Application-Layer Protocols
Overview

- Application-layer protocols define:
  - The types of messages exchanged
  - The syntax and semantics of messages
  - The rules for when and how messages are sent

- Public protocols (defined in RFCs)
  - HTTP, FTP, SMTP, POP, IMAP, DNS

- Proprietary protocols
  - RealAudio, RealVideo
  - Skype
  - …
Hypertext Transfer Protocol -- HTTP/1.1

Abstract

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers [47]. A feature of HTTP is the typing and negotiation of data representation, allowing systems to be built independently of the data being transferred.

HTTP has been in use by the World-Wide Web global information initiative since 1990. This specification defines the protocol referred to as "HTTP/1.1", and is an update to RFC 2068 [33].

Application-Layer Protocols

Overview

- Application-layer protocols define:
  - The types of messages exchanged
  - The syntax and semantics of messages
  - The rules for when and how messages are sent

- Public protocols (defined in RFCs)
  - HTTP, FTP, SMTP, POP, IMAP, DNS

- Proprietary protocols
  - RealAudio, RealVideo
  - Skype
  - Skype
  - …
**Application-Layer Protocols**

**Overview**

- Application developers write programs that:
  - Run on (different) end systems
  - Communicate over network

- Note: application developers don't need to write code for network-core devices
  - Network devices do not run user applications or application layer protocols

**Application-Layer Protocols**

**Outline**

- The architecture of distributed systems
  - Client/Server computing
  - Peer-to-Peer computing
  - Content delivery networks

- The programming model used in constructing distributed systems
  - Socket programming
Application-Layer Protocols

Outline

- Example client/server systems and their application-level protocols:
  - The World-Wide Web (HTTP)
  - Reliable file transfer (FTP)
  - E-mail (SMTP & POP)
  - Internet Domain Name System (DNS)
- Example p2p applications systems:
  - BitTorrent
- Other protocols and systems:
  - Streaming media — DASH
  - Content delivery networks (CDNs)

The Application Layer

The client-server paradigm

- Typical network application has two pieces: client and server
- Client:
  - Initiates contact with server ("speaks first")
  - Requests service from server
  - For Web, client is implemented in browser; for e-mail, in mail reader
- Server:
  - Provides requested service to client
  - "Always" running
  - May also include a "client interface"
  - A server may be a logical machine
    - Implemented by one of thousands of physical servers in a data center
The Application Layer
The peer-to-peer-paradigm

- No “always-on” server
- Arbitrary end systems directly communicate
  - Peers request service from other peers, provide service in return to other peers
- Self scalability – new peers bring new service capacity, as well as new service demands
- Peers are intermittently connected and change IP addresses
- Complex management

Application-Layer Protocols
Outline

- Example client/server systems and their application-level protocols
  - The World-Wide Web (HTTP)
  - Reliable file transfer (FTP)
  - E-mail (SMTP & POP)
  - Internet Domain Name System (DNS)
- Protocol design issues:
  - In-band vs. out-of-band control signaling
  - Push vs. pull protocols
  - Persistent vs. non-persistent connections
- Client/server service architectures
  - Contacted server responds vs. forwards request
Client/Server Paradigm
Socket programming

- Sockets are the fundamental building block for client/server systems
- Sockets are created and managed by applications
  » Strong analogies with files

- Two types of transport services are available via the socket API:
  » UDP sockets: unreliable, datagram-oriented communications
  » TCP sockets: reliable, stream-oriented communications

Client/Server Paradigm
A quick aside on processes

- A process is the OS term for a program running within a host
- On the same host, two processes communicate using inter-process communication
  » A service defined by the OS
- Processes on different hosts communicate by exchanging messages
  » By using some protocol!
**Client/Server Paradigm**

*Socket-programming using TCP*

- A socket is an application created, OS-controlled interface into which an application can both send and receive messages to and from another application
  - A “door” between application processes and end-to-end transport protocols

**Client/Server Paradigm**

*Addressing processes*

- To receive messages, a process must have an identifier
  - How does a client identify a server process?
- We know that a host device has unique 32-bit IP address
- But does the IP address of host suffice for identifying the destination process?
  - No! Many processes can be (and are!) running on the same host

---

**Diagram Notes:**

- **Host (end system)**
- **Internet**
- **Network**
- **Application**
- **Physical Layer**
- **Transport Layer**
- **Regional ISP**
- **Institutional Network**
- **Transport Layer**
- **Physical Layer**
Client/Server Paradigm
Addressing processes

- Processes are identified by a “port number”
  » Sort of like a socket identifier
- The “server” identifier includes both an IP address and port numbers associated with the server process on the host
- Example port numbers:
  » HTTP server: 80
  » mail server: 25
- For a browser to send an HTTP message to www.cs.unc.edu the request is addressed to IP address 152.2.131.244 and port 80

Socket-programming using TCP
TCP socket programming model

- A TCP socket provides a reliable, bi-directional, byte-stream communications channel from one process to another
  » A “pair of pipes” abstraction
Socket-programming using TCP

Network addressing for sockets

- Sockets are addressed using an IP address and port number

Socket programming in Python

- Client creates a local TCP socket specifying the host and port number of server process
  - Python resolves host names to IP addresses using DNS
- Client contacts server
  - Server process must be running
  - Server must have created socket that “welcomes” client’s contact
- When the client creates a socket, the client’s TCP establishes connection to server’s TCP
- When contacted by a client, server creates a new socket for server process to communicate with client
  - This allows the server to talk with multiple clients
Socket-programming using TCP
Socket creation in the client-server model

- Process
  - Client socket
  - Socket
  - TCP 3-way handshake
  - Internet

- Server
  - Server socket
  - Socket
  - Connection socket
  - 3-way handshake

Simple client-server example

- **The client reads a line of text from standard input and sends the text to the server via a socket**
- **The server receives the line of text from the client and converts the line of characters to all uppercase**
- **The server sends the converted line back to the client**
- **The client receives the converted text and writes it to standard output**
Socket programming with TCP Example
Client/server TCP socket interaction in Python

**Server (running on swan.cs.unc.edu)**
- create socket for incoming request (port=6789)
  - serverSocket = socket(...)
- wait for incoming connection request
  - connectionSocket = serverSocket.accept()
- read request from connectionSocket
  - write request to connectionSocket
  - close connectionSocket

**Client (running on classroom.cs...)**
- create socket, connect to swan.cs.unc.edu, port=6789
  - clientSocket = socket(...) (socket library)
- read request from connectionSocket
  - write request using clientSocket
  - read reply from clientSocket
  - close clientSocket

**TCP connection setup**
**Program flow**

**Socket Programming with TCP Example**
**Python client**
- from socket import *
- serverName = 'snapper.cs.unc.edu'
- serverPort = 12000
- include Python's socket library
- create TCP socket to server on port 12000
  - clientSocket = socket(AF_INET, SOCK_STREAM)
  - clientSocket.connect((serverName, serverPort))
- get user keyboard input
  - sentence = raw_input('Input lowercase sentence:')
  - change text into a sequence of bytes before sending
  - clientSocket.send(sentence.encode())
- receive data from server in a buffer
  - modifiedSentence = clientSocket.recv(1024)
  - print ('From Server: ', modifiedSentence.decode())
- close connection
  - clientSocket.close()
Socket Programming with TCP Example
Python server

```python
from socket import *
serverPort = 12000
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
print 'The server is ready to receive'
while True:
    connectionSocket, addr = serverSocket.accept()
    sentence = connectionSocket.recv(1024).
    decode()
    capitalizedSentence = sentence.upper()
    connectionSocket.send(capitalizedSentence.encode())
    connectionSocket.close()
```

Socket Programming with TCP Example
Client/server TCP socket interaction in Python

```python
Server (running on snap.cs.unc.edu)
create socket for incoming request (port=6789)
serverSocket = socket(...) 
wait for incoming connection request 
connectionSocket, addr = serverSocket.accept()

TCP connection setup
write request using clientSocket
write reply to connectionSocket
read reply from connectionSocket

Client (running on classroom.cs...)
create socket, connect to snap.cs.unc.edu, port=6789
clientSocket = socket(...)

TCP connection setup
read request from connectionSocket
write reply to connectionSocket
read reply from clientSocket
```
Socket-programming using UDP

UDP socket programming model

- A UDP socket provides an unreliable bi-directional communication channel from one process to another
  » A "datagram" abstraction

Socket programming with UDP Example

Client/server UDP socket interaction in Python

Server (running on 152.2.131.245)

- create socket for incoming request (port=9876)
  - serverSocket = socket(AF_INET,SOCK_DGRAM)

- read request from serverSocket
- write reply to serverSocket specifying client IP address and port number

Client

- create socket
  - clientSocket = socket(AF_INET,SOCK_DGRAM)

- read address (152.2.131.245, port = 9876) and send datagram using clientSocket
- read reply from clientSocket
- close clientSocket
Python client

```python
from socket import *
serverName = 'hostname'
serverPort = 12000

clientSocket = socket(AF_INET, SOCK_DGRAM)
message = raw_input('Input lowercase sentence:')
clientSocket.sendto(message.encode(), (serverName, serverPort))

modifiedMessage, serverAddress = clientSocket.recvfrom(2048)
print modifiedMessage.decode()
clientSocket.close()
```

Python server

```python
from socket import *
serverPort = 12000

serverSocket = socket(AF_INET, SOCK_DGRAM)
serverSocket.bind(('', serverPort))

print ('The server is ready to receive')

while True:
    message, clientAddress = serverSocket.recvfrom(2048)
    modifiedMessage = message.decode().upper()
    serverSocket.sendto(modifiedMessage.encode(), clientAddress)
```
Socket Programming
Services provided by Internet transport protocols

- TCP service:
  - connection-oriented: setup required between client, server
  - reliable transport: between sending and receiving process
  - flow control: sender won’t overwhelm receiver
  - congestion control: throttle sender when network overloaded
  - does not provide: timing, minimum bandwidth guarantees

- UDP service:
  - unreliable data transfer between sending and receiving process
  - does not provide: connection setup, reliability, flow control, congestion control, timing, or minimum bandwidth guarantees

Why bother? Why is there a UDP?