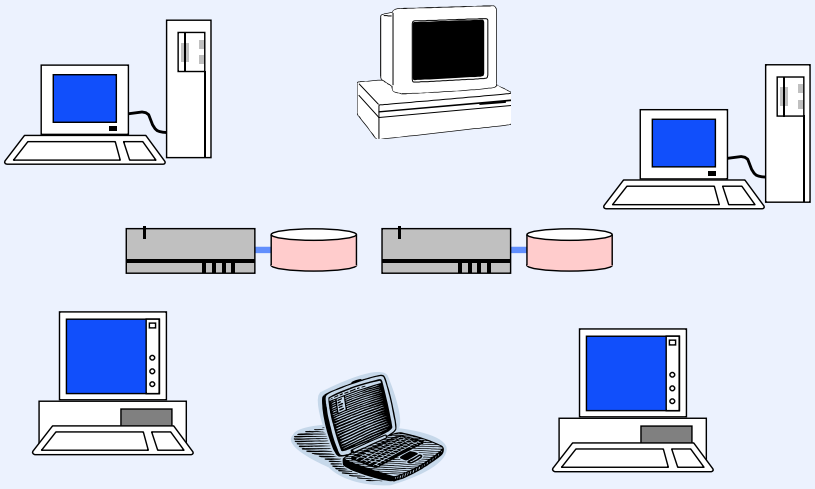
Introduction to Networking

Distributed File Systems

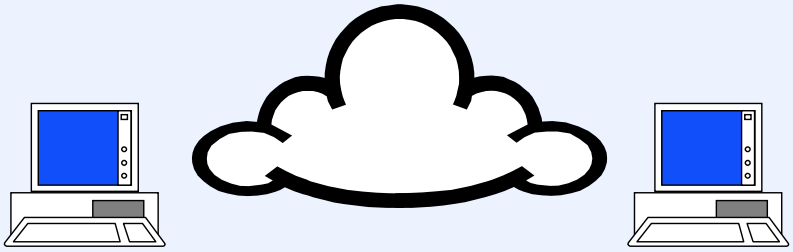


We cover networking as a basis for our discussion, in the next lectures, of remote procedure call protocols and distributed file systems.

What We Cover

- Communication protocols
- Remote procedure call protocols
- **Distributed file systems**

Communication



Data is communicated through some sort of network. We usually try to hide the details beneath several layers of abstraction. Our concern in this lecture is a particular approach to communicating data, known as **packet switching**.

Some Issues

- Quantity: how many are communicating?
  - unicast
  - broadcast
  - multicast
- Quality: how good/reliable is the communication?
  - best effort
  - fully reliable
  - guaranteed bandwidth and delay

Some More Issues

- Naming and addressing
  - www.cs.brown.edu vs. 128.148.32.110
- Routing
- Congestion

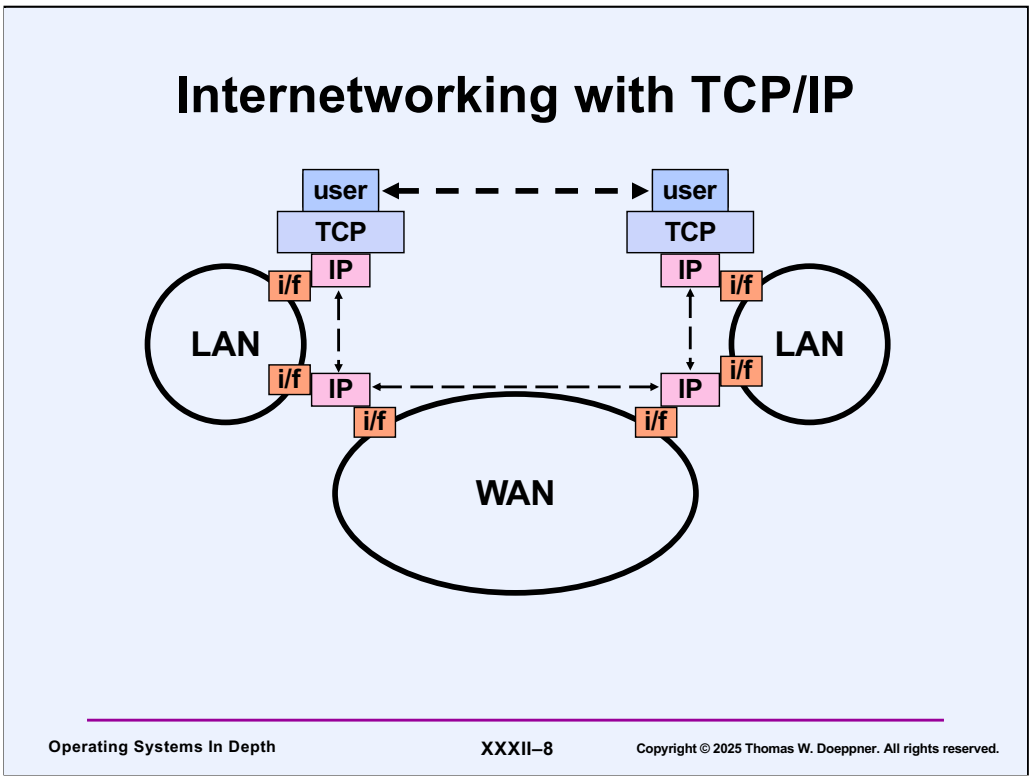
Circuits vs. Packets

- **Circuit switching**
  - **reserve a circuit between communicating entities**
- **Packet switching**
  - **break data into packets**
  - **transmit each packet separately — each packet could travel a different route**

There are (at least) two models for communicating data. The first is known as **circuit switching**. This works in the same way that domestic telephone connections work—to communicate, the two parties first establish (reserve) a circuit between themselves. They then communicate using this reserved communication path.

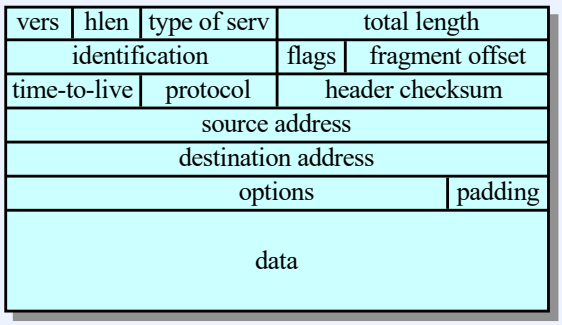
The second approach is known as **packet switching**. In this model no physical connection is established (though one may think of a logical connection as being established). Instead, data is broken up into packets, each packet is transmitted separately, and each packet is independently routed through the internetwork.

The advantage of circuit switching is that once the circuit is reserved, communication is relatively fast and inexpensive. The disadvantage is that you must pay for the circuit even when you are not actively communicating. The advantage of packet switching is that you only pay when you communicate (i.e. send packets). The disadvantage is that more work is required for each packet. An analogy (put forth by proponents of packet switching) is suppose that you are driving from Providence to San Francisco. Using the circuit switching approach, you would first reserve those highways that you will be travelling over for your exclusive use. Once everyone else is off these roads, then you travel (since you've also kicked off the cops, you probably can go pretty quickly). With packet switching, you drive from here to San Francisco, competing for highways with everyone else, and you make local decisions about which is the best road to take (and you have to watch out for cops).



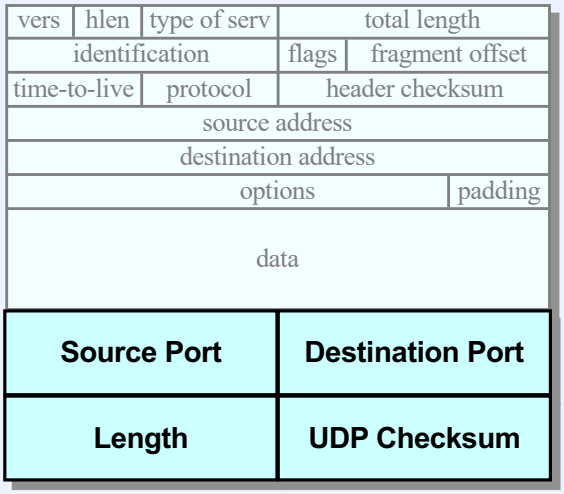
TCP/IP is a family of protocols that makes internetworking possible. Applications use TCP (transmission control protocol) to send data reliably to other applications—it is known as an end-to-end protocol in that it used only at the endpoints of the communication path. To handle the transmission of data between endpoints and gateways and between gateways, IP (internet protocol) is used. It is responsible, among other things, for finding a route to the ultimate destination. The job of actually communicating data to the next machine is handled by the network interface. It, for example, might implement the Ethernet protocol.

IP Header



We discuss IP version 4 (IPV4), which was supposed to have been replaced with IPV6 many years ago, but now coexists with IPV6. A major difference between the two is the IPV4 has 32-bit network addresses, while IPv6 has 128-bit network addresses.

User Datagram Protocol



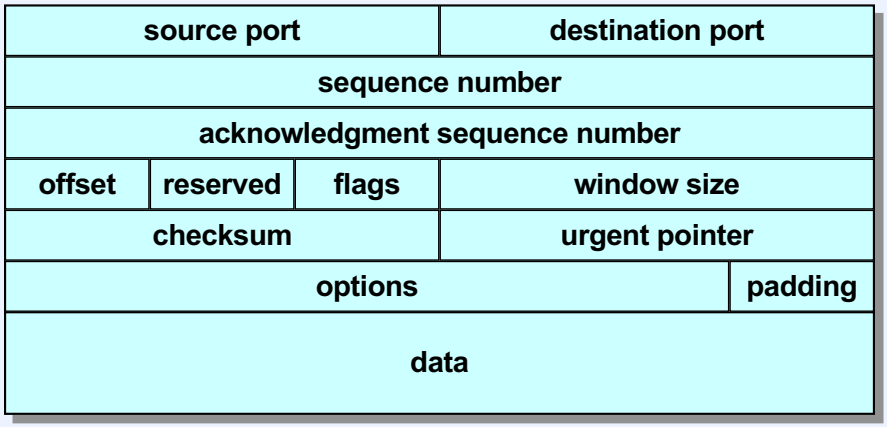
The source and destination ports indicate which application programs on the source and destination computers sent and received the packet.

Transmission Control Protocol (TCP)

- A multiplexing service built on top of IP
- A full-duplex reliable stream protocol
- Provides reliable data transfer
- Packet boundaries are not seen by the application
  - it sees a sequence of bytes, not a sequence of packets
- A connection must be established for communication to take place
  - connections are flow-controlled
  - connections are congestion-controlled

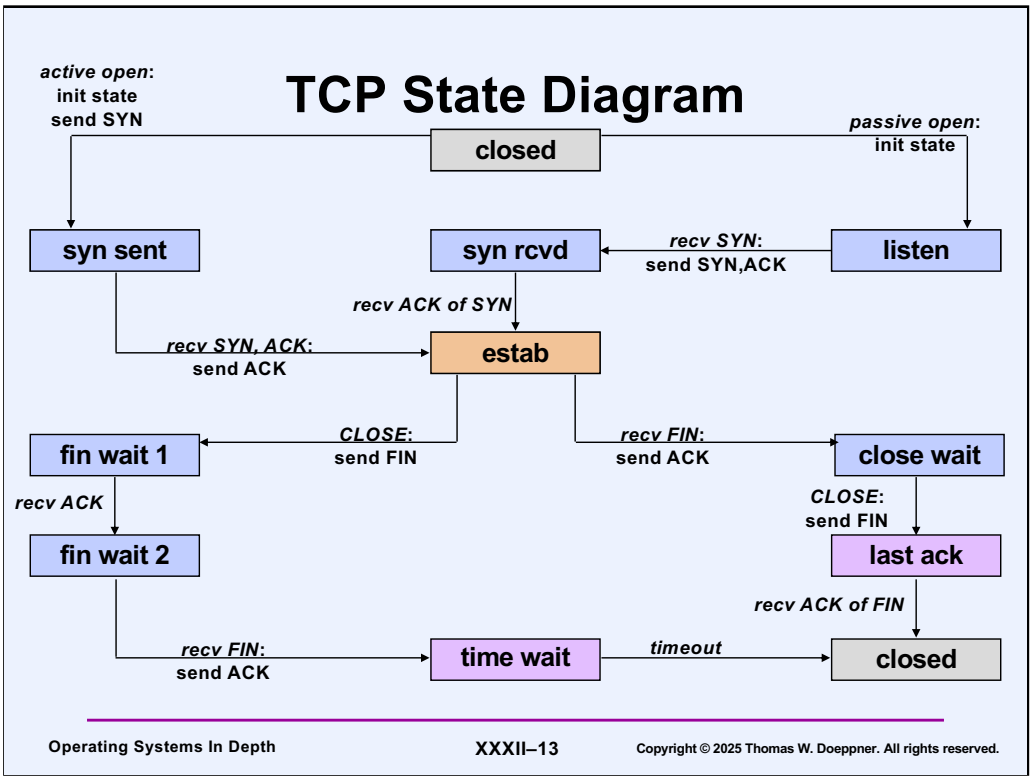
All bytes are delivered to the receiving application in the order in which they were sent, providing a reliable data transfer mechanism. A connection must first be established for communication to take place. Connections are flow-controlled so that neither side's buffers are overrun. Connections are also congestion-controlled so that data is not being transmitted faster than the internet can handle it.

TCP Header Components

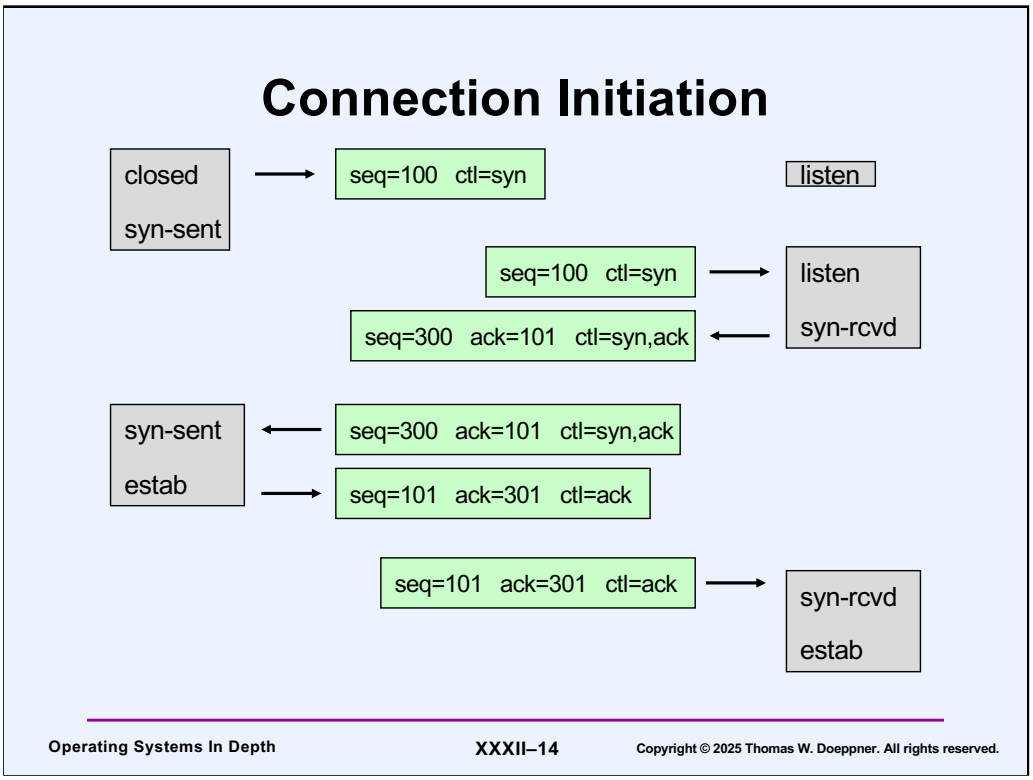


Each segment of data transmitted by TCP starts with a header. It indicates the source port and destination port and identifies where, in the logical sequence of bytes being transmitted, this segment resides. We assume that logically associated with each byte of data is a sequence number. The sequence-number field of the header indicates the sequence number of the first byte of the segment's data. The other bytes follow in sequence.

If A is communicating with B, the acknowledgment sequence number of segments from A indicates what A has received so far from B: if A has received everything sent by B up through the byte with sequence number i, then the acknowledgment sequence number is i+1 (it's the sequence number of the next byte A expects to receive from B).



Each end of a TCP connection goes through a series of states as part of connection establishment and shutdown. This diagram illustrates the TCP state machine (in somewhat simplified form). The text labeling each arrow gives the event (in *italic* type) that causes the transition to be taken and the action (in roman type) taken in response.

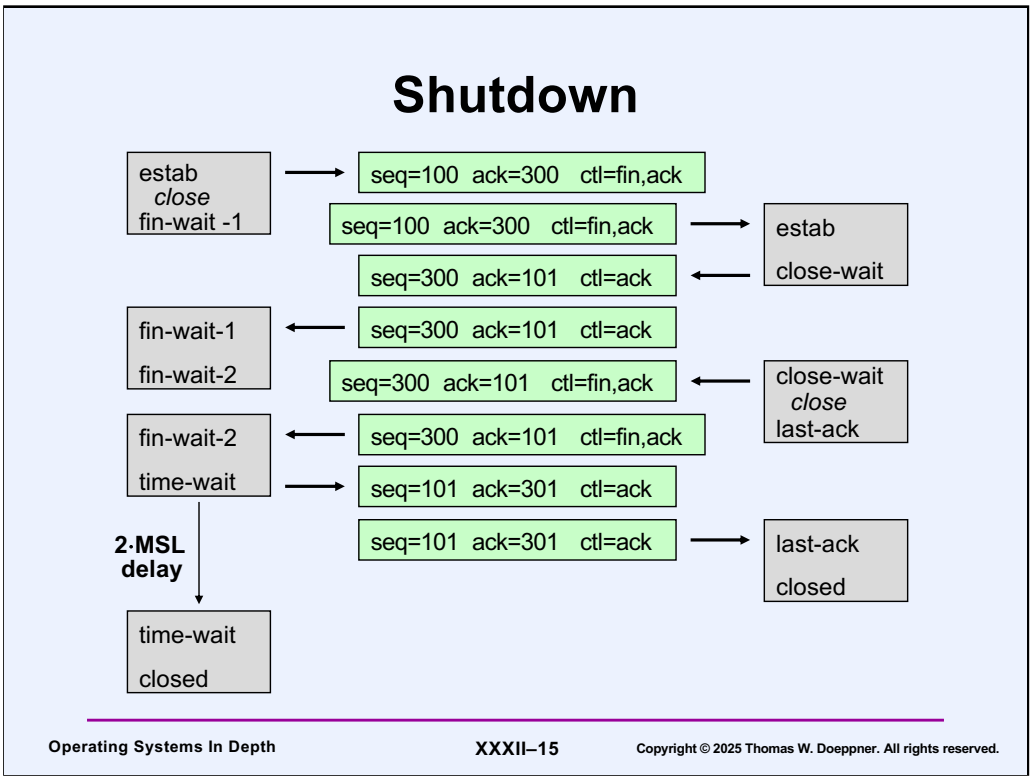


Here is a detailed example of the three-message exchange that takes place to establish a TCP connection. The active party (on the left side) transmits a packet whose sequence number is 100 and whose **syn** bit is set. Though this packet does not contain normal data, the **syn** bit must be transferred reliably and hence this bit that occupies position 100 in the sequence-number space. After sending the packet, the sender goes into the **syn-sent** state.

The passive party (on the right side) receives the first packet and acknowledges it by responding with a packet whose sequence number is 300 (the passive party's initial sequence number) and whose **syn** and **ack** bits are set. The acknowledgment field is set to 101, indicating receipt of the active party's **syn** bit. The passive party enters the **syn-rcvd** state.

The passive party's packet is received by the active party, who goes into the **established** state and acknowledges the packet's receipt. Note that the sequence number is now 101—though there is no data (or **syn** bit) being transmitted, the sequence number is advanced so as to refer to the sequence number following the last one sent. If this were not done (i.e., the packet's sequence number were set to 100), then the receiver of the packet would ignore it, treating it as a duplicate of what was already received.

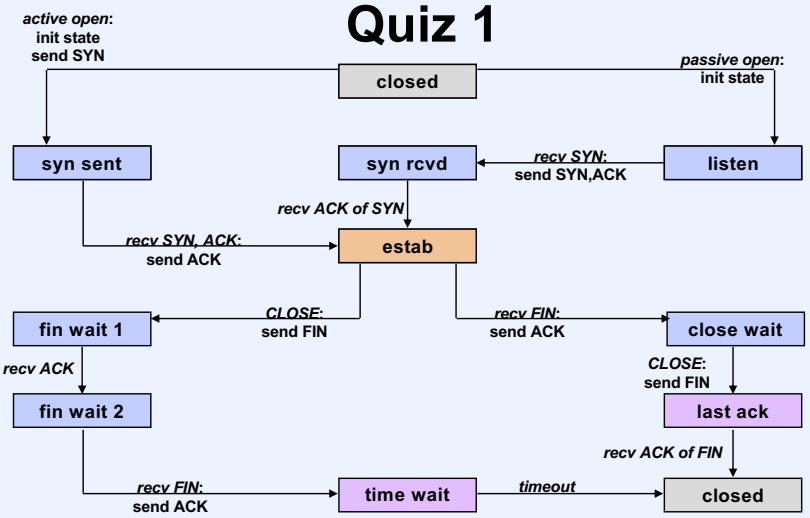
When the packet is received by the passive side, it transitions to the **established** state.



Here is an example of the exchange of messages required for the shutdown of a TCP connection. What we have here is a "graceful" shutdown—the party on the left indicates that it has nothing more to send, but it is still willing to receive data. The party on the right acknowledges this, and subsequently states (probably after transmitting more data) that it also has no more data to send. The party on the left acknowledges this. We have a potential problem if this last acknowledgment is lost—the party on the right will retransmit its termination message and the party on the left must stick around to (re)acknowledge it. How long should it stay around? The TCP specification states that it should refrain from terminating for twice the "maximum segment lifetime" (MSL), which is the maximum amount of time required for a message to travel to the farthest reaches of the Internet—usually conservatively considered to be about a minute.

Also possible is a "forced termination" in which one side unilaterally shuts down both directions of communication. This is usually an error condition, as when the terminating side's application has crashed.

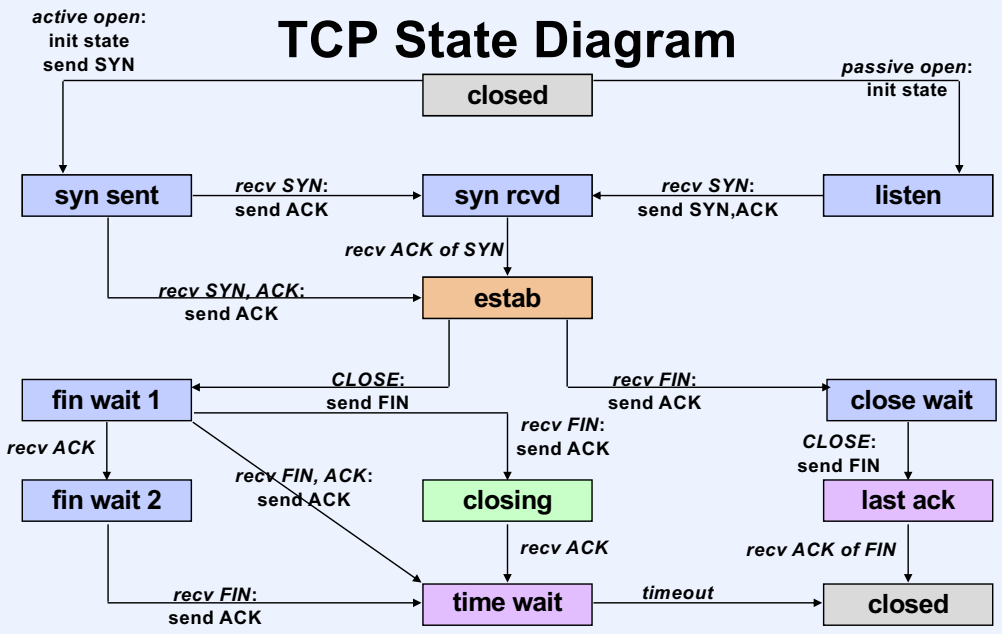
Quiz 1



Suppose both sides send the other a FIN while in the estab state; both then go to fin-wait-1. What happens next?

a) Both go to fin-wait-2, then time-wait
b) Both go to a new state that's not in the diagram
c) Such an event can't happen

TCP State Diagram

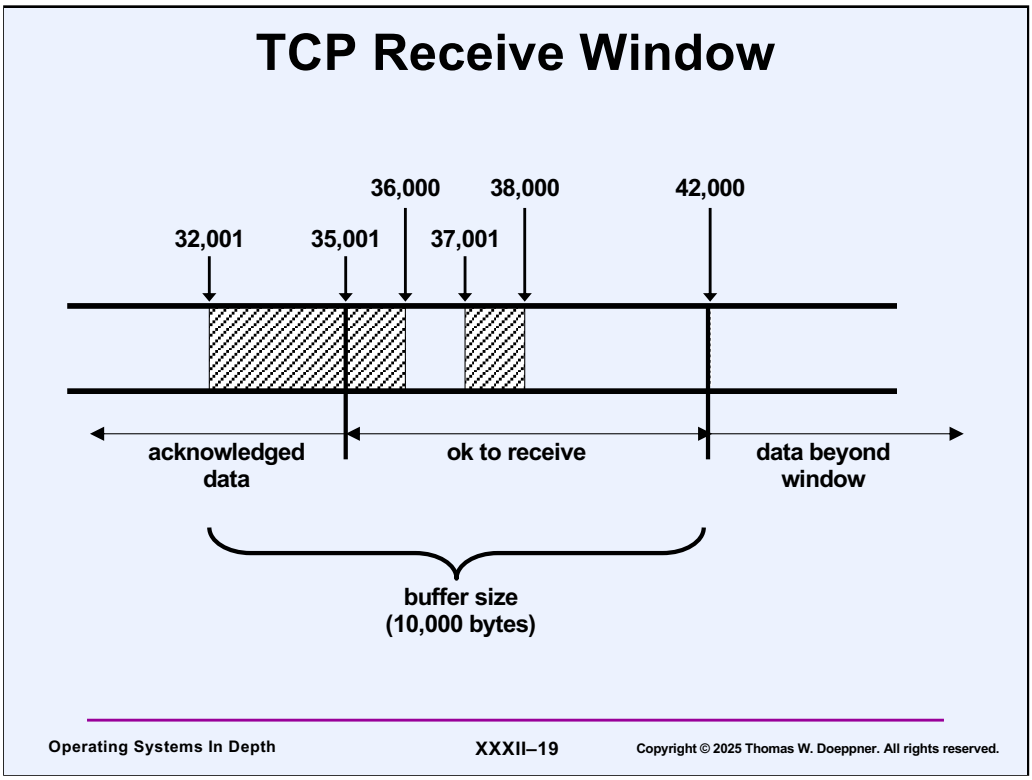


Sliding-Window Protocol

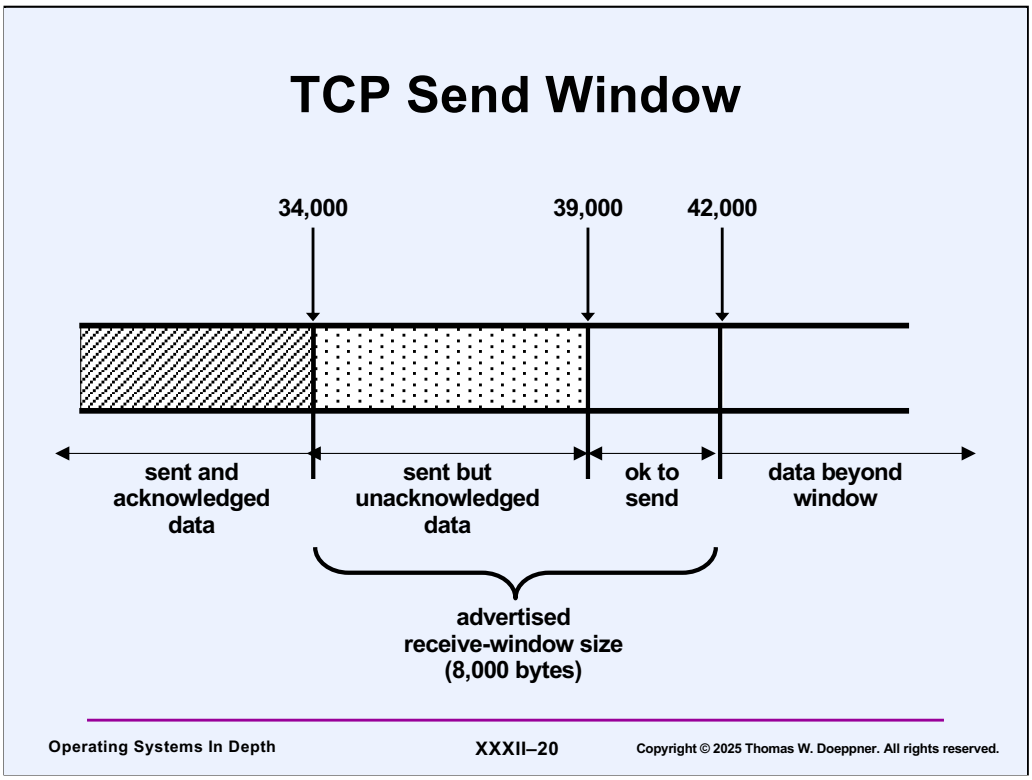
- **Used for:**
  - **reliable delivery**
  - **flow control**
- **Windows**
  - **send**
  - **receive**

Everything that must be delivered reliably is assigned a sequence number.

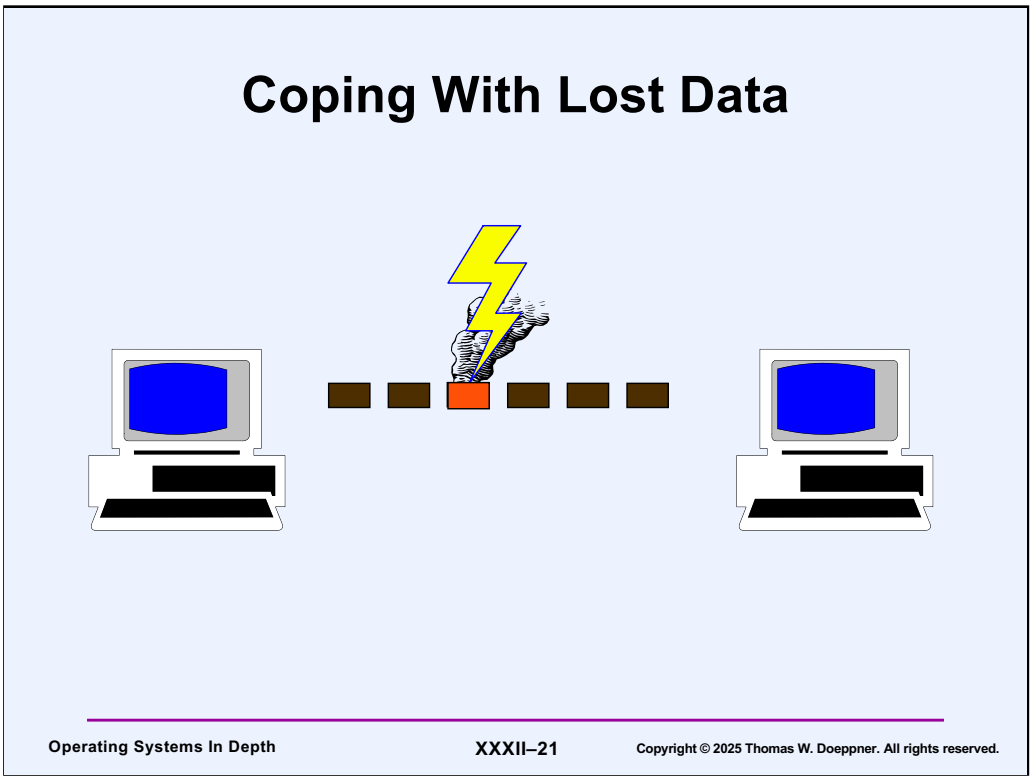
Sequence numbers are used both as a means for providing reliable delivery and as a means for flow control. Each party maintains a send window and a receive window. The receive window is the range of sequence numbers representing the space that a party has to receive data. The send window is a side's understanding of what the other side's receive window is. If a side has received all data through sequence number **i**, it acknowledges this by reporting back sequence number **i+1**.



Let's look at things from the receiver's point of view by describing its **receive window** (in the slide the shaded regions represent received data that have not yet been consumed by the application). Let's say the receiver has 10,000 bytes of buffer space for incoming data. It's already received and acknowledged 15,000 data bytes with sequence numbers from 20,001 through 35,000. Moreover, the application has consumed the first 12,000 bytes of these. Thus 3000 bytes of data in its buffer have been acknowledged but not consumed—it must hold on to this data until the application consumes it. It's also received data with sequence numbers 35,001 through 36,000 and 37,001 though 38,000 that it hasn't yet acknowledged (and, of course, it must not acknowledge data in the latter range until the data in the range 36,001 through 37,000 arrives). Thus there are an additional 2000 bytes of data in its buffer, leaving room for 5000 more bytes. If it receives new data with sequence numbers in the range 20,001 through 36,000 or in the range 37,001 through 38,000, it may assume these are duplicates and discard them. However, new data in the range 36,001 through 37,000 and in the range 38,001 though 42,000 are not duplicates and must be stored in the buffer. If it receives anything whose sequence number is greater than 42,000, it must discard it because it doesn't have room for it.



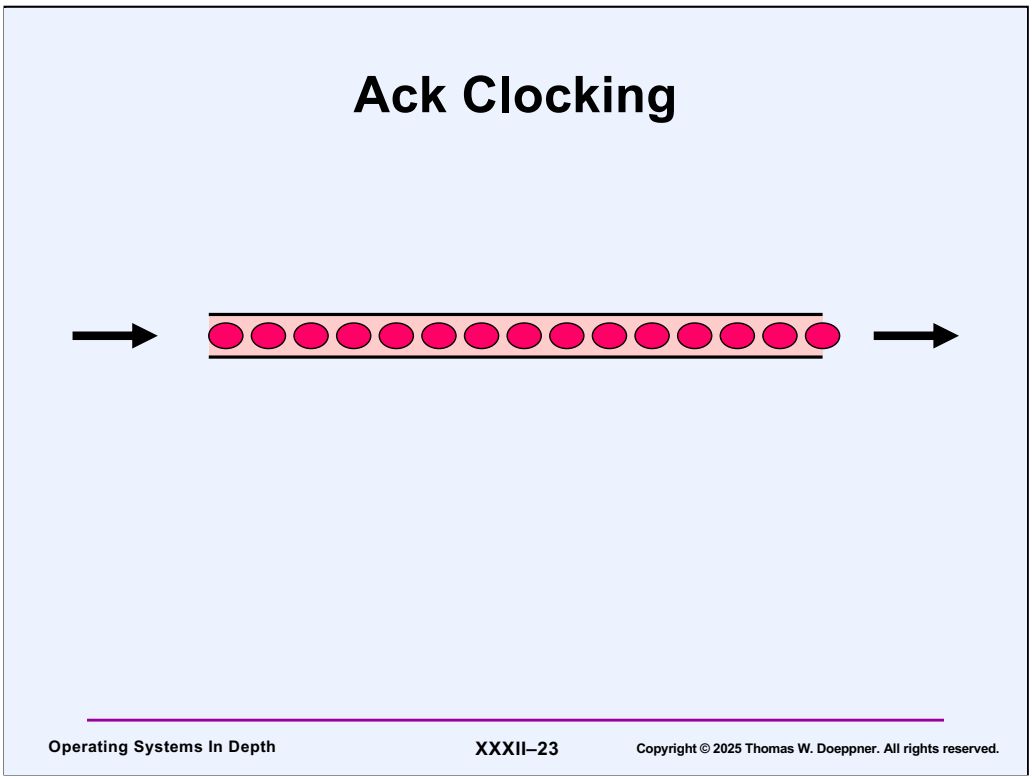
Now for the sender's point of view—we look at its **send window**. It has sent out and received acknowledgments for bytes with sequence numbers from 20,001 through 34,000. The sender has also sent out but hasn't received acknowledgments for data bytes with sequence numbers from 34,001 through 39,000. It must retain a copy of these data bytes just in case it has to resend them. The most recent segment the sender has received from the other side indicates the receive window is 8000 bytes. However, since it knows 5000 bytes' worth of data have been sent but not acknowledged, it knows that it really can send no more than 3,000 additional bytes of data—up through sequence number 42,000.



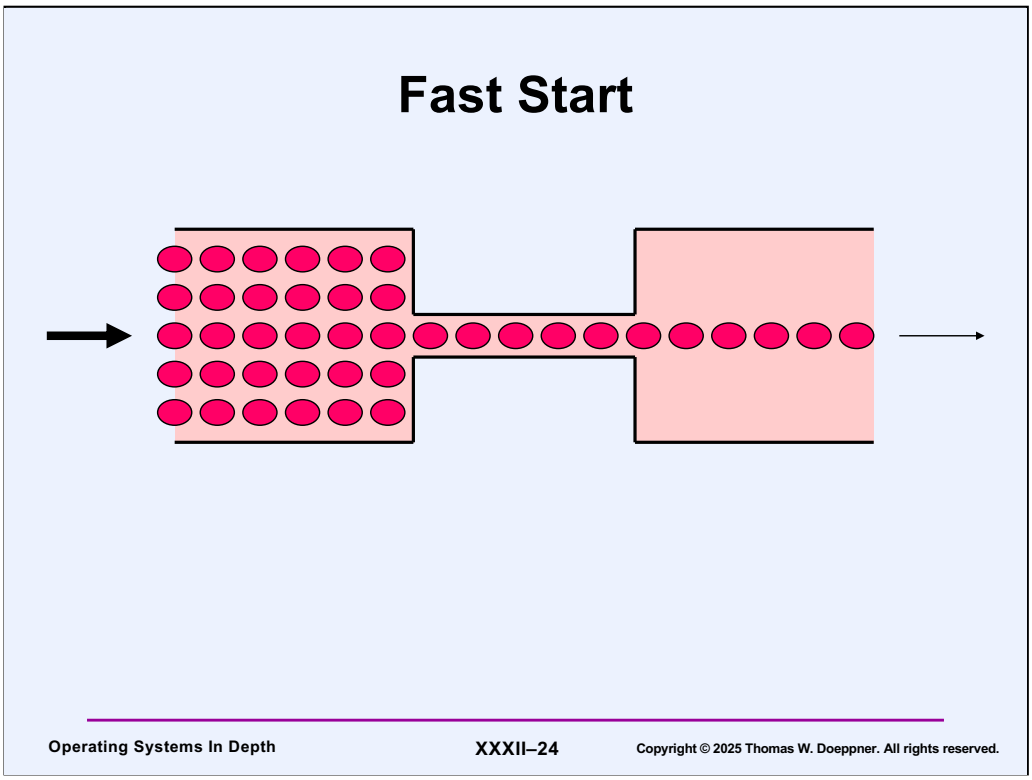
Some data inevitably will be lost and must be retransmitted. The sender decides that data has been lost if it does not receive an acknowledgment for it; but how long should it wait for the acknowledgment before giving up and retransmitting? If it doesn't wait long enough, data will be retransmitted needlessly, using up valuable network resources. If it waits too long, then there are needless communication delays. The approach taken is to keep track, for each connection, of the average amount of time between when a segment is transmitted and when the acknowledgment is received. If appreciably more time than the average is taken to receive an acknowledgment, then the segment is resent. A formula for this is suggested in the TCP specification, though it gives plenty of leeway to each implementation.

When to Retransmit?

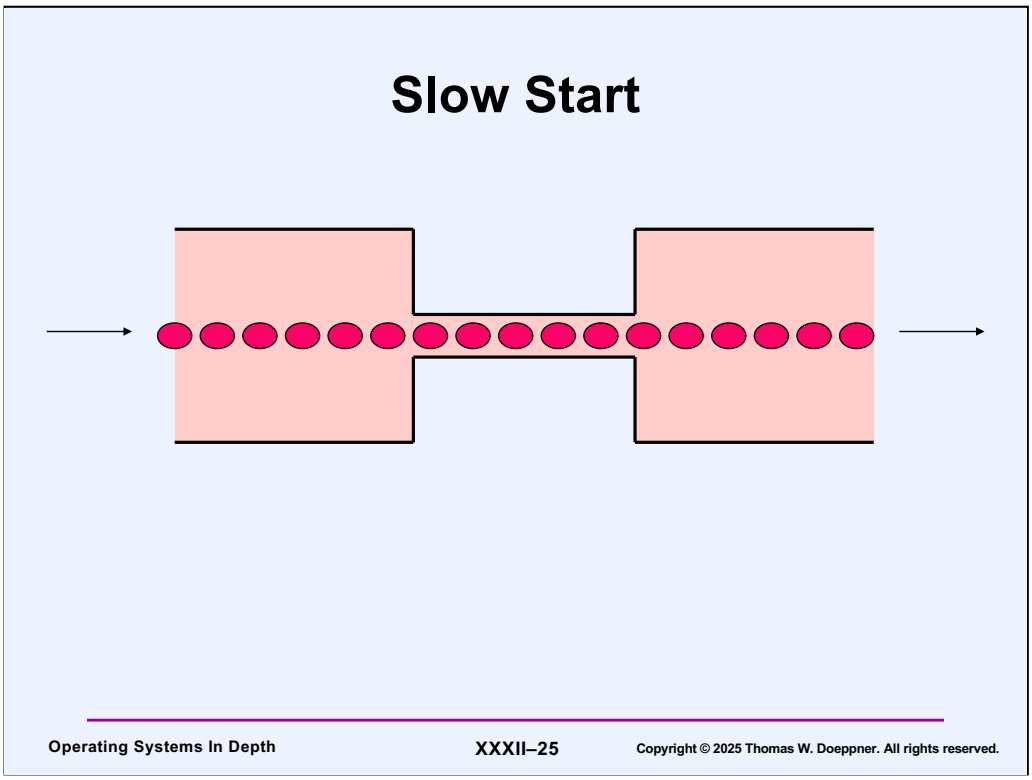
- After waiting significantly longer than the average ("smoothed") roundtrip time
- How much longer?
  - significantly longer than the average deviation
- What if one retransmission doesn't do it?
  - transmit again …
- When?
  - use exponential backoff
- When does one give up?
  - eventually …



One might think of network connections as being like pipelines—they have finite and varying capacities. Data flows through pipelines and obeys Kirchoff's current laws: the flows in must equal the flows out. Thus when a data pipeline is running at capacity, a new packet should be transmitted at one end for every packet received at the other. Since acks are being generated for the packets being received, the sender simply transmits a new packet every time it receives an ack.



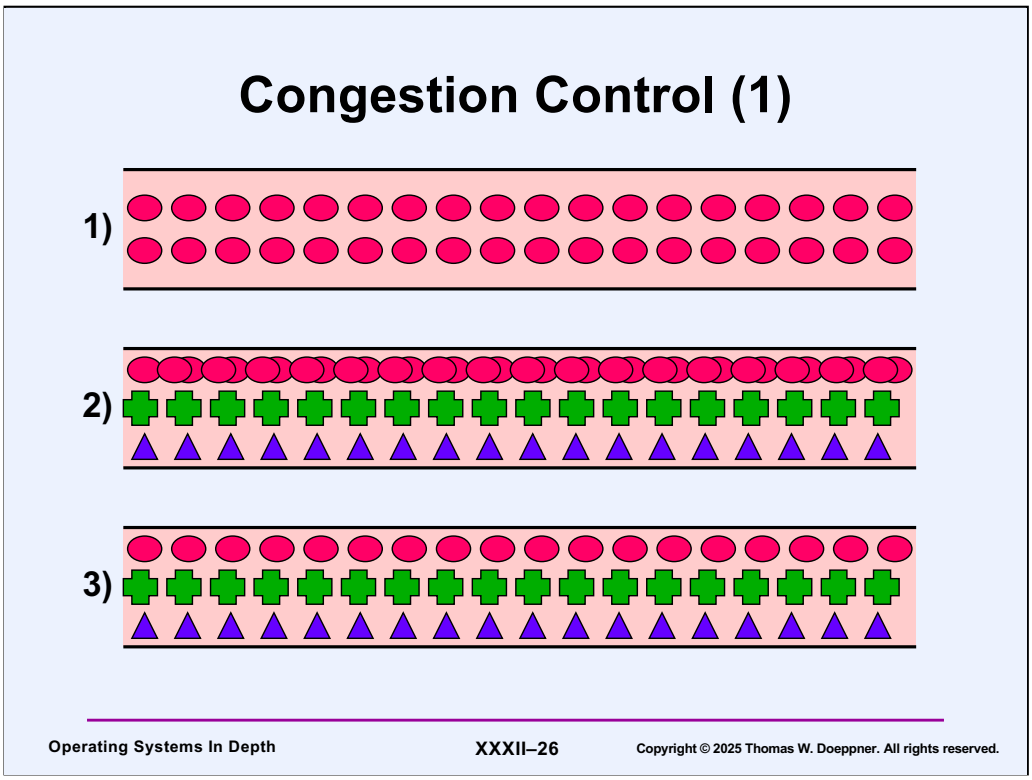
The problem with the "ack clocking" approach is getting the pipeline to capacity (and determining when it's there). Suppose a pipeline has a capacity of one gigabyte (capacity is bandwidth times the time it takes a packet to traverse the pipeline; this is known as the **bandwidth-delay product**). A naive (and fast) sender might start transmitting at its maximum speed, say 10 megabytes/second—since the pipeline is empty, it seems reasonable to fill it as quickly as possible. However, one or more downstream routers might be able to handle data at only 1 megabyte/second. Packets will start piling up at the first such router, which will soon start discarding them, forcing retransmissions by the sender.



A better approach is the one called **slow start**: the sender starts transmitting slowly but speeds up until it fills the pipeline and is transmitting no faster than the pipeline can handle (as governed by ack clocking).

More precisely, each sender maintains a **congestion window**. The amount of data the sender is allowed to transmit is the minimum of the sizes of the congestion and send windows. When transmitting data into an idle connection (i.e., an empty pipeline—one with no acks outstanding), the congestion window is set to one. Each time an ack is received, the congestion window size is increased by one. This results in exponential growth of the window size over time: after each roundtrip time, assuming no packets are lost, the window size doubles.

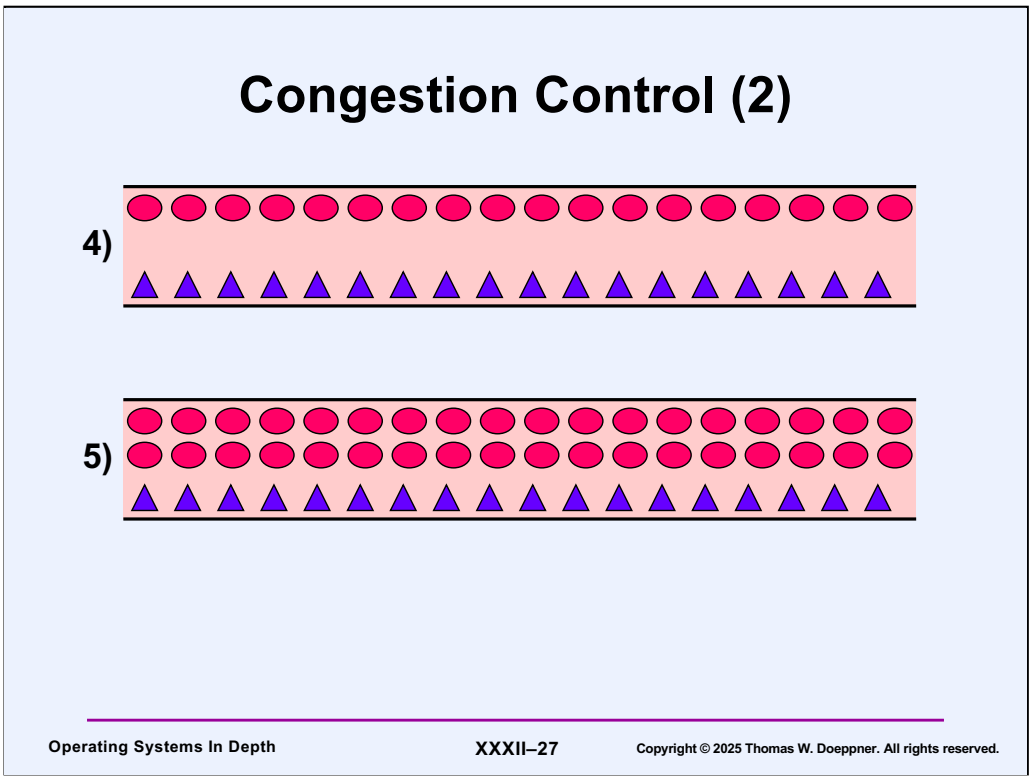
Of course, we need to know when to stop this exponential growth; we take this up in the next pair of slides.



Let's assume that, somehow, we have found the capacity of the pipeline. Using ack clocking, we are maintaining an optimal data rate (line 1 of the slide). Suddenly, some additional connections are competing with us for bandwidth — congestion occurs and many of our packets are dropped (line 2 of the slide).

In earlier versions of TCP, it was up to the router that dropped a packet to inform the sender that there is a problem. The sender was then to slow down. In practice, this approach failed miserably. A better approach, by Van Jacobson (see "Congestion Avoidance and Control", Proceedings of SIGCOMM '88) is for the sender to recognize that there is a problem as soon as one of its packets is not acknowledged. That a packet is lost might be due to its being corrupted, but the far most likely cause is that it was dropped by a router (due to congestion). Thus the response to a lost packet is to instantly lower the transmission rate by a factor of two (i.e., halve the congestion window size) (line 3 of the slide).

An unfortunate complication is that by the time the sender times out waiting for the dropped packet to be ack'd, the pipeline has emptied. Thus it must now use the slow-start approach to get the pipeline going again.



Let's assume that the sender has once again found the capacity of the pipeline and is transmitting according to ack clocking. Now (line 4), a connection that was using the pipeline drops out and there is surplus bandwidth. It's important that our sender discover this so it can transmit at a faster rate. Since the parties that ceased to communicate are not going to announce this fact to others, our sender must discover it on its own.

This discovery is implemented by having our sender gradually raise its transmission rate. It might do this exponentially, as in slow start, but this would quickly cause it to exceed the pipeline's capacity and it would have to lower the rate by a factor of two. This would result in fairly wild oscillations. A better approach, suggestion in the previously cited paper by Jacobson and implemented in most TCP implementations, is to raise the transmission rate linearly, as opposed to exponentially. Thus the rate rises slowly until it's too high and a packet is dropped, at which point it is again halved. The net effect is a slow oscillation around the optimal speed.

To deal with both slow start and congestion control, we must remember that whenever we timeout on an ack, the pipeline is emptied. When this happens, the current value of the congestion window size is halved and stored in another per-connection variable, **ssthresh** (slow-start threshold). The congestion window is then set to one and the slow-start, exponential growth approach (which isn't all that slow) is used until the window size reaches ssthresh. At this point the transmission rate through the pipeline has reached half of what it was before and we turn off slow start and switch from exponential growth of the congestion window size to linear growth, as discussed in the previous paragraph.

Acks

- When a segment is received, the highest permissible Ack is sent back
  - if data up through i has been received, the ack sequence number is i+1
  - if data up through i has been received, as well as i+100 through i+200, the ack sequence number is i+1
    - a higher value would imply that data in the range [i+1, i+99] has been received
- Every segment sent contains the most up-to-date Ack

A TCP receiver responds with an Ack to everything received.

Quiz 2

A TCP sender has sent four hundred-byte segments starting with sequence numbers 1000, 1100, 1200, and 1300, respectively. It receives from the other side three consecutive ACKs, all mentioning sequence number 1100. It may conclude that

a) The first segment was received, but nothing more

b) The first, third, and fourth segments were received, but not the second

c) The first segment was received, but not the second; nothing is known about the others

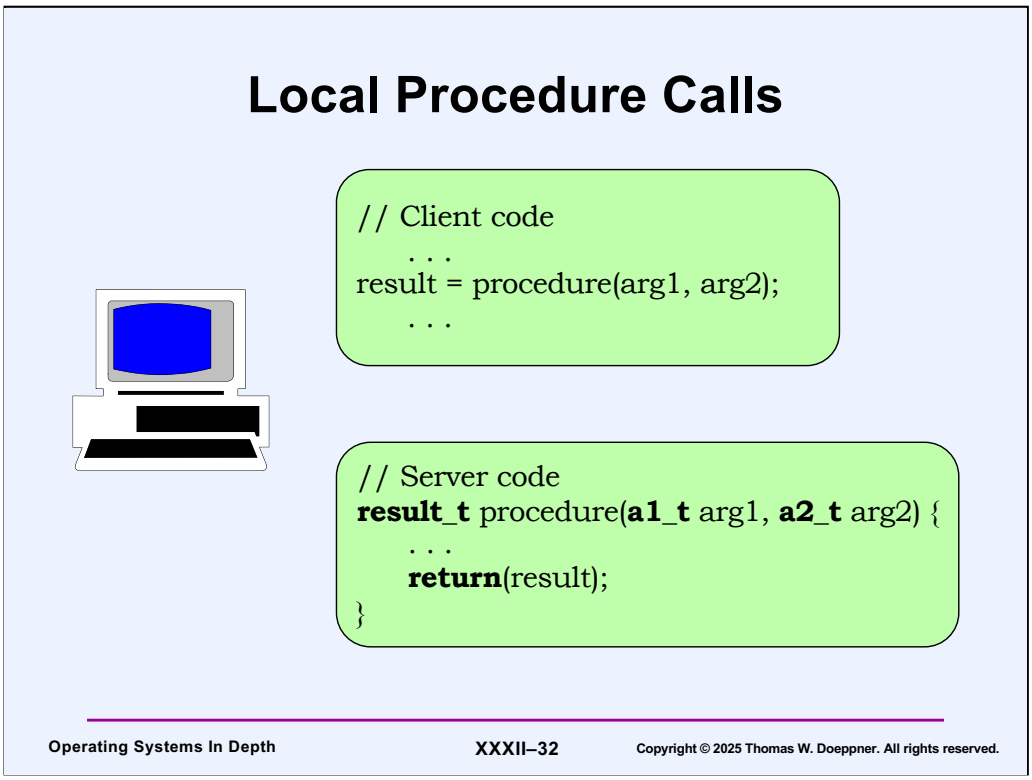
d) There's a bug in the receiver

Fast Retransmit and Recovery

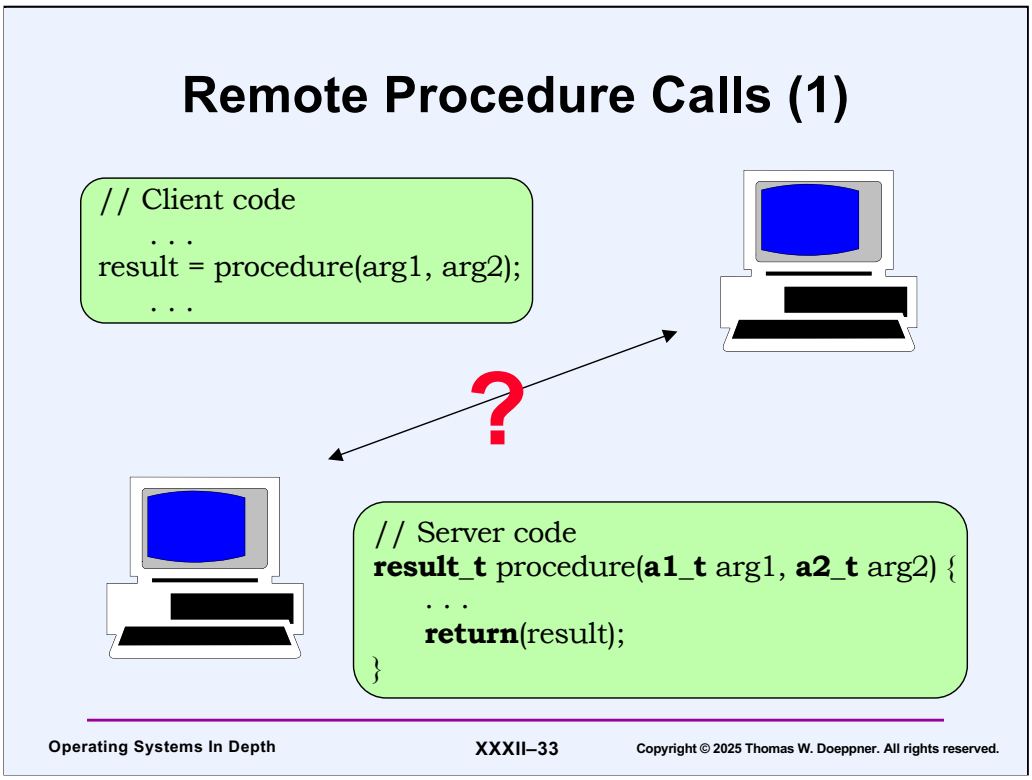
- Waiting an entire RTO before retransmitting causes the "pipeline" to become empty
  - must slow-start to get going again
- If one receives three acks that all repeat the same sequence number:
  - some data is getting through
  - one segment is lost
  - immediately retransmit the lost segment
  - halve the congestion window (i.e., perform congestion control)
  - don't slow-start (there is still data in the pipeline)

Note that RTO stands for retransmission timeout–how long the system waits for an acknowledgement before giving up and retransmitting.

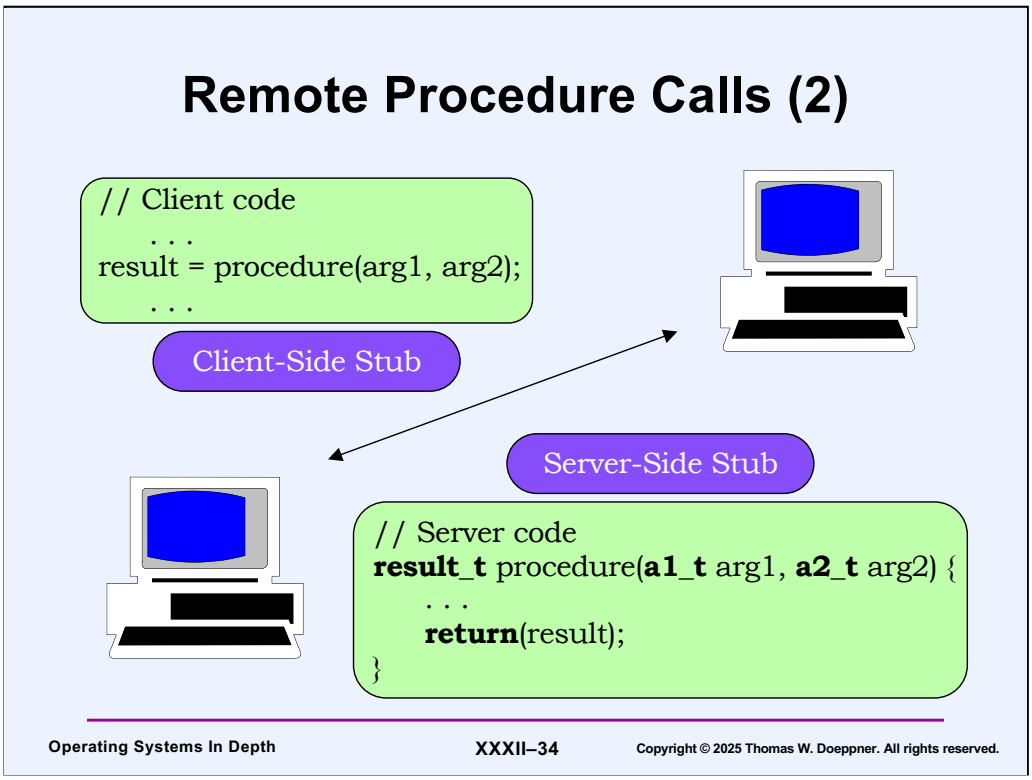
Remote Procedure Call Protocols



The basic theory of operation of RPC is pretty straightforward. But, to understand **remote** procedure calls, let's first make sure that we understand **local** procedure calls. The client (or caller) supplies some number of arguments and invokes the procedure. The server (or callee) receives the invocation and gets a copy of the arguments (other languages, such as C++, provide other argument-passing modes, but copying is all that is provided in C). In the usual implementation, the callee's copy of the arguments have been placed on the runtime stack by the caller—the callee code knows exactly where to find them. When the call completes, a return value may be supplied by the callee to the caller. Some of the arguments might be out arguments—changes to their value are reflected back to the caller. This is handled in C indirectly—the actual argument, passed by copying, is a pointer to some value. The callee follows the pointer and modifies the value.

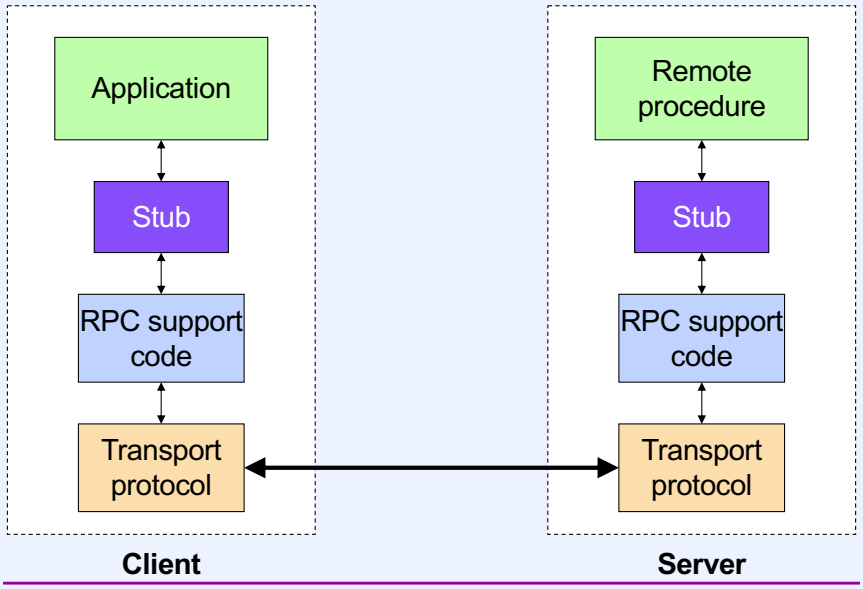


Now suppose that the client and server are on separate machines. As much as possible, we would like remote procedure calling to look and behave like local procedure calling. Furthermore, we would like to use the same languages and compilers for the remote case as in the local case. But how do we make this work? A remote call is very different from a local call. For example, in the local call, the caller simply puts the arguments on the runtime stack and expects the callee to find them there. In C, the callee returns data through out arguments by following a pointer into the space of the caller. These techniques simply don't work in the remote case.



The solution is to use **stub procedures**: the client places a call to something that has the name of the desired procedure, but is actually a proxy for it, known as the **client-side stub**. This proxy gathers together all of the arguments (actually, just the in and in-out arguments) and packages them into a message that it sends to the server. The server has a corresponding **server-side stub** that receives the invocation message, pulls out the arguments, and calls the actual (remote) procedure. When this procedure returns, returned data is packaged by the server-side stub into another message, which is transmitted back to the client-side stub, which pulls out the data and returns it to the original caller. From the points of view of the caller and callee procedure, the entire process appears to be a local procedure call—they behave no differently for the remote case.

Block Diagram



ONC RPC

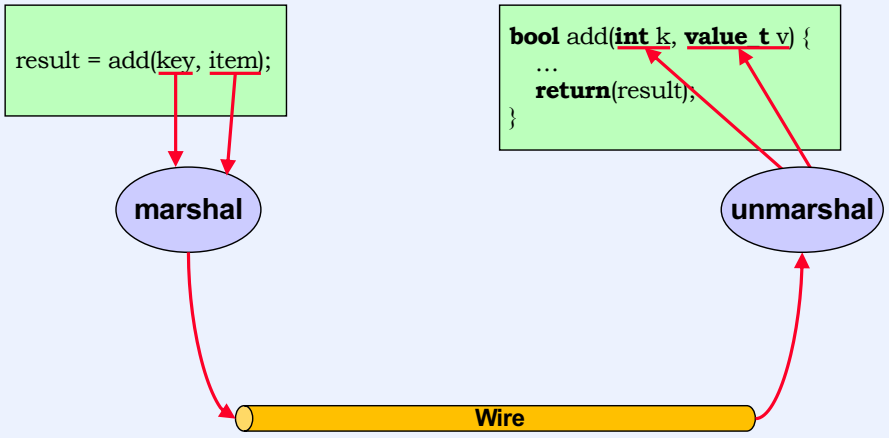
- Used with NFS
- eXternal Data Representation (XDR)
  - specification for how data is transmitted
  - language for specifying interfaces

ONC stands for open network computing and was originally designed by Sun Microsystems in the 1980s.

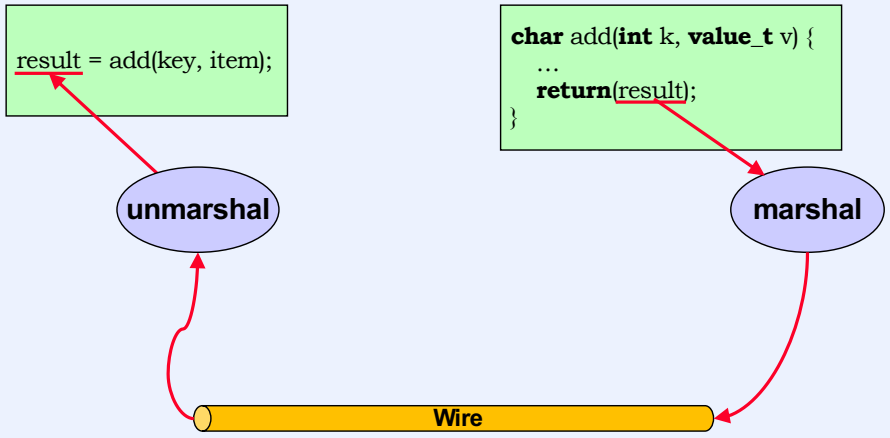
Example

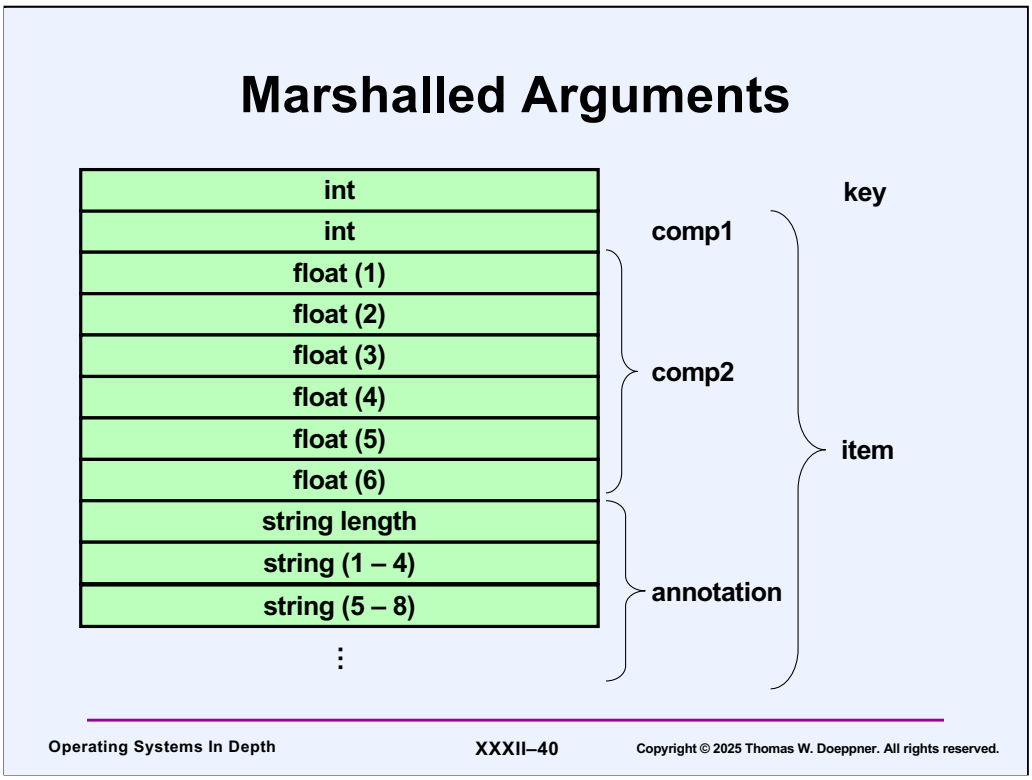
```
typedef struct {
   int   comp1;
   float comp2[6];
   char  *annotation;
} value_t;

typedef struct {
   value_t  item;
   list_t   *next;
} list_t;

bool add(int key, value_t item);
bool remove(int key, value_t item);
list_t query(int key);
```

Placing a Call



Returning From the Call

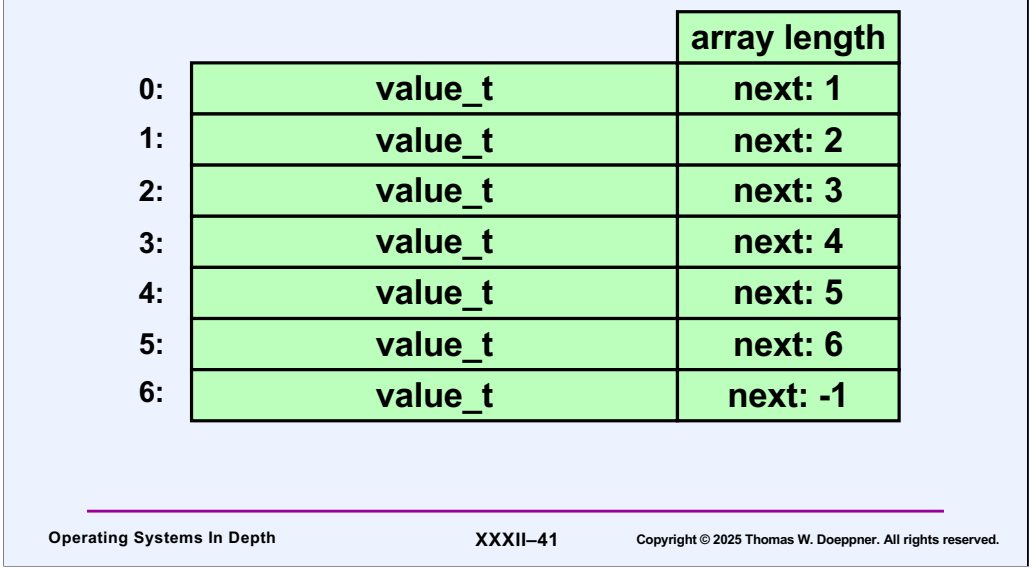




Here is what is sent over the wire after marshalling an item of type value_t.

Note that, as part of marshalling, the length of the string is determined and included with the marshalled arguments.

Marshalled Linked List



To marshal a list, one might represent it as an array with the links represented as array indices.