a linux net interface.
- Higher queues: more packets can be buffered and hence not lost.
- In TCP, an overflow of this queue will cause loss
  - TCP will enter in the congestion control mode

Additional tuning

- Verify that the following variables are all set to the default value of 1

```
net.ipv4.tcp_window_scaling
net.ipv4.tcp_timestamps
net.ipv4.tcp_sack
```

Otherwise set them using

```
$ sysctl -w net.ipv4.tcp_window_scaling = 1
$ sysctl -w net.ipv4.tcp_timestamps = 1
$ sysctl -w net.ipv4.tcp_sack = 1
```

What not to change

- We suggest not to adjust *tcp_mem* unless there is some specific need.
- It is an array that determines how the system balances the total network buffer space
  - against all other LOWMEM memory usage.
- Initialized at boot time to appropriate fractions of the available system memory.
- In the same way there is normally no need to adjust *rmem_default* or *wmem_default*
  - These are the default buffer sizes for non-TCP sockets (e.g. unix domain and UDP sockets).

Congestion window and slow start

- **Congestion window**:
  - Estimation how much congestion there is between sender and receiver
  - It is maintained at the sender
- **Slow start**: increase the congestion window after a connection is initialized and after a timeout.
  - It starts with a window of 1 maximum segment size (MSS).
  - For every packet acknowledged, the congestion window increases by 1 MSS
  - The congestion window effectively doubles for every round trip time (RTT).
  - Actually not so slow…

TCP Congestion control

- Initially one algorithm available Reno
- Linear increment of the congestion window
- It typically drops to half the size when a packet is lost
- Starting from Linux 2.6.7, alternative congestion control algorithms were implemented
  - recover quickly from packet loss on high-speed and high BDP networks.
- The choice of congestion control options is selected when the kernel is built.

Some congestion control examples

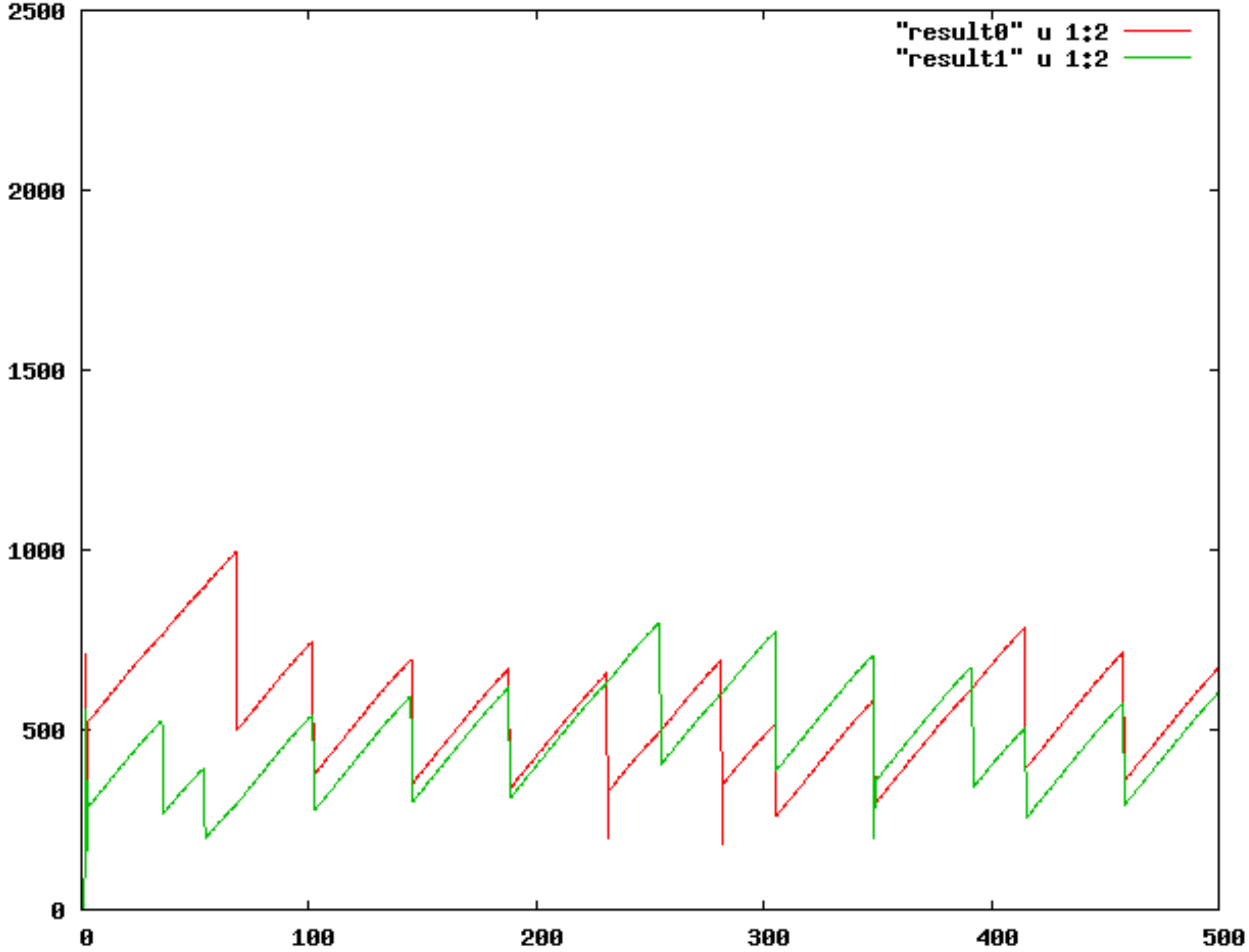
The following are some of the options are available in the 2.6 kernel:

- **reno**: Traditional TCP used by almost all other OSes (default with old Linux kernel).
  - It adjusts congestion window based on packet loss.
  - The slow start has an additive Increase window on each Ack and
  - a Multiplicative Decrease on loss
- **cubic**: Faster (cubic function) recovery on packet loss
  - Efficient for high-BDP network
- **bic**: Combines two schemes called additive increase and binary search increase.
  - It promises fairness as well as good scalability.
  - Under small congestion windows, binary search increase is designed to provide TCP friendliness.
  - Default congestion-control in many Linux distribution

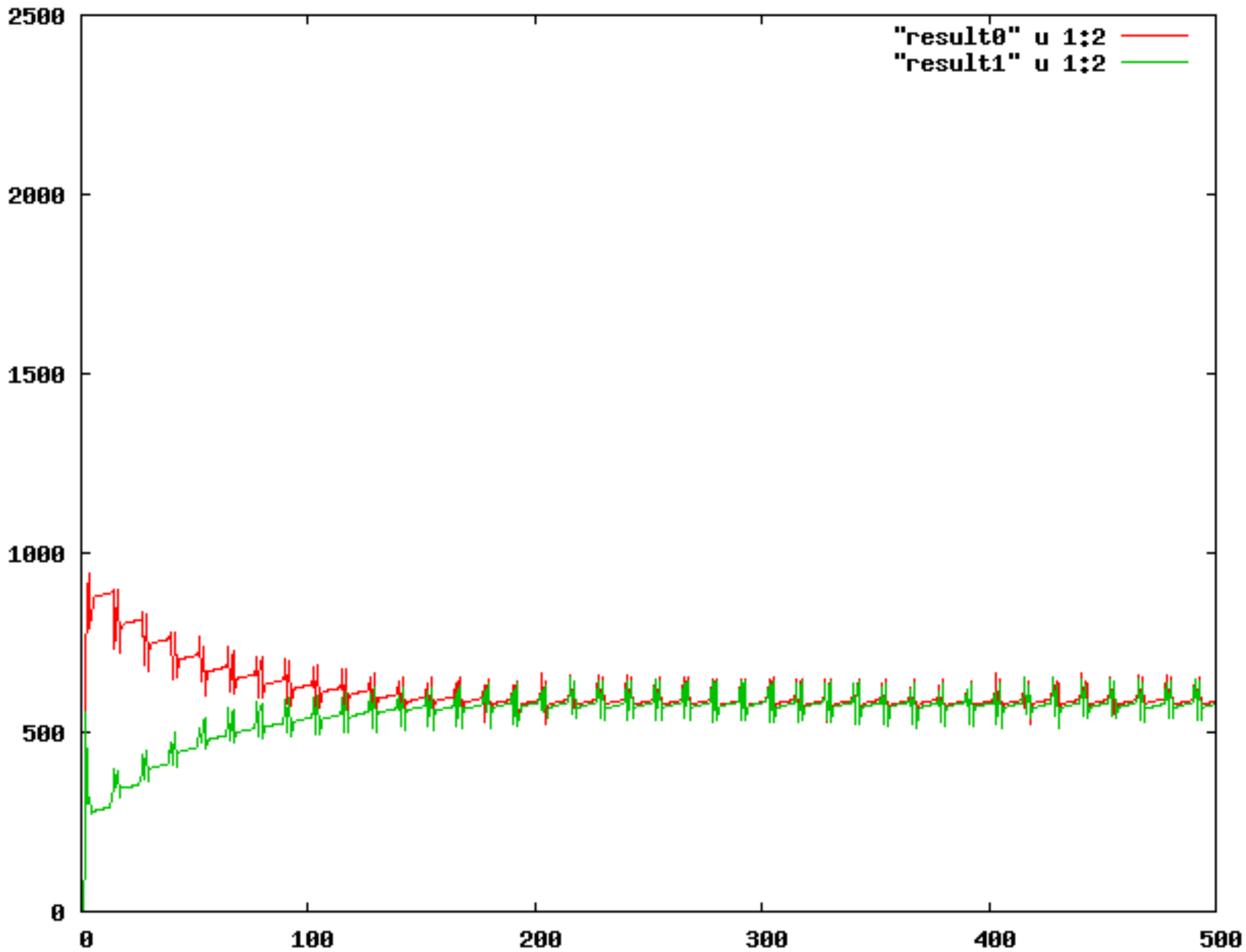
Some congestion control examples Cont.

- **hstcp:** An adaptive algorithm that:
  - Increases its additive increase parameter and
  - decreases its decrease parameter in relation to the current congestion window size.
- **vegas:** It measure bandwidth based on RTT and adjust congestion window on bandwidth
- **westwood:** optimized for lossy networks. The focus in on wireless networks (where packet loss does not necessarily mean congestion).
- **htcp**: Hamilton TCP: Optimized congestion control algorithm for high speed networks with high latency (LFN: Long Fat Networks).
  - Hamilton TCP increases the rate of additive increase as the time since the previous loss increases.
  - This avoids the problem of making flows more aggressive if their windows are already large (cubic).

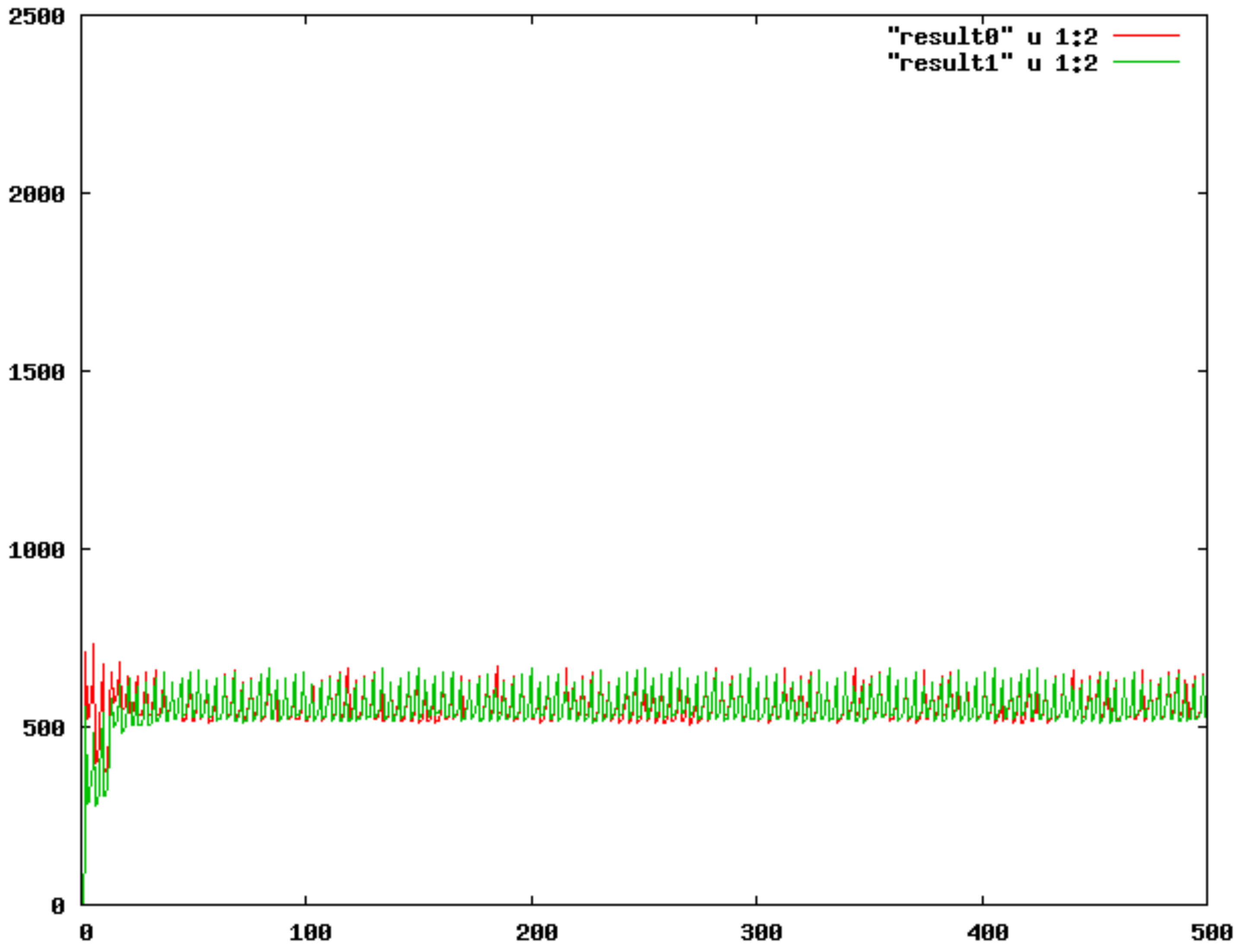
Congestion control: Reno with two flows



Congestion control: BIC with two flows



Congestion control: Hamilton with two flows



Checking and setting congestion control

- To get a list of congestion control algorithms that are available in your kernel, run:

```
$ sysctl net.ipv4.tcp_available_congestion_control
net.ipv4.tcp_available_congestion_control = cubic
reno bic
```

- To know which is the congestion control in use

```
$ sysctl net.ipv4.tcp_congestion_control
reno
```

- To set the congestion control

```
sysctl -w net.ipv4.tcp_congestion_control=cubic
```

Final considerations

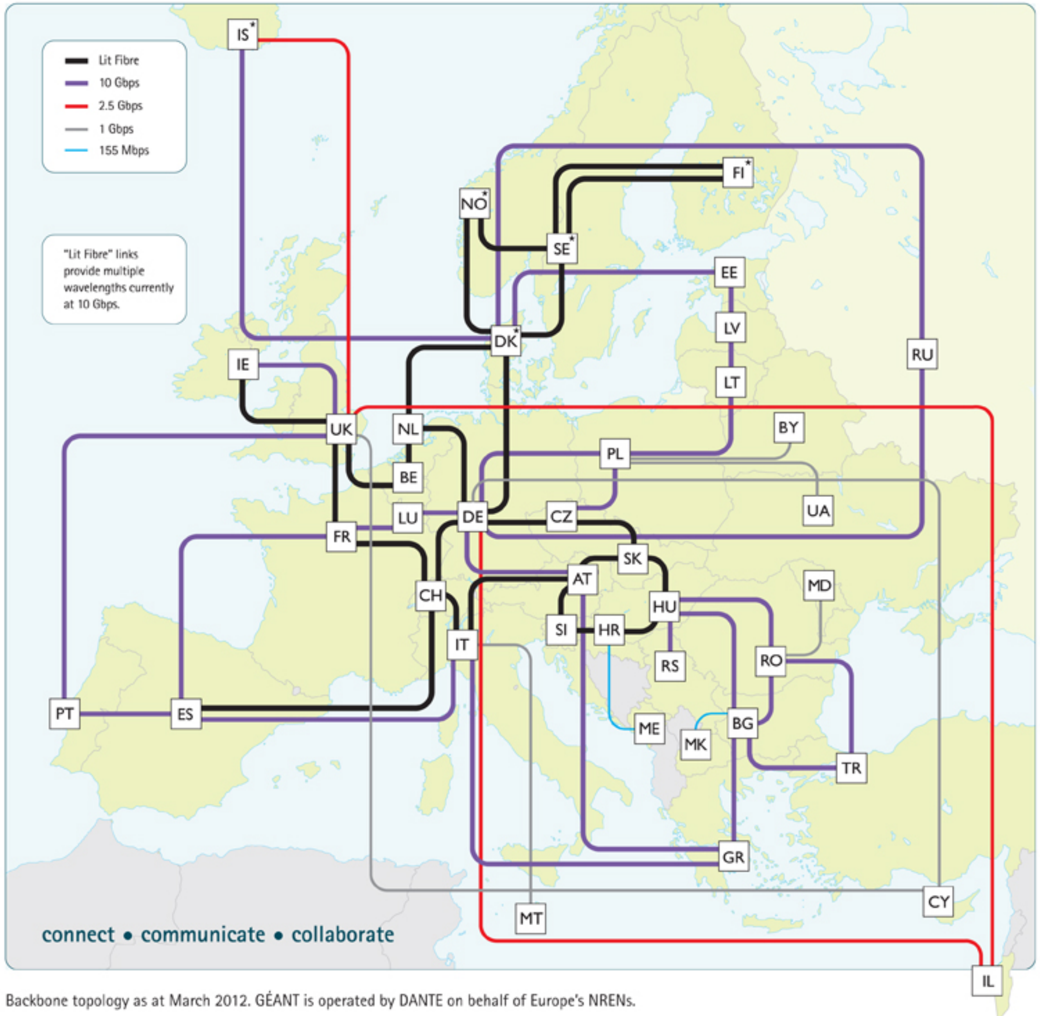
- *Large MTUs*:
  - Linux host is configured to use 9K MTUs
  - But the connection is using 1500 byte packets,
  - ➔ then it is actually needed 9/1.5 = 6 times more buffer space in order to fill the pipe.
  - ➔ In fact some device drivers only allocate memory in power of two sizes, so it could be even needed 16/1.5 = 11 times more

Final considerations (cont.)

- *Very large BDP paths*:
  - For very large BDP links (>20 MB) there could be some Linux SACK implementation problem.
  - Too many packets in flight when it gets a SACK event
    - ➔ *too long to locate the SACKed packet*
    - ➔ *TCP timeout and CWND goes back to 1 packet.*
  - Restricting the TCP buffer size to about 12 MB seems to avoid this problem
    - *but clearly limits the total throughput.*
  - Another solution is to disable SACK

GEANT Slides

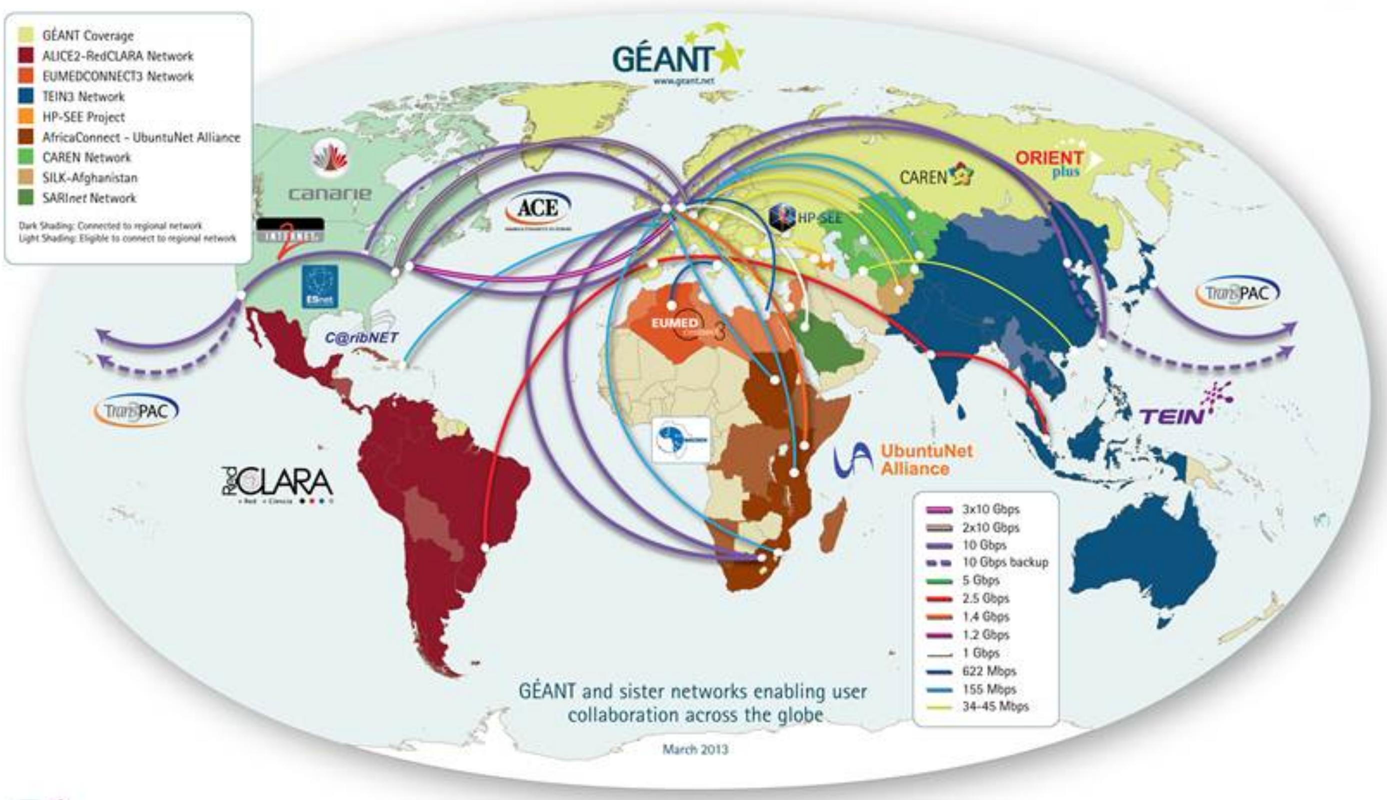
Europe's 100Gbps Network
- e-Infrastructure for the "data deluge"



Backbone topology as at March 2012. GÉANT is operated by DANTE on behalf of Europe's NRENs.

- Latest transmission and switching technology
- Routers with 100Gbps capability
- Optical transmission platform designed to provide 500Gbps super-channels
- 12,000km of dark fibre
- Over 100,000km of leased capacity (including transatlantic connections)
- 28 main sites covering European footprint

GÉANT Global Connectivity
- at the heart of global research networking



DANTE www.dante.net GÉANT www.geant.net

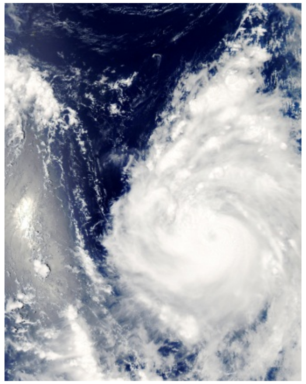
connect • communicate • collaborate

GÉANT is co-funded by the European Union within its 7th R&D Framework Programme.

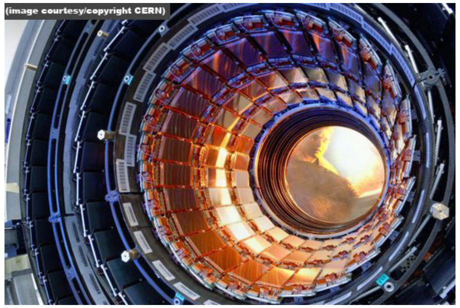
GÉANT connects 65 countries outside of Europe, reaching all continents through international partners

Supporting the growth of R&E Communities
- transforming how researchers collaborate

- GÉANT delivers real value and benefit to society by enabling research communities to transform the way they collaborate on ground breaking research



Together with Europe's NRENs, GÉANT connects 50 million users in 10,000 institutions across Europe

Health and Medicine | **Energy** | Environment | **Particle Physics**
Radio Astronomy | Arts & Education | **Society**

Innovation through collaboration
- for delivery of advanced networking services

- Building the GÉANT "eco-system" through development and delivery of a world-class networking service portfolio:
  - Flexible connectivity options & test-bed facilities
  - Performance tools & expertise
  - Advanced AAI, cloud and mobility services

- Collaborative research into state-of-the-art technology
  - network architectures - mobility, cloud, sensor, scientific content delivery, high-speed mobile
  - identity and trust technologies
  - paradigm shifts in service provisioning and management
  - influencing global standards development

- **Open Calls** to widen the scope and agility for innovation

Delivering innovative services to end users, their projects and institutions across Europe and beyond: secure access to the network and resources they need, when and where they want it.