Transport Protocols

Jasleen Kaur

Spring 2012

BASIC INTERFACE

Multiplexing, Connection Oriented Semantics
Transport Protocols: Basic Functionality

- Provide a “process-to-process” communication channel
  - Versus the host-to-host abstraction provided by lower layers

- Wish-list from such a channel:
  - Guarantees message delivery
    - Guarantees in-order delivery
    - Guarantees no duplicate messages
  - Supports arbitrarily large messages
  - Supports multiple application processes on each host
  - Guarantees bounded delay
    - Helps support synchronization between sender and receiver
  - Allows receiver to control data flow from sender
  - Secure and private

**How to provide these services on top of IP's best-effort service?**

User Datagram Protocol (UDP)

- Simplest possible service
  - Extend from host-to-host to process-to-process communication
  - Simply add a level of de-multiplexing
    - Since there may be several processes running on a host

- How to identify processes?
  - Ports/mailboxes:
    - Abstract locator for sending messages to, and receiving messages from
  - Allow multiple channels to be established in same process
  - Host-local scope:
    - host ID + port number uniquely identify a channel

- How to learn of destination port number?
  - Servers use well-known port numbers (or port-mapper service)
**UDP Segment Format**

- **Includes**
  - 16-bit port numbers
  - Checksum (UDP segment + pseudo-header)
  - Length
  - Data (from application)

```
<table>
<thead>
<tr>
<th>0</th>
<th>16</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>SrcPort</td>
<td>DstPort</td>
<td></td>
</tr>
<tr>
<td>Checksum</td>
<td>Length</td>
<td></td>
</tr>
<tr>
<td>Data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Transmission Control Protocol (TCP)**

- **Basic interface:**
  - **Byte-stream service:**
    - Application writes bytes
    - TCP sends segments
    - Application reads bytes
  - **Full-duplex:**
    - Data can be transferred (independently) in both directions
  - **Connection-oriented:**
    - Connection state is established and used
**Additional TCP Services Offered**

- Reliable delivery
- In-order delivery
- Receiver-limited flow control
- Congestion control

**TCP Segment Format**

- Each byte has a sequence number (byte stream)
- Each side has independent seq numbers (full-duplex)
- ACKs used to confirm delivery (reliability, in-order)
- AdvertisedWindow used to control sender (flow-control)
- Flags used for control information (connection-oriented)
**Connection Establishment**

- **Three-way Handshake:**
  - Two sides agree on starting sequence numbers to use
    - Why not start from 0?
  - SequenceNum = Next sequence expected

**Diagram:**

- SYN, SequenceNum = x
- SYN + ACK, SequenceNum = y
- Acknowledgment = x + 1
- ACK, Acknowledgment = y + 1

**Complex State Management Protocol**

- **Just a partial view...**

**Diagram:**

- States: CLOSED, LISTEN, SYN_RCVD, ESTABLISHED, FIN_WAIT_1, FIN_WAIT_2, CLOSING, TIME_WAIT, LAST_ACK, CLOSED
- Transitions with events like SYN, SYN + ACK, FIN, ACK, FIN + ACK, CLOSE_WAIT, LAST_ACK
RELIABILITY & FLOW CONTROL

Sliding Window & RTOs

Recovering Lost Data: ARQ

- Segment loss may occur due to:
  - non-recoverable bit errors
  - buffer overflow
  - multiple vain attempts at shared medium access
- Need mechanisms to recover lost frames
- Basic idea: use Automatic Repeat Request (ARQ)
  - Acknowledgements (sent by receiver)
    - Indicate successful delivery
  - Timeouts (maintained by sender)
    - Indicate when packet is likely to be lost
ARQ Example: Stop-and-Wait Protocol

- Sender sends next packet only after it receives ACK for transmitted packet
  - At most one unacknowledged packet in flight
  - Uses sequence numbers to distinguish retransmissions from new packets
    - How many distinct sequence numbers needed?

Stop-and-Wait: Inefficiency

- Expected throughput in a stop-and-wait transfer?
  - May prevent transfer from utilizing available capacity
  - e.g., 1 KB frame size, 1.5 Mbps link, 45 ms RTT
    - $\Rightarrow$ 1/8th link utilization

- Keeping the pipe full:
  - How much data should be in transit to completely utilize the bottleneck bandwidth?
  - 1.5Mbps link x 45ms RTT = 67.5Kb (8KB)

Should allow multiple un-acknowledged frames
**Efficient ARQ Example: Sliding Window**

- Allows multiple unacknowledged packets
  - Number of unacknowledged packets upper bounded by window

**Sender Receiver**

**Time**

- Sender assigns seq no. to every byte
- Sender maintains:
  - Last Byte Written (LBWritten)
  - Last Byte Sent (LBSent)
  - Last Byte Acked (LBAcked)
- Invariant: LBAcked \( \leq \) LBSent \( \leq \) LBWritten
- Sender buffers only bytes between: (LBAcked, LBWritten]
  - Retransmits packets that time-out waiting for ACKs

**Reliable, In-order, Byte-stream Delivery**

- Receiver maintains:
  - Last Byte Read (LBRead)
  - Next Byte Expected (NBExp)
  - Last Byte Received (LBRcvd)
- Receiver (cumulative) ACKs send NBExp
- Invariant: LBRead < NBExp \( \leq \) LBRcvd + 1
- Receiver buffers only bytes between: (LBRead, LBRcvd]
### Sliding Window: Flow Control

- **New invariants:**
  - $LBSent - LBAcked \leq AdvWin$
  - $EffectiveWin = AdvWin - (LBSent - LBAcked)$
- **Sender buffer limits** (MaxSendBuf) add invariant:
  - $LBWritten - LBAcked \leq MaxSendBuf$
- **Receiver buffer limits** (MaxRcvBuf) add new invariant
  - $LBRead - LBRcvd \leq MaxRcvBuf$
- **Receiver advertises to sender:**
  - $AdvWin = MaxRcvBuf - [(NBExp - 1) - LBRead]$

**Is this guaranteed to prevent buffer overflow at either ends?**
**Is it possible that an ACK arrival will not allow further transmissions?**

### Sequence Number Space Required

- **How large should the seq number space be?**
  - **As large as the maximum window size?**
  - **MaxSeqNum > 2*Window**

---

1/12/2012
Setting the Retransmission Timer (RTO)

- Central question: How to set retransmission timeout?
  - Internet paths can vary significantly in their propagation length
  - End-to-end RTTs can vary significantly over time
    - RTO should be an adaptive function of current RTT conditions

- How variables can RTTs be in practice?
  - UNC study of 1 million TCP transfers
  - Estimated the per-segment RTT in each transfer

RTT Variability in Practice

- UNC study measured per-segment RTTs for 1 million TCP transfers
  - 70% transfers see a standard deviation of 10 ms or more
RTT Variability in Practice

- Median RTTs can be more than twice of min RTTs for 30% of transfers

Jacobson/Karel’s Algorithm

- Incorporates RTT variability
  - Use variance in RTTs to estimate timeout
- Algo:
  - Calculate running averages of both RTT and its variation
    - $\text{Diff} = \text{SampledRTT} - \text{EstimatedRTT}$
    - $\text{EstimatedRTT} = \text{EstimatedRTT} + (1-a)\text{Diff}$
    - $\text{Deviation} = \text{Deviation} + (1-b)(|\text{Diff}| - \text{Deviation})$
  - $\text{RTO} = \mu \text{EstimatedRTT} + \rho \text{Deviation}$
    - Typically, $\mu = 1$, $\rho = 4$, $a = b = 0.875$
    - Large variance causes Deviation to dominate calculation of RTO
RTO Timers in Practice

- Most Unix TCP implementations check to see if a timeout should occur, only once every clock tick
  - Ultimately, algorithm only as good as granularity of system clocks
    - 10-100 ms for current OSes
- Most implementations take only one RTT sample per RTT
- Most implementations use a minRTO value of 200-1000 ms

```
Timeouts may happen no earlier than 1 second after segment was transmitted!
```

- TCP extensions:
  - Instead of relying on coarse-grained timers for measuring RTTs, use per-packet timestamps to measure RTT