ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Chapter 2a
SOCKET PROGRAMMING

Computer Networks
Timothy Roscoe
Summer 2007

Overview

- Basic socket concepts
- Java socket programming
  - Client & server
  - TCP & UDP
  - Threads
- C socket programming
  - API details
  - TCP client and server
  - Asynchronous I/O and events
- Bonus: EiffelNet API slides

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

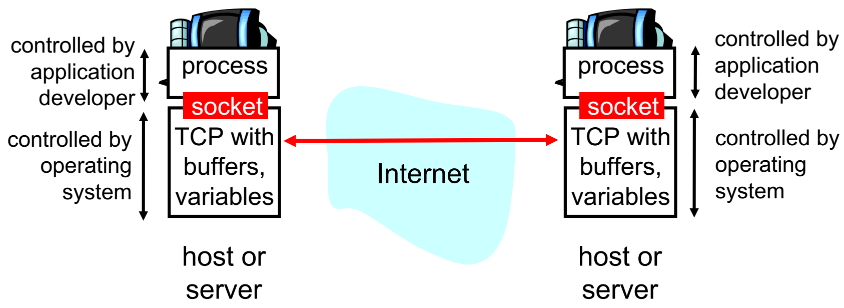
Socket programming with TCP

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 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

application viewpoint

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

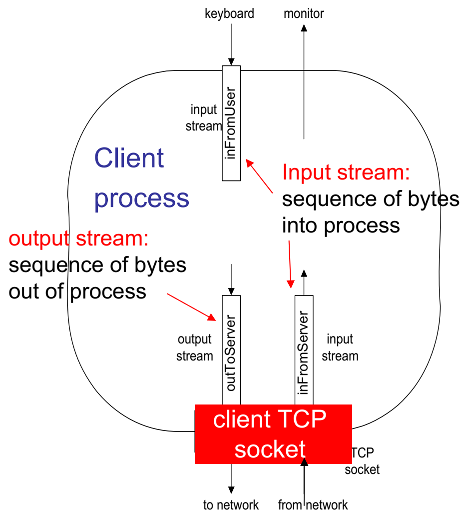
- UDP: transmitted data may be received out of order, or lost

Java API vs. C API

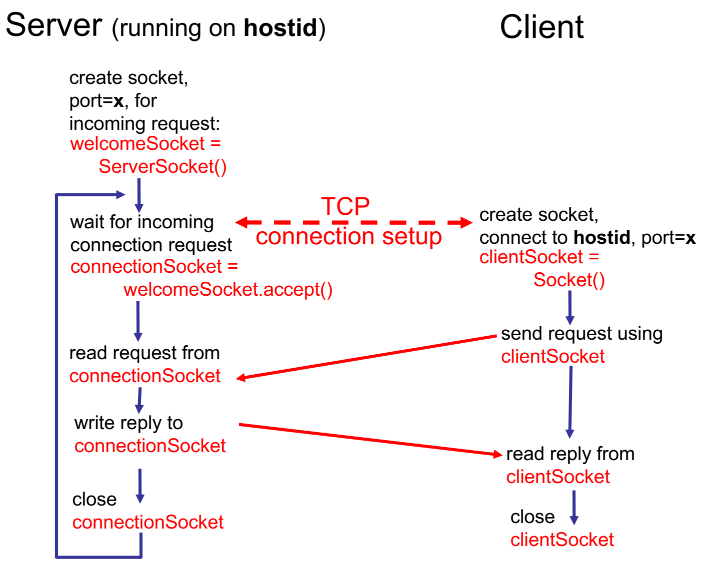
- Java:
  – High-level, easy to use for common situations
  – Buffered I/O
  – Failure abstracted as exceptions
  – Less code to write
- C:
  – Low-level ⇒ more code, more flexibility
  – Original interface
  – Maximum control
  – Basis for all other APIs in Unix (and Windows)

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)



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()));

sentence = inFromUser.readLine();
```

Send line to server

```
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);

    while(true) {
```

Wait on welcoming socket for contact by client

```
      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

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

Read in line from socket

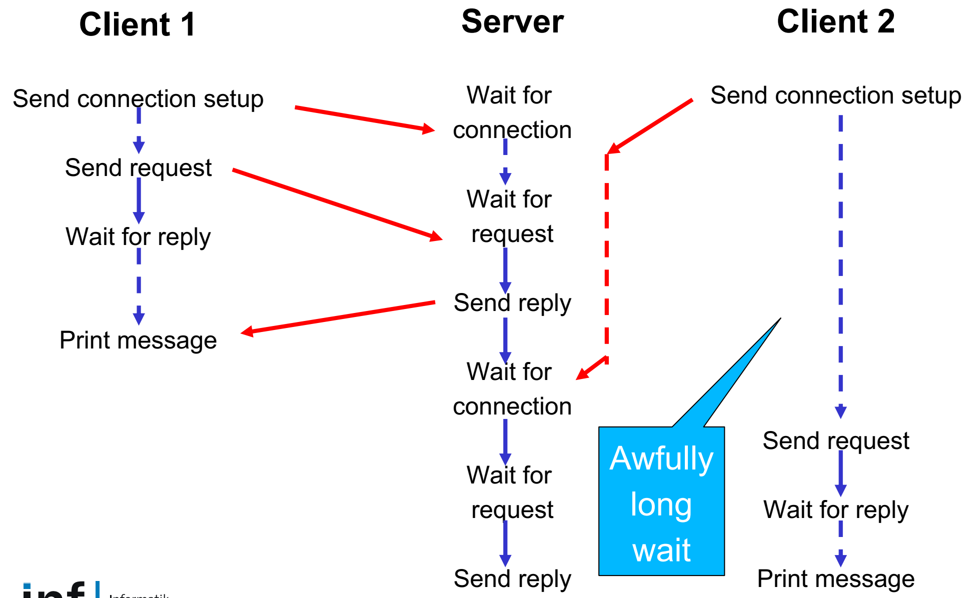
```java
clientSentence = inFromClient.readLine();

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

Write out line to socket

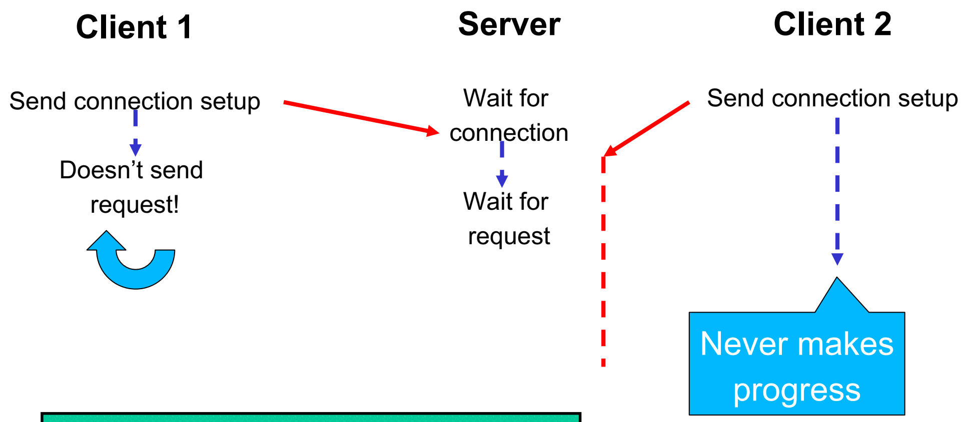
```java
outToClient.writeBytes(capitalizedSentence);
      }
    }
  }
```

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

Problem: One client can delay other clients



In fact, one client can block other clients



More generally, only one machine (client or server) can run at a once!

The Problem: Concurrency

- Networking applications are
  - Inherently *concurrent*
  - Prone to *partial failure*
- Hence, "blocking" (waiting for something else) can
  - Slow things down (only one machine running at a time)
  - REALLY slow things down (mostly, no machines running at a time)
  - Stop things (something stops and everything else waits)
- Central problem of *distributed systems*
  - Not really networking, but probably should be

One solution: Threads

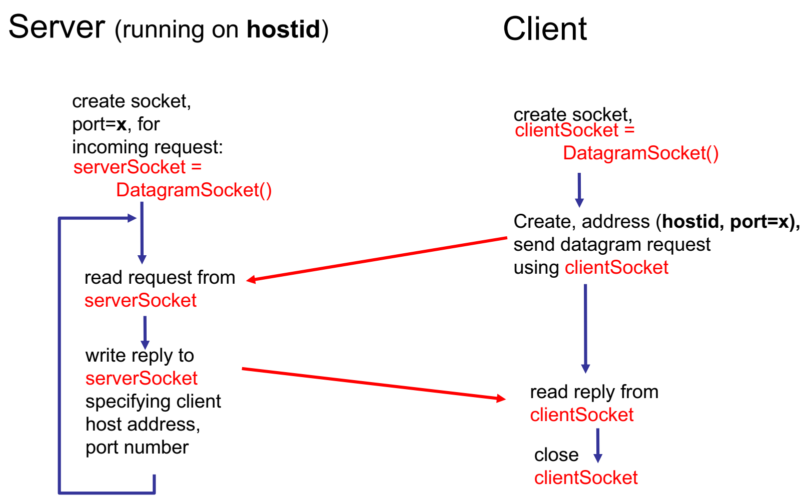
```
ServerSocket welcomeSocket = new ServerSocket(6789);
while(true) {
        Socket connectionSocket = welcomeSocket.accept();
        ServerThread thread = new ServerThread(connectionSocket);
        thread.start();
}
public class ServerThread extends Thread {
        /* … */
        BufferedReader inFromClient = new BufferedReader(new
            InputStreamReader(connectionSocket.getInputStream()));
        DataOutputStream outToClient = new DataOutputStream(
            connectionSocket.getOutputStream());
        clientSentence = inFromClient.readLine();
        capitalizedSentence = clientSentence.toUpperCase() + '\n';
        outToClient.writeBytes(capitalizedSentence);
        /* … */
    }
}
```

Does this solve the problem?

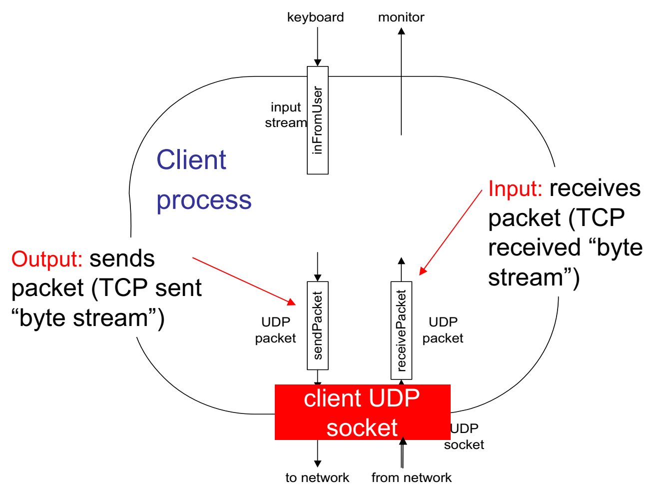
Threads

- Threads are programming abstractions of separate activities
- Still need to worry about resources:
  - How many threads?
  - How long should each thread live for?
- Many programming patterns:
  - Thread-per-request
  - Worker pools
  - Etc.
- See distributed systems course for more on these

Client/server socket interaction: UDP (Java)



Example: Java client (UDP)



Example: Java client (UDP)

Create input stream

Create client socket

Translate hostname to IP Address using DNS

```
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();
```


Example: Java client (UDP), continued

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

Send datagram to server

Read datagram from server

```
      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();
      }
}
```


Example: Java server (UDP)

Create datagram socket at port 9876

Create space for received datagram

Receive datagram

```
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), continued

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

```
      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);
      }
    }
}
```

TCP Client in C step by step…

- Create a socket
- Bind the socket
- Resolve the host name
- Connect the socket
- Write some data
- Read some data
- Close
- Exit

General flavour: much lower level…

C API: `socket()`

```
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
                                  ...
          int s = socket( AF_INET, SOCK_STREAM, 0);
```

socket descriptor: small integer (as with file descriptors)

Address family or domain: In this case IPv4.

Service type requested, e.g. `SOCK_STREAM` or `SOCK_DGRAM`.

Protocol within a service type; 0 ⇒ OS chooses: `IPPROTO_TCP` (often only one!)

C API: Specifying local address

```
int bind(int s, const struct sockaddr *a, socklen_t len);

  struct in_addr {
      u_int32_t s_addr;
  };

  struct sockaddr_in {
      short sin_family;
      u_short sin_port;
      struct in_addr sin_addr;
      char sin_zero[8];
  };
```

- `struct in_addr` → IPv4 address: 4 bytes packed in an integer
- `sin_family` → Set to `PF_INET` (why?)
- `sin_port`, `sin_addr` → Port number and *local* address in *network endian* byte order
- `sin_zero[8]` → Padding (why?)

What is `bind()` *actually for?*

C API: Usage of `bind()`

```
struct sockaddr_in sa;

memset(&sa, 0, sizeof(sa);
sa.sin_family = PF_INET;
sa.sin_port = htons(0);
sa.sin_addr = htonl(INADDR_ANY);
If (bind (s, (struct sockaddr *)&sa, sizeof(sa)) < 0) {
    perror("binding to local address");
    close(s);
    return -1;
}
```

Seems like a lot of work…

C sockets: resolving a host name

```
struct hostent *h;

h = gethostbyname(host)
if (!h || h->h_length != sizeof(struct in_addr)) {
     fprintf(stderr, "%s: no such host\n", host);
     return -1;
}
```

Result: `h->h_addr` points to the address of the machine we want to talk to.

C API: Connecting (finally!)

```
struct sockaddr_in sa;

sa.sin_port = htons(port)
sa.sin_addr = *(struct sockaddr *)h->h_addr;

if (connect (s, (struct sockaddr *)&sa, sizeof(sa)) < 0 {
     perror(host);
     close(s);
     return -1;
}
```

Filled in from previous slide

Address structure gives all needed info about the remote end-point

Sending and receiving data

```
ssize_t send(int s, const void *buf, size_t len, int flags);
```

- With no flags (0), equivalent to `write( s, buf, len )`

```
ssize_t recv(int s, void *buf, size_t len, int flags);
```

- With no flags, equivalent to `read( s, buf, len )`

```
ssize_t sendto(int s, const void *buf, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);

ssize_t recvfrom(int s, void *buf, size_t len, int flags,
               struct sockaddr *from, socklen_t *fromlen);
```

- And these two are for… ?

Putting it all together – the "W" client.

TCP server programming in C

```
int listen(int sockfd, int backlog);
```

- Takes a bound (but not connected!) socket
- Turns it into a listener for new connections
- Returns immediately
- `backlog`: number of outstanding connection attempts
  - See `accept()` on next slide
  - Traditionally, 5 (not any more…)
- What do you do with a listening socket?

TCP server programming in C

```
int accept(int sockfd,
        struct sockaddr *addr, socklen_t *addrlen);
```

- Takes a listening socket sockfd
- Waits for a connection request, and accepts it (!)
  - You don't get to say "no"…
- Returns a *new* socket for the connection
  - Plus the address of the remote peer

TCP server: example pattern

1. Create a server socket and bind to a local address
2. Call listen()
3. Loop:
   1. Call accept() and get a new ("connection") socket back
   2. Read client's request from the connection socket
   3. Write response back to the connection socket
   4. Close the connection socket

- See real example server...

Asynchronous programming: `O_NONBLOCK`

```
if ((n = fcntl (s, F_GETFL)) < 0
  || fcntl(s, F_SETFL, n | O_NONBLOCK) < 0) {
  perror("O_NONBLOCK");
}
```

Socket descriptor now behaves differently:

- `read/recv`: as normal if there is data to read. EOF returns `0`. Otherwise, returns `-1` and `errno` set to `EAGAIN`.
- `write/send`: if data cannot yet be sent, returns `-1` and `errno = EAGAIN`
- `connect`: if no immediate success, returns `-1` and `errno = EINPROGRESS`
- `accept`: if no pending connections, returns `-1` and `errno = EWOULDBLOCK`

Asynchronous programming: `select()`

```
int select(int nfds,
           fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *timeout);
```

Sets of file descriptors to watch for activity

```
void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

- Returns when anything happens on any set file (i.e. socket) descriptor, or the timeout occurs.
- The fd_sets are modified to indicate fds that are active%%

A basic event loop

- Operations to register callbacks for
  - File (socket) descriptors
  - Timeout events
- Map from socket descriptor $\rightarrow$ callback
- Priority queue of timer events
- Loop:
  - Process timeouts
  - Call select with next timeout
  - Process any active socket descriptors

Event programming:

- Event programming is hard
  - Callbacks $\Rightarrow$ need to maintain state machine for each activity ("stack ripping")
  - Anything that blocks has to be handled with a callback
  - Hard to deal with long-running operations
- But…
  - No need for synchronization (at least, with one processor)
  - *Very* scalable (only one thread)
  - Model similar to interrupts $\Rightarrow$ close to how one needs to implement a practical networking stack

More information on TCP and C

- Upcoming labs…
- Some of this material is from the excellent:

  "Using TCP Through Sockets",

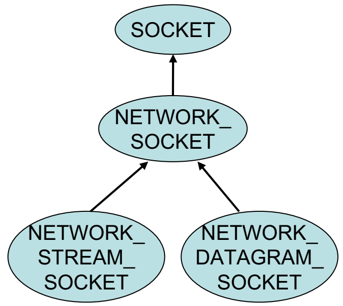
  by David Mazières, Frank Dabek, and Eric Peterson.

  http://people.inf.ethz.ch/troscoe/teaching/net2-1.pdf

Finally…

- Backup slides also cover Eiffel networking classes
  - Exercises/labs will be Java and C
  - Eiffel abstracts events into “pollers” and related objects
- Next week:
  - Java development
  - Eclipse tutorial for Java and C
- Then:
  - Transport protocols.

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
                    soc1.cleanup
                end
              rescue soc1.cleanup
              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), contd.

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

```
    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
```

- the server obtains access to the server
- *accept* - ensures synchronization to with the client
- *accept* - creates a new socket which is accesible 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
                  rescue
                   cleanup
                  end
            ...
```

The message exchanged between server and client

1. Creates a socket and setup the communication

2. Builds the list of strings

3. Sends the list of strings to the server

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), 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)
                        . . .
```

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), 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)
                   . . .
```

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), 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
```

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)

```
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
```

Receive a packet of size 10 characters

Prints the packet number of the packet

Prints all the caracters from the packet

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"

Example: Eiffel Command class (UDP), contd.

```
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 succesive 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
              -- succesive 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"