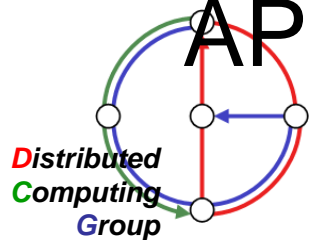


# Chapter 2

# APPLICATIONS



Computer Networks  
Summer 2005

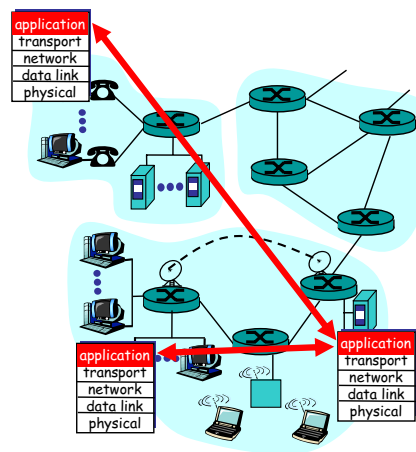
## Overview

- Learn specific application layer protocols
  - http, ftp, smtp, pop, dns, etc.
- How to program network applications?
- Socket API for Java and Eiffel
- Goals
  - learn about protocols by examining popular application-level protocols
  - conceptual and implementation aspects of network application protocols
  - client-server paradigm
  - service models



## Applications vs. Application-Layer Protocols

- Application: communicating, distributed process
  - running in network hosts in “user space”
  - exchange messages to implement application
  - e.g. email, ftp, web
- Application-layer protocol
  - one part of application
  - define messages exchanged by applications and actions taken
  - use communication services provided by transport layer protocols (TCP, UDP)



## Network applications: some jargon

- Process: program running within a host
  - within same host, two processes communicate using interprocess communication (defined by Operating System).
  - processes running on different hosts communicate with an application-layer protocol through messages
- User agent: software process, interfacing with user “above” and network “below”
  - implements application-level protocol
  - Examples
    - Web: browser
    - E-mail: mail reader
    - streaming audio/video: media player



## Client-server paradigm

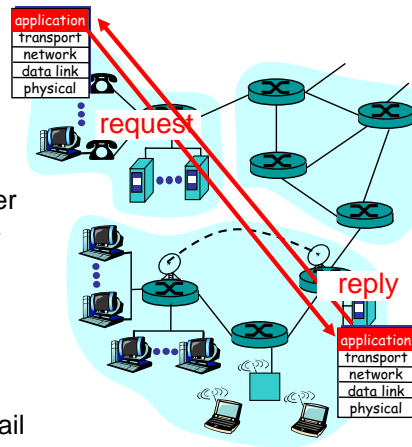
Typical network app has two parts: **Client** and **Server**

### Client

- initiates contact with server (“client speaks first”)
- typically requests service from server
- Web: client implemented in browser
- email: client in mail reader

### Server

- provides requested service to client
- e.g. Web server sends requested Web page, mail server delivers e-mail



## API: Application Programming Interface

- Defines interface between application and transport layers
- socket: Internet API
- two processes communicate by sending data into socket, reading data out of socket
- How does a process identify the other process with which it wants to communicate?
  - IP address of host running other process
  - “port number”: allows receiving host to determine to which local process the message should be delivered
- lots more on this later...



## What transport service does an application need?

### Data loss

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

### 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

### Timing

- some apps (e.g. Internet telephony, interactive games) require low delay to be “effective”



## Transport service requirements of common applications

Application	Data loss	Bandwidth	Time Sensitive
file transfer	no loss	elastic	no
e-mail	no loss	elastic	no
Web documents	loss-tolerant	elastic	no
real-time audio/video	loss-tolerant	audio: 5Kb-1Mb video: 10Kb-5Mb	yes, 100's msec
stored audio/video	loss-tolerant	same as above	yes, few secs
interactive games	loss-tolerant	few Kbps up	yes, 100's msec
financial apps	no loss	elastic	yes and no



## Internet transport protocols services

### 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 bandwidth guarantee
- Why bother? Why is there a UDP service at all?!?



## Internet apps: application, transport protocols

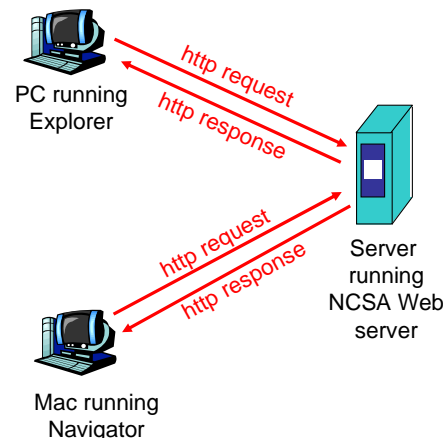
Application	Application layer protocol	Underlying transport protocol
e-mail	smtp [RFC 821]	TCP
remote terminal access	telnet [RFC 854]	TCP
Web	http [RFC 2068]	TCP
file transfer	ftp [RFC 959]	TCP
streaming multimedia	proprietary (e.g. Quicktime)	TCP or UDP
remote file server	NFS	TCP or UDP
Internet telephony	proprietary (e.g. Vocaltec)	typically UDP



## The Web: The http protocol

### http: hypertext transfer protocol

- Web's application layer protocol
- client/server model
  - *client*: browser that requests, receives, and "displays" Web objects
  - *server*: Web server sends objects in response to requests
- http 1.0: RFC 1945
- http 1.1: RFC 2616



## More on the http protocol

- client initiates TCP connection (creates socket) to server, port 80
- server accepts TCP connection from client
- http messages (application-layer protocol messages) exchanged between browser (http client) and Web server (http server)
- TCP connection closed

### http is "stateless"

- server maintains no information about past client requests

**aside**

- Protocols that maintain "state" are complex!
- past history (state) must be maintained
- if server/client crashes, their views of "state" may be inconsistent, must be reconciled



## Example for http

Suppose user enters URL `www.inf.ethz.ch/education/index.html`  
(assume that web page contains text, references to 10 jpeg images)

1. http client initiates TCP connection to http server (process) at `www.inf.ethz.ch`. Port 80 is default for http server.
2. http server at host `www.inf.ethz.ch` waiting for TCP connection at port 80, "accepts" connection, notifies client
3. http client sends http request message (containing URL) into TCP connection socket
4. http server receives request message, forms *response message* containing requested object (`index.html` in directory `education`), sends message into socket

time



## Example for http (continued)

5. http server closes TCP connection
6. http client receives response message containing html file, displays html. Parsing html file, finds 10 referenced jpeg pictures

Then...

Steps 1-6 repeated for each of the 10 jpeg objects

time



## Non-persistent vs. persistent connections

### Non-persistent

- http/1.0
- server parses request, responds, closes TCP connection
- 2 RTTs (round-trip-time) to fetch object
  - TCP connection
  - object request/transfer
- each transfer suffers from TCP's initially slow sending rate
- many browsers open multiple parallel connections

### Persistent

- default for http/1.1
- on same TCP connection: server, parses request, responds, parses new request,...
- client sends requests for all referenced objects as soon as it receives base HTML
- fewer RTTs, less slow start



## http message format: request

- two types of http messages: *request*, *response*
- http request message: ASCII (human-readable format)

request line (GET, POST, HEAD commands)

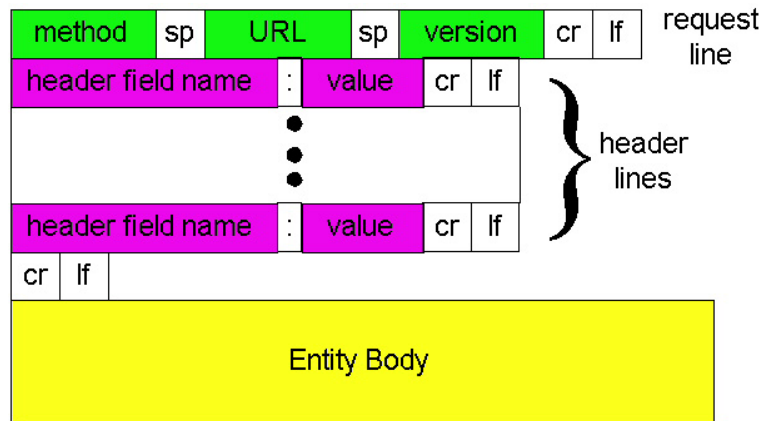
header lines

```
GET /somedir/page.html HTTP/1.1
Host: www.servername.com
User-agent: Mozilla/4.0
Accept-language: de
```

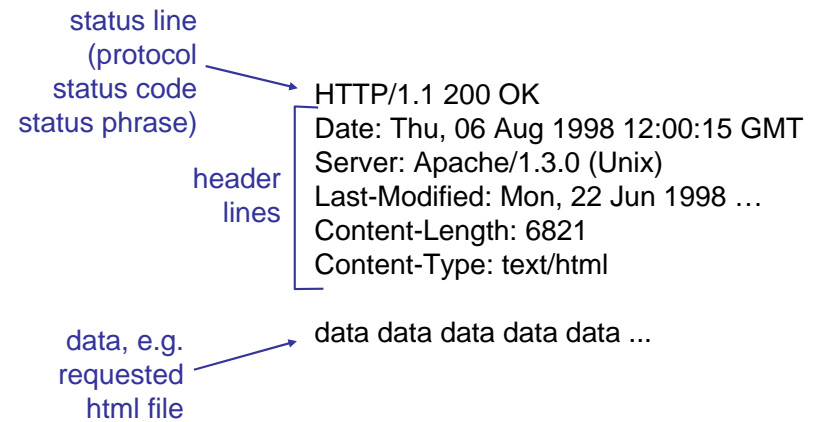
Carriage return and line feed indicate end of message (extra carriage return, line feed)



## http request message: the general format



## http message format: response



## http response status codes

First line of server/client response message.  
A few sample codes:

### 200 OK

- request succeeded, requested object later in this message

### 301 Moved Permanently

- requested object moved, new location specified later in this message (Location:)

### 400 Bad Request

- request message not understood by server

### 404 Not Found

- requested document not found on this server

### 505 HTTP Version Not Supported



## Be your own http client

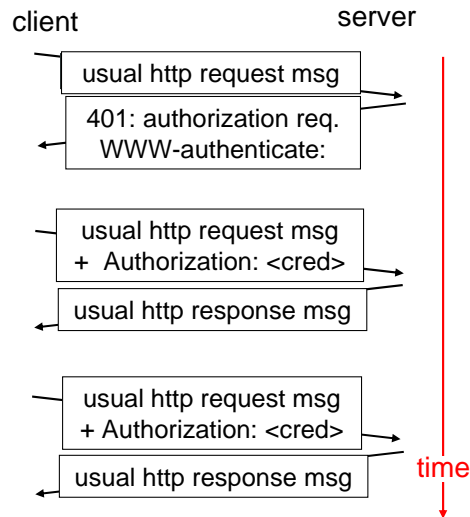
1. Telnet to your favorite Web server:  
`telnet www.sbb.ch 80`
  - Opens TCP connection to port 80 (default http server port) at www.sbb.ch.
2. Type in a GET http request:  
`GET /index.htm HTTP/1.0`
  - Anything typed in sent to port 80 at www.sbb.ch
  - By typing this (hit carriage return twice), you send this minimal (but complete) GET request to http server
3. Check out response message sent by http server...

Could you check the SBB timetable from within your own application?!?



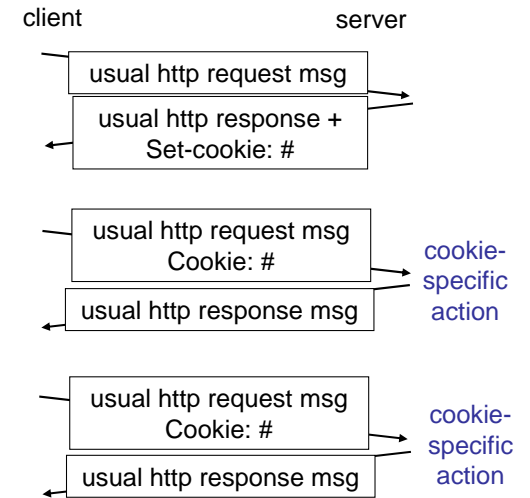
## User-server interaction: authentication

- Authentication: control access to server content
- authorization credentials: typically name and password
- stateless: client must present authorization in *each* request
  - authorization: header line in each request
  - if no authorization: header, server refuses access, sends  
**WWW authenticate:**  
  
header line in response



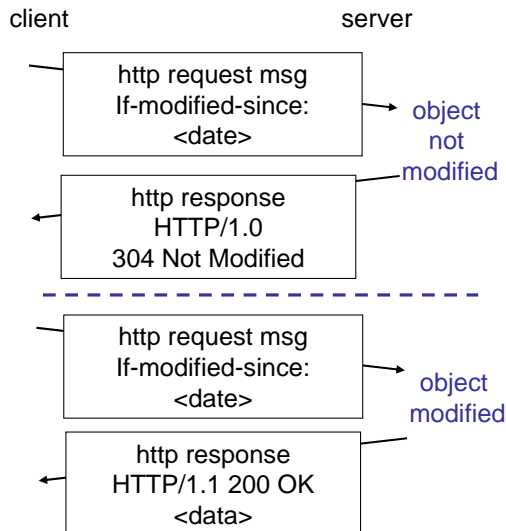
## Cookies: keeping "state"

- server-generated #, server-remembered #, later used for
  - authentication
  - remembering user preferences
  - remembering previous choices
  - (...privacy?)
- server sends "cookie" to client in response msg  
**Set-cookie: 1678453**
- client presents cookie in later requests  
**Cookie: 1678453**



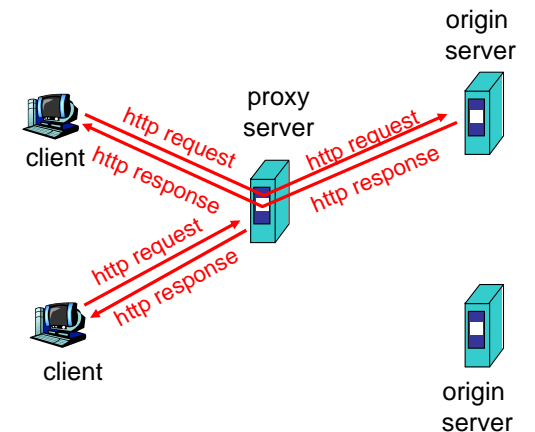
## Conditional GET: client-side caching

- Goal: don't send object if client has up-to-date cached version
- Client: specify date of cached copy in http request  
**If-modified-since: <date>**
- Server: response contains no object if cached copy is up-to-date:  
**HTTP/1.0 304 Not Modified**



## Web Caches (a.k.a. proxy server)

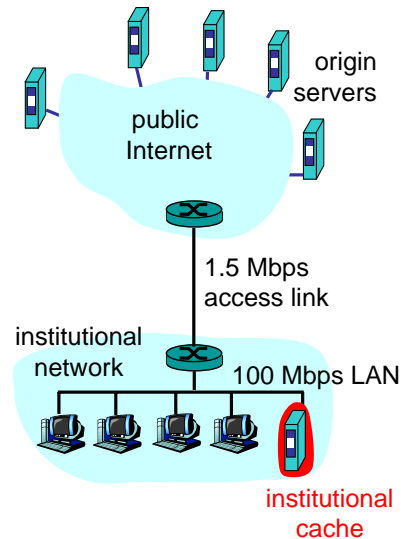
- Goal: satisfy client request without involving origin server
- User sets browser: Web accesses via web cache
- Client sends all http requests to web cache
  - object in web cache: web cache returns object
  - else web cache requests object from origin server, then returns object to client





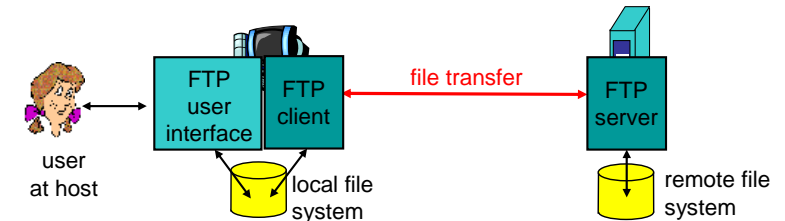
## Why Web Caching?

- Assumption: cache is “close” to client (e.g. in same network)
- Smaller response time: cache “closer” to client
- Decrease traffic to distant servers
- Link out of institutional/local ISP network is often a bottleneck



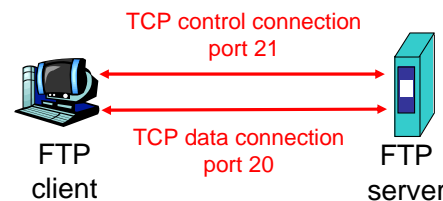
## ftp: The file transfer protocol

- 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 and 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 and responses

### Sample commands

- sent as ASCII text over control channel
- **USER *username***
- **PASS *password***
- **LIST** returns list of files 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**



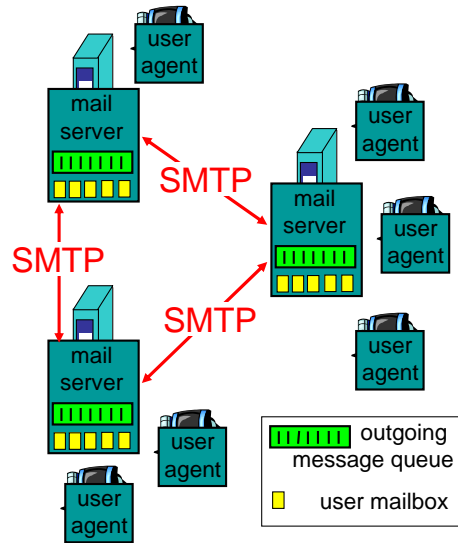
## Electronic Mail

### Three major components

- user agents
- mail servers
- simple mail transfer protocol: smtp

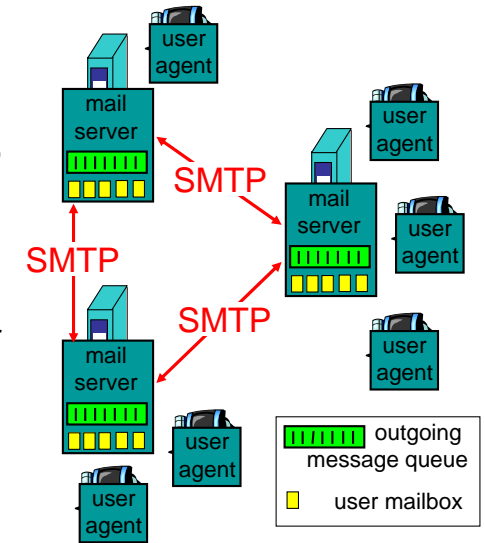
### User Agent

- a.k.a. "mail reader"
- composing, editing, reading mail messages
- Examples: Outlook, Netscape Messenger, elm, Eudora
- outgoing, incoming messages stored on server



## Electronic Mail: 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 servers to send email messages
  - "client": sending mail server
  - "server": receiving mail server



- Why not sending directly?



## Electronic Mail: SMTP

- uses TCP to reliably transfer email message from client to server, on port 25
- direct transfer: sending server to receiving server
- three phases of transfer
  - handshake (greeting)
  - transfer of messages
  - closure
- command/response interaction
  - commands: ASCII text
  - response: status code and phrase
- SMTP: RFC 821

## 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
```

You can be your own smtp client: telnet to a mail server you know (telnet mail.inf.ethz.ch 25) and play with the protocol...





## smtp: more details

- smtp uses persistent connections
- smtp requires message (header & body) to be in 7-bit ASCII
- certain character strings not permitted in msg (e.g., **CRLF.CRLF**, which is used to determine the end of a message by the server).
- Thus msg has to be encoded (usually into either base-64 or quoted printable)

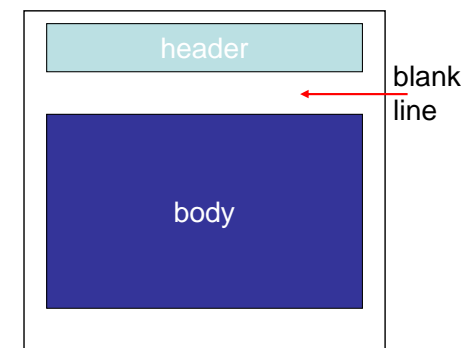
### Comparison with http

- http: pull
- email: push
- both have ASCII command/response interaction and status codes
- http: each object encapsulated in its own response msg (1.0), or by use of content-length field (1.1)
- smtp: multiple objects sent in multipart msg (as we will see on the next slides)



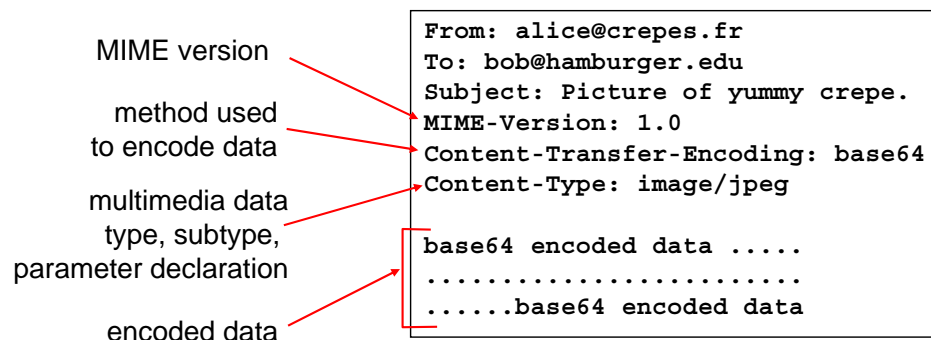
## Mail message format

- smtp: protocol for exchanging email msgs
- RFC 822: standard for text message format:
- header lines, e.g.
  - To:
  - From:
  - Subject:
 (!) Caution: these are not smtp commands! They are like the header of a letter, whereas smtp commands are like the address on the envelope
- body
  - the “message”
  - ASCII characters only



## Message format: multimedia extensions

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



## MIME types

### Content-Type: type/subtype; parameters

#### Text

- example subtypes: **plain**, **enriched**, **html**

#### Video

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

#### Image

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

#### Application

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

#### Audio

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



## MIME Multipart Type

```
From: alice@crepes.fr
To: bob@hamburger.edu
Subject: Picture of yummy crepe.
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=98766789
```

```
--98766789
Content-Transfer-Encoding: quoted-printable
Content-Type: text/plain
```

```
Dear Bob,
Please find a picture of a crepe.
```

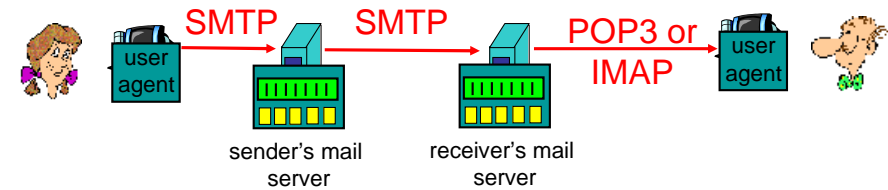
```
--98766789
Content-Transfer-Encoding: base64
Content-Type: image/jpeg
```

```
base64 encoded data .....
.....base64 encoded data
--98766789--
```



## 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 2060]
    - more features (more complex)
    - manipulation of stored messages on server
  - HTTP: Hotmail, Yahoo! Mail, etc.



## POP3 protocol

### Authorization phase

- client commands:
  - user: declare username
  - pass: password

### server responses

- +OK
- -ERR

### Transaction phase

- client commands
  - list: list message numbers
  - retr: retrieve message by number
  - dele: delete
  - quit

```
S: +OK POP3 server ready
C: user alice
S: +OK
C: pass hungry
S: +OK user successfully logged on

C: list
S: 1 498
S: 2 912
S: .
C: retr 1
S: <message 1 contents>
S: .
C: dele 1
C: retr 2
S: <message 1 contents>
S: .
C: dele 2
C: quit
S: +OK POP3 server signing off
```



## DNS: Domain Name System

### People have many identifiers

- passport number, AHV number, student number, name, etc.

### Internet hosts, routers

- IP address (129.132.130.152); used for addressing datagrams
- Name (photek.ethz.ch); used by humans
- We need a map from names to IP addresses (and vice versa?)

### Domain Name System

- *distributed database* implemented in hierarchy of many *name servers*
- *application-layer protocol* host, routers, name servers to communicate to *resolve* names (name/address translation)
  - note: is a core Internet function, but only implemented as application-layer protocol
  - complexity at network's "edge"



## DNS name servers

### Why not centralize DNS?

- single point of failure
- traffic volume
- distant centralized database
- maintenance

...it does not *scale!*

- no server has all name-to-IP address mappings

### local name servers

- each ISP, company has *local (default) name server*
- host DNS query first goes to local name server

### authoritative name server

- for a host: stores that host's IP address, name
- can perform name/address translation for that host's name



## DNS: Root name servers

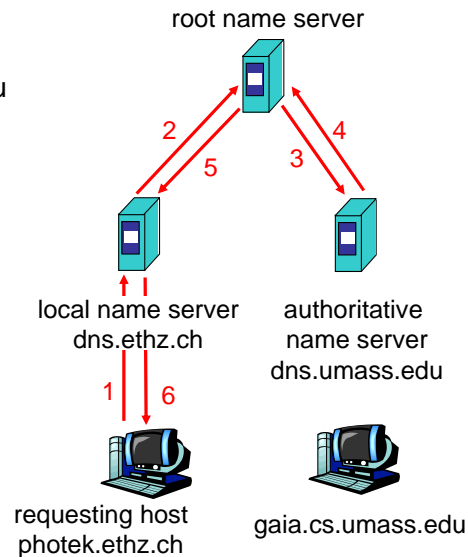
- contacted by local name server that cannot resolve name
- root name server
  - contacts authoritative name server if name mapping not known
  - gets mapping
  - returns mapping to local name server
  - currently 13 root name servers worldwide



## Simple DNS example

- host photek.ethz.ch wants IP address of gaia.cs.umass.edu

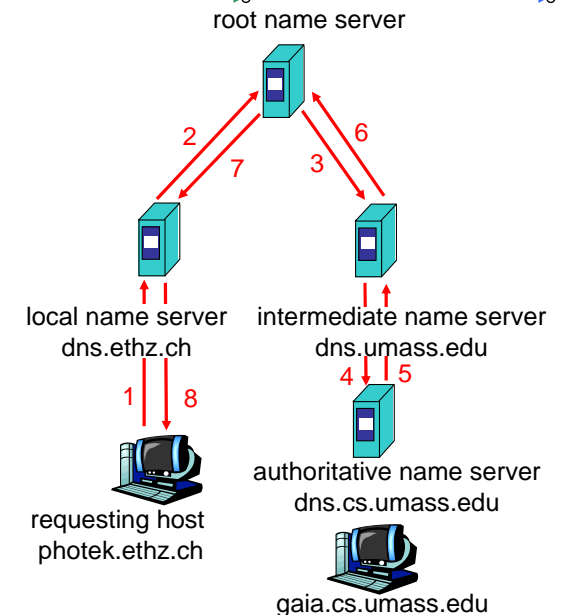
1. contact local DNS server, dns.ethz.ch
2. dns.ethz.ch contacts root name server, if necessary
3. root name server contacts authoritative name server, dns.umass.edu, if necessary



## DNS extended example

### Root name server:

- may not know authoritative name server
- may know *intermediate name server*: who to contact to find authoritative name server



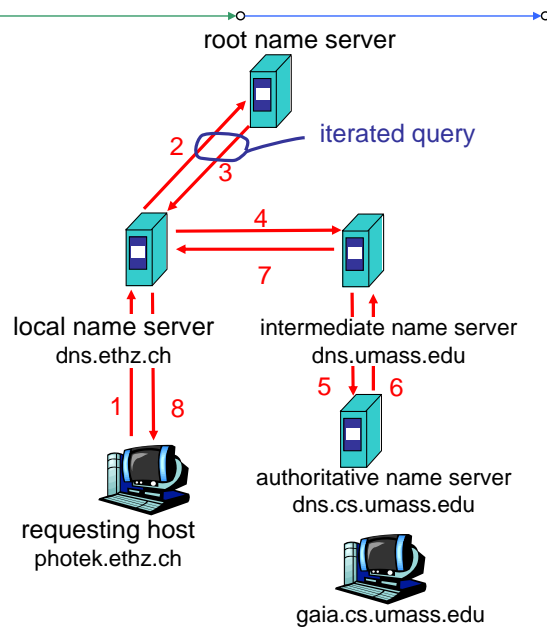
## DNS Iterated queries

### Recursive query

- puts burden of name resolution on contacted name server
- heavy load?

### Iterated query

- contacted server replies with name of server to contact
- "I don't know this name, but ask this server"



## DNS: Caching and updating records

- once (any) name server learns mapping, it *caches* mapping
  - cache entries timeout (disappear) after some time
- update/notify mechanisms under design by IETF
  - RFC 2136
  - <http://www.ietf.org/html.charters/dnsind-charter.html>



## DNS resource records

DNS: distributed database storing resource records (RR)

**RR format: (name, ttl, class, type, value)**

- Type=A
  - name is hostname
  - value is IP address
- Type=NS
  - name is domain (e.g. foo.com)
  - value is IP address of authoritative name server for this domain
- Type=CNAME
  - name is alias name for some "canonical" (the real) name
  - value is canonical name

www.ibm.com is really servereast.backup2.ibm.com
- Type=MX
  - value is name of mail server associated with name



## Example of DNS lookup

```
host -v dcg.ethz.ch
Trying "dcg.ethz.ch"
;; ->HEADER<<- opcode: QUERY, status: NOERROR, id: 27554
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 3,
    ADDITIONAL: 3

;; QUESTION SECTION:
;dcg.ethz.ch.                IN      ANY

;; ANSWER SECTION:
dcg.ethz.ch.                86400  IN      CNAME   dcg.inf.ethz.ch.

;; AUTHORITY SECTION:
ethz.ch.                    3600000 IN      NS      dns1.ethz.ch.
ethz.ch.                    3600000 IN      NS      dns2.ethz.ch.
ethz.ch.                    3600000 IN      NS      dns3.ethz.ch.

;; ADDITIONAL SECTION:
dns1.ethz.ch.               86400  IN      A       129.132.98.12
dns2.ethz.ch.               86400  IN      A       129.132.250.220
dns3.ethz.ch.               86400  IN      A       129.132.250.2
```



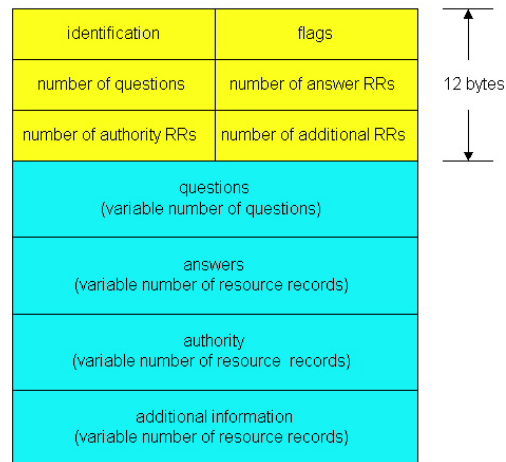
## DNS protocol, messages

### DNS protocol

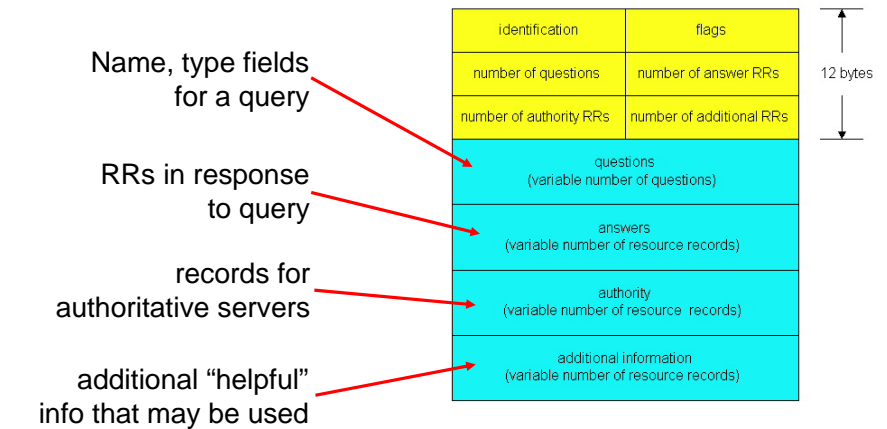
- *query* and *reply* messages, both with same *message format*

### msg header

- identification: 16 bit number for query, reply to query uses same number
- flags:
  - query or reply
  - recursion desired
  - recursion available
  - reply is authoritative



## DNS protocol, messages



## Socket programming

### Goal

- Learn building client/server applications that communicate using sockets, the standard application programming interface

### Socket API

- introduced in BSD4.1 UNIX, 1981
- explicitly created, used, released by applications
- 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



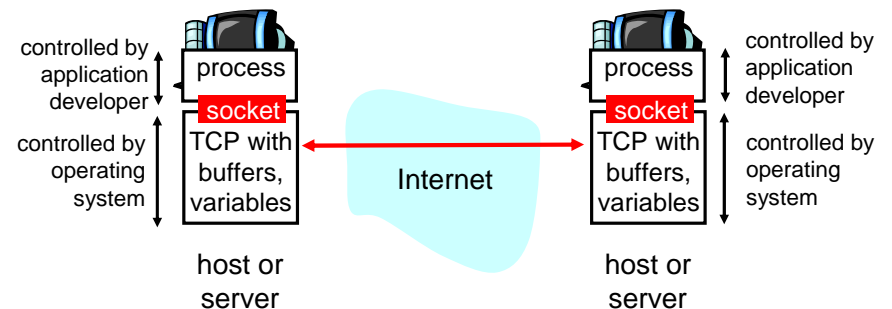
## Socket programming with TCP

### 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 already
- server must have created socket ("door") that welcomes client's contact

Client contacts server by

- creating client-local TCP socket
- specifying IP address and 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

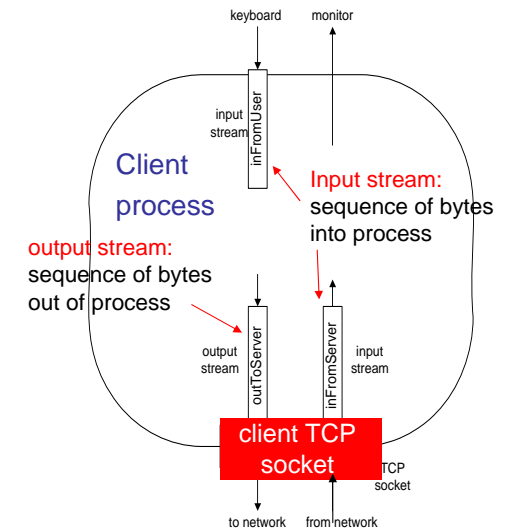
*TCP provides reliable, in-order transfer of bytes ("pipe") between client and server*



## Socket programming with TCP (Java)

Example client-server application

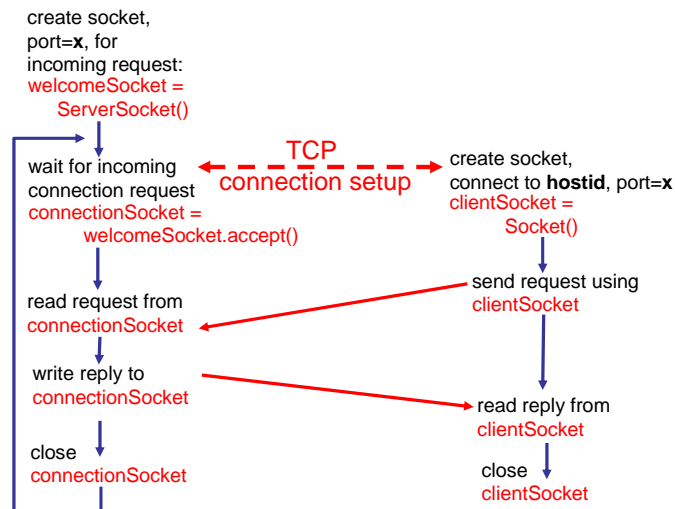
- client reads line from standard input (`inFromUser` stream), sends to server via socket (`outToServer` stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads and prints modified line from socket (`inFromServer` stream)



## Client/server socket interaction with TCP (Java)

Server (running on `hostid`)

Client



## Example: Java client (TCP)

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

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

        Create input stream → BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));


        Create client socket, connect to server → Socket clientSocket = new Socket("hostname", 6789);

        Create output stream attached to socket → DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
    }
}
```





## Example: Java client (TCP), continued



```

Create input stream attached to socket →
BufferedReader inFromServer =
    new BufferedReader(new
        InputStreamReader(clientSocket.getInputStream()));


Send line to server →
sentence = inFromUser.readLine();
outToServer.writeBytes(sentence + '\n');

Read line from server →
modifiedSentence = inFromServer.readLine();
System.out.println("FROM SERVER: " + modifiedSentence);

clientSocket.close();
    }
}
    
```



## Example: Java server (TCP)



```

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

class TCPServer {

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


        Create welcoming socket at port 6789 →
        ServerSocket welcomeSocket = new ServerSocket(6789);

        Wait on welcoming socket for contact by client →
        while(true) {
            Socket connectionSocket = welcomeSocket.accept();

            Create input stream, attached to socket →
            BufferedReader inFromClient =
                new BufferedReader(new
                    InputStreamReader(connectionSocket.getInputStream()));
        }
    }
}
    
```



## Example: Java server (TCP), continued



```

Create output stream, attached to socket →
DataOutputStream outToClient =
    new DataOutputStream(connectionSocket.getOutputStream());

Read in line from socket →
clientSentence = inFromClient.readLine();

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

Write out line to socket →
outToClient.writeBytes(capitalizedSentence);
    }
}

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



## Problem: One client can block other clients

Problem can be solved with threads:

```

ServerSocket welcomeSocket = new ServerSocket(6789);
while(true) {
    Socket connectionSocket = welcomeSocket.accept();
    ServerThread thread = new ServerThread(connectionSocket);
    thread.start();
}

public class ServerThread extends Thread {
    /* Handles connection socket */
    /* "More or less" code of old server loop */
}
    
```

Alternative solution: Client opens socket *after* reading input line



## Socket programming with UDP

Remember: 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

application viewpoint

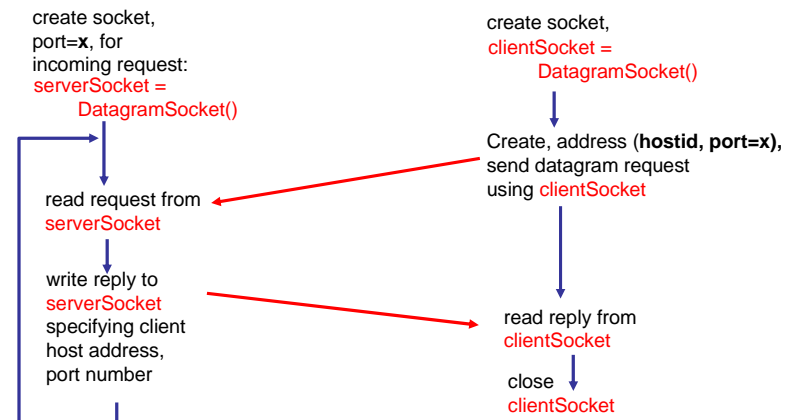
*UDP provides unreliable transfer of groups of bytes (“datagrams”) between client and server*



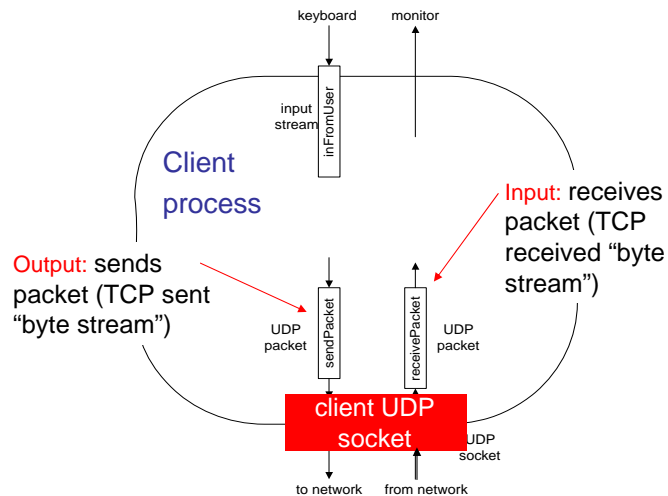
## Client/server socket interaction: UDP (Java)

Server (running on `hostid`)

Client



## Example: Java client (UDP)



## Example: Java client (UDP)

```

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

class UDPClient {
    public static void main(String args[]) throws Exception
    {
        Create input stream → BufferedReader inFromUser =
            new BufferedReader(new InputStreamReader(System.in));
        Create client socket → DatagramSocket clientSocket = new DatagramSocket();
        Translate hostname to IP Address using DNS → InetAddress IPAddress = InetAddress.getByName("hostname");

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

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



## Example: Java client (UDP), continued

```

Create datagram with data-to-send, length, IP addr, port → DatagramPacket sendPacket =
new DatagramPacket(sendData, sendData.length, IPAddress, 9876);

Send datagram to server → clientSocket.send(sendPacket);

Read datagram from server → DatagramPacket receivePacket =
new DatagramPacket(receiveData, receiveData.length);
clientSocket.receive(receivePacket);

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

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



## Example: Java server (UDP)

```

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

class UDPServer {
public static void main(String args[]) throws Exception
{
Create datagram socket at port 9876 → DatagramSocket serverSocket = new DatagramSocket(9876);

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

while(true)
{
Create space for received datagram → DatagramPacket receivePacket =
new DatagramPacket(receiveData, receiveData.length);

Receive datagram → serverSocket.receive(receivePacket);
}
}
    
```



## Example: Java server (UDP), continued

```

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

Get IP addr, port #, of sender → InetAddress IPAddress = receivePacket.getAddress();
int port = receivePacket.getPort();

String capitalizedSentence = sentence.toUpperCase();

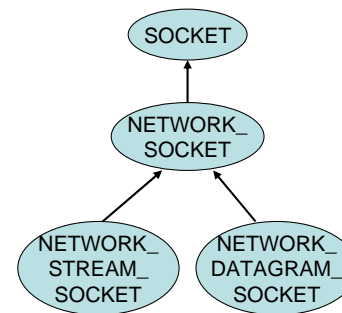
sendData = capitalizedSentence.getBytes();

Create datagram to send to client → DatagramPacket sendPacket =
new DatagramPacket(sendData, sendData.length, IPAddress,
port);

Write out datagram to socket → serverSocket.send(sendPacket);
}
}
End of while loop, loop back and wait for another datagram
    
```



## EiffelNet: Sockets and communication modes



- **Two modes of socket communication:**
  - **stream communication**
  - **datagram communication**
- **Stream socket:**
  - provided by the **STREAM\_** classes
  - provides sequenced communication without any loss or duplication of data
  - **synchronous**: the sending system waits until it has established a connection to the receiving system and transmitted the data
- **Datagram socket:**
  - provided by the **DATAGRAM\_** classes
  - **asynchronous**: the sending system emits its data and does not wait for an acknowledgment
  - efficient, but it does not guarantee sequencing, reliability or non-duplication



## Example: Eiffel Server (TCP - stream socket)

```

class OUR_SERVER
inherit
    SOCKET_RESOURCES
    STORABLE
create
    make
feature
    soc1, soc2: NETWORK_STREAM_SOCKET
    make [argv: ARRAY [STRING]] is
        local
            count: INTEGER
        do
            if argv.count /= 2 then
                io.error.putstring ("Usage: ")
                io.error.putstring [argv.item [0]]
                io.error.putstring ("portnumber")
            else
                create soc1.make_server_by_port [argv.item [1].to_integer]
                from
                    soc1.listen [5]
                    count := 0
                until
                    count := 5
                loop
                    process
                        count := count + 1
                    end
                end
                soc1.cleanup
            end
        rescue soc1.cleanup
        end
end

```

**CLIENT:**

- 1) Sends to the server a list of strings
- 5) Receives the result from the server and print it

**SERVER:**

- 2) Receives the corresponding object structure
- 3) Appends to it another string
- 4) Returns the result to the client

Accepts communication with the client and exchange messages

Create server socket on 'portnumber'

Listen on socket for at most '5' connections

- Accepts communication with the client
- Receives a message from the client
- Extends the message
- Sends the message back to the client

Closes the open socket and frees the corresponding resources

## Example: Eiffel Server (TCP - stream socket), continued

```

process is
local
    our_new_list: OUR_MESSAGE
do
    soc1.accept
    soc2 := soc1.accepted
    our_new_list := retrieved (soc2)
from
    our_new_list.start
until
    our_new_list.after
loop
    io.putstring [our_new_list.item]
    our_new_list.forth
    io.new_line
end
our_new_list.extend ("Server message. %N")
our_new_list.general_store (soc2)
soc2.close
end
end

```

Receives a message from the client, extend it, and send it back.

- the server obtains access to the server
- *accept* - ensures synchronization to with the client
- *accept* - creates a new socket which is accessible through the attribute *accepted*
- the *accepted* value is assigned to *soc2* - this makes *soc1* available to accept connections with other clients

The message exchanged between server and client is a linked list of strings

```

class OUR_MESSAGE
inherit
    LINKED_LIST [STRING]
    STORABLE
undefine
    is_equal, copy
end
create
    make
end

```

Extends the message received from the client

Sends the extended message back to the client

Closes the socket

## Example: Eiffel Client (TCP - stream socket)

```

class OUR_CLIENT
inherit
    NETWORK_CLIENT
    redefine
        received
    end
create
    make_client
feature
    our_list: OUR_MESSAGE
    received: OUR_MESSAGE
    make_client [argv: ARRAY [STRING]] is
        -- Build list, send it, receive modified list, and print it.
        do
            if argv.count /= 3 then
                io.error.putstring ("Usage: ")
                io.error.putstring [argv.item [0]]
                io.error.putstring ("hostname portnumber")
            else
                make [argv.item [2].to_integer, argv.item [1]]
                build_list
                send [our_list]
                receive
                process_received
                cleanup
            end
        end
    rescue
        cleanup
    end
end

```

1. Creates a socket and setup the communication

2. Builds the list of strings

3. Sends the list of strings to the server

The message exchanged between server and client

4. Receives the message from the server

5. Prints the content of the received message

6. Closes the open socket and free the corresponding resources

## Example: Eiffel Client (TCP - stream socket), continued

```

build_list is
do
    create our_list.make
    our_list.extend ("This ")
    our_list.extend ("is ")
    our_list.extend ("a ")
    our_list.extend ("test.")
end

process_received is
do
    if received = Void then
        io.putstring ("No list received.")
    else
        from received.start until received.after loop
            io.putstring (received.item)
            received.forth
        end
    end
end
end

```

Builds the list of strings 'our\_list' for transmission to the server

Prints the content of the received message in sequence

## Example: Eiffel Server (UDP - datagram socket)

```

class OUR_DATAGRAM_SERVER
create
  make
feature
  make (argv: ARRAY [STRING]) is
    local
      soc: NETWORK_DATAGRAM_SOCKET
      ps: MEDIUM_POLLER
      readcomm: DATAGRAM_READER
      writecomm: SERVER_DATAGRAM_WRITER
    do
      if argv.count /= 2 then
        io.error.putstring ("Usage: ")
        io.error.putstring (argv.item (0))
        io.error.putstring (" portnumber")
      else
        create soc.make_bound (argv.item (1).to_integer)
        create ps.make

        create readcomm.make (soc)
        ps.put_read_command (readcomm)
        create writecomm.make (soc)
        ps.put_write_command (writecomm)
        ...
      end
    end
end
  
```

1. Creates read and write commands  
2. Attach them to a poller  
3. Set up the poller for execution

Creates a network datagram socket bound to a local address with a specific port

Creates poller with multi-event polling

1. Creates a read command which it attaches to the socket  
2. Enters the read command into the poller  
3. Creates a write command which it attaches to the socket  
4. Enters the write command into the poller



## Example: Eiffel Server (UDP - datagram socket), continued

```

...
  ps.make_read_only
  ps.execute (15, 20000)
  ps.make_write_only
  ps.execute (15, 20000)
  soc.close
end
rescue
  if not soc.is_closed then
    soc.close
  end
end
end
  
```

1. Sets up the poller to accept read commands only and then executes the poller -- enable the server to get the read event triggered by the client's write command

2. Reverses the poller's set up to write-only, and then executes the poller

Monitors the sockets for the corresponding events and executes the command associated with each event that will be received



## Example: Eiffel Client (UDP - datagram socket)

```

class OUR_DATAGRAM_CLIENT
create
  make
feature
  make (argv: ARRAY [STRING]) is
    local
      soc: NETWORK_DATAGRAM_SOCKET
      ps: MEDIUM_POLLER
      readcomm: DATAGRAM_READER
      writecomm: CLIENT_DATAGRAM_WRITER
    do
      if argv.count /= 3 then
        io.error.putstring ("Usage: ")
        io.error.putstring (argv.item (0))
        io.error.putstring ("hostname portnumber")
      else
        create soc.make_targeted_to_hostname (argv.item (1), argv.item (2).to_integer)
        create ps.make

        create readcomm.make (soc)
        ps.put_read_command (readcomm)
        create writecomm.make (soc)
        ps.put_write_command (writecomm)
        ...
      end
    end
end
  
```

1. Create read and write commands  
2. Attach them to a poller  
3. Set up the poller for execution

Command executed in case of a read event

Command executed by the client when the socket "is ready for writing"

Create a datagram socket connected to 'hostname' and 'port'

Creates poller with multi-event polling

1. Creates a read command which it attaches to the socket  
2. Enters the read command into the poller  
3. Creates a write command which it attaches to the socket  
4. Enters the write command into the poller



## Example: Eiffel Client (UDP - datagram socket), continued

```

...
  ps.make_write_only
  ps.execute (15, 20000)
  ps.make_read_only
  ps.execute (15, 20000)
  soc.close
end
rescue
  if not soc.is_closed then
    soc.close
  end
end
end
  
```

1. Sets up the poller to write commands only and then executes the poller

2. Reverses the poller's set up to accept read commands only, and then executes the poller -- enables the client to get the read event triggered by the server's write command

Monitors the sockets for the corresponding events and executes the command associated with each event that will be received



## Example: Eiffel Command class (UDP - datagram socket)

```

class OUR_DATAGRAM_READER
inherit
  POLL_COMMAND
  redefine
    active_medium
  end
create
  make
feature
  active_medium: NETWORK_DATAGRAM_SOCKET
  execute [arg: ANY] is
    local
      rec_pack: DATAGRAM_PACKET
      i: INTEGER
    do
      rec_pack := active_medium.received (10, 0)
      io.putint (rec_pack.packet_number)
      from i := 0 until i > 9 loop
        io.putchar (rec_pack.element (i))
        i := i + 1
      end
    end
end
  
```

### Commands and events:

- Each system specify certain communication events that it wants to monitor, and certain commands to be executed on occurrence of the specified events
- The commands are objects, instances of the class `POLL_COMMAND`
- The class `POLL_COMMAND` has the procedure `execute` which executes the current command

### Command classes:

- `OUR_DATAGRAM_READER` - represents operations that must be triggered in the case of a read event
- `CLIENT_DATAGRAM_WRITER` - command executed by the client when the socket "is ready for writing"
- `SERVER_DATAGRAM_WRITER` - command executed by the server when the socket "is ready for writing"

Receive a packet of size 10 characters

Prints the packet number of the packet

Prints all the characters from the packet



## Example: Eiffel Command class (UDP - datagram socket), cont

```

class CLIENT_DATAGRAM_WRITER
inherit
  POLL_COMMAND
  redefine
    active_medium
  end
create
  make
feature
  active_medium: NETWORK_DATAGRAM_SOCKET
  execute [arg: ANY] is
    local
      sen_pack: DATAGRAM_PACKET
      char: CHARACTER
    do
      -- Make packet with 10 characters 'a' to 'j'
      -- in successive positions
      create sen_pack.make (10)
      from char := 'a' until char > 'j' loop
        sen_pack.put_element (char |-'a')
        char := char.next
      end
      sen_pack.set_packet_number (1)
      active_medium.send (sen_pack, 0)
    end
end
  
```

Command executed by the client when the socket "is ready for writing"

```

class SERVER_DATAGRAM_WRITER
inherit
  POLL_COMMAND
  redefine
    active_medium
  end
create
  make
feature
  active_medium: NETWORK_DATAGRAM_SOCKET
  execute [arg: ANY] is
    local
      sen_pack: DATAGRAM_PACKET
      i: INTEGER
    do
      -- Make packet with 10 characters 'a' in
      -- successive positions
      create sen_pack.make (10)
      from i := 0 until i > 9 loop
        sen_pack.put_element ('a', i)
        i := i + 1
      end
      sen_pack.set_packet_number (2)
      active_medium.send (sen_pack, 0)
    end
end
  
```

Command executed by the server when the socket "is ready for writing"

