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

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
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)

• Big Issue in both worlds: Enable concurrency
• Two ways to enable multiple clients and concurrency
  • Threads or Events
• Demonstrate “Threads” using Java
• Demonstrate “Events” using C
Socket programming with TCP (Java)

Example client-server application

- client reads line from standard input (\texttt{inFromUser} stream), sends to server via socket (\texttt{outToServer} stream)
- server reads line from socket
- server converts line to uppercase, sends back to client
- client reads and prints modified line from socket (\texttt{inFromServer} stream)
Client/server socket interaction with TCP (Java)

Server (running on **favServer**)  

- create socket, port=6789, for incoming request:  
  ```java  
  welcomeSocket = ServerSocket();  
  ```
- wait for incoming connection request:  
  ```java  
  connectionSocket = welcomeSocket.accept();  
  ```
- read request from connectionSocket
- write reply to connectionSocket
- close connectionSocket

Client

- create socket, connect to **favServer**, port=6789:  
  ```java  
  clientSocket = Socket();  
  ```
- send request using clientSocket
- read reply from clientSocket
- close clientSocket

TCP connection setup
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("favServer", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());

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

        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
```
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("favServer", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());

        Create client socket, connect to server

        Socket clientSocket = new Socket("favServer", 6789);
        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
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("favServer", 6789);

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

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

        DataOutputStream outToServer =
            new DataOutputStream(clientSocket.getOutputStream());
```
Example: Java client (TCP), continued

```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 client (TCP), continued

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 client (TCP), continued

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()));

            System.out.println("Create welcoming socket at port 6789");

            String clientSentence = inFromClient.readLine();
            String capitalizedSentence = clientSentence.toUpperCase();
            System.out.println("Capitalized sentence: "+capitalizedSentence);

        }
    }
}
```
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()));

            // Wait on welcoming socket for contact by client
            ...
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()));
        }
    }
}
Example: Java server (TCP), continued

Create output stream, attached to socket

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

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '
';

outToClient.writeBytes(capitalizedSentence);
```
Example: Java server (TCP), continued

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

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '
';

outToClient.writeBytes(capitalizedSentence);
```
Example: Java server (TCP), continued

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

clientSentence = inFromClient.readLine();

capitalizedSentence = clientSentence.toUpperCase() + '
';

outToClient.writeBytes(capitalizedSentence);
```

Write out line to socket
Example: Java server (TCP), continued

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

clientSentence = inFromClient.readLine();

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

outToClient.writeBytes(capitalizedSentence);
```

End of while loop, loop back and wait for another client connection
Problem: One client can delay other clients
Problem: One client can delay other clients
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 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);
    /* ... */
}
One solution: Threads

```java
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());
    String clientSentence = inFromClient.readLine();
    String 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.
• Beyond the scope of this course
Client/server sockets: UDP (Java)

Server (running on favServer)

- create socket, port=9876, 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 (favServer, port=9876), send datagram request using clientSocket
- read reply from clientSocket
- close clientSocket
Example: Java client (UDP)

Client process

Input: receives packet (TCP received “byte stream”)

Output: sends packet (TCP sent “byte stream”)

sendPacket to network from network
receivePacket inFromUser

client UDP socket

keyboard monitor

input stream

stream

UDP packet

TCP sent “byte stream”

TCP received “byte stream”
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("favServer");

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

        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();
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("favServer");

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

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

Create

client socket

```
        clientSocket.send(new DatagramPacket(sendData, sendData.length, IPAddress, 1234));
        System.out.println("Waiting for response...\n");

        DatagramPacket incomingPacket = new DatagramPacket(receiveData, receiveData.length);
        try {
            clientSocket.receive(incomingPacket);
            String receivedLine = new String(incomingPacket.getData(), 0, incomingPacket.getLength());
            System.out.println("Server: " + receivedLine);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```
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));

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

        byte[] sendData = new byte[1024];
        byte[] receiveData = new byte[1024];
        String sentence = inFromUser.readLine();
        sendData = sentence.getBytes();

        DatagramSocket clientSocket = new DatagramSocket();
        DatagramPacket sendPacket = new DatagramPacket(sendData, sendData.length, IPAddress, 7777);
        clientSocket.send(sendPacket);

        DatagramPacket receivePacket = new DatagramPacket(receiveData, receiveData.length);
        clientSocket.receive(receivePacket);
        String receivedData = new String(receiveData, 0, receivePacket.getLength());
        System.out.println("Received: " + receivedData);
    }
}
```

Translate hostname to IP Address using DNS
Example: Java client  
(UDP, contd)

```java
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 client (UDP, contd)

```java
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();
```

Send datagram to server
Example: Java client (UDP, contd)

```java
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)

```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);
            // Process incoming data...
        }
    }
}
```
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);
        }
    }
}
```
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, contd)

```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);
```
Example: Java server (UDP, contd)

```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);
```
Example: Java server (UDP, contd)

```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);
}
```
Example: Java server (UDP, contd)

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 datagram
TCP Client in C step by step...

- Resolve the host name: typically with DNS
- Create a socket
- Bind the socket support for multiple NICs
- Connect the socket connect to other machine
- Write some data
- Read some data
- Close
- Exit

*General flavour: much lower level...*

*more flexibility: can do everything, must do everything*

- support for several NICs on one machine
- support several hostnames for the same NIC
- support for IPC on the same machine
C sockets: resolving a host name

```c
struct addrinfo hints, *res;

memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;  // protocol
hints.ai_socktype = SOCK_STREAM;   // "TCP" vs. "UDB"
int error = getaddrinfo("localhost", "8080", &hints, &res);

#SBATCH  create, bind, work...

freeaddrinfo(res);   // release memory
```
Create socket

... int s = socket(res->ai_family, res->socktype, res->ai_protocol);

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!)
Bind: hostname + port

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

bind(s, res->ai_addr, res->ai_addrlen);
struct addrinfo hints, *res;

// resuse hints from binding to localhost
int error = getaddrinfo("favserver", "http", &hints, &res);
connect(s, res->ai_addr, res->ai_addrlen);

• Binding to localhost and connecting to remote host are
done in exactly the same way.

• N.B. "old style" of doing this in different ways is
discouraged.

• Details: look into man pages of "getaddrinfo"
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...?
TCP server programming in C

```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

```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

• Alternative to threads: “Event Driven”
• Only one thread: manage each activity explicitly
• Basic idea:
  – Don’t wait for anything to happen, if there’s something else you can do in the meantime.
• Requires:
  1. Some way to stop system calls from waiting (“Blocking”)
  2. Some way to wait for more than one thing (i.e., “wait until one of the following things happens”).
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()

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

```c
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
Asynchronous programming:

select()

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

```c
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 → 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
    ⇒ 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 ⇒ 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.
Finally...

• Backup slides also cover Eiffel networking classes
  – Exercises/labs/exams will be Java and C
  – Eiffel abstracts events into “pollers” and related objects

• Next week:
  – Queueing Theory,
  – Reliability
  – Transport protocols...
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
process is
class OUR_MESSAGE
crete make
is_equal, copy
end
end

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

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

- the server obtains access to the server
- accept - ensures synchronization to 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

Extends the message received from the client

Sends the extended message back to the client

Closes the socket

Example: Eiffel Server (TCP), contd.
Example: Eiffel Client (TCP - stream socket)

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

```eiffel
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
**Example: Eiffel Client (TCP), continued**

```eiffel
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)
    end
  end
end
```

- **build_list** is responsible for building the list of strings for transmission to the server.
- **process_received** handles the received message and prints its content in sequence.

- **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. Attaches them to a poller
3. Sets 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
```

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)

```eiffel
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), continued

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

```
... 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
Example: Eiffel Command class (UDP)

```eiffel
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
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”