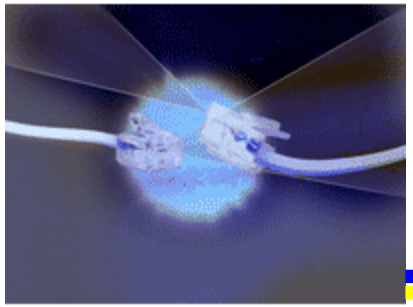
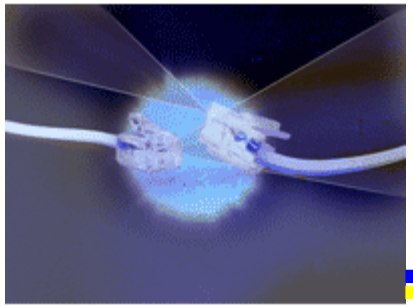


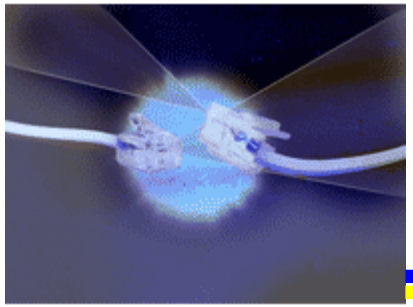
User Datagram Protocol (UDP)

- Standard connectionless protocol for the transport layer of the Internet architecture
- Only adds demultiplexing capability to basic best-effort delivery provided by IP
- Needs to identify target process for msg
  - Could use some direct identifier like process ID, but that might not work with all OSes
  - Instead uses indirect handle, the port number



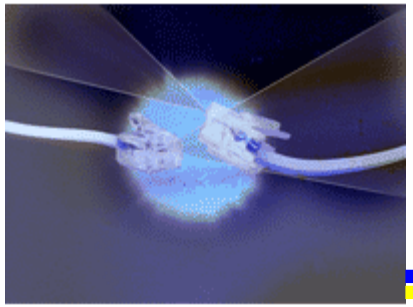
UDP (cont.)

- UDP header contains source port, destination port, length, and checksum (all two bytes)
- Source and destination ports are only unique on the respective hosts – key is pair of (host, port) values



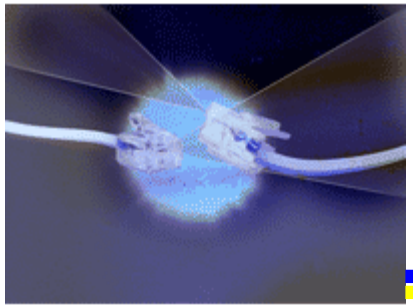
UDP (cont.)

- UDP does ensure correctness of packet using checksum.
  - Optional in current UDP, required in IPv6
  - Checksum computed over message data, UDP header, and *pseudoheader* – protocol number and source and destination IP addresses, plus UDP length
  - Uses same checksum as IP



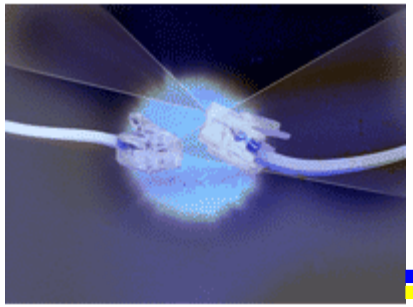
Obtaining Port Numbers

- Need host IP and port to talk to server
  - Once server has address, it can respond to address in packet it received
- Different techniques for getting port #
  - Use a well-known port (i.e. DNS uses 53)
    - Values found in /etc/services
  - Use a *port mapper* – single process that runs on the server and knows the ports for different services
  - Use a *directory service* that runs on the network and knows the port numbers for services on any host



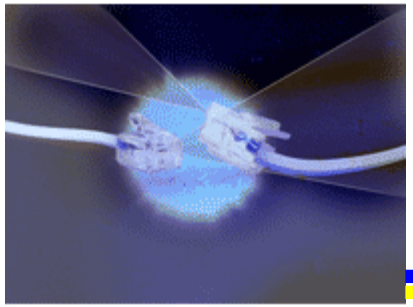
Implementation

- Typically, a port is implemented by OS as a message queue
  - Incoming messages added to queue for specified port
  - Messages removed by application when it reads the port
  - Messages discarded if queue is full
  - Process blocks if queue is empty when it reads



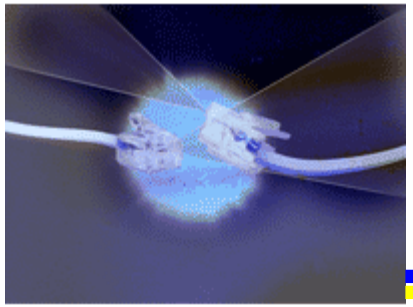
Transmission Control Protocol (TCP)

- Standard connection-oriented protocol used in Internet architecture
- Guarantees reliable, in-order delivery of byte stream
  - Stream is full-duplex, and each direction provides flow control so receiver can limit amount of data sender can transmit
- Like UDP, TCP uses ports to select app



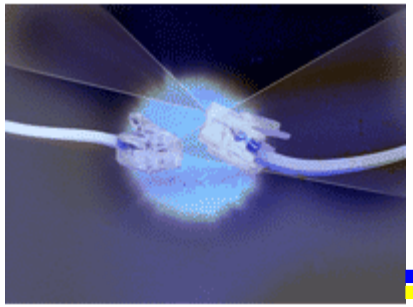
TCP (cont.)

- TCP also includes congestion control
  - Flow control keeps sender from overrunning receiver
  - Congestion control keeps sender from overrunning network
- Uses a sliding window protocol for reliability
  - Requires connection setup (like VC setup) and connection teardown phases



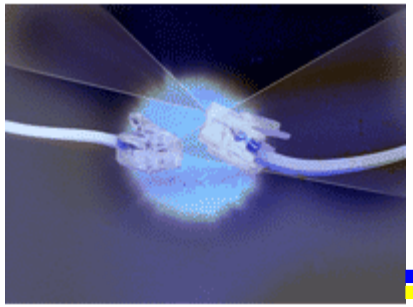
TCP Sliding Windows

- RTT might vary widely over different connections, and even with same connection over time, so retransmit timeout must be adaptive
- Packets may be reordered crossing internet, which can't happen on point-to-point links
  - TCP knows that packets will expire, so it assumes *maximum segment lifetime* (MSL) – currently 120 sec.



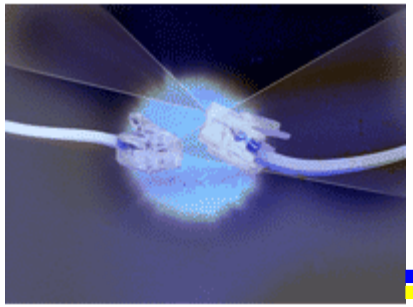
TCP Sliding Windows (cont.)

- Can't tailor size of window to link's gain-bandwidth product, so sender must learn how many resources like buffers receiver has (flow control problem)
- Sender may also overload a slow intermediate network link, so it must learn where bottlenecks are (congestion problem)



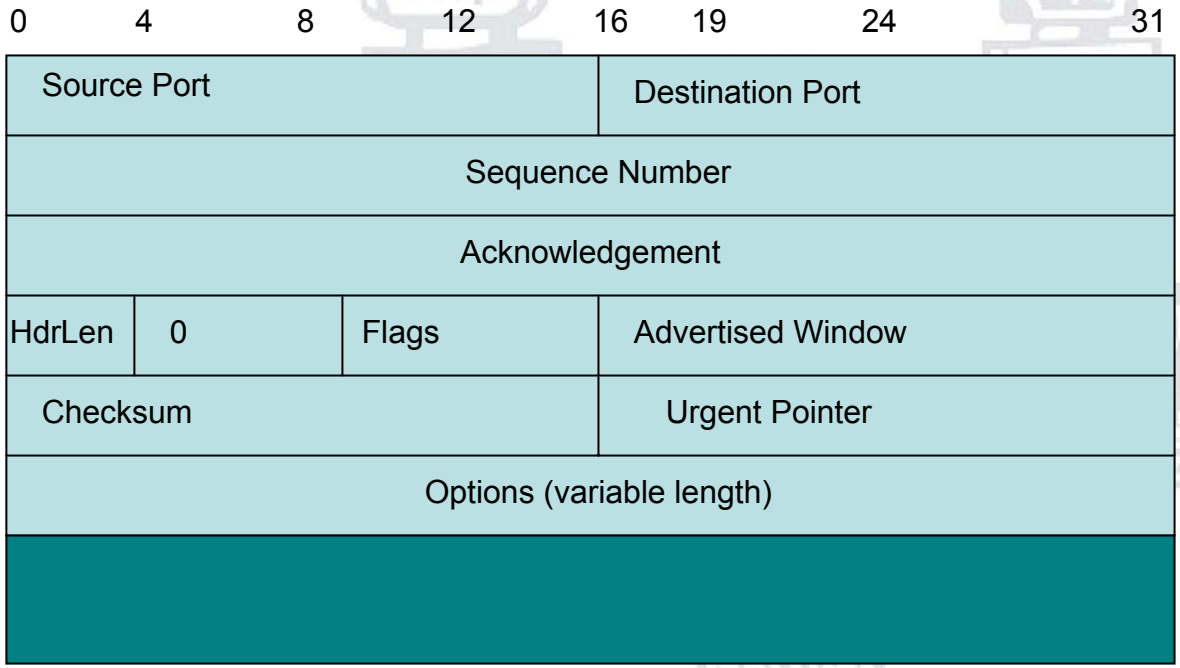
TCP Segments

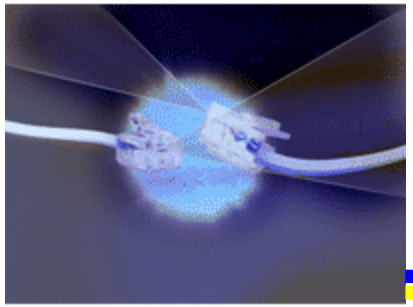
- TCP provides byte stream service to apps, but breaks stream into *segments* for transmission
  - TCP provides send and receive buffers to handle this for the app
- TCP uses same ports as UDP to identify target process



TCP Segment Header

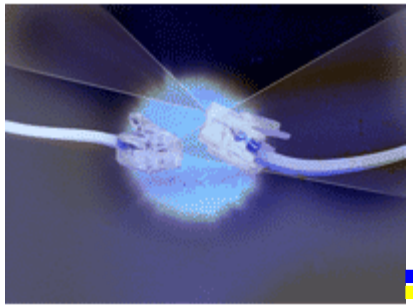
- At least 20 byte header





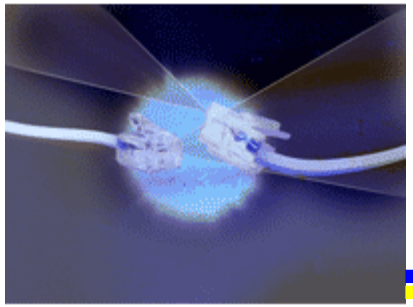
TCP Header (cont.)

- Connection identified by (src IP, src port, dest IP, dest port)
  - Connection might be created, destroyed, and recreated; can have multiple *incarnations*
- Sequence Num, Acknowledgement, and Advertised Window used by sliding window protocol
  - Each byte has sequence – header field is value for first byte of data in segment



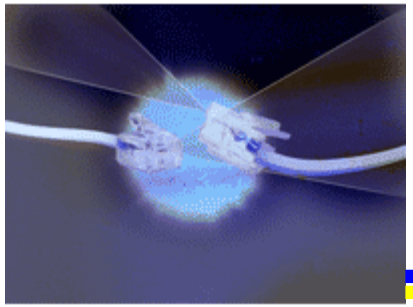
TCP Header (cont.)

  - Acknowledgement and advertised window used to return data from receiver to sender
- Flags include SYN, FIN, RESET, PUSH, URG, and ACK
  - SYN for establishing connection
  - FIN for tearing down connection
  - ACK set whenever Acknowledgement valid
  - URG indicates segment contains urgent data
  - PUSH causes receiver to notify app (OOB data)
  - RESET allows receiver to panic and kill connection



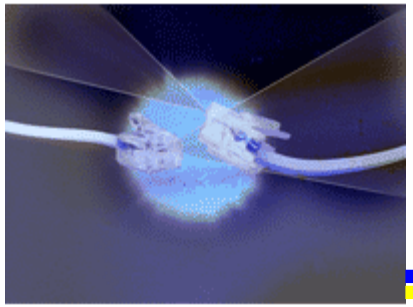
Connection Setup

- Client exchanges messages with server to establish connection
  - Client is doing *active* open, while server has done *passive* open
- Three-way handshake process
  - Client sends SYN with starting sequence #, *x*
  - Server returns msg with ACK, ack = *x* + 1, SYN, and starting sequence #, *y*
  - Client sends ACK, ack = *y* + 1



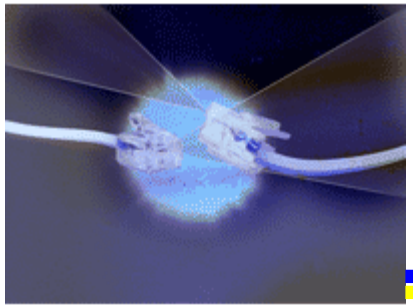
Connection Setup (cont.)

- Ack indicates “next seq # expected”
- Timer started for each segment – retransmits if response not received
- Starting sequence must be chosen at random, to minimize the chance of second incarnation of connection mistaking an old packet from earlier incarnation



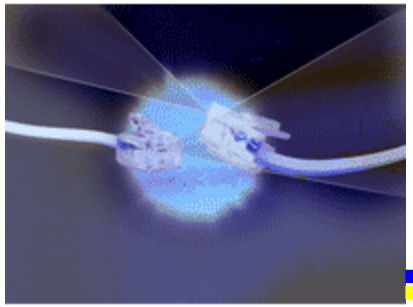
State Transition

- See Fig. 5.7 in text, p. 386
  - Arcs labeled by event / action
  - Events can be network-related or application generated
- Note that last ACK from client to server can be lost – server still in ESTABLISHED state
  - All following segments contain ACK and Acknowledged even if no new data received



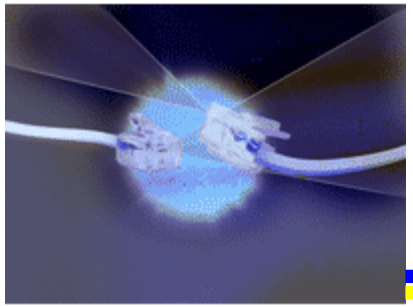
State Transition (cont.)

- Diagram should also include arcs for timeouts – each state will retry send several times. If it fails, return to CLOSED state



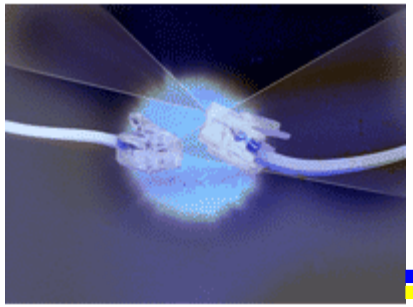
Connection Teardown

- Each side of connection must close its half of connection
- Cannot close connection without waiting two MSLs after sending ACK
  - Waiting to make sure other side doesn't retransmit FIN



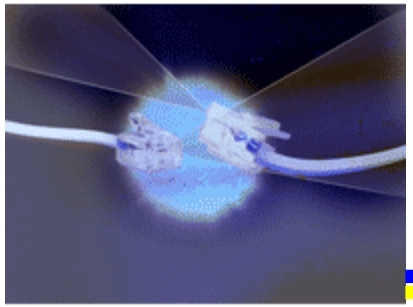
Sliding Window

- TCP adds flow control to basic sliding window protocol
- Receiver advertises window size to sender
  - Sender can have no more than that many unacknowledged bytes of data outstanding
- Sender maintains LastByteAcked, LastByteSent, LastByteWritten
- Receiver maintains LastByteRead, NextByteExpected, LastByteRcvd



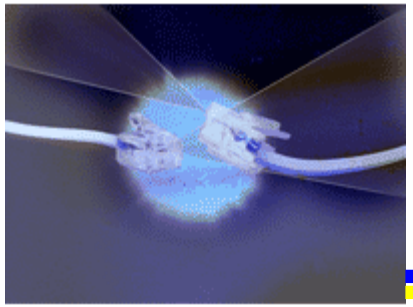
Sliding Window (cont.)

- If sender computes EffectiveWindow = AdvertisedWindow – (LastByteSent – LastByte Acked) and this value is 0, it cannot send
  - It may send TCP message anyway to ACK, but data length will be 0
- Sender must also block application to make sure it doesn't overflow MaxSendBuffer



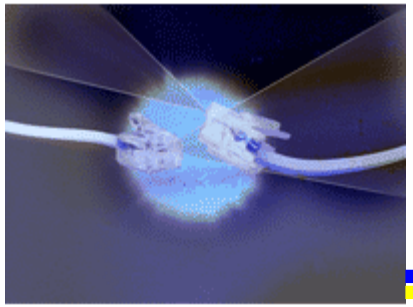
Sliding Window (cont.)

- When receiver shuts window down to 0, sender continues sending 1-byte messages periodically, so receiver can respond with ack when buffer space freed
- Wrap-around of seq. #s can occur, even with 32-bit numbers, on very fast networks within a short period of time
  - New version of TCP will extend numbers



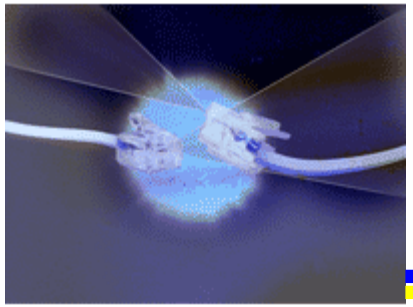
Sliding Window (cont.)

- Worse problem with AdvertisedWindow – it isn't big enough to allow pipe to be kept full if round trip time isn't large
  - New TCP version also extends this number



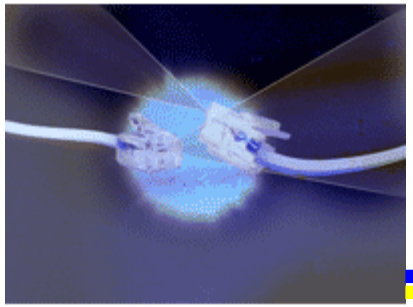
Sliding Window (cont.)

- Also have MaxSendBuffer and MaxRcvBuffer
  - Receiver requires LastByteRcvd – LastByteRead <= MaxRcvBuffer
  - It sets AdvertisedWindow = MaxRcvBuffer – ((NextByteExpected – 1) – LastByteRcvd) to slow down sender
  - Sender must guarantee that LastByteSent – LastByteAcked <= AdvertisedWindow



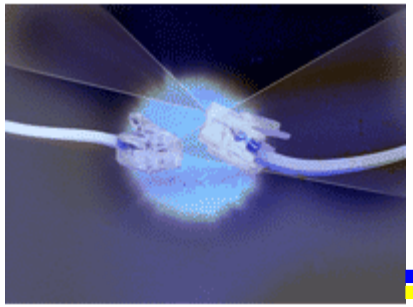
Triggering Transmission

- TCP must decide when to send segment
  - Buffering bytes for outgoing stream, so there is no absolute event like *sendto()* to trigger
- TCP has three mechanisms:
  - When Max Segment Size (MSS) bytes ready
    - MSS usually set to largest value to fit in MTU
  - When sending process requests *push* to flush buffer
  - When transmit timer expires



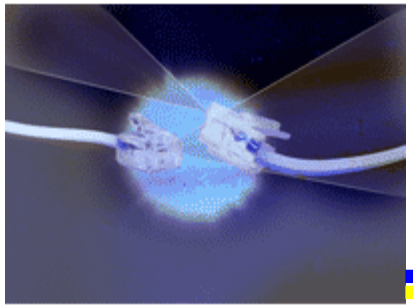
Silly Window Syndrome

- TCP must also consider flow control (receiver's advertised window size)
- If window is closed (window size = 0) and MSS bytes are accumulated, then window opens to MSS/2 bytes, should sender immediately send a half-full segment?
  - Greedy sending causes *silly window syndrome*, where sender sends small packet, receiver acks, sender immediately sends another small packet, etc.



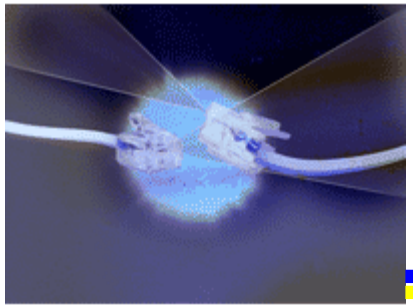
Silly Window Syndrome (cont.)

  - Inefficient use of network, because of TCP overhead for each packet and ACK
- Receiver could wait until it has at least MSS bytes free before advertising open window
- Receiver could also try to consolidate small segments into larger ones by delaying ACKs, sending a single ACK for several small segments instead of individual ACKs, but it does not know how long it can wait



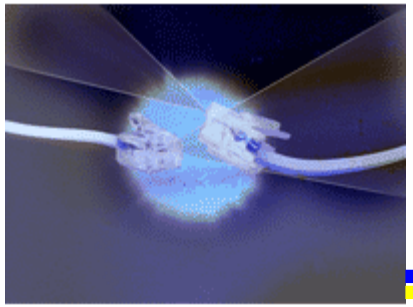
Nagle's Algorithm

- Sender needs to decide when to send a segment
  - Too much delay is bad for interactive apps
  - Too little delay hurts network performance
- Use timer to decide; instead of clock-based timer, use Nagle's *self-clocking* scheme



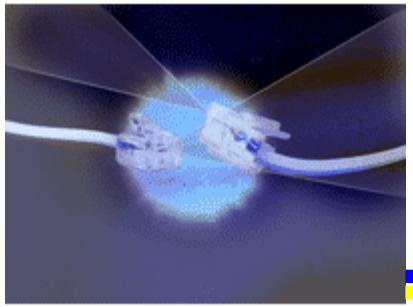
Nagle's Algorithm (cont.)

- Send a full segment if window allows
- Send small segment if there is nothing in flight
- If data already in flight, wait for ACK to send next segment
  - Can disable using the TCP_NODELAY option on the socket



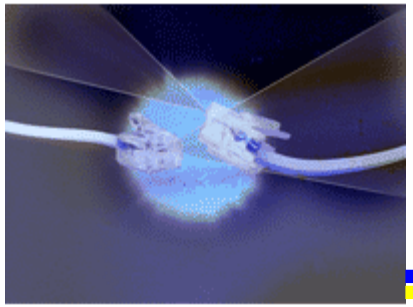
Adaptive Retransmission

- TCP retransmits if no ACK received within timeout period
  - Timeout is function of RTT
  - RTT might vary widely between connections or on the same connection over time
  - Need to adapt timeout



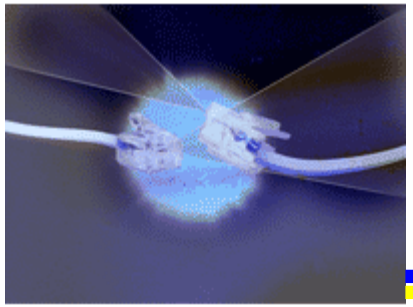
Adaptive Retransmission – Original Algorithm

- Maintain running average of RTT
  - Record time when each segment is sent, time when ACK is received
  - Difference is sample RTT – compute weighted average as
    EstRTT = α * EstRTT + (1-α)*SampleRTT
  - α is *smoothing parameter*; small value weights more on latest sample, large value more stable but less adaptable. Typically use value between .8 and .9
  - Timeout = 2 * EstRTT



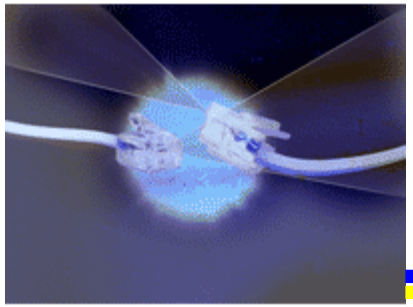
Karn/Partridge Algorithm

- Original algorithm had a problem; ACK doesn't acknowledge transmission, only receipt of data
  - If segment is retransmitted then ACK arrives, it is not possible to tell if the ACK is for the first or second transmission of the segment
  - If you associate the ACK with the wrong transmission of the segment, it skews it one way or the other



Karn/Partridge Algorithm (cont.)

- To resolve, TCP doesn't measure sample RTT when it has to retransmit
- Karn/Partridge also added exponential backoff on timeout time
  - Next timeout is twice last timeout if retransmitting
  - Makes sender react more cautiously as segments are lost, which helps in the case of network congestion



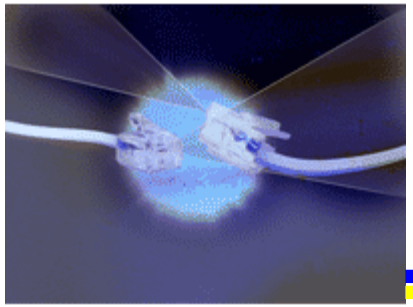
Jacobsen/Karels Algorithm

- Karn/Partridge improved congestion problem, but did not eliminate it
- Jacobson/Karels was bigger change to TCP to address congestion
  - Want to avoid retransmitting unnecessarily
  - Keep track of variance among samples, and use it to scale the timeout adjustment

    Diff = SampleRTT – EstRTT

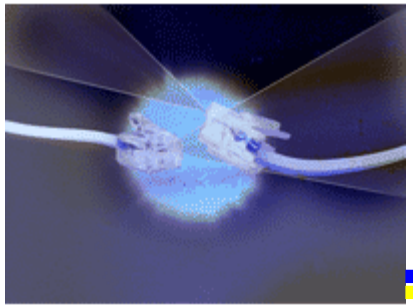
    EstRTT = EstRTT + (δ * Diff)

    Dev = Dev + δ (|Diff| - Dev)
  - Timeout = μ * EstRTT + φ * Dev (typically μ=1, φ=4)



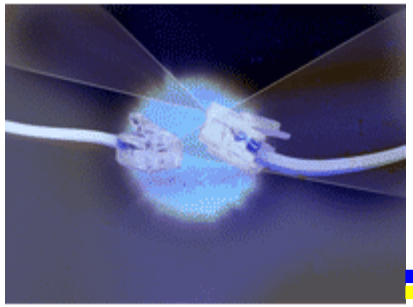
Record Boundaries

- Two mechanisms for inserting record markers in stream
  - Using the “Urgent data” feature, UrgPtr
  - Using the “push” operation – partial segment sent with PUSH flag set; receiver must notify app that it was a push



TCP Extensions

- Both sides of connection agree on which extensions to allow during connection setup
  - Sender puts send timestamp into extension header, receiver echoes value in ACK, sender compare value with current time to compute difference
  - To increase SequenceNum, prepend timestamp to number. Modified field only used to prevent wrapping, not for ordering



TCP Extensions

– Advertise larger window than provided by 16-bit field in header – both sides agree on scaling factor, so window is in chunks instead of bytes