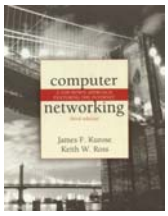
Chapter 2 (continued) Application Layer – part 2



Computer Networking: A Top Down Approach Featuring the Internet, 3rd edition. Jim Kurose, Keith Ross Addison-Wesley, July 2004.

A note on the use of these ppt slides:
We're making these slides freely available to all (faculty, students, readers). They're in PowerPoint form so you can add, modify, and delete slides (including this one) and slide content to suit your needs. They obviously represent a *lot* of work on our part. In return for use, we only ask the following:
- If you use these slides (e.g., in a class) in substantially unaltered form, that you mention their source (after all, we'd like people to use our book!)
- If you post any slides in substantially unaltered form on a www site, that you note that they are adapted from (or perhaps identical to) our slides, and note our copyright of this material.

Thanks and enjoy! JFK/KWR

Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

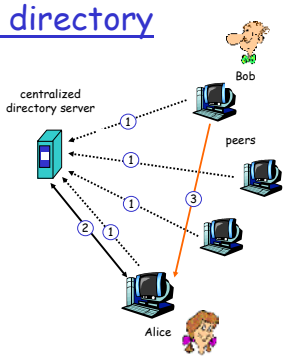
P2P file sharing

Example
- Alice runs P2P client application on her notebook computer
- Intermittently connects to Internet; gets new IP address for each connection
- Asks for "Hey Jude"
- Application displays other peers that have copy of Hey Jude.
- Alice chooses one of the peers, Bob.
- File is copied from Bob's PC to Alice's notebook: HTTP
- While Alice downloads, other users uploading from Alice.
- Alice's peer is both a Web client and a transient Web server.

All peers are servers = highly scalable!

P2P: centralized directory

original "Napster" design
1) when peer connects, it informs central server:
   - IP address
   - content
2) Alice queries for "Hey Jude"
3) Alice requests file from Bob



P2P: problems with centralized directory

- Single point of failure
- Performance bottleneck
- Copyright infringement

file transfer is decentralized, but locating content is highly decentralized

Query flooding: Gnutella

- fully distributed
  - no central server
- public domain protocol
- many Gnutella clients implementing protocol

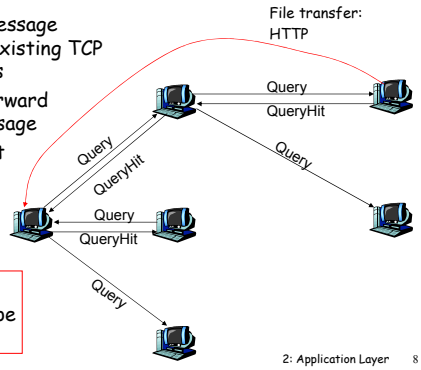
overlay network: graph
- edge between peer X and Y if there's a TCP connection
- all active peers and edges is overlay net
- Edge is not a physical link
- Given peer will typically be connected with < 10 overlay neighbors

Gnutella Messages

Descriptor	Description
Ping	Used to actively discover hosts on the network. A servent receiving a Ping descriptor is expected to respond with one or more Pong descriptors.
Pong	The response to a Ping. Includes the address of a connected Gnutella servent and information regarding the amount of data it is making available to the network.
Query	The primary mechanism for searching the distributed network. A servent receiving a Query descriptor will respond with a QueryHit if a match is found against its local data set.
QueryHit	The response to a Query. This descriptor provides the recipient with enough information to acquire the data matching the corresponding Query.
Push	A mechanism that allows a firewalled servent to contribute file-based data to the network.

Gnutella: protocol

- Query message sent over existing TCP connections
- peers forward Query message
- QueryHit sent over reverse path



Scalability: limited scope flooding

Gnutella Connection Setup

GNUTELLA CONNECT/<protocol version string>\n\n

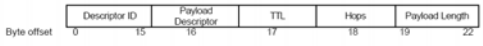
where <protocol version string> is defined to be the ASCII string "0.4" (or, equivalently, "\x30\x2e\x34") in this version of the specification.

A servent wishing to accept the connection request must respond with

GNUTELLA OK\n\n

Gnutella Message Header

Descriptor Header



Descriptor ID *A 16-byte string uniquely identifying the descriptor on the network*

Payload Descriptor
0x00 = Ping
0x01 = Pong
0x40 = Push
0x80 = Query
0x81 = QueryHit

Gnutella Message Header (Cont.)

TTL *Time To Live. The number of times the descriptor will be forwarded by Gnutella servents before it is removed from the network. Each servent will decrement the TTL before passing it on to another servent. When the TTL reaches 0, the descriptor will no longer be forwarded.*

Hops *The number of times the descriptor has been forwarded. As a descriptor is passed from servent to servent, the TTL and Hops fields of the header must satisfy the following condition:*

TTL(0) = TTL(i) + Hops(i)

Where TTL(*i*) and Hops(*i*) are the value of the TTL and Hops fields of the header at the descriptor's *i*-th hop, for *i* >= 0.

Payload Length *The length of the descriptor immediately following this header. The next descriptor header is located exactly Payload_Length bytes from the end of this header i.e. there are no gaps or pad bytes in the Gnutella data stream.*

Ping Message

Ping (0x00)
Ping descriptors have no associated payload and are of zero length. A Ping is simply represented by a Descriptor Header whose Payload_Descriptor field is 0x00 and whose Payload_ Length field is 0x00000000.

A servent uses Ping descriptors to actively probe the network for other servents. A servent receiving a Ping descriptor may elect to respond with a Pong descriptor, which contains the address of an active Gnutella servent (possibly the one sending the Pong descriptor) and the amount of data it's sharing on the network.

This specification makes no recommendations as to the frequency at which a servent should send Ping descriptors, although servent implementers should make every attempt to minimize Ping traffic on the network.

Pong Message

Pong (0x01)

Byte offset	Port	IP Address	Number of Files Shared	Number of Kilobytes Shared
	0 1	2 5	6 9	10 13

Port *The port number on which the responding host can accept incoming connections.*

IP Address *The IP address of the responding host.*

This field is in big-endian format.

Number of Files Shared *The number of files that the servent with the given IP address and port is sharing on the network.*

Number of Kilobytes Shared *The number of kilobytes of data that the servent with the given IP address and port is sharing on the network.*

Pong descriptors are only sent in response to an incoming Ping descriptor. It is valid for more than one Pong descriptor to be sent in response to a single Ping descriptor. This enables host caches to send cached servent address information in response to a Ping request.

Big-Endian vs. Little-Endian

big-endian

The adjectives *big-endian* and *little-endian* refer to which bytes are most significant in multi-byte data types and describe the order in which a sequence of bytes is stored in a computer's memory.

In a big-endian system, the most significant value in the sequence is stored at the lowest storage address (i.e., first). In a little-endian system, the least significant value in the sequence is stored first. For example, consider the number 1025 (2 to the tenth power plus one) stored in a 4-byte integer:

00000000 00000000 00000100 00000001

Address	Big-Endian representation of 1025	Little-Endian representation of 1025
00	00000000	00000001
01	00000000	00000100
02	00000100	00000000
03	00000001	00000000

Many mainframe computers, particularly IBM mainframes, use a big-endian architecture. Most modern computers, including PCs, use the little-endian system. The PowerPC system is *bi-endian* because it can understand both systems.

Query Message

Query (0x80)

Byte offset	Minimum Speed	Search criteria
	0 1	2 ...

Minimum Speed *The minimum speed (in kb/second) of servents that should respond to this message. A servent receiving a Query descriptor with a Minimum Speed field of n kb/s should only respond with a QueryHit if it is able to communicate at a*

speed >= n kb/s

Search Criteria *A nul (i.e. 0x00) terminated search string. The maximum length of this string is bounded by the Payload_Length field of the descriptor header.*

QueryHit Message

QueryHit (0x81)

Byte offset	Number of Hits	Port	IP Address	Speed	Result Set	Servent Identifier
	0	1 2	3 6	7 10	11 ... n	n + 16

Number of Hits *The number of query hits in the result set (see below).*

Port *The port number on which the responding host can accept incoming connections.*

IP Address *The IP address of the responding host.*

This field is in big-endian format.

Speed *The speed (in kb/second) of the responding host.*

QueryHit Message (cont.)

Result Set *A set of responses to the corresponding Query. This set contains Number_of_Hits elements, each with the following structure:*

Byte offset	File Index	File Size	File Name
	0 3	4 7	8 ...

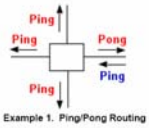
File Index *A number, assigned by the responding host, which is used to uniquely identify the file matching the corresponding query.*

File Size *The size (in bytes) of the file whose index is File_Index.*

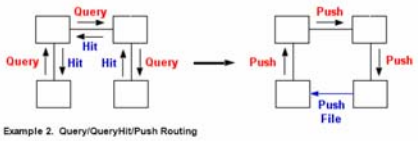
File Name *The double-nul (i.e. 0x0000) terminated name of the file whose index is File_Index.*

The size of the result set is bounded by the size of the Payload_Length field in the Descriptor Header.

Query Routing



Example 1. Ping/Pong Routing



Example 2. Query/QueryHit/Push Routing

File Download

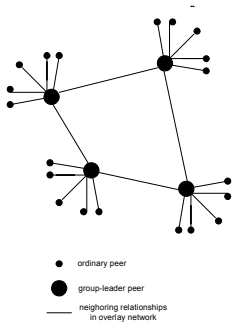
GET /get/<File Index>/<File Name>/ HTTP/1.0\r\n
Connection: Keep-Alive\r\n
Range: bytes=0-\r\n
User-Agent: Gnutella\r\n[3]
\r\n

where <File Index> and <File Name> are one of the File Index/File Name pairs from a QueryHit descriptor's Result Set. For example, if the Result Set from a QueryHit descriptor contained the entry

File Index	*2468*
File Size	*4356789*
File Name	*Foobar.mp3\x00\x00*

Exploiting heterogeneity: Gnutella v. 2

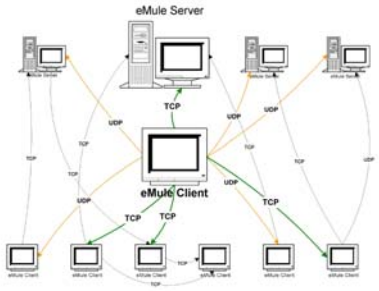
- Each peer is either a supernode or assigned to a supernode.
  - TCP connection between peer and its group leader.
  - TCP connections between some pairs of group leaders.
- Supernode tracks the content in all its children.



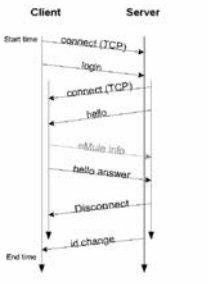
Gnutella v. 2: Querying

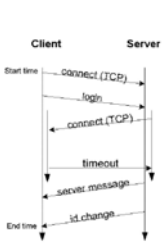
- On connection client updates its supernode with all its files
- Client sends keyword query to its supernode
- Supernode responds with matches:
- Supernode forwards query to other supernodes
- Client then selects files for downloading

eMule



eMule connection setup





Connection startup

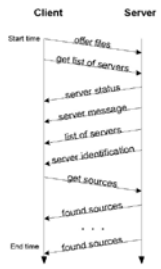


Figure 2.4: Connection startup sequence

File Search

Figure 2.5: File search sequence

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

Socket programming

Goal: learn how to build client/server application that communicate using sockets

Socket API
- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by apps
- client/server paradigm
- two types of transport service via socket API:
  - unreliable datagram
  - reliable, byte stream-oriented

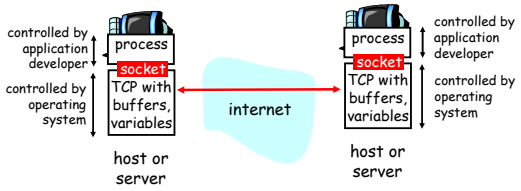
socket

a *host-local, application-created, OS-controlled* interface (a "door") into which application process can both send and receive messages to/from another application process

Socket-programming using TCP

Socket: a door between application process and end-end-transport protocol (UCP or TCP)

TCP service: reliable transfer of **bytes** from one process to another



Socket programming *with TCP*

Client must contact server
- server process must first be running
- server must have created socket (door) that welcomes client's contact

Client contacts server by:
- creating client-local TCP socket
- specifying IP address, port number of server process
- When client creates socket: client TCP establishes connection to server TCP

- When contacted by client, server TCP creates new socket for server process to communicate with client
  - allows server to talk with multiple clients
  - source port numbers used to distinguish clients (more in Chap 3)

application viewpoint

TCP provides reliable, in-order transfer of bytes ("pipe") between client and server

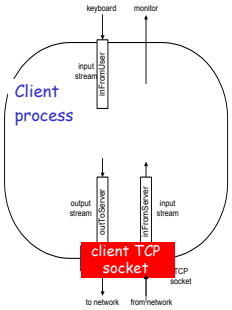
Stream jargon

- A stream is a sequence of characters that flow into or out of a process.
- An input stream is attached to some input source for the process, eg, keyboard or socket.
- An output stream is attached to an output source, eg, monitor or socket.

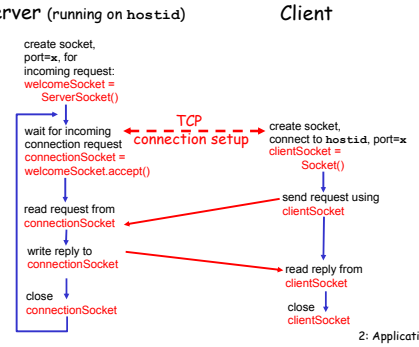
Socket programming with TCP

Example client-server app:

1) client reads line from standard input (`inFromUser` stream) , sends to server via socket (`outToServer` stream)
2) server reads line from socket
3) server converts line to uppercase, sends back to client
4) client reads, prints modified line from socket (`inFromServer` stream)



Client/server socket interaction: TCP



Example: Java client (TCP)

```
import java.io.*;
import java.net.*;
class TCPClient {

   public static void main(String argv[]) throws Exception
   {
      String sentence;
      String modifiedSentence;

      BufferedReader inFromUser =
        new BufferedReader(new InputStreamReader(System.in));

      Socket clientSocket = new Socket("hostname", 6789);

      DataOutputStream outToServer =
        new DataOutputStream(clientSocket.getOutputStream());
```

Create input stream

Create client socket, connect to server

Create output stream attached to socket

Example: Java client (TCP), cont.

```
      BufferedReader inFromServer =
        new BufferedReader(new
        InputStreamReader(clientSocket.getInputStream()));

      sentence = inFromUser.readLine();

      outToServer.writeBytes(sentence + '\n');

      modifiedSentence = inFromServer.readLine();

      System.out.println("FROM SERVER: " + modifiedSentence);

      clientSocket.close();

   }
}
```

Create input stream attached to socket

Send line to server

Read line from server

Example: Java server (TCP)

```
import java.io.*;
import java.net.*;

class TCPServer {

  public static void main(String argv[]) throws Exception
   {
     String clientSentence;
     String capitalizedSentence;

     ServerSocket welcomeSocket = new ServerSocket(6789);

     while(true) {

       Socket connectionSocket = welcomeSocket.accept();

       BufferedReader inFromClient =
         new BufferedReader(new
         InputStreamReader(connectionSocket.getInputStream()));
```

Create welcoming socket at port 6789

Wait, on welcoming socket for contact by client

Create input stream, attached to socket

Example: Java server (TCP), cont

```
       DataOutputStream  outToClient =
         new DataOutputStream(connectionSocket.getOutputStream());

       clientSentence = inFromClient.readLine();

       capitalizedSentence = clientSentence.toUpperCase() + '\n';

       outToClient.writeBytes(capitalizedSentence);
     }
   }
}
```

Create output stream, attached to socket

Read in line from socket

Write out line to socket

End of while loop, loop back and wait for another client connection

Chapter 2: Application layer

- 2.1 Principles of network applications
- 2.2 Web and HTTP
- 2.3 FTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

Socket programming *with UDP*

UDP: no "connection" between client and server

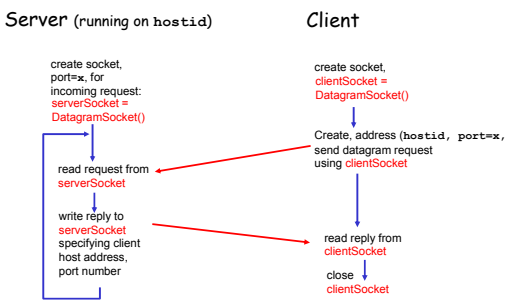
- no handshaking
- sender explicitly attaches IP address and port of destination to each packet
- server must extract IP address, port of sender from received packet

UDP: transmitted data may be received out of order, or lost

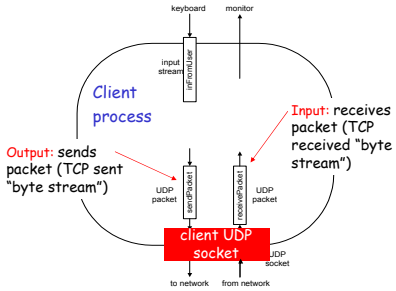
application viewpoint

UDP provides unreliable transfer of groups of bytes ("datagrams") between client and server

Client/server socket interaction: UDP



Example: Java client (UDP)



Example: Java client (UDP)

```
import java.io.*;
import java.net.*;

class UDPClient {
   public static void main(String args[]) throws Exception
   {
     BufferedReader inFromUser =
       new BufferedReader(new InputStreamReader(System.in));

     DatagramSocket clientSocket = new DatagramSocket();

     InetAddress IPAddress = InetAddress.getByName("hostname");

     byte[] sendData = new byte[1024];
     byte[] receiveData = new byte[1024];

     String sentence = inFromUser.readLine();
     sendData = sentence.getBytes();
```

Create input stream

Create client socket

Translate hostname to IP address using DNS

Example: Java client (UDP), cont.

```
     DatagramPacket sendPacket =
       new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

     clientSocket.send(sendPacket);

     DatagramPacket receivePacket =
       new DatagramPacket(receiveData, receiveData.length);

     clientSocket.receive(receivePacket);

     String modifiedSentence =
       new String(receivePacket.getData());

     System.out.println("FROM SERVER:" + modifiedSentence);
     clientSocket.close();
     }
}
```

Create datagram with data-to-send, length, IP addr, port

Send datagram to server

Read datagram from server

Example: Java server (UDP)

```
import java.io.*;
import java.net.*;

class UDPServer {
  public static void main(String args[]) throws Exception
    {
      DatagramSocket serverSocket = new DatagramSocket(9876);

      byte[] receiveData = new byte[1024];
      byte[] sendData  = new byte[1024];

      while(true)
        {
          DatagramPacket receivePacket =
            new DatagramPacket(receiveData, receiveData.length);
          serverSocket.receive(receivePacket);
```

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

Example: Java server (UDP), cont

```
          String sentence = new String(receivePacket.getData());
          InetAddress IPAddress = receivePacket.getAddress();
          int port = receivePacket.getPort();
          String capitalizedSentence = sentence.toUpperCase();
          sendData = capitalizedSentence.getBytes();
          DatagramPacket sendPacket =
            new DatagramPacket(sendData, sendData.length, IPAddress,
                               port);
          serverSocket.send(sendPacket);
        }
    }
}
```

Get IP addr port #, of sender

Create datagram to send to client

Write out datagram to socket

End of while loop, loop back and wait for another datagram

Chapter 2: Application layer

- 2.1 Principles of network applications
  - app architectures
  - app requirements
- 2.2 Web and HTTP
- 2.4 Electronic Mail
  - SMTP, POP3, IMAP
- 2.5 DNS
- 2.6 P2P file sharing
- 2.7 Socket programming with TCP
- 2.8 Socket programming with UDP
- 2.9 Building a Web server

Building a simple Web server

- handles one HTTP request
- accepts the request
- parses header
- obtains requested file from server's file system
- creates HTTP response message:
  - header lines + file
- sends response to client
- after creating server, you can request file using a browser (eg IE explorer)
- see text for details

Chapter 2: Summary

Our study of network apps now complete!

- Application architectures
  - client-server
  - P2P
  - hybrid
- application service requirements:
  - reliability, bandwidth, delay
- Internet transport service model
  - connection-oriented, reliable: TCP
  - unreliable, datagrams: UDP
- specific protocols:
  - HTTP
  - FTP
  - SMTP, POP, IMAP
  - DNS
- socket programming

Chapter 2: Summary

Most importantly: learned about *protocols*

- typical request/reply message exchange:
  - client requests info or service
  - server responds with data, status code
- message formats:
  - headers: fields giving info about data
  - data: info being communicated
- control vs. data msgs
  - in-band, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- "complexity at network edge"