SCALE and heterogeneity

Grid Computing (M)

Lecture 16

UNIVERSITY of GLASGOW

Case Study 2

Large delay-bandwidth product networks

The situation

- In Grid environments, access to distributed data is very important
- Distributed scientific and engineering applications require:
  - Transfers of large amounts of data between storage systems
  - Access to large amounts of data by many geographically distributed applications and users for analysis, visualization etc
- What mechanisms do we have at our disposal?
  - FTP: standard protocol that exploits TCP to move data files: get and put commands for pull and push mode transfers
  - HTTP: standard protocol for fetching files (pull mode) that was designed for use with web content; layered over TCP
  - Bulk transfer-specific protocols (SRB, BBFTP, …): specially designed protocols for providing access to scientific data sets; again, layered over TCP

The problem

- Core networks support high bandwidths edge-to-edge [o(10 Gbit/sec)]. As the interconnection bandwidths for end systems continues to grow (100 Mbit/sec previously, 1 Gbit/sec now, 10 Gbit/sec in the near future), users expect to be able to exploit large portions of this bandwidth
- For example, if there is a 1 Gbit/sec path end-to-end available in hardware, a user should be able to consume most of that bandwidth for a data transfer if there are no other competing flows along the path.
- If the one-way delay along such a path is τ seconds and the bandwidth along the path is β bits/sec, then the delay-bandwidth product is $\tau \times \beta$ bits – i.e. $\tau \times \beta$ unacknowledged bits can be in flight along the path

The problem (2)

- For example, assume that a 10 Gbit/sec path is available end-to-end, and that the one-way delay is 0.4 seconds; this means that 4 Gbits/500 MBytes of unacknowledged data can be in transit between sender and receiver
- Each of the mechanisms on slide 3 transfers each file over a single TCP flow; therefore, the file transfer performance is dependent upon the steady-state, bulk data flow characteristics of TCP
- Therefore, the primary question is whether TCP will permit a single flow to exploit the full capacity of the network in steady state
- And if not, how should we address the problem?

TCP

- Original goals of the TCP protocol
  - Provide reliable end-to-end transmission of data
  - Minimize the time required to transmit all of the data
  - All of this is achieved by the end systems, w/o involvement from the core routers
- How does it do this?
  - During connection establishment, the receiver of the data indicates the receive window for this flow – i.e. the number of bytes that it can buffer
  - Reliable transmission is achieved using a sliding window protocol for flow control – i.e. after sending a window's worth of data, the sender must wait for an ACK from the receiver
  - Each ACK indicates the number of the last contiguous byte in the stream that was successfully received
  - Upon receipt of an ACK, the sender is able to send more bytes up to the window size
  - Transmission time can be reduced if a larger window size is used

TCP (2)

- This particular approach caused congestion collapse of the Internet at the end of the 1980's:
  - Senders were sending their bytes into the network as fast as the receive window would allow
  - Congestion would occur at some router, causing packets (bytes) to be dropped
  - Hosts would time out waiting for ACKs and then retransmit their bytes, exacerbating the problem
- What's missing?
  - The senders were not determining the capacity available in the network, thus overloading the network given the receive window information from the receivers
  - Note also that there was nothing in the original design that indicated a requirement for fair sharing of the network resources

TCP Congestion Control (1)

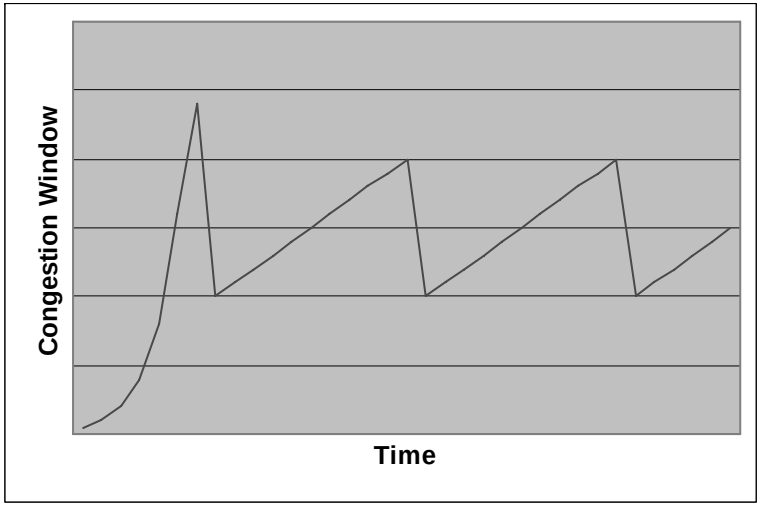
- Revised goals of the TCP protocol
  - Provide reliable end-to-end transmission of data
  - Enable fair sharing of network resources by multiple flows
  - Minimize the time required to transmit all of the data on a single flow commensurate with the fairness criterion
  - Still achieved by end systems
- How can this be done?
  - A sender must determine how many packets it can safely have in transit given the available capacity in the network – i.e. the congestion window
  - As before, it uses the arrival of an ACK to signal that one of its packets has left the network, and it can safely insert a new packet into the network (this is known as self clocking)
  - The difficulty is determining the available capacity in the first place, and to change the congestion window over time as other TCP flows come and go

TCP Congestion Control (2)

- Basics of TCP Congestion Control
  - Must determine the congestion window, decreasing it when the level of congestion in the network goes up and increasing it when the congestion level goes down
  - It turns out that the overall system will be stable if the reduction in the congestion window upon congestion increase is much more drastic than the increase in the window when the congestion decreases
  - It has been mathematically shown that stability is only attained if the window changes conform to additive increase, multiplicative decrease (AIMD) (it is a necessary condition)
  - Simply put, at every time when the congestion window is evaluated, if no additional congestion has been detected, then the congestion window is increased by a fixed amount (additive increase); if, on the other hand, additional congestion has been detected, then the congestion window is halved (multiplicative decrease)

TCP Congestion Control (3)

- The "slow start" aspect of the protocol causes the congestion window to grow exponentially until congestion is detected
- In steady state, the congestion window shows a sawtooth behaviour over time



- How does the sender determine if the network is congested?
- **ASSUMPTION**: the primary reason that a packet is not delivered, and that a timeout occurs, is that the packet was dropped due to congestion; therefore missing ACKs (or duplicate ACKs) are indicative of congestion
- How often does TCP adjust the congestion window? Once per round trip time between sender and receiver.

So, what's the problem?

- Recall that for a large delay-bandwidth product network, we could have 500 MBytes of unacknowledged data between sender and receiver; assuming 1000 byte packets, this means 500,000 packets in transit
- The one-way delay was 0.4 seconds, which implies that the RTT is 0.8 seconds
- Now, assume that the window size reached 500,000 packets and a loss was detected, so that the window size is reduced to 250,000
- At the rate of adding 1 packet to the window size for each 0.8 seconds, it will take over **55 hours** for the window to get back to 500,000 again!
- During this entire time, the flow is using the network resources sub-optimally!
- Bottom line – the congestion control protocol must drive the system into a congested state to determine that congestion exists, and then drastically reduces the window – optimal situation would be for the window to be maintained just below the onset of congestion

Potential solutions – GRID FTP

- GRID FTP: a set of extensions to standard FTP that uses multiple, parallel flows for transferring a single file
  - i.e. Multiple TCP streams between a single source and destination; the file to be transported is broken up into blocks and distributed across the multiple streams
- Parallel data transfer require support for out-of-order data delivery
- Extended block mode supports out-of-sequence data delivery
- Extended block mode header

Descriptor 8 bits	Byte Count 64 bits	Offset 64 bits

- Descriptor is used to indicate if this block is a EOD marker, restart marker etc

Potential solutions – GRID FTP (2)

- Does this solve the problem of exploiting the large delay-bandwidth product?
  - Instead of one flow trying to use up all of the network capacity, N flows are each trying to use 1/N of the capacity
  - If the N flows were at different stages of their steady-state behaviour, then one would expect some to be near to W_{max} while others are near to $W_{max}/2$, with others spread in between
  - Unfortunately, all N flows are started at the same time, and essentially synchronize their behaviour, so the anticipated averaging behaviour in the previous point is unlikely
  - Suppose that we have 10 flows; in a fair world, then each flow would have 50,000 unacknowledged packets en route at the same time; if congestion on one of these flows is detected, the window is reduced to 25,000 packets, and the flow will require 5.5 hours to get back to W_{max}
  - The congestion control mechanism still suffers from the same two problems:
    - It must drive the system into a congested state to detect congestion
    - Its response to congestion is too drastic

TCP Vegas – smoother congestion response

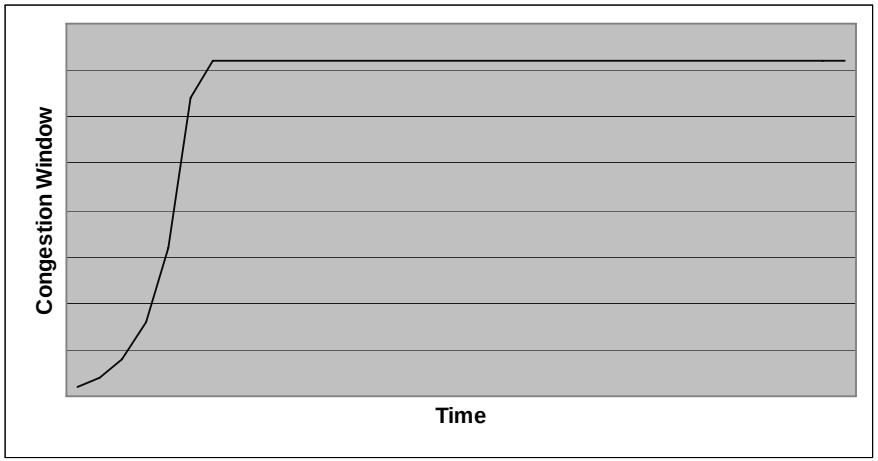
- The traditional congestion control algorithm, called TCP Reno, relies upon packet loss indications to trigger its control response – i.e. it has too little information to be able to PREDICT when the flow is likely to encounter congestion; it is a reactive control mechanism
- If we can obtain additional information such that we can PREDICT the onset of congestion, we should be able to avoid the multiplicative decrease of the congestion window except in very severe congestion situations
- TCP Vegas monitors the round trip delay experienced by packets and their ACKs to provide this additional information needed to proactively avoid congestion – the measure of queueing delay
- In essence, Vegas attempts to keep the congestion window just below the W_{max} that would trigger packet loss and multiplicative decrease of the congestion window

How does Vegas congestion avoidance work?

- A given flow's *BaseRTT* is defined to be the RTT of a packet when the flow is not congested
- Then the expected throughput is *Expected=WindowSize/BaseRTT*
- Calculate the current actual sending rate as the number of bytes transmitted between the time that a packet is sent and its ACK is received divided by the RTT
- *Difference = Expected – Actual*; note that *Difference* ≥ 0
- Define two thresholds *low* < *high*, corresponding to having too little/too much extra data in the network
- If *Difference* < *low*, increase the congestion window linearly during the next RTT
- If *Difference* > *high*, decrease the congestion window linearly during the next RTT
- Leave the window unchanged if *low* < *Difference* < *high*

So, does this solve the problem?

- It certainly avoids the sawtooth behaviour in steady state
- If congestion is detected, though, it still performs multiplicative decrease



- The effectiveness for exploiting large delay-bandwidth product networks depends upon the choice of the parameters *low* and *high*
- The additive increase to the window will still be too slow in adjusting to large increases in available capacity
- In order to truly address the problem, one must consider the flow-level design of the congestion avoidance algorithm to achieve high utilization, low queueing delay and loss, fairness, and stability

FAST TCP

- Several individuals have attempted to address this problem by looking at the flow-level design:
  - FAST TCP – California Institute of Technology
  - HTCP – Hamilton Institute
  - HSTCP – Cambridge University
- FAST TCP is essentially a variant of TCP Vegas that is focused on high delay-bandwidth product networks
- FAST updates the congestion window according to the following equation

$$w \leftarrow \min \{ 2*w, (1 - \gamma)*w + \gamma *(BaseRTT/RTT*w + \alpha)\}$$

- where $0 < \gamma <= 1$ and α is a positive protocol parameter that determines the number of packets queued in routers in equilibrium along the flow's path

Characteristics of FAST TCP

- When the congestion window is very small relative to the available capacity, it experiences multiplicative increase
- Once the window is within half of w_{max}, the window grows by a weighted average of the current window, the ratio of BaseRTT and current RTT, and the total queueing capacity of the routers carrying the flow
- In particular, if BaseRTT/RTT ~ 1, then
  $w := \min \{2w, w + \gamma * \alpha\}$
- If BaseRTT/RTT ~ 0, then
  $w := \min \{2w, w + \gamma*(\alpha - w)\}$
- Efficacy of the mechanism depends upon good choices for γ and α

How well does FAST work?

B. Case study: dynamic scenario I

In the first dynamic test, the number of flows was small so that throughput per flow, and hence the window size, was large. There were three TCP flows, with propagation delays of 100, 150, and 200ms, that started and terminated at different times, as illustrated in Figure 2.

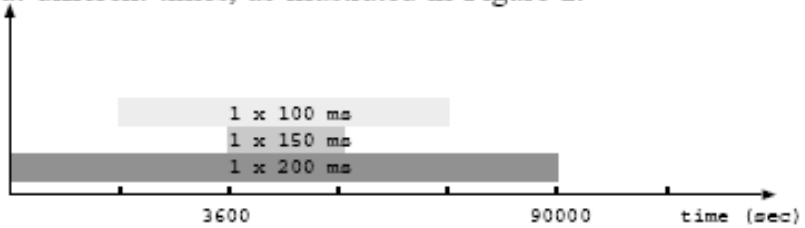
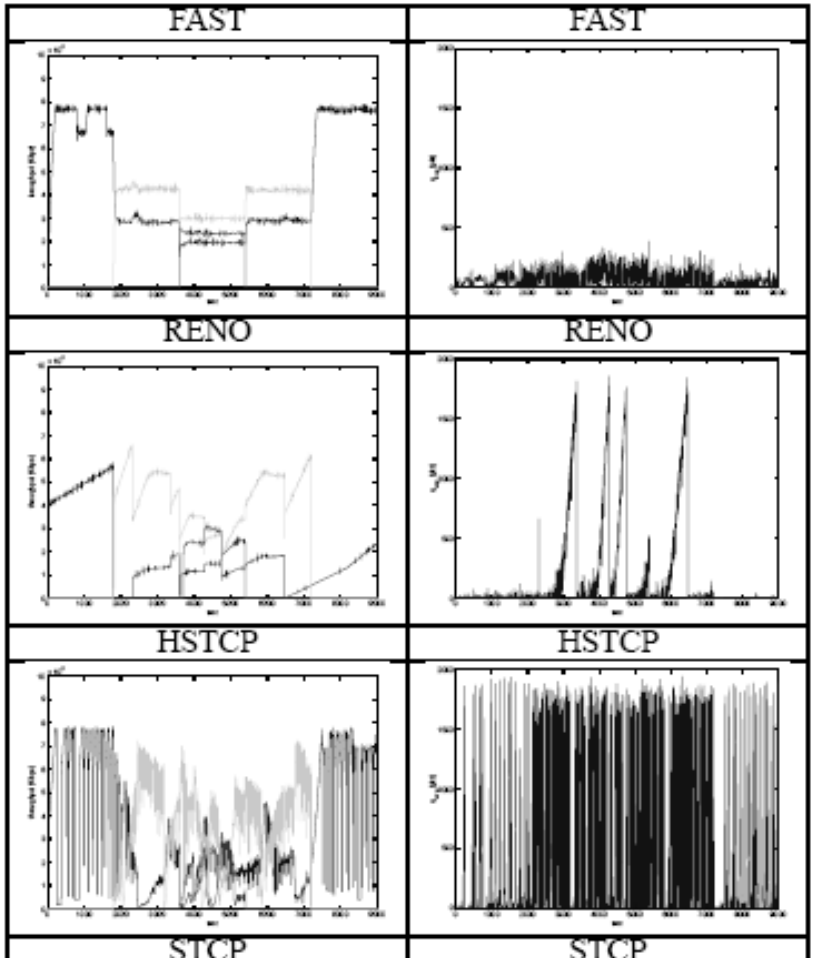


Fig. 2. Dynamic scenario I (3 flows): active periods.

For each dynamic experiment, we generated two sets of figures. From the sender monitor, we obtained the trajectory of individual connection throughput (in Kbps) over time. They are shown in Figure 3. As new flows joined or old flows left, FAST TCP converged to the new equilibrium rate allocation rapidly and stably (left column). Reno's throughput was also relatively smooth because of the slow (linear) increase between packet losses. It incurred inefficiency towards the end of the experiment when it took 30 minutes for a flow to consume the spare capacity made available by the departure of another flow. HSTCP, STCP, and BIC-TCP responded more quickly but also exhibited significant fluctuation in throughput.



How well does FAST work?

of the protocols is similar in this case as in scenario I, and in fact is amplified as the number of flows increases.

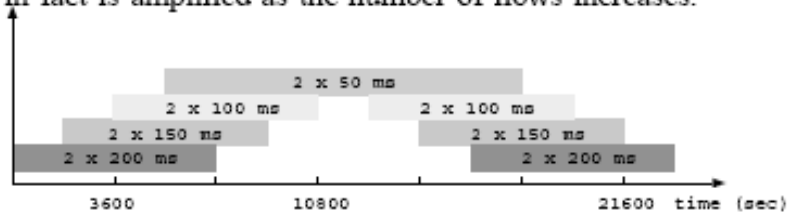
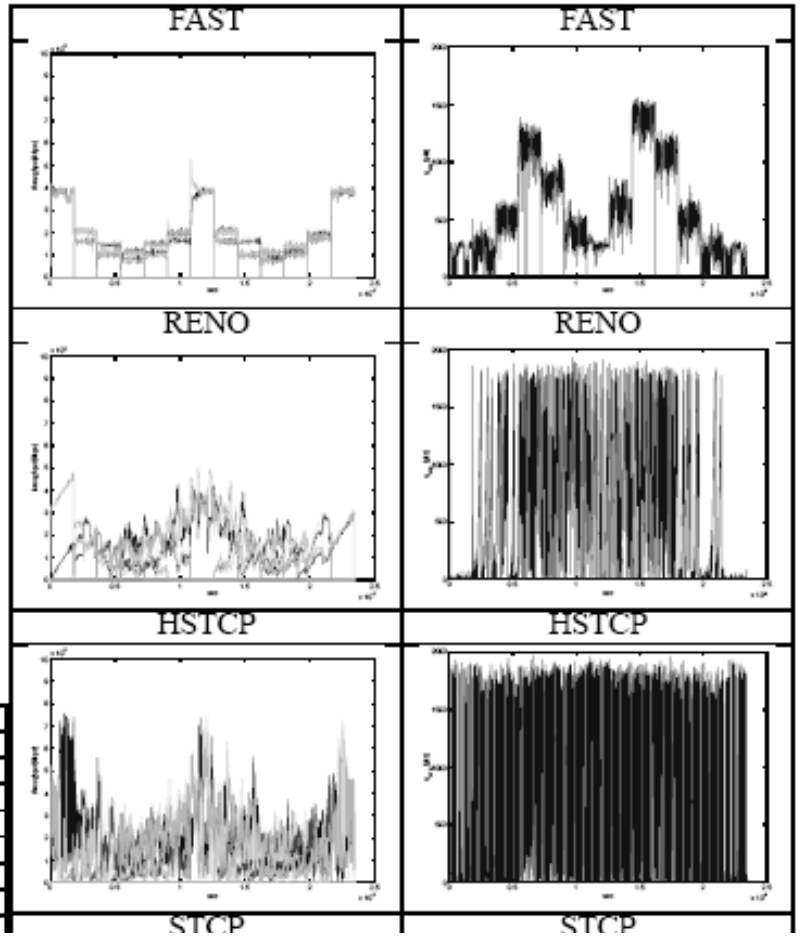


Fig. 4. Dynamic scenario II (8 flows): active periods.

Specifically, as the number of competing sources increases, stability became worse for the loss-based protocols. As shown in Figure 5, oscillations in both throughput and queue size are more severe for loss-base protocols. Packet loss was more severe. The performance of FAST TCP did not degrade in any significant way. Connections sharing the link achieved very similar rates. There was a reasonably stable queue at all times, with little packet loss and high link utilization. Intra-protocol fairness is shown in Table IV, with no significant variation in the fairness of FAST TCP.

Time (sec)	Sources	FAST	Reno	HSTCP	STCP	BIC
0 – 1800	2	1.000	.711	.806	.999	.979
1800 – 3600	4	.987	.979	.940	.721	.971
3600 – 5400	6	.976	.978	.808	.631	.876
5400 – 7200	8	.977	.830	.747	.566	.858
7200 – 9000	6	.970	.845	.800	.613	.856
9000 – 10800	4	.989	.885	.906	.636	.973
10800 – 12600	2	.998	.993	.996	.643	1.000
12600 – 14400	4	.989	.782	.843	.780	.936



Summary

- Effective utilization of high delay-bandwidth product networks requires that the TCP congestion control algorithm be able to PREDICT the onset of congestion in order to avoid the multiplicative decrease required for stability
- A proactive congestion control algorithm needs additional information beyond packet loss to do its job; packet loss indications are already too late – i.e. congestion has already occurred
- In order to avoid the slow growth of the congestion window resulting from additive increase, FAST has introduced a multiplicative increase aspect to its algorithm; this is moderated by terms that depend upon the prediction of onset of congestion
- FAST's scalability has resulted from:
  - Smooth out control response
  - Paradigm shift