Application Layer

Srinidhi Varadarajan
Application Layer: Applications and Protocols

**Application: communicating, distributed processes**
- running in network hosts in “user space”
- exchange messages to implement app
- e.g., email, file transfer, the Web

**Application-layer protocols**
- one “piece” of an app
- define messages exchanged by apps and actions taken
- user services provided by lower layer protocols
Definitions

- A **process** is a program that is running within a host.

- Within the same host, two processes communicate using **interprocess communication (IPC)** defined by the OS.
  - E.g. pipes, named pipes, FIFO queues, mailboxes

- Processes running in different hosts communicate with an **application-layer protocol**

- A **user agent** is an interface between the user and the network application.
  - Web: browser
  - E-mail: mail reader
  - Streaming audio/video: media player
Client-server paradigm

Client:
- initiates contact with server ("speaks first")
- typically requests service from server,
- for Web, client is implemented in browser; for e-mail, in mail reader

Server:
- provides requested service to client
- e.g., Web server sends requested Web page, mail server delivers e-mail
Application Layer Programming

API: application programming interface
- defines interface between application and transport layer

sockets: Internet API
- two processes communicate by sending data into socket, reading data out of socket
How do you identify the client and server in a client-server environment?

- Each end-host on the Internet is identified by a unique 32 bit address (IPv4). IPv6 uses 64 bit addresses.
- IP addresses are hierarchical. The address consists of two parts, a network address and a host address.
- To identify a network application within a end-host, the transport protocol (TCP or UDP) defines a multiplexing/demultiplexing mechanism called a *port*.
- A port is a 16 bit unsigned integer.
- A port is *owned* by at most one network application. The converse is not true. A network application may own multiple ports.
- When the network stack at an end host receives a packet, it uses the dest. port to determine the target application of the packet.
- Client server communication occurs between application specified by <source IP, source port> and <dest. IP, dest. port>.
Choice of Transport Protocols

Data loss
- some apps (e.g., audio) can tolerate some loss
- other apps (e.g., file transfer, telnet) require 100% reliable data transfer

Temporal constraints
- some apps (e.g., Internet telephony, interactive games) require low delay to be “effective”
- Temporal constraints may be expressed as:
  - End-to-end delay: delay from packet transmission at source to reception by target application
  - Delay Jitter: Variation in delay observed by successive packets

Bandwidth
- some apps (e.g., multimedia) require minimum amount of bandwidth to be “effective”
- other apps (“elastic apps”) make use of whatever bandwidth they get
## Application/Data Characteristics

<table>
<thead>
<tr>
<th>Application</th>
<th>Data loss</th>
<th>Bandwidth</th>
<th>Time Sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td>file transfer</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>e-mail</td>
<td>no loss</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>Web documents</td>
<td>loss-tolerant</td>
<td>elastic</td>
<td>no</td>
</tr>
<tr>
<td>real-time audio/video</td>
<td>loss-tolerant</td>
<td>audio: 5Kb-1Mb</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td></td>
<td></td>
<td>video: 10Kb-5Mb</td>
<td></td>
</tr>
<tr>
<td>stored audio/video</td>
<td>loss-tolerant</td>
<td>same as above</td>
<td>yes, few secs</td>
</tr>
<tr>
<td>interactive games</td>
<td>loss-tolerant</td>
<td>few Kbps up</td>
<td>yes, 100’s msec</td>
</tr>
<tr>
<td>financial apps</td>
<td>no loss</td>
<td>elastic</td>
<td>yes and no</td>
</tr>
</tbody>
</table>
## Transport Protocols of Internet Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Application layer protocol</th>
<th>Underlying transport protocol</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-mail</td>
<td>smtp [RFC 821]</td>
<td>TCP</td>
</tr>
<tr>
<td>remote terminal access</td>
<td>telnet [RFC 854]</td>
<td>TCP</td>
</tr>
<tr>
<td>Web</td>
<td>http [RFC 2068]</td>
<td>TCP</td>
</tr>
<tr>
<td>file transfer</td>
<td>ftp [RFC 959]</td>
<td>TCP</td>
</tr>
<tr>
<td>streaming multimedia</td>
<td>proprietary (e.g. RealNetworks)</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>remote file server</td>
<td>NFS</td>
<td>TCP or UDP</td>
</tr>
<tr>
<td>Internet telephony</td>
<td>proprietary (e.g., Vocaltec)</td>
<td>typically UDP</td>
</tr>
</tbody>
</table>
Socket Interface. What is it?

- Gives a file system like abstraction to the capabilities of the network.

- Each transport protocol offers a set of services. The socket API provides the abstraction to access these services.

- The API defines function calls to create, close, read and write to/from a socket.
Creating a socket

int socket(int domain, int type, int protocol)

Protocol Family:
PF_INET or PF_UNIX

Communication semantics:
SOCK_STREAM or SOCK_DGRAM

Usually UNSPEC

The call returns a integer identifier called a handle
What do you need for socket communication?

- Basically 4 parameters
  - Source Identifier (IP address)
  - Source Port
  - Destination Identifier
  - Destination Port

- In the socket API, this information is communicated by binding the socket.
Binding a socket

int bind (int socket, struct sockaddr *address, int addr_len)

• This call is executed by:
  – Server in TCP
  – Both sides in UDP

• It binds the socket to the specified address. The address parameter specifies the local component of the address, e.g. IP address and UDP/TCP port
TCP Server Side: Listen

int listen (int socket, int backlog)

- This server side call specifies the number of pending connections on the given socket.

- When the server is processing a connection, “backlog” number of connections may be pending in a queue.
TCP Server Side: Passive Open

int accept (int socket, struct sockaddr *address, int *addr_len)

- This call is executed by the server.
- The call does not return until a remote client has established a connection.
- When it completes, it returns a new socket handle corresponding to the just-established connection.
TCP Client Side: Active Open

int connect (int socket, struct sockaddr *address, int *addr_len)

- This call is executed by the client. *address contains the remote address.

- The call attempts to connect the socket to a server. It does not return until a connection has been established.

- When the call completes, the socket “socket” is connected and ready for communication.
Sockets: Summary

- **Client:**
  
  int socket(int domain, int type, int protocol)
  int connect (int socket, struct sockaddr *address, int addr_len)

- **Server:**

  int socket(int domain, int type, int protocol)
  int bind (int socket, struct sockaddr *address, int addr_len)
  int listen (int socket, int backlog)
  int accept (int socket, struct sockaddr *address, int *addr_len)
Message Passing

- int send (int socket, char *message, int msg_len, int flags) (TCP)
- int sendto (int socket, void *msg, int len, int flags, struct sockaddr *to, int tolen); (UDP)
- int write(int socket, void *msg, int len); /* TCP */

- int recv (int socket, char *buffer, int buf_len, int flags) (TCP)
- int recvfrom(int socket, void *msg, int len, int flags, struct sockaddr *from, int *fromlen); (UDP)
- int read(int socket, void *msg, int len); (TCP)