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/owned, OS-controlled interface (a “door”) into which application process can both send and receive messages to/from another (remote or local) application process

Socket-programming using TCP

**Socket:** a door between application process and end-end-transport protocol (UDP 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

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

Socket programming with TCP

Example client-server app:
- client reads line from standard input (inFromUser stream), sends to server via socket (outToServer stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads, prints modified line from socket (inFromServer stream)

Input stream: sequence of bytes into process
Output stream: sequence of bytes out of process
**Client/server socket interaction: TCP**

**Server (running on hostid)**

- create socket, port=x, for incoming request:
  - welcomeSocket = ServerSocket()
- wait for incoming connection request:
  - connectionSocket = welcomeSocket.accept()
- read request from connectionSocket
- write reply to connectionSocket
- close connectionSocket

**Client**

- create socket, connect to hostid, port=x
- clientSocket = Socket()
- send request using clientSocket
- read reply from connectionSocket
- close clientSocket

---

**Example: Java client (TCP)**

```java
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());
        BufferedReader inFromServer = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        String response = inFromServer.readLine();
        System.out.println("Response: "+response);
    }
}
```
Example: Java client (TCP), cont.

```java
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();
```

Example: Java server (TCP)

```java
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 input stream attached to socket

Create welcoming socket at port 6789

Send line to server

Read line from server
Example: Java server (TCP), cont

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

clientSentence = inFromClient.readLine();
capitalizedSentence = clientSentence.toUpperCase() + '
';
outToClient.writeBytes(capitalizedSentence);
}
}
```

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

Socket programming with UDP

UDP: no “connection” between
client and server

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

UDP: transmitted data may be
received out of order, or lost
Client/server socket interaction: UDP

Server (running on hostid)

- Create socket, port=x, for incoming request:
  - serverSocket = DatagramSocket()

- Read request from serverSocket

- Write reply to serverSocket specifying client host address, port number

Client

- Create socket, clientSocket = DatagramSocket()

- Create, address (hostid, port=x), send datagram request using clientSocket

- Read reply from clientSocket

- Close clientSocket

Example: Java client (UDP)

```java
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();
        InetAddress hostAddress = InetAddress.getLocalHost();
        sendPacket = new DatagramPacket(sendData, sendData.length, hostAddress, port);
        clientSocket.send(sendPacket);
    }
}
```
**Example: Java client (UDP), cont.**

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

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

Send datagram to server

```java
DatagramPacket receivePacket =
    new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);
```

Read datagram from server

```java
String modifiedSentence =
    new String(receivePacket.getData());
System.out.println("FROM SERVER:" + modifiedSentence);
clientSocket.close();
```

---

**Example: Java server (UDP)**

```java
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);
        }
    }
}
```
Example: Java server (UDP), cont

```java
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);
```

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

Electronic Mail

Three major components:
- user agents
- mail servers
- simple mail transfer protocol: smtp

User Agent
- a.k.a. "mail reader"
- composing, editing, reading mail messages
- e.g., Eudora, pine, elm, Netscape Messenger
- outgoing, incoming messages stored on server
Electronic Mail: mail servers

Mail Servers
- **mailbox** contains incoming messages (yet to be read) for user
- **message queue** of outgoing (to be sent) mail messages
- **smtp protocol** between mail server to send email messages
  - client: sending mail server
  - “server”: receiving mail server

Electronic Mail: smtp [RFC 821]

- uses tcp to reliably transfer email msg from client to server, port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshaking (greeting)
  - transfer
  - closure
- command/response interaction
  - **commands**: ASCII text
  - **response**: status code and phrase
Sample smtp interaction

S: 220 hamburger.edu
C: HELO crepes.fr
S: 250 Hello crepes.fr, pleased to meet you
C: MAIL FROM: <alice@crepes.fr>
S: 250 alice@crepes.fr... Sender ok
C: RCPT TO: <bob@hamburger.edu>
S: 250 bob@hamburger.edu ... Recipient ok
C: DATA
S: 354 Enter mail, end with "." on a line by itself
C: Do you like ketchup?
C: How about pickles?
C: .
S: 250 Message accepted for delivery
C: QUIT
S: 221 hamburger.edu closing connection

smtp: final words

try smtp interaction for yourself:
- telnet servername 25
- see 220 reply from server
- enter HELO, MAIL FROM, RCPT TO, DATA, QUIT commands above lets you send email without using email client (reader)

Comparison with http
- http: pull
- email: push
- both have ASCII command/response interaction, status codes
- http: multiple objects in file sent in separate connections
- smtp: multiple message parts sent in one connection
Mail message format

smtp: protocol for exchanging email msgs
RFC 822: standard for text message format:
- header lines, e.g.,
  - To:
  - From:
  - Subject: different from smtp commands!
- body
  - the “message”, ASCII characters only
- line containing only `.'

Message format: multimedia extensions

- MIME: multimedia mail extension, RFC 2045, 2056
- additional lines in msg header declare MIME content type

From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Transfer-Encoding: base64
Content-Type: image/jpeg

base64 encoded data ......
...........................
......base64 encoded data

- blank line
- header
- body
**MIME types**

- **Text**
  - example subtypes: `plain`, `html`

- **Image**
  - example subtypes: `jpeg`, `gif`

- **Audio**
  - example subtypes: `basic` (8-bit mu-law encoded), `32kadpcm` *(32 kbps coding)*

- **Video**
  - example subtypes: `mpeg`, `quicktime`

- **Application**
  - other data that must be processed by reader before “viewable”
  - example subtypes: `msword`, `octet-stream`

---

**Mail access protocols**

- SMTP: delivery/storage to receiver’s server
- Mail access protocol: retrieval from server
  - POP: Post Office Protocol [RFC 1939]
    - authorization (agent <-->server) and download
  - IMAP: Internet Mail Access Protocol [RFC 1730]
    - more features (more complex)
    - manipulation of stored msgs on server
POP3 protocol

authorization phase
- client commands:
  - user: declare username
  - pass: password
- server responses
  - +OK
  - -ERR

transaction phase, client:
- list: list message numbers
- retr: retrieve message by number
- dele: delete
- quit

POP3 (more) and IMAP

More about POP3
- Previous example uses “download and delete” mode.
- Bob cannot re-read email if he changes client
- “Download-and-keep”: copies of messages on different clients
- POP3 is stateless across sessions

IMAP
- Keep all messages in one place: the server
- Allows user to organize messages in folders
- IMAP keeps user state across sessions:
  - names of folders and mappings between message IDs and folder name
ftp: the file transfer protocol

- transfer file to/from remote host
- client/server model
  - client: side that initiates transfer (either to/from remote)
  - server: remote host
- ftp: RFC 959
- ftp server: port 21

ftp: separate control, data connections

- ftp client contacts ftp server at port 21, specifying TCP as transport protocol
- two parallel TCP connections opened:
  - control: exchange commands, responses between client, server.
    - "out of band control"
  - data: file data to/from server
- ftp server maintains "state": current directory, earlier authentication
ftp commands, responses

Sample commands:
- sent as ASCII text over control channel
- USER username
- PASS password
- LIST return list of file in current directory
- RETR filename retrieves (gets) file
- STOR filename stores (puts) file onto remote host

Sample return codes
- status code and phrase (as in http)
- 331 Username OK, password required
- 125 data connection already open; transfer starting
- 425 Can’t open data connection
- 452 Error writing file

Application architectures

- Client-server
- Peer-to-peer (P2P)
- Hybrid of client-server and P2P
**Client-server architecture**

**server:**
- always-on host
- permanent IP address
- server farms for scaling

**clients:**
- communicate with server
- may be intermittently connected
- may have dynamic IP addresses
- do not communicate directly with each other

---

**Pure P2P architecture**

- no always-on server
- arbitrary end systems directly communicate
- peers are intermittently connected and change IP addresses
- example: Gnutella

Highly scalable but difficult to manage
Generations of P2P

- 1st Generation: Centralized file list
  - Napster
  - He who controls central file is responsible legally

- 2nd Generation: Decentralized file lists
  - Gnutella, FastTrack
  - Improvements – optimizations of decentralized search

- 3rd Generation: No file lists
  - Freenet, WASTE, Entropy, MUTE
  - Anonymity built in

The Good, Bad, and Ugly of P2P

- The Good
  - Security based on social contract
  - Free exchange of ideas
  - Everyone’s computer can contribute to the greater good

- The Bad
  - Avoids most security: Can be used for piracy
  - “The Tragedy of the Commons” – not supported by all IP
Hybrid of client-server and P2P

Skype
- Internet telephony app
- Finding address of remote party: centralized server(s)
- Client-client connection is direct (not through server)

Instant messaging
- Chatting between two users is P2P
- Presence detection/location centralized:
  - User registers its IP address with central server when it comes online
  - User contacts central server to find IP addresses of buddies

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 centralized
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 are part of the overlay net
- edge is not a physical link
- given peer will typically be connected with < 10 overlay neighbors

Gnutella: protocol

- Query message sent over existing TCP connections
- peers forward Query message
- QueryHit sent over reverse path
- File transfer: HTTP

Scalability: limited scope flooding
Gnutella: Peer joining

1. Joining peer X must find some other peer in Gnutella network: use list of candidate peers
2. X sequentially attempts to make TCP with peers on list until connection setup with Y
3. X sends Ping message to Y; Y forwards Ping message.
4. All peers receiving Ping message respond with Pong message
5. X receives many Pong messages. It can then setup additional TCP connections

Exploiting heterogeneity: KaZaA

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

- Each file has a hash and a descriptor
- Client sends keyword query to its group leader
- Group leader responds with matches:
  - For each match: metadata, hash, IP address
- If group leader forwards query to other group leaders, they respond with matches
- Client then selects files for downloading
  - HTTP requests using hash as identifier sent to peers holding desired file

KaZaA tricks

- Limitations on simultaneous uploads
- Request queuing
- Incentive priorities
- Parallel downloading

For more info:
- J. Liang, R. Kumar, K. Ross, "Understanding KaZaA,"
  (available via cis.poly.edu/~ross)
BitTorrent

- Scalable way of downloading one file
  - Break file into multiple data blocks
  - After downloading a data block, clients help upload it to other clients
- Accounts for one third of all Web traffic
  - Many legitimate uses
  - Software distribution
    - Unix distributions, e.g., Linux
    - Valve (Steam), e.g., Halflife2, Counterstrike1.6
  - Media distribution
    - bittorrent.org (commercial service)

BitTorrent terms

- Client
  - Leecher (Sauger): initial state
  - Seeder: once it has the full dataset (file)
- Tracker
  - Coordinator
- Torrent file
  - Meta information
- Torrent
  - The overlay
BitTorrent – joining a torrent

Peers divided into:
- **seeds**: have the entire file
- **leechers**: still downloading

1. obtain the *metadata file*
2. contact the *tracker*
3. obtain a *peer list* (contains seeds & leechers)
4. contact peers from that list for data

BitTorrent – exchanging data

- Verify *pieces* using hashes
- Download sub-pieces *in parallel*
- *Advertise* received pieces to the entire peer list
- Look for the *rarest* pieces
BitTorrent – philosophy

- Philosophy
  - Based on Tit-for-tat
  - Incentive - Uploading while downloading

- Choking algorithm
  - Controls whom you are sending data
  - Choke – disable sending of data to peer
  - Unchoke – enable sending of data to peer
  - Preferred peers – those that are sending you data
  - Optimistic unchoke – to enable download from peers
  - Snubbing – disconnect bad peers

BitTorrent - unchoking

- Periodically calculate data-receiving rates
- Upload to (unchoke) the fastest downloaders

- Optimistic unchoking
  - Periodically select a peer at random and upload to it
  - Continuously look for the fastest partners
Overall Architecture

Web Server
Tracker
Torrent

Peer [Leech]
Downloader “US”

Peer [Seed]

Overall Architecture

Web Server
Tracker
Get-announce

Peer [Leech]
Downloader “US”

Peer [Seed]
Overall Architecture

Web page with link to .torrent

Response-peer list

Web Server

Tracker

Peer [Leech]

Downloader “US”

Peer [Leech]

Peer [Seed]

Overall Architecture

Web page with link to .torrent

Shake-hand

Web Server

Tracker

Peer [Leech]

Downloader “US”

Peer [Leech]

Peer [Seed]
Chat systems

- Popular
- Highly interactive services
- Allow to exchange text messages interactively with other people around the world
- Types
  - IRC (Internet relay chat)
  - Web-Chat
    - HTML based
    - Applet based
  - Instant messagengers (ICQ, AIM, MIM, ...)
  - Otheres
IRC

- Widely used – relatively old
  - 2003: 5 biggest networks > 250,000 users
- Channels
- User = unique nickname
- Commands: PRIVMSG, JOIN, ISON, NICK
  - User input: /MSG nick2 Hi
  - IRC Format: :nick1 PRIVMSG nick2 :Hi!
  - Output: *nick1* Hi!

Properties of IRC

- IRC operators administer IRC network
- Port 6667
- Well-defined protocol (RFC 1495)
- Client/server to server network
  - Allows direct computer-to-computer connections (DCC)
- Usage
  - User connects to public IRC server
  - Joins channel
  - Chats with others
  - Shares files via DCC
Sample IRC network

Server: A, B, C, D, E
Client: 1, 2, 3, 4

IRC: Dangers

- Means of spreading worms
- Susceptible to flooding
- Can be embedded in trojans and act as hostile server unnoticed
Malicious IRC usages

- Exchange exploit information
  - Passwords, etc...
- Common protocol for communication between attackers
- Use IRC as control channel
  - Use private IRC networks
- Distributed Denial of Service attacks
  - Clone/Flood/War bots simulate multiple users in a channel
  - Bots spread and infect lots of computers that log into the channel
  - Attacker sends a command through IRC causing all bots to simultaneously flood packets to a target, e.g., Yahoo, eTrade, Amazon.com, ...)

Web-Chat

- Widely used – newer
- Simple user interface
- Lots of systems using different protocols
- HTML based
  - Interface: browser
  - Protocol: HTTP
- Applet based
  - Interface: applet window
  - Protocol: customer or IRC
Application-Layer: 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-based, out-of-band
- centralized vs. decentralized
- stateless vs. stateful
- reliable vs. unreliable msg transfer
- “complexity at network edge”