Homework 3

Due: 22 April, 2011, 4pm

Problem 1 - TCP

a. What is the difference between TCP Tahoe and TCP Reno? Why is Reno an improvement?

> TCP Reno introduces Fast Recovery: on triple duplicate ACKs, the congestion window is halved instead of reset to 1; the missing segment is retransmitted immediately (fast retransmit); and the algorithm remains in congestion avoidance mode, growing the window using AIMD. On timeout, TCP Reno is identical to TCP Tahoe. TCP Reno is an improvement because the congestion window remains larger and more data can be sent. timeout, fast retransmit higher

b. Why does TCP ignore retransmitted segments when calculating the RTT?

> When an ACK for the retransmitted segment returns, it is not possible to determine whether the ACK is for the first transmission of the segment, or for a later retransmission. Therefore, it is not possible to calculate the RTT for the segment.

c. We saw that an approximation for the average throughput of TCP is given by

$$T = \frac{1.22MSS}{RTT\sqrt{L}},$$

where L is the loss rate. For a path with 1,500 byte segments, RTT of 100ms, what is the loss rate if we want to saturate a link of 100Mbps? What about a link of 10Gbps?

First, we re-write the formula to solve for L,

$$L = \left(\frac{1.22 \times MSS}{T \times RTT}\right)^2$$

Next, we convert our MSS to bits or our throughput to bytes to solve for 100Mbps:

$$\left(\frac{8 \times 1.22 \times 1500}{100 \times 10^6 \times 100 \times 10^{-3}}\right)^2 \approx 2.143 \times 10^{-6}$$

And for 10Gbps:

$$\left(\frac{8 \times 1.22 \times 1500}{10 \times 10^9 \times 100 \times 10^{-3}}\right)^2 \approx 2.143 \times 10^{-10}$$

Note that this probability is very low, which is difficult to achieve in practice. For high-speed Ethernet transmission, it is common to use much larger segment sizes – frames up to 9000 bytes are known as "Jumbo Frames."

d. Suppose you have two TCP connections sharing a bottleneck link on the network: connection A has an RTT of 50ms and connection B has an RTT of 100ms, and they are both operating in TCP Reno congestion avoidance mode. In the long run, will they reach the same throughput? Justify your answer using a Chiu-Jain phase plot (lecture 11). Assume the flows start at a point well below the full utilization of the link, and that they both reduce their rates by half whenever they cross the full utilization line. Hint: draw what happens at every increment of 50ms.

   No, they will not reach the same throughput. A's throughput will be twice B's throughput. This is the result as outlined in slide 31 of lecture 12 ("Increasing cwnd Faster"), except that A is not cheating by increasing its cwnd by 2 each RTT – it has an RTT which is half of B's.

Problem 2 - TCP and HTTP

Suppose that a Web browser has to download 30 objects from the same server to properly display a page. Assume that these objects are all 15KB long and that the MSS for the connection is 1KB. The communication between the client and the server has to go through a bottleneck link that has a total bandwidth of 1MB/s, and there is one TCP flow already present in this bottleneck link. Assume that the RTT for your flows, as well as for the other flow already there, is 100ms.

In this problem you have to answer the following question: how many TCP connections should the browser open to the server to finish downloading all of the files in the fastest way?

We are going to assume an idealized version of TCP, that has only two phases after regular connection establishment: slow start and constant rate. You can assume that TCP will magically know the size of the `cwnd` that will lead to the "fair" sharing of the bottleneck link. In other words, assume that TCP will start each connection in slow start and then stop growing the window once it reaches the "fair" size of the congestion window. You can also ignore the closing of the connections.

a. If yours were the only flow in the bottleneck link, what would be the size of the congestion window that would maximally utilize the bottleneck link? What would the window size be if you added the other flow we mentioned above? (Assume this extra flow is going to be there for all subsequent questions).

   If we are the only flow in the bottleneck link, then to fill the pipe we need a congestion window of 1024 KB / sec * 0.1 sec = 102.4 KB. To share fairly with the other flow, we will need to split the bandwidth evenly, resulting in a congestion window of 51.2 KB.

b. How large would an object have to be for your connection to reach this window size?

For our connection to reach this window size, we consider what happens during the slow start regime: On each RTT, the window size (in bytes) doubles, until we cross the slow-start threshold. This is equivalent to increasing the window size by one MSS on each ACK, since we receive an ACK for each of the w/MSS packets which were sent. Our window's initial starts out at 1 MSS (1 KB). After the first RTT, it is doubled to 2 KB. Then, 4 KB, 8 KB, 16 KB, and 32 KB. After the ACKs from the data sent with the 32 KB cwnd, the cwnd would be doubled to 64 KB, but our magic TCP would then set it to 51.2 KB for fair-sharing. Thus, we needed to have sent (1+2+4+8+16+32) = 63 KB to lead to a full-sized cwnd.

c. What would be the size of the congestion window if you now have n flows (plus the one extra flow that is not yours and it already there) ?

TCP is designed to share the bandwidth evenly across all $n+1$ flows. Therefore, each flow's cwnd will be $\frac{102.4KB}{n+1}$.

d. If the browser can open just one concurrent TCP connection to the server, using HTTP/1.0, how long would it take to transfer all of the 30 objects?

Since each object is 15 KB, we will send with the cwnd's 1, 2, 4, and 8 (totaling 15 KB), which takes 4 RTTs. There is an additional RTT to open the TCP connection (we assume the HTTP request and the last ACK of the three-way handshake are the same packet), for a total of 5 RTTs per object. With 30 objects and a 100ms RTT, this process will 15 seconds.

e. What if the browser switches to HTTP/1.1 and requests a persistent connection to the server?

With a persistent connection, TCP will only go through the slow-start phase once. The first object will require 4 RTTs to be transmitted (1 MSS, 2 MSS, 4 MSS, and 8 MSS). On the request for the second object (ACK'ing the last of the 8 MSS transmissions, which ended the first object), the cwnd will now be 16 MSS, which is larger than the size of the object (15 MSS). Therefore, from the second object onward, each object will require 1 RTT for retrieval. This gives us a total of 29 RTT + 4 RTT + 1 RTT for the TCP handshake = 34 RTT or 3400 ms.

f. What if the browser adds HTTP/1.1 pipelining? (Assume all of the 30 requests can fit in one segment), how long does it take now?

With both a persistent connection and pipelining, TCP will only go through the slow-start phase once and all other bytes will be transferred *together* at the fair rate (which has a cwnd of 51.2 KB). We have a total of 30 objects $\times$ 15 KB/object to transmit (450 KB). As calculated above, the first 63 KB will go out during the slow-start phase. This process took 600ms. The remaining $450 - 63 = 387$ KB will require $\lceil \frac{387}{51.2} \rceil = 8$ RTTs. Finally, we add the additional RTT to establish the connection, for a total of 15 RTTs or 1500 ms. This is an order of magnitude improvement over part d!

Remember, from the perspective of the sending server, TCP provides a reliable byte-oriented stream. The HTTP server will write out the responses for all 30 objects to the stream and TCP will manage the transmission of the stream.

g. Now the browser gets greedy, and decides to open 30 parallel connections to the server, requesting one object on each connection. Again assume that TCP will do slow start, and magically stop growing the window once (and if) it reaches the fair size. How long will it take to transfer all files?

With 30 parallel connections, the fair size will be 102.4 KB / 31 = 3.3 KB. Our window sizes on each round-trip will be 1 KB, 2 KB, 3.3 KB, 3.3 KB, ... Including the connection setup, the first 3 KB of each object will take 3 RTTs. Transmitting the remaining bytes requires $\lceil \frac{15-3}{3.3KB} \rceil = 4$ RTTs, for a total of 7 RTTs or 700 ms.

h. How many connections should the browser open (between 1 and 30) to minimize the total transfer time?

The browser should open 30 connections to minimize the total transfer time. We can see this because we want the browser to take as much of the bottleneck link as possible as quickly as possible. By opening a large number of connections, we make the "fair" congestion window as small as possible, which means that TCP slow-start will reach the "fair" window as fast as possible.

Problem 3 - DNS

a. Give two reasons for which the DNS system scales to serve the entire Internet.

The DNS system can scale to serve the entire Internet because:

- It is a distributed and hierarchical system. Changes to DNS do not need to be propagated to all Internet hosts, which also do not have to store records for all other hosts. Even in the absence of caching, no single server needs to contain the entire DNS database.
- DNS lookups are mostly reads, which makes caching very effective. The DNS system can also tolerate inconsistencies, which allows distributed caching near end-hosts to be effective.
- DNS delegates authority for zones in order to spread and share responsibility for record updates.

- The DNS record format allows for multiple IP addresses to be returned for a lookup, which can be used to implement load-balancing.

b. Give a scenario in which you would set the TTL for a DNS mapping to a large value. What is the down side?

> If you are running a service for which you get many DNS requests (stable service) and cannot support a high load, you will want the answers to be cached for a long time. The down side is that if you want to change the mapping, you will need to wait for a long time until the TTL expires on all the records for your change to fully propagate.

c. Give a scenario in which you would set the TTL to a very small value. What is the down side?

> As a first example, consider the design of Akamai's content distribution network, which uses DNS responses to do load-balancing across many requests, by sending different clients to different IP addresses on each lookup. This is done with small TTL values and changing records. The down side is that the DNS servers will have to support a high request load. A second example is dynamic DNS records for home internet connections. If you wish to run a website such as "www.myhome.org" from your home connection, you may need a small TTL value so that IP addresses changes will propagate quickly.

Problem 4 - Consistent Hashing

We talked about consistent hashing in class. Consider two situations, and explain why CH is better than modulo hashing for each one.

a. When you have a fixed set of cache servers implementing consistent hashing, and a population of clients who have incomplete views of the system, *i.e.* each client only knows about a fraction of the servers.

> Consistent hashing is better than modulo hashing in this scenario because it has a lower "spread." That is, clients with an incomplete view of the system will continue to map cached objects to the same (consistently) hashed value, which means that the objects will be stored "close" together on the ring, even if stored at different servers due to the incomplete view. In modulo hashing, there is no guarantee that two different incomplete views will map objects close together in the hashed space.

b. When you have a set of cache servers that changes (nodes come and go).

> Only a few items stored in the distributed hash table will need to be transferred when the cache servers come and go. In traditional hashing, the keys

are stored modulo the number of storage buckets – removing one storage bucket will affect all keys.

Problem 5 - P2P

Chord In class we discussed how the STABILIZE and NOTIFY operations in Chord will handle joins.

a. Explain why that doesn't quite work for node failures. As an example, take the ring in slide 39 of lecture 16, and assume node 50 fails.

  Nodes only keep a single predecessor and successor. The STABILIZE and NOTIFY operations would not know how to reach their new successor and predecessor after the node between them fails.

b. What would you change in the description of the protocol to make the ring stabilize again after a failure? (You can, for example, change what nodes do when they detect that their predecessors fail, or change the set of nodes to which nodes store pointers).

  In common practice, nodes increase the size of the set of nodes to which they store pointers – that is, they store multiple successors which can be used if the primary successor fails.

BitTorrent

a. What does a *tracker* do in BitTorrent? How would you use a DHT to replace BitTorrent trackers? Who would be the peers? What would be the keys, and the values stored for each key? How would you bootstrap the system?

  The BitTorrent tracker contains a list of peers in the BitTorrent swarm from which you can download pieces of the file. In a trackerless BitTorrent, a DHT can be used to lookup which peers (the values) are participating in the swarm for each file (the keys). The peers in the DHT could be all nodes participating in any BitTorrent swarm. To bootstrap the system, an existing general DHT could be used to store values for trackerless BitTorrent clients. The DHT could also be pre-loaded with values from existing torrent files distributed by a community.