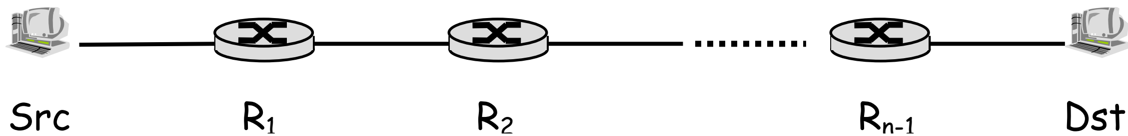
CS 5565 Sample Midterm Spring 2006

1 Cut-through Switching (13 pts)

Unlike store-and-forward, cut-through switching or forwarding allows a node inside a network to start forwarding a packet before it has been received in its entirety. Consider a path in a network that connects a host "Src" to a host "Dst" via n links (1..n) through n-1 intermediate routers, each employing cut-through forwarding.
Ignore processing delay and assume that there is no queuing delay.



Assume that all links have the same length d and that the speed of light in the link medium is s. Assume that each link has bandwidth R and that packets of length L bits are sent through the network. Each switch can start forwarding the packet after L_c bits have been received.

a) (5 pts) What is the total latency for a single packet (counted from first bit sent at Src until last bit received at Dst)?

$$Latency = t_{prop} + t_{trans} = n\frac{d}{s} + \frac{L}{R} + (n-1)\frac{L_C}{R}$$

b) (4 pts) By how much did cut-through switching reduce the latency for sending a single packet end-to-end, compared to store-and-forward?

Store and forward would mean that $L_c = L$*, so the savings are*

$$\Delta_{Latency} = (n-1)\frac{L - L_C}{R}$$

c) (4 pts) Now suppose a message of F bits is sent, which is sent as multiple packets. Will the latency savings increase, decrease, or stay the same as in the case of a single packet? Justify your answer.

The savings would be the same, because after the first bit of the first packet arrives, the remaining bits will in both cases arrive after F/R seconds (ignoring headers), because the source and all routers will be continuously operating.

[If you include headers, it will take longer – but the transmission delay would be increased in both cases by the same amount, but you'd need those headers no matter whether the switches operate using cut-through or not.]

2 Application Layer Protocols: HTTP 1.1 (9 pts)

In 1994, Jeff Mogul from DEC's Western Research Laboratory proposed the use of persistent and pipelined connections in HTTP 1.1. Some people at the time doubted the necessity of connection persistence and pipelining and suggested the use of multiple, non-persistent connections as an alternative.

a) (6 pts) Name two advantages of persistent, pipelined connections over multiple connections.

Using a persistent connections instead of multiple connections reduces the number of times a connection must be established. This reduces not only the number of necessary TCP segments, and as such the number of roundtrips required until a document is retrieved, it saves processing time, and it reduces the number of TCP connections a server has to manage at any given time. In addition, being able to reuse a connection means that the slow-start cost of TCP congestion control algorithm are paid only once, reducing delay. Server authentication also needs to be done only once (if authentication is done via SSL.)

See Padmanabhan, V. N. and J. Mogul, "Improving HTTP Latency," Computer Networks and ISDN Systems, v.28, pp. 25-35, Dec. 1995. Slightly Revised Version in Proceedings of the 2nd International WWW Conference '94: Mosaic and the Web, Oct. 1994. http://www.geocities.com/archu_s_r/itp/httplatency.pdf

Note that multiple connections do give you the ability to submit new requests before all currently running requests have completed, so this is not an advantage of pipelined persistent connections over them.

b) (3 pts) Name one advantage of multiple connections over persistent, pipelined connections.

Multiple connections can reduce average latency, if the server uses a server model that can handle multiple requests simultaneously. For instance, consider a case where a page contains one large and many small images, and assume that the client happens to submit the request for the large image first – then the small images will all arrive after the large page. If multiple connections are used, most small images will have completed by the time the large image is received. (In scheduling terms, persistent pipelined connections use a FCFS scheduler, whereas multiple connections would allow for Round-Robin scheduling.)

3 XMPP & Server Models (8 pts)

We had discussed the following options for implementing TCP servers in class:

A – for each connection, a new process is spawned
B – for each connection, a new thread is spawned
C – a constant number of processes that handle connections
D – a constant number of threads that handle connections
E – a single thread that multiplexes multiple connections using select() or a similar mechanism

Which of these models would you use if you had to implement an XMPP server? Give two reasons for your choice. Be specific. (For instance, simply saying "better scalability" is not specific enough – you would have to say what scales better and why.)

The typical use case for an XMPP server would be to maintain long-lived connections that are only sporadically active, say when a stanza is received. This will make it hard to use either approach A or B, because of the large number of processes and threads that the system would need to continuously maintain. On the other hand, C and D would severely restrict the number of users that server could support. (Popular XMPP servers easily support dozens of thousands of users.) This leaves E, which is the most practical approach when it comes to scaling to large numbers of user/connections.

Another reason to use E is that most XMPP requests (once an entire stanza has been received) are typically short and do not involve I/O bound activity (except when the roster database is updated or off-line messages must be stored.) Approach E also allows for easier communication – say if a message is sent from one user to another without using off-line storage, no thread-to-thread communication is necessary.

In practice, one would probably use multiple threads to multiplex the requests coming from multiple connections, without however dedicating a thread to a particular connection.

4 Reliable Data Transmission (20 pts)

a) Reordered Acknowledgements (6 pts)

Assume that an alternating bit protocol is used over an asymmetric channel where data packets always arrive in the order in which they are sent, but acknowledgments may be reordered. (This can happen, for instance, if multipath routing is used in the direction from receiver to sender, but not the other way around.) Give a sender and receiver timeline that shows that an alternating-bit protocol such as rdt3.0 will fail if acknowledgements are reordered. (Show the timeline for the sender on the left, for the receiver on the right, and include the type and sequence numbers of messages exchanged.)

Sender sends Pkt0

Sender retransmits Pkt0

Sender sends Pkt1

Error: sender mistakes reordered, delayed Ack0 for acknowledgement and thinks that 2nd Pkt0 has been received.

Pkt0

Ack0

Pkt0

Ack0

Pkt1

Ack1

Pkt0

⊗ Lost!

Pkt1

Receiver expected Pkt0, and sends Ack0 when it arrives. Expects Pkt1 next.

Receiver discards duplicate Pkt0, resends Ack0.

Receiver acknowledges packet 1, expects Pkt0 next

Receiver will keep waiting for Pkt0, which will never come

Sender

Receiver

b) Flow Control (6 pts)

Your project partner suggests to add flow control to your implementation of Project 1B. He suggests to change the protocol such that the receiver buffers received in-order segments, but delays sending acknowledgements for them until the receiver application has picked up the data that was received. (Assume that a downcall model is used on the receiver side.)

- Explain why this would indeed achieve flow control!

If no acknowledgements are forthcoming, the sender cannot send new data. If the sender retransmits, the receiver can drop the retransmissions as duplicate without requiring buffer space. Recall that the goal of flow control is to avoid a situation where the receiver in a connection runs out of resources (CPU, buffer space) because the sender does not slow down.

- Identify the big disadvantage of this approach and suggest a way to fix it.

Obviously, the sender will continuously try to retransmit the data that wasn't acknowledged, wasting network resources and possibly causing congestion. This could be fixed, for instance, by only retransmitting the oldest unacknowledged

packet and by continuously increasing the timeouts, like Karn's algorithm would do.

- TCP uses persistence timers to avoid deadlock in its flow control scheme. Would you need a similar timer here?

No, because the sender will keep trying to send. Note that if the sender's timeout backs off so much that its value becomes equal to TCP's persistence timer, you have almost the same scenario (except that TCP probes do not contain retransmitted data as all data has been acknowledged.)

c) Connection Management & Two-Army Problem (4 pts)

We had seen that the two-army problem implied that it was impossible to be 100% sure that each side has sent all data it wants to send when a TCP connection is closed.
Does the two-army problem also affect connection establishment?
If not, why not? If so, do we need to worry about it?

Short answer: Yes – but we don't need to worry about it since we will only start transmitting data if a connection establishment was successful.

Long answer: The two-army problem says that it is impossible to devise a distributed algorithm that guarantees that a system reaches agreement in the presence of unreliable communication. It does not say that it is impossible to reach agreement in those systems. It also does not say that it is impossible to notice when agreement has been reached.

The difference between connection establishment & teardown is what this agreement is about. If it's about when to close a connection, we are in a situation where we must reach agreement or else data might be lost. If it's about the fact that a connection has been set up, then all we need to be able to do is realize when that agreement has been reached. Still, if all connection request packets are lost, it is not possible to guarantee that a connection will be established.

Note that "a 3-way handshake is used" is not a correct answer, because if a 3-way handshake could avoid the two-army problem, we could simply use one to disconnect also. Also, note that the two-army problem is not about who wants to disconnect, it's about whether the other side knows that one side wants to close its side of the connection.

d) Long Fat Networks (4 pts)

Long fat networks are networks that have high delay (a "long" pipe) but large bandwidth (a "fat" pipe.) Name one reason why such an environment is challenging for sliding window protocols!

There are many reasons.

A large bandwidth-delay product requires large window sizes, which in turns requires large retransmission buffers at the sender side and large receive buffers at the receiver side. Large window sizes might also mean a large number of retransmissions if a single packet is lost, unless selective acknowledgement is deployed. Large bandwidth by itself means that slow-start congestion control algorithms will take longer to identify the achievable throughput.
Note that "high delays cause more timeouts" is not a fully correct answer, because as long as RTT estimation works, the estimated timeout period would be set high enough to avoid timeouts even for the larger delay. So the number of timeouts does not have to increase, although the impact of an individual timed-out packet will.