ISCTE IUL
Instituto Universitário de Lisboa

Introduction to Distributed Applications in Java

Programação Concorrente e Distribuída

Parallel and Distributed Programming

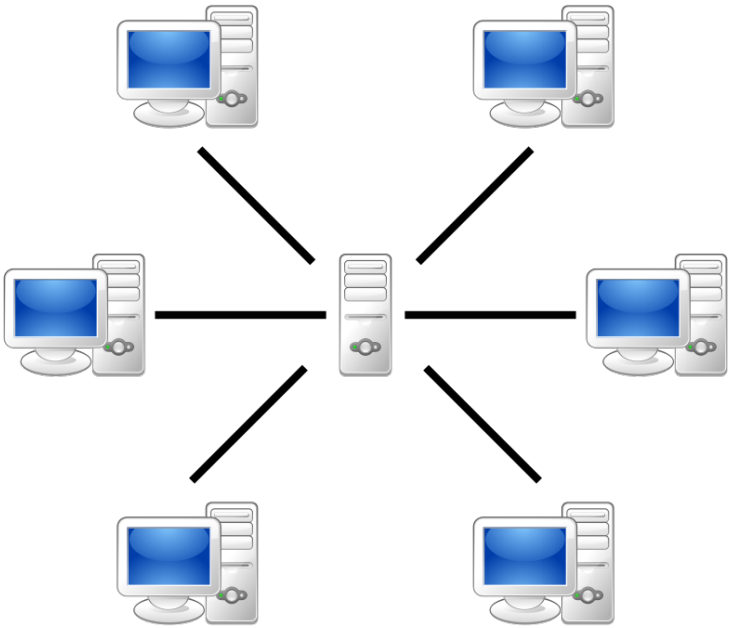
2012-2013

After this class you will be able to…

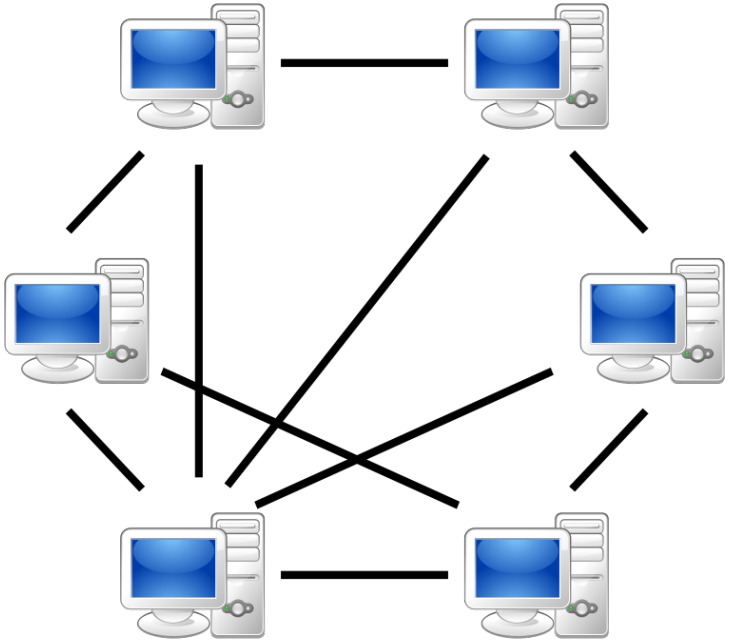
- Understand what is a distributed application.
- Know how to connect two applications using TCP/IP sockets.
- Understand how to send and receive messages from other applications.
- Know how to design a client/server application.
- Comprehend the mechanisms behind java serialization.
- Control java serialization.
- Send and receive java objects from other applications.

Distributed Applications

Client Server



Peer 2 Peer



Fonte imagens: http://en.wikipedia.org/wiki/Peer-to-peer

Distributed Java Programming

- Programs on different machines can exchanges messages and/or objects – *basic network programming through sockets*.

- Invocation of methods on object that reside on remote machines as if the objects existed on the same machine - *Remote Method Invocation (RMI);*

- A connection to a relational database that can reside on a remote machine - *Java DataBase Connectivity (JDBC);*

Distributed Java Programming

- Services accessed though a webpage – *Servlets e Java Server Pages (JSP);*
- Connecting JAVA application to remote applications possibly developed in other languages (for instance C++) though a commen standard– for instance: *Common Object Request Broker Architecture (CORBA);*
- Separation of business logic form the communication logic in client/server applications, namely with respect to transactions and security - *Enterprise JavaBeans (EJBs);*
- Service discovery and mobile code (Jini – Apache River).

Part I

Basic network programming

Basic Network Programming

- Identification of a machine on the Internet: IP (*Internet Protocol),* can have two forms:
  1. DNS (*Domain Name System). Ex.: www.iscte.pt*
  2. Numerical address
     - IPv4. Ex.: 123.255.28.120 (32 bits)
     - IPv6. Ex.: fe80::4c4f:bfa3:a634:1f82 (128 bits)

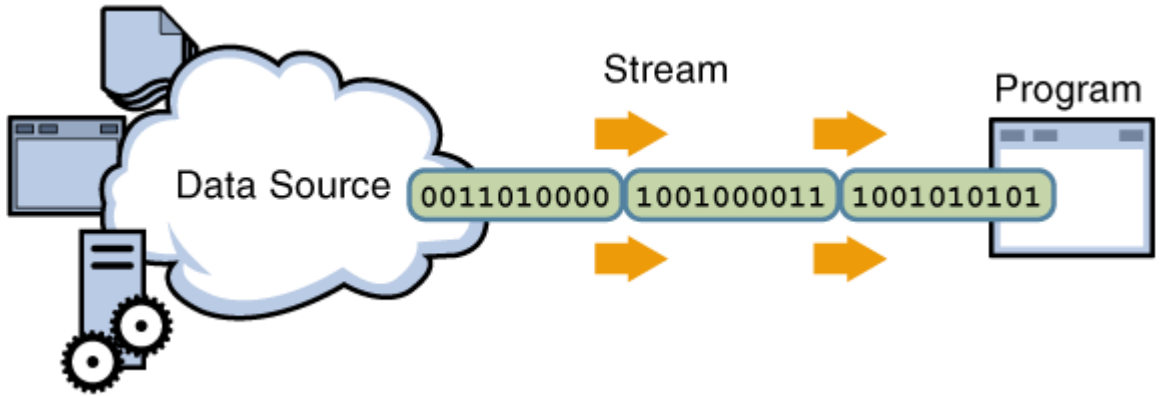
- One IP address can be represented by an object of type InetAddress. This IP address is returned by the static method InetAddress.getByName(String host).
  - This object is then used to build a socket, that is fundamental to the establishment of a connection between two machines in the network.

Example

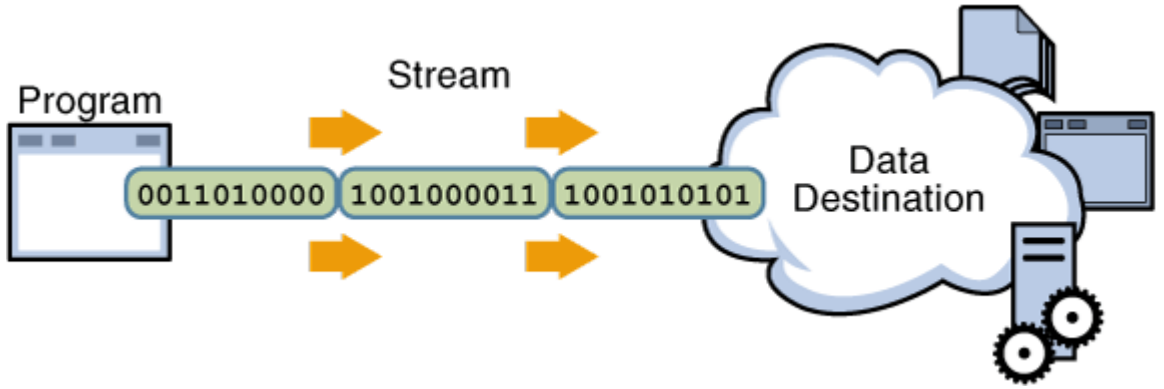
```
import java.net.*;

public class QuemSouEu {
    public static void main(String[] args)
           throws Exception {
        if (args.length != 1) {
            System.err.println("Utilização:" +
             "QuemSouEu NomeDaMáquina");
            System.exit(1);
        }
        InetAddress a =
             InetAddress.getByName(args[0]);
        System.out.println(a);
    }
}
```

Streams (revision)

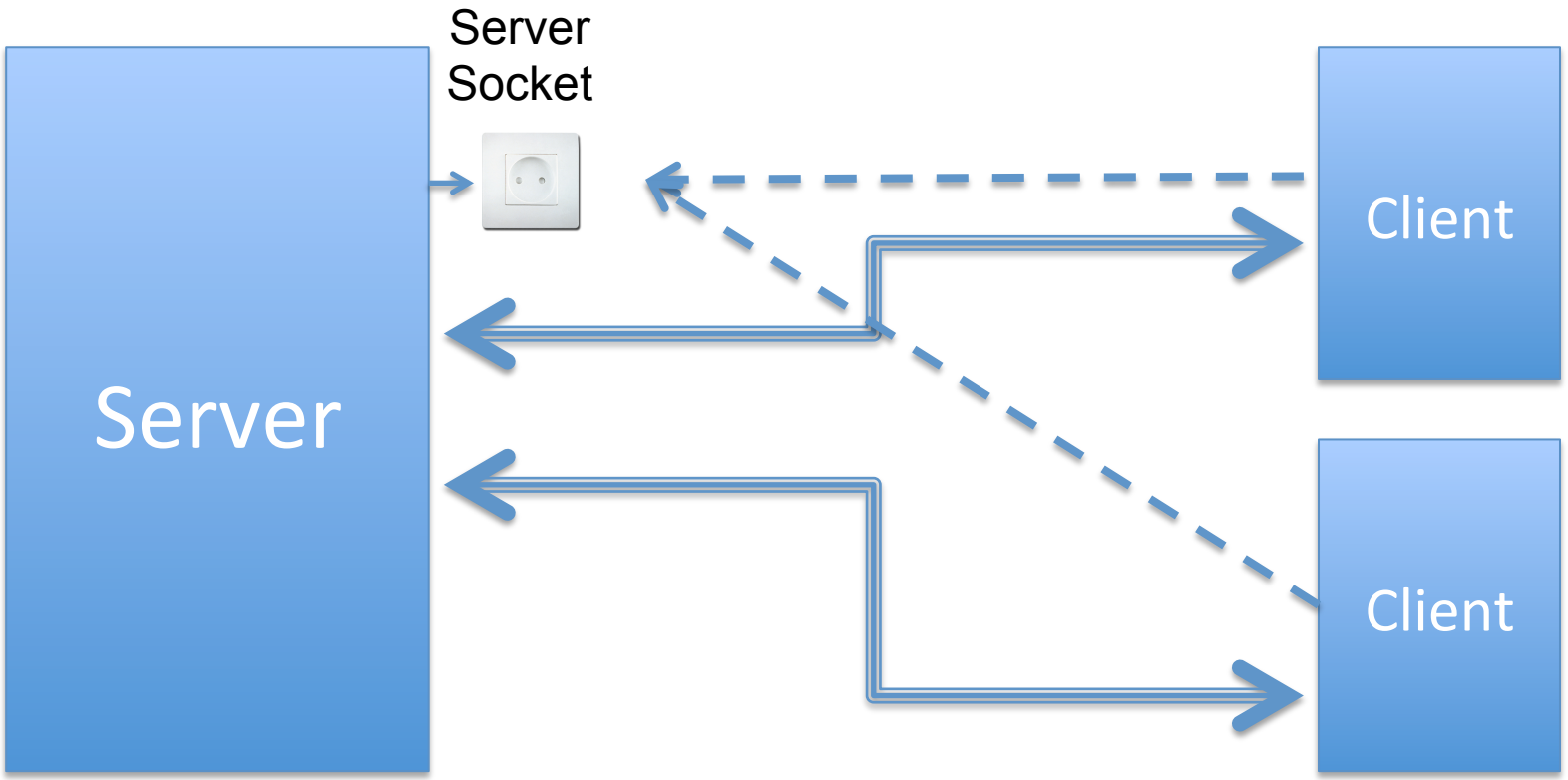
Input stream:



Output stream:



TCP connection flow



Socket

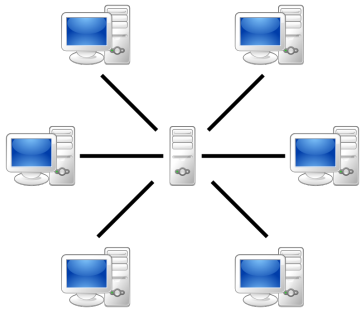
- A socket represents the endpoint of a connection between two machines:
  - For each connection, two sockets exist: one on each of the connected machines.
  - From a socket, a program can create an InputStream and an OutputStream, which allows the program to respectively receive and send data.
- Initially, an instance of the class ServerSocket exist on the server. The server socket enables a machine to accept connections from other remote computers.
  - When another program connects to the server socket and when the server accepts the connection, the server socket returns an instance of the class Socket.

Ports

- Whenever one machine connects to another, it connects on a certain port.
- Depending on the port that client tries to connect to, a server can offer different services.
- Normally, system services are reserved for ports 1 to 1024. A port is a logical (and not physical) concept.
- On Linux/Unix machines typically only root/admin processes can listen on ports bellow 1024.

Part II

Basic client/server model



Servers and Clients

Server – a computer or an application that offers a service and that waits for one or more clients to connect.

Client – an application or a computer that connects to a server

Simple Server

```
public class SimpleServer {
	public static final int PORTO = 8080;
	private BufferedReader in;
	private PrintWriter out;

	public static void main(String[] args) {
		try {
			new SimpleServer().startServing();
		} catch (IOException e) {
			// TODO Auto-generated catch block
		}
	}
}
```

Simple Server

```
public void startServing() throws IOException {
   ServerSocket s = new ServerSocket(PORTO);
   System.out.println("Lançou ServerSocket: " + s);
   try {
      Socket socket = s.accept();
      try {
         System.out.println("Conexão aceite: " +
                            socket);
         doConnections(socket);
         serve();
      } finally {
         System.out.println("a fechar...");
         socket.close();
      }
   } finally {
      s.close();
   }
}
```

Simple Server

```
private void doConnections(Socket socket)
                                        throws IOException {
      in = new BufferedReader(new InputStreamReader(
            socket.getInputStream()));
      out = new PrintWriter(new BufferedWriter(
               new OutputStreamWriter(
                  socket.getOutputStream())), true);
}

private void serve() throws IOException {
      while (true) {
         String str = in.readLine();
         if (str.equals("FIM"))
            break;
         System.out.println("Eco: " + str);
         out.println(str);
      }
}
```

- **Server sockets** have to be closed by calling close() on the socket. If sockets are not closed properly, they will remain open for a while. This takes up system resources and prevents new server sockets from being opened on the same port (for a while).

- What is the purpose of the try-finally block *inside* the method `startServing` that is declared as "throws IOException "?

Simple Client

```
public class SimpleClient {

    private BufferedReader in;
    private PrintWriter out;
    private Socket socket;




    public static void main(String[] args) {
          new SimpleClient().runClient();
    }
```

Simple Client

```
public void runClient() {
	try {
		connectToServer();
		sendMessages();

	} catch (IOException e) {
		// ERRO...
	} finally {
		System.out.println("a fechar...");
		try {
			socket.close();
		} catch (IOException e) {
			e.printStackTrace();
		}
	}
}
```

Simple Client

```
private void connectToServer() throws IOException {
  InetAddress endereco =InetAddress.getByName(null);
  System.out.println("Endereço = " + endereco);
  socket = new Socket(endereco, SimpleServer.PORTO);

  System.out.println("Socket = " + socket);

  in = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));

  out = new PrintWriter(new BufferedWriter(new
    OutputStreamWriter(
        socket.getOutputStream())), true);
}
```

Simple Client

```
private void sendMessages() throws IOException {
          for (int i = 0; i < 10; i++) {
                out.println("Olá " + i);
                String str = in.readLine();
                System.out.println(str);
          }
          out.println("FIM");
}
```

Local host

- It is possible to test the client and server programs without a network connection.
- The address *localhost* denotes the local machine*:*
  - InetAddress addr = InetAddress.getByName(null);

    or
  - InetAddress addr = InetAddress.getByName("127.0.0.1");

    or
  - InetAddress addr = InetAddress.getByName("localhost");

Outputs

Output of the simple Client:

```
Endereço = localhost/127.0.0.1
Socket = Socket[addr=localhost/
  127.0.0.1,
  port=8080,localport=1094]
Olá 0
Olá 1
Olá 2
Olá 3
Olá 4
Olá 5
Olá 6
Olá 7
Olá 8
Olá 9
a fechar...
```

Output of the simple server:

```
Lançou ServerSocket:
ServerSocket[addr=0.0.0.0/0.0.0.0,
  port=0,localport=8080]
Conexão aceite:
Socket[addr=127.0.0.1/127.0.0.1,
  port=1094,localport=8080]
Eco: Olá 0
Eco: Olá 1
Eco: Olá 2
Eco: Olá 3
Eco: Olá 4
Eco: Olá 5
Eco: Olá 6
Eco: Olá 7
Eco: Olá 8
Eco: Olá 9
a fechar...
```

Part III

Serialization and object streams

Serialization

- Object serialization is the process of transforming the state of an object into a sequence of bytes;
- It also allows the inverse process, to rebuild the state of an Object based on a sequence of bytes;
- These two processes allow you to send an Object through a channel like a socket or to store that object in the filesytem to use later.

Some uses of Serialization

- **Network**: Serialization is much easier than writing messages in special formats for each type of messages to be passed between client and server. Just send the common Objects back and forward.
- **File I/O**: You can save the state of an application to disk to latter re-use.
- **RDBMS**: You can store entire objects in relational databases as Blobs.

Serialization in Java

Java API implements objects serialization in a very simple way.

The *Serializable* interface

- To add persistence to an Object it has to implement the *Serializable* interface*:*

```
public class PersistentTime implements Serializable {
      private Date time;
      public PersistentTime(){
         time = Calendar.getInstance().getTime();
      }
      public Date getTime(){
         return time;
      }
}
```

- This indicates to the JVM that this class can be serialized. (Acts like a marker)

ObjectOutputStream

Serialization is done by the class ObjectOutputStream

```
public class FlattenTime{
  public static void main(String [] args){
    PersistentTime time = new PersistentTime();
    try {
      FileOutputStream fos = new
                 FileOutputStream("time.ser");
     ObjectOutputStream out = new ObjectOutputStream(fos);
            out.writeObject(time);
            out.close();
     }
     catch(IOException ex)     {
        ex.printStackTrace();
      }
   }
}
```

ObjectInputStream

The recovery of a serialized object is done by the class ObjectInputStream.

```
public class InflateTime{
  public static void main(String [] args){
    try{
        FileInputStream fis = new FileInputStream("time.ser");
            ObjectInputStream in = new ObjectInputStream(fis);
        PersistentTime time = (PersistentTime)in.readObject();
        in.close();
    }
    catch(IOException ex){ ex.printStackTrace(); }
    catch(ClassNotFoundException ex){ ex.printStackTrace();}
    System.out.println("Flattened time: " + time.getTime());
    System.out.println("Current time: " +
    Calendar.getInstance().getTime());
 }
}
```

Non-serializable Objects

- Only objects that implement the Serializable interface can be serialized;
- The base class *Object* doesn't implement *Serializable* interface
- Usually you don't want to serialize some objects:
  - *Sockets*
  - *Threads*
  - *OutputStream*
  - *…*

You don't usually want to serialize a *Thread:*

```
public class PersistentAnimation implements
                              Serializable, Runnable {
   transient private Thread animator;
   private int animationSpeed;
   public PersistentAnimation(int animationSpeed){
      this.animationSpeed = animationSpeed;
      animator = new Thread(this);
      animator.start();
   }
   public void run(){
      while(true) {
       // do animation here
      }
    }
}
```

transient Objects

- Writing example:

```
PersistentAnimation animation = new
                          PersistentAnimation(10);
FileOutputStream fos = ...
ObjectOutputStream out = new ObjectOutputStream(fos);
out.writeObject(animation);
```

- What about reading? (readObject)
  - The constructor is only called when the object is instantiated
  - When reading we are NOT creating a new instance
  - After reading a new thread is not instantiated

Redefinition of the serialization process

One can specify how the object serialization is done:

Write Objects

```
private void writeObject(ObjectOutputStream out)
                    throws IOException;
```

Read Objects

```
private void readObject(ObjectInputStream in)
          throws IOException,
          ClassNotFoundException;
```

Example

In the previous example

```
private void writeObject(ObjectOutputStream out)
    throws IOException {
  out.defaultWriteObject();
}
private void readObject(ObjectInputStream in)
     throws IOException, ClassNotFoundException {
  in.defaultReadObject();
  startAnimation();
}
private void startAnimation(){
  animator = new Thread(this);
  animator.start();
}
```

Non-serializable objects

How can we prevent a class from being serializable when it extends a serializable class?

We can't unimplement an interface!

We can stop the serialization process by throwing an exception: `NotSerializableException:`

```
private void writeObject(ObjectOutputStream out)
                                    throws  IOException {
       throw new NotSerializableException("Not today!");
}
private void readObject(ObjectInputStream in)
                                     throws IOException {
       throw new NotSerializableException("Not today!");
}
```

The object stream's cache

- To be efficient the *ObjectOutputStream* keeps track of the references that have been sent.
- The *ObjectInputStream* on the other side keeps a cache of the objects that have been received.
- This interesting and efficient mechanism can be problematic.

The object stream's cache

```
ObjectOutputStream out = new ObjectOutputStream(...);
  MyObject obj = new MyObject();
  // has to be serializable
  obj.setState(100);
  out.writeObject(obj);
  // sends the object with state 100
  obj.setState(200);
  out.writeObject(obj);
  // will just send the reference of the obj.
```

On the *ObjectInputStream* the reference is used to search the cache for the object. What is found is the object in state 100.

The object stream's cache

There are two solutions to deal with this problem:

- Close the *Stream* after each write.
- Use `ObjectOutputStream.reset()` method to reset the cache.

More information

Java I/O Streams:

http://docs.oracle.com/javase/tutorial/essential/io/

Java Lesson: All About Sockets:

http://docs.oracle.com/javase/tutorial/networking/sockets/

Discover the secrets of the Java Serialization API

http://java.sun.com/developer/technicalArticles/Programming/serialization/

Java Object Serialization Specification

http://download.oracle.com/javase/6/docs/platform/serialization/spec/serialTOC.html

Summary

- Network programming in Java
- Streams (revision)
- Sockets
- Client-server example
- Java Object Serialization
  - *Serializable interface*
  - *ObjectOutputStream*
  - *ObjectInputStream*
  - *Non serializable attributes*
  - *Redefining the serialization process*
  - *Non serializable Objects*
  - *Stream's caches*