# Java Sockets network programming

Patrice Torguet
IRIT/VORTEX
Paul Sabatier University

# Schedule

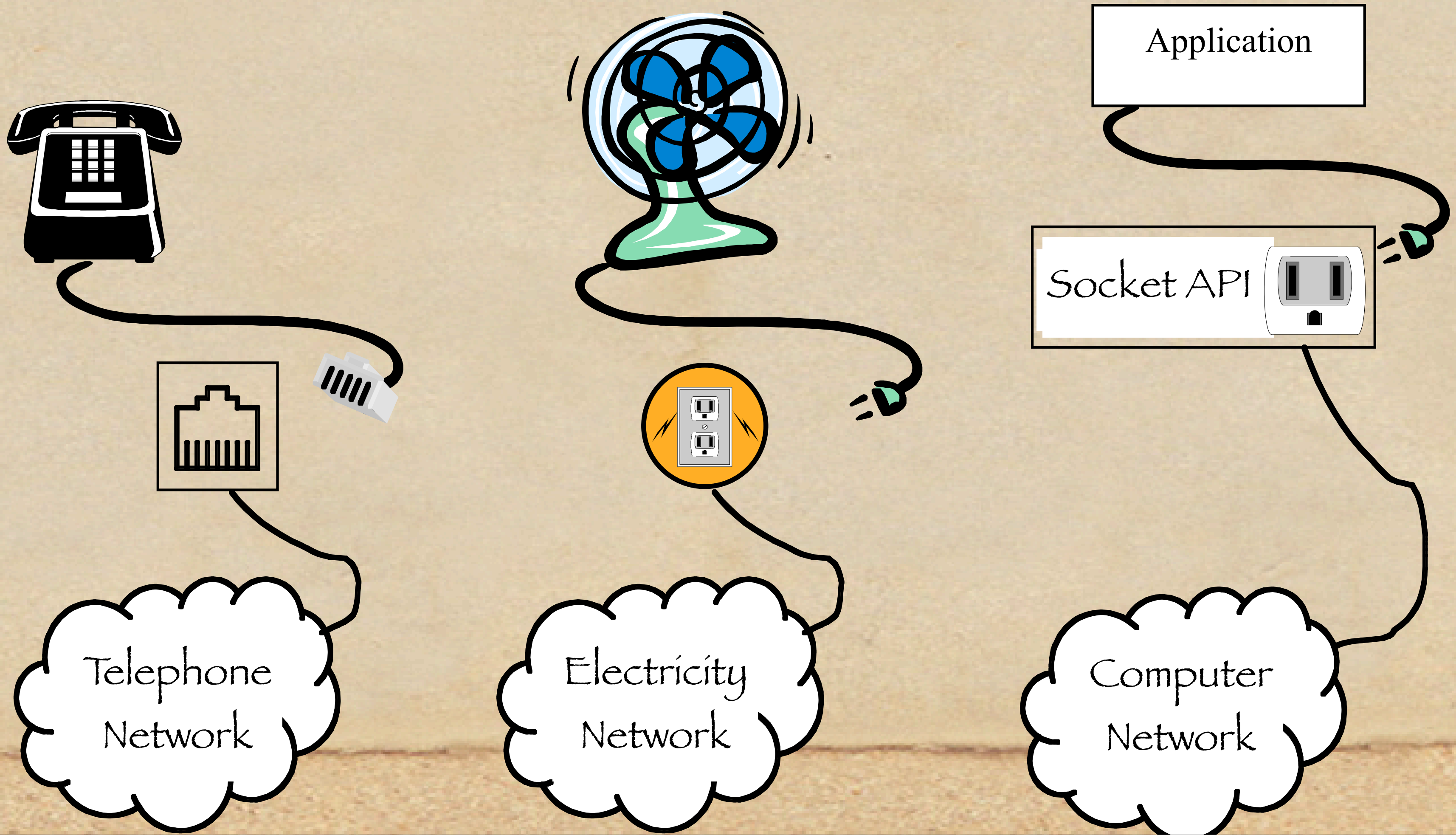
- Introduction to BSD Sockets
- Transport Protocols
- Java socket programming
- Conclusion

# BSD Sockets

- BSD (Berkeley Software Distribution) is a UNIX like OS developed by Berkeley University since 1977
- It introduced Sockets (version 4.3 de BSD - 1983)
- Sockets are now in every OS

# BSD Sockets

- A socket is an abstract object that represents the end-points of a communication channel
- The socket term comes from an electricity/ phone socket metaphor



# BSD Sockets

- Sockets are also an API for:
  - Manipulating data related to communication (source and destination addresses, port numbers...)
  - Creating a communication channel (if it is needed)
  - Sending and receiving application level PDU (protocol data units)
  - Controlling and customizing communication

# BSD Sockets

- BSD sockets allow both
  - Process to process communications (AF_UNIX domain - not available with Java or on Windows e.g.)
  - Networked communications
    - using TCP/IP (AF_INET domain)
    - or other protocol suites (e.g. ATM)

# BSD Sockets

- Sockets can be an abstraction related to
  - The network
    - for TCP/IP a socket is a quintuplet: local IP @, local port, remote IP @, remote port, transport protocol (TCP or UDP)
  - Computer programming
    - a socket can be manipulated like a file descriptor (similarly to FIFOs and pipes)

# Transport Protocols

- Using the AF_INET domain you can communicate
  - with virtual channels (STREAM)
    - uses TCP connections
  - with independent messages (DGRAM)
    - uses UDP datagrams
    - allows for point to point or multipoint delivery (broadcast / multicast)

# Transport Protocols

- Port numbers
  - On one computer you can have several applications that use the network at the same time
  - Problem: how can we identify with which application we want to talk
  - Solution: each application is identified by a unique id (unique for a computer and for a protocol) called a port number (16 bits integer - 65535 different ports - 0 is not used)

# Transport Protocols

- Several types of port numbers
  - System or well known ports (1-1023) - OS reserved (example 80 - web servers)
  - User or registered ports (1024- 49152) - reserved to specific applications (like the first ones) registered with IANA (Internet Assigned Numbers Authority - www.iana.org) (example 26000 - Quake)
  - Private or dynamic ports (others) - used by unregistered applications and TCP clients

# IP: Internet Protocol

- Manages
  - Addressing (IP @) and routing in the Internet
  - Fragmentation in order to adapt to the low-level network protocols maximum PDU size (MTU)
    - Attention: this increases loss probability (if a fragment is lost, the whole datagram is lost)
  - TTL: maximum number of routers that the datagram can cross

# STREAM / TCP Sockets

- Manages:
- a bidirectional byte stream which is
  - Reliable (no loss, no duplication) and ordered
  - "Virtual connection" between both applications (we can detect connection failures)
  - The most used protocol today (mail, web, ftp...)
- Need to code 3 phases: connection, dialogue, dis-connection

# DGRAM / UDP Sockets

- Manages:
- Independent message transfers using UDP datagrams
    - Non reliable and non ordered: best effort
    - Faster than TCP
    - Mostly used by multimedia applications (audio, video, games) and for LAN only applications
- Send/receive messages with a socket

# DGRAM / UDP Sockets

- Advantages
- Simpler protocol (no virtual connections, no reliability management) and therefore less CPU hungry
- Faster protocol (no order management and congestion avoidance): messages are sent directly (no need to wait when the reception window is full) and delivered directly to the application (no reordering)

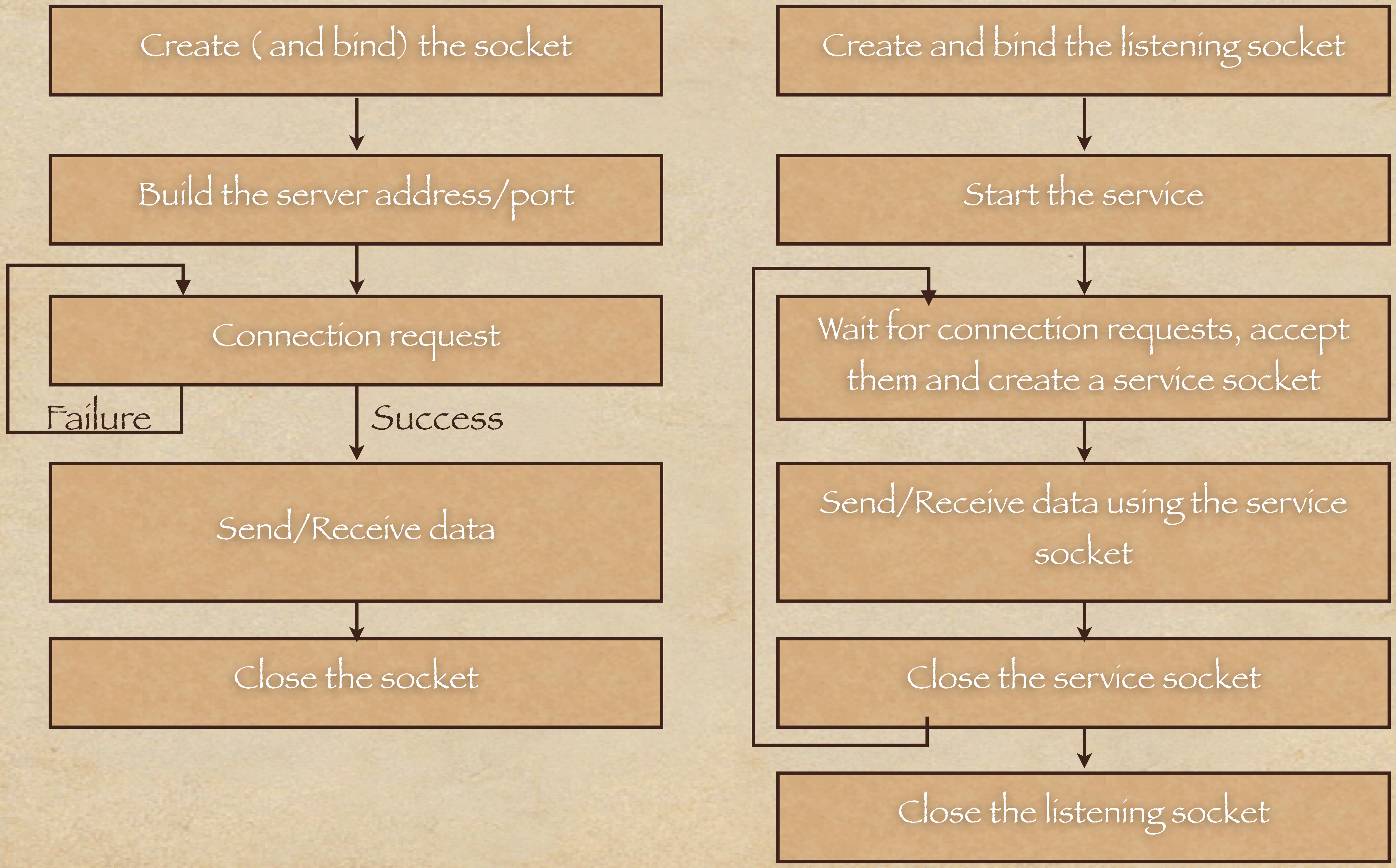
# DGRAM / UDP Sockets

- Advantages
- OSes limit the number of simultaneous TCP connections
- UDP hasn't this problem because a UDP socket can send/receive to/from several destinations. It is therefore more adapted to large scale applications (P2P for example)
- Moreover you can broadcast/multicast with UDP (not with TCP)

# DGRAM / UDP Sockets

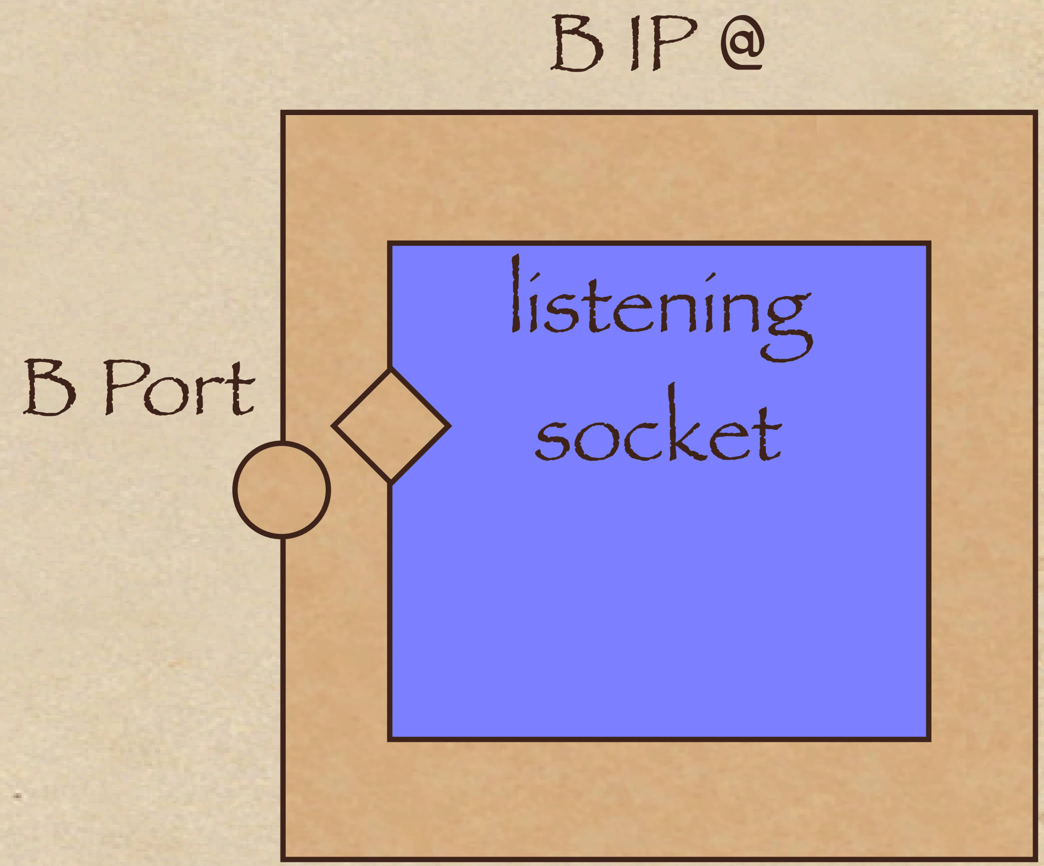
- Disadvantages
- Security problem: a UDP socket can receive data from any computer/application
- Therefore, most firewalls are configured to block incoming UDP traffic

# TCP and the client/server model



Create ( and bind) the socket
Build the server address/port
Connection request
Failure
Success
Send/Receive data
Close the socket
Create and bind the listening socket
Start the service
Wait for connection requests, accept them and create a service socket
Send/Receive data using the service socket
Close the service socket
Close the listening socket

# Create + bind the listening socket



listening s.

Local @ = B IP @ or Any

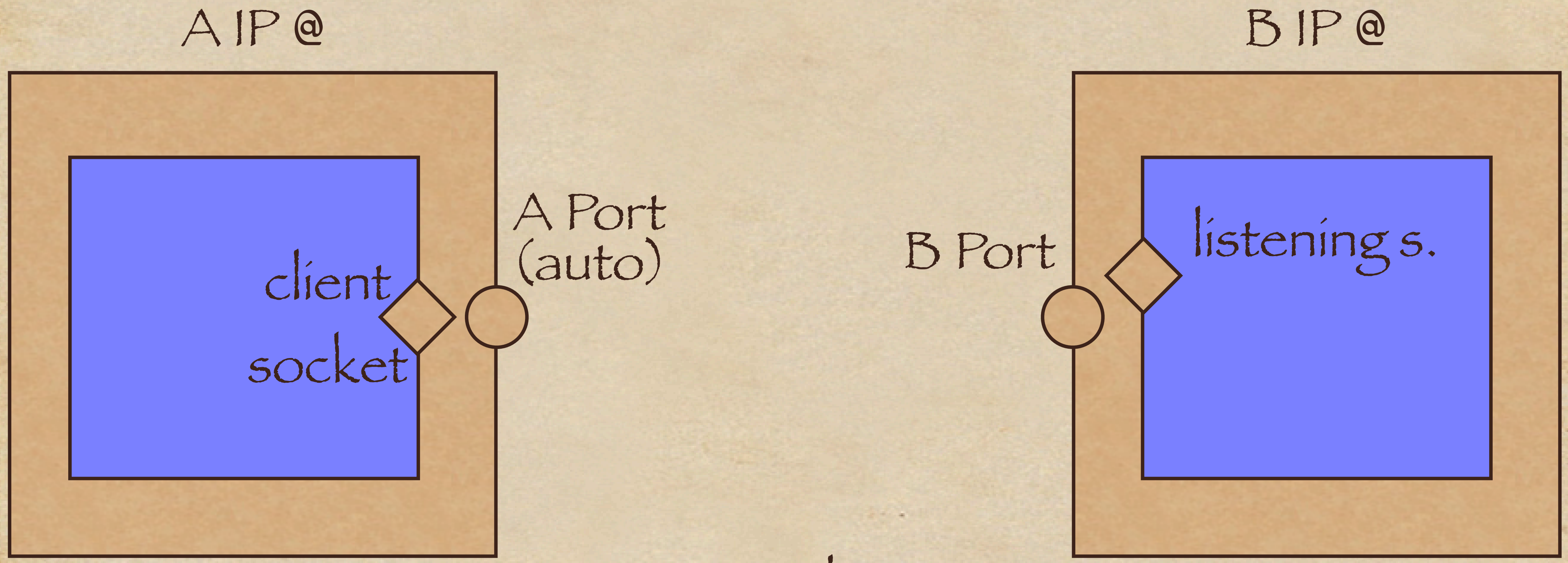
Local port = B port

Remote @ = Any

Remote port = 0

Protocol = TCP

# Create + bind client socket



client socket

@ loc = @ IP A

port loc = port A

@ dist = Any

port dist = 0

proto = TCP

listening s.

Local @ = B IP @ or Any

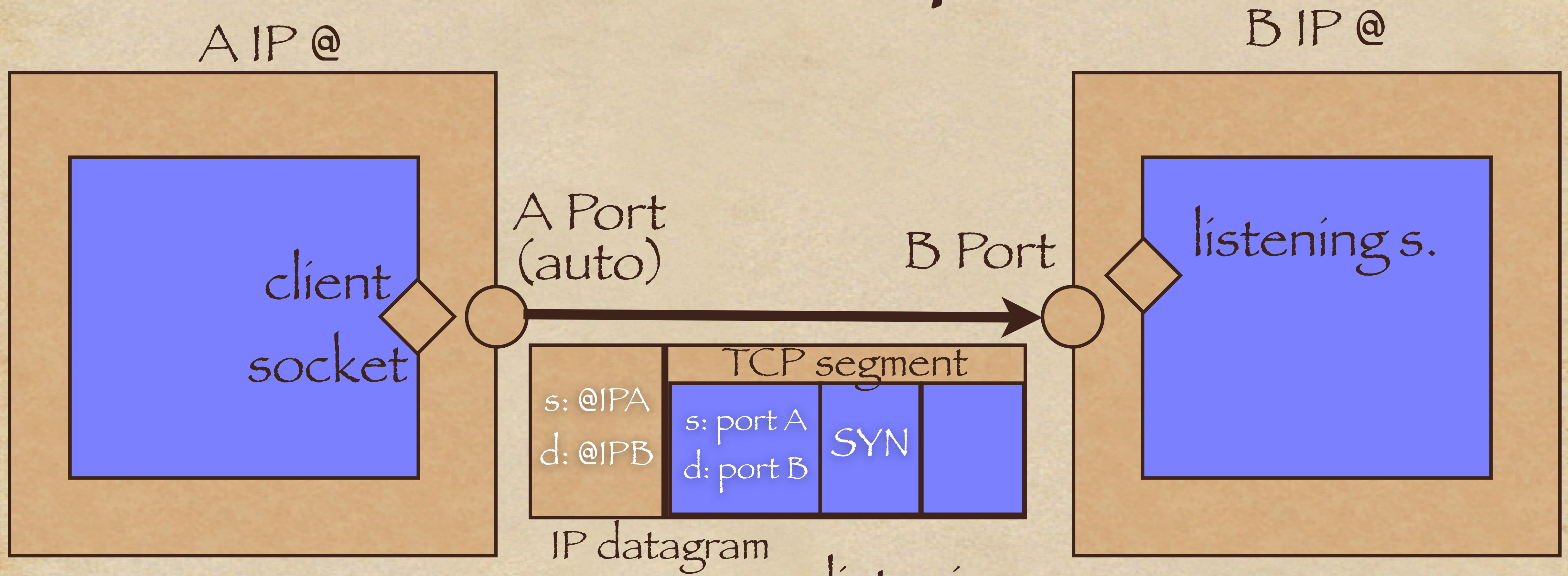
Local port = B port

Remote @ = Any

Remote port = 0

Protocol = TCP

# Connection request



client socket

loc @ = A IP @

loc port = A port

rem @ = B IP @

rem port =B port

proto = TCP

listening s.

Local @ = B IP @ or Any

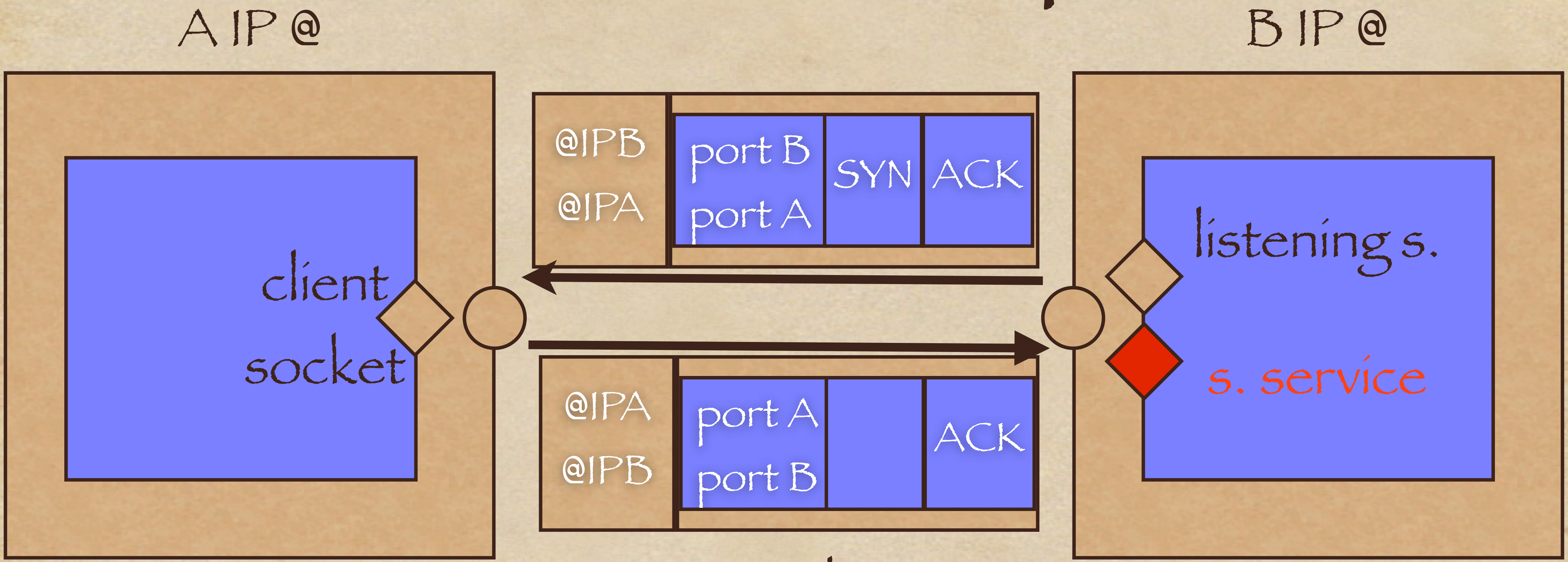
Local port = B port

Remote @ = Any

Remote port = 0

Protocol = TCP

# Connection Acceptation

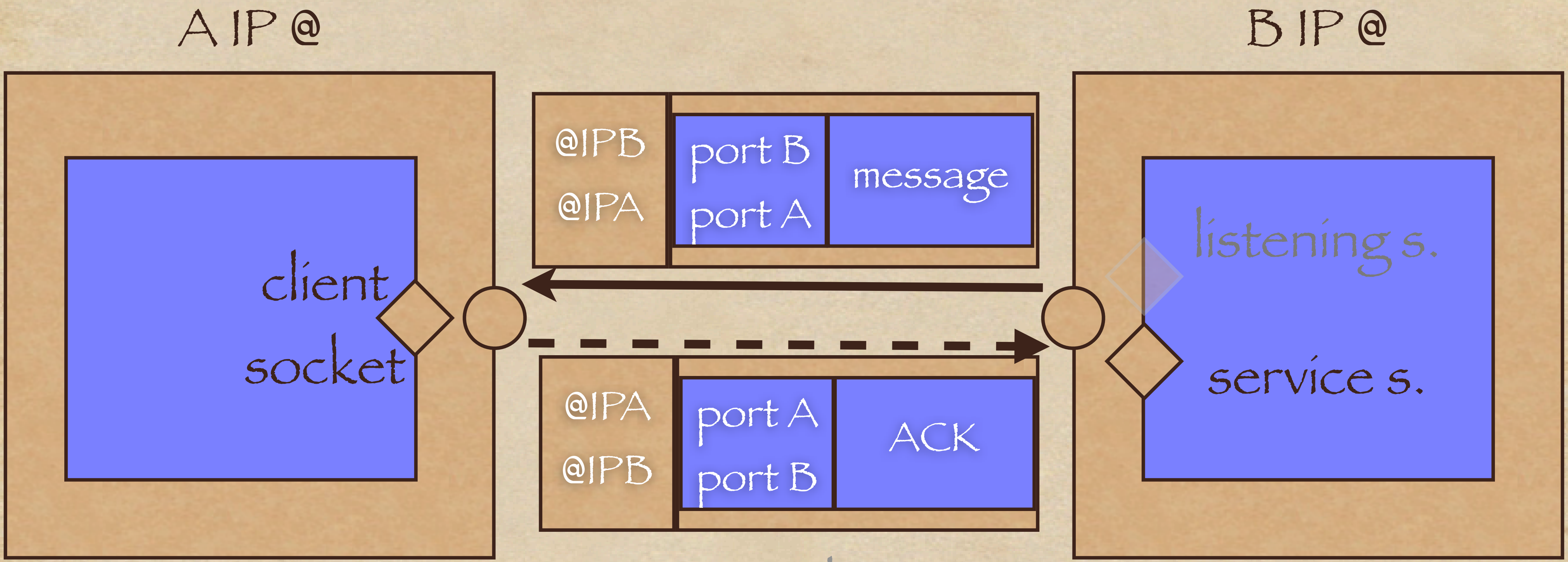


client socket
- loc @ = A IP @
- loc port = A port
- rem @ = B IP @
- rem port =B port
- proto = TCP

listening s.
- Local @ = B IP @ or Any
- Local port = B port
- Remote @ = Any
- Remote port = 0
- Protocol = TCP

service s.
- loc @ = B IP @
- loc port = B port
- rem @ = A IP @
- rem port = A port
- proto = TCP

# Communication



client socket

loc @ = A IP @

loc port = A port

rem @ = B IP @

rem port =B port

proto = TCP

listening s.

Local @ = B IP @ or Any

Local port = B port

Remote @ = Any

Remote port = 0

Protocol = TCP

service s.

loc @ = B IP @

loc port = B port

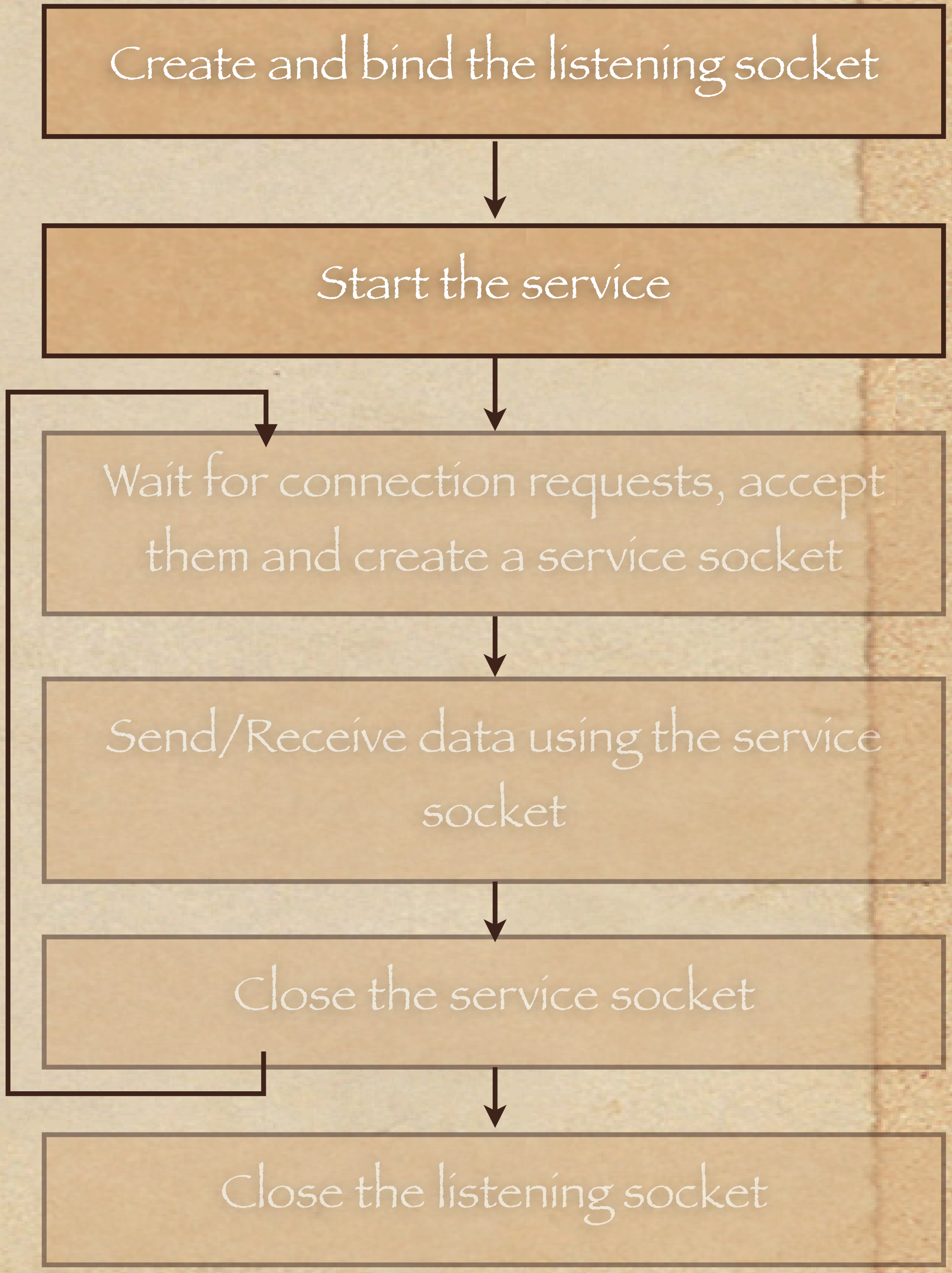
rem @ = A IP @

rem port = A port

proto = TCP

# Java: TCP Server

- java.net.ServerSocket class
  - listening sockets
  - most used constructor allows to chose the port (or 0 for OS automatic port)
  - Other constructors exist that let you choose the local IP @ and/or the size of the listening queue (see the java doc for the java.net package)



Create and bind the listening socket

Start the service

Wait for connection requests, accept them and create a service socket

Send/Receive data using the service socket

Close the service socket

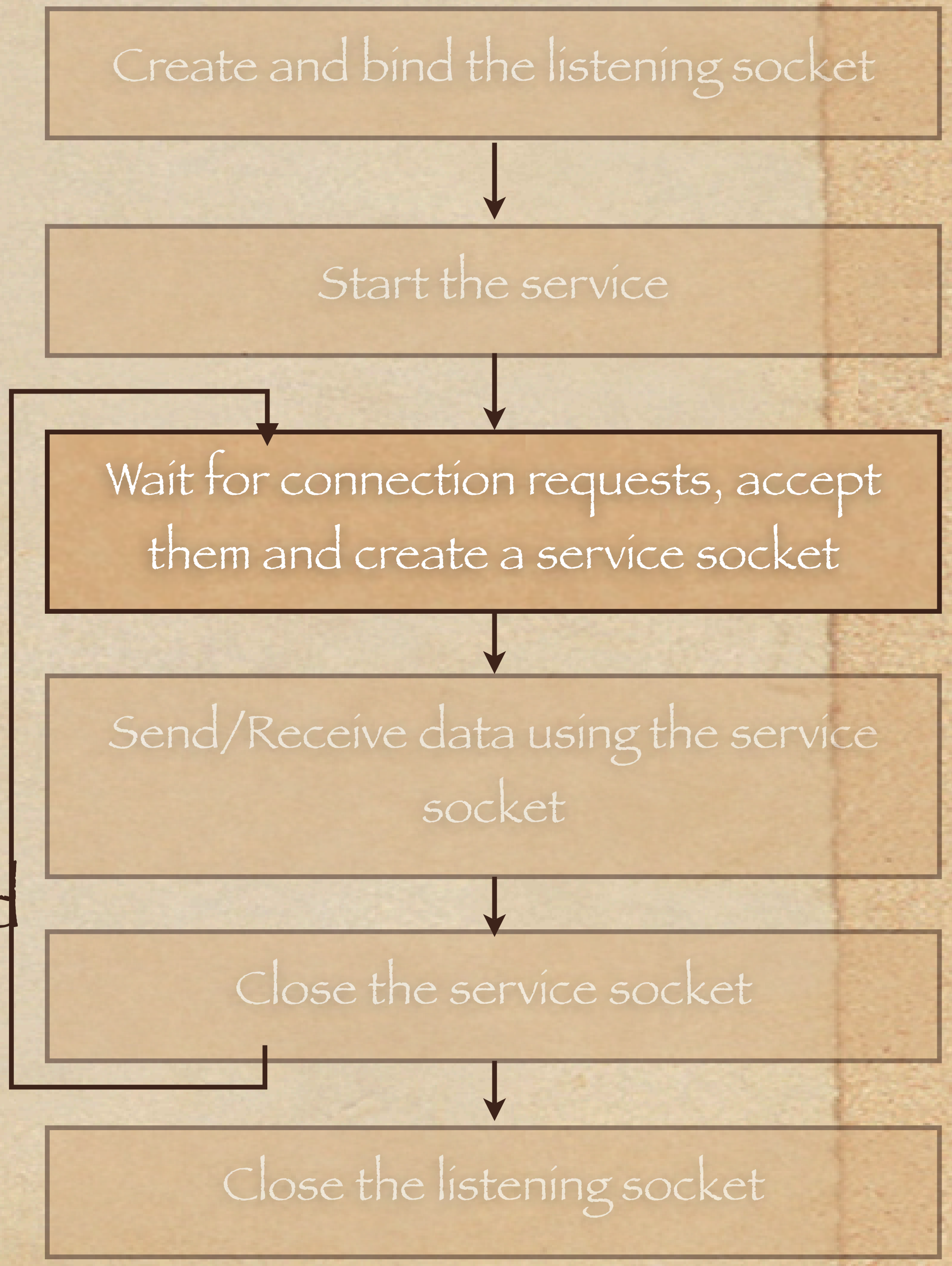
Close the listening socket

# Java: TCP Server

```
import java.net.ServerSocket;
import java.net.Socket;
import java.io.IOException;
import java.io.DataInputStream;
import java.io.DataOutputStream;

ServerSocket listeningSock;                    // ServerSocket declaration

//  constructs a server socket and chose a port number
try {
     listeningSock = new ServerSocket(13214);
}
catch(IOException ioe) {
    System.out.println("Server socket creation error: " + ioe.getMessage());
    return;
}
```

# Java: TCP Server

- accept method
  - Waits for a connection request
  - When we accept a request, it creates a service socket (Socket class instance)
- Socket is the type used for service and client sockets

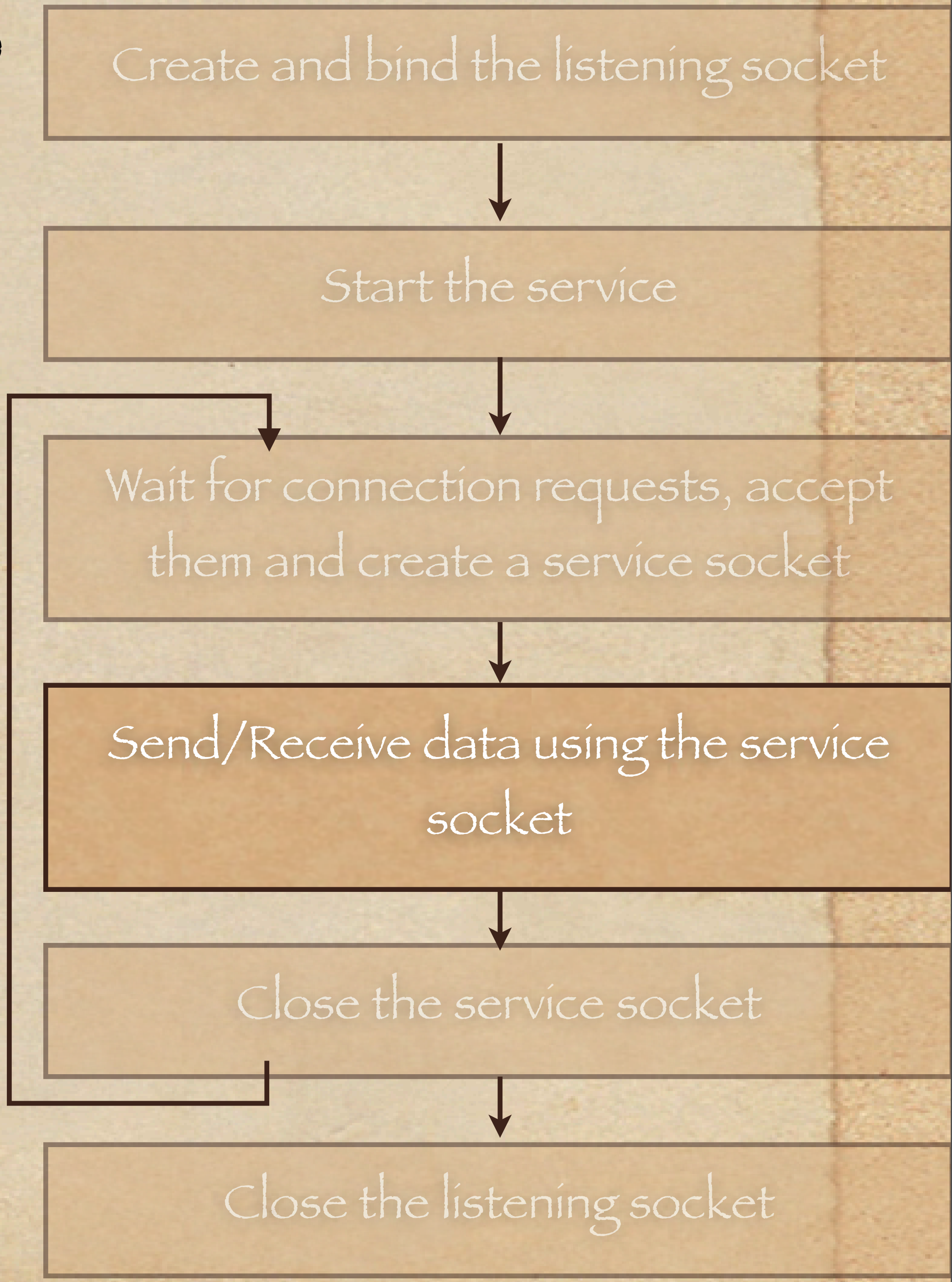

Create and bind the listening socket
Start the service
Wait for connection requests, accept them and create a service socket
Send/Receive data using the service socket
Close the service socket
Close the listening socket

# Java: TCP Server

```
Socket serviceSock;  // service socket declaration

//  We call accept on the listening socket to wait for connection requests
// when a conn. request is received a new Socket object is created
// this object manages connection with the client which sent the request

while(true) {
      try {
            serviceSock = listeningSock.accept();
      }
      catch(IOException ioe) {
            System.out.println("Accept error: " + ioe.getMessage());
            break;
      }
      /* ... Manage connection with the client ... */
}
```

# Java: TCP Server

- Uses java input/output classes (java.io package)
- Methods : getOutputStream and getInputStream of Socket
- Return basic binary I/O streams that we will be able to encapsulate in more complex streams (BufferedReader, BufferedWriter, DataInputStream, DataOutputStream, ObjectInputStream, ObjectOutputStream...)


Create and bind the listening socket
Start the service
Wait for connection requests, accept them and create a service socket
Send/Receive data using the service socket
Close the service socket
Close the listening socket

# Java: TCP Server

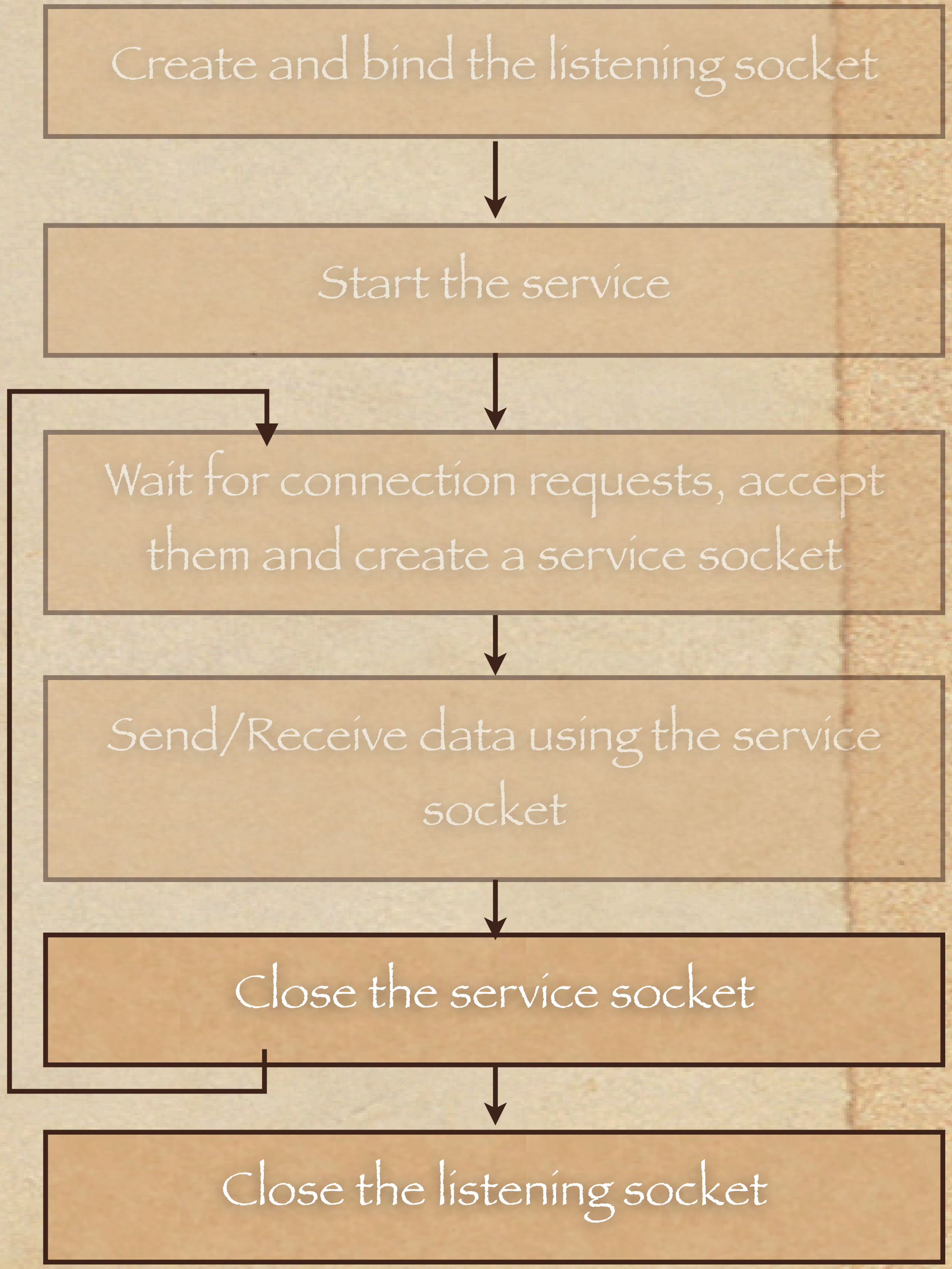
```
try{
    //  Creates a data input stream that will work on the socket basic input stream
    DataInputStream iStream = new DataInputStream(serviceSock.getInputStream());

    //  Reads a string an an integer. Those are received from the client.
    String helloString = iStream.readUTF();
    int three = iStream.readInt();
}
catch(IOException ioe) {
    System.out.println("Socket read error: " + ioe.getMessage());
}
```

# Java : TCP Serv

```
try{
    //  Creates a data output stream which will work on the socket’s basic output stream
    DataOutputStream oStream = new DataOutputStream(
                                                        serviceSock.getOutputStream());

    //  Writes a string and a float. The socket sends them to the client.
    oStream.writeUTF("Hello!");
    oStream.writeFloat(3.14f);
}

catch(IOException ioe) {
   System.out.println("Socket write error: " + ioe.getMessage());
}
```

# Java : TCP Serv

- close method of Socket and ServerSocket
  - code: sock.close() + try/catch IOException



Create and bind the listening socket

Start the service

Wait for connection requests, accept them and create a service socket

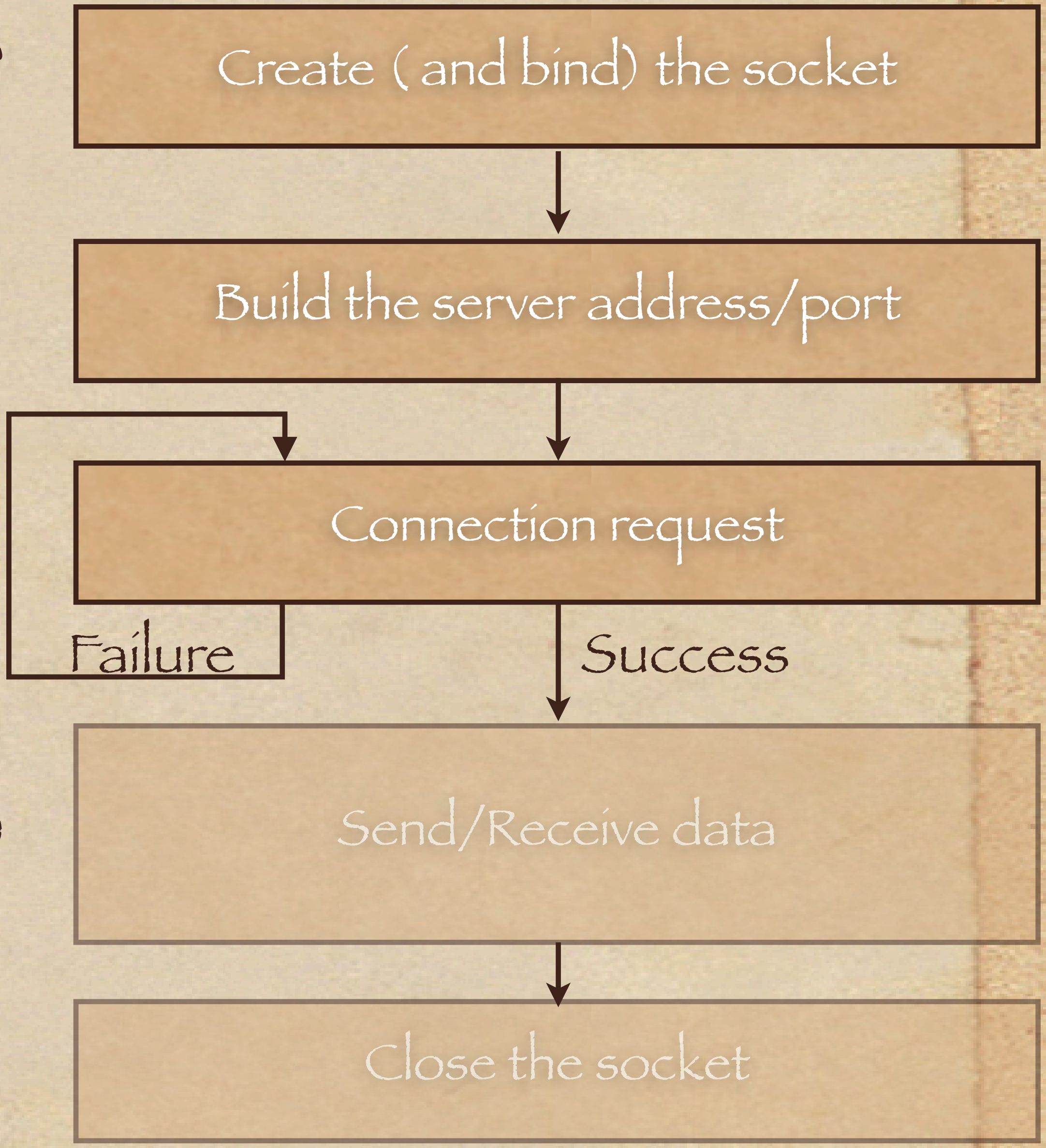
Send/Receive data using the service socket

Close the service socket

Close the listening socket

# Java : TCP Client

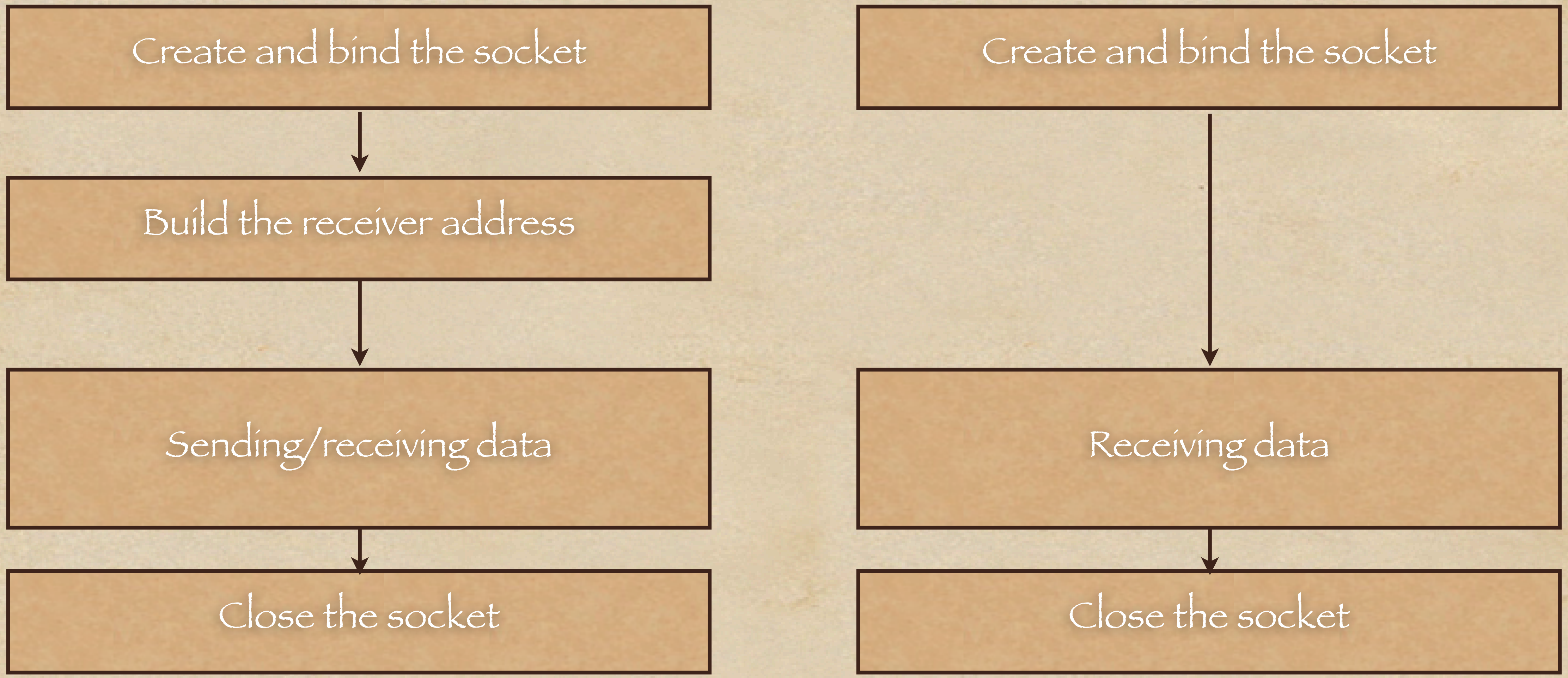
- Socket class
  - Use one of the constructors
  - Each create the socket, binds it and sends the connection request to the server
  - The most used one allows to give the name of the computer (or its IP @) and the application port



# Java : TCP Client

```
import java.net.Socket;
import java.io.IOException;
import java.io.DataInputStream;
import java.io.DataOutputStream;

Socket sock;                    //  Client socket declaration

//  Creates a socket and give the computer name and port for the server
try {
    sock = new Socket("marine.edu.ups-tlse.fr", 13214);
    // another solution:
    // sock = new Socket("10.5.4.1", 13214);
}
catch(IOException ioe) {
    System.out.println("Connection creation error: "
                                + ioe.getMessage());
    return;
}
```

# Sending/receiving with UDP



Create and bind the socket

Build the receiver address

Sending/receiving data

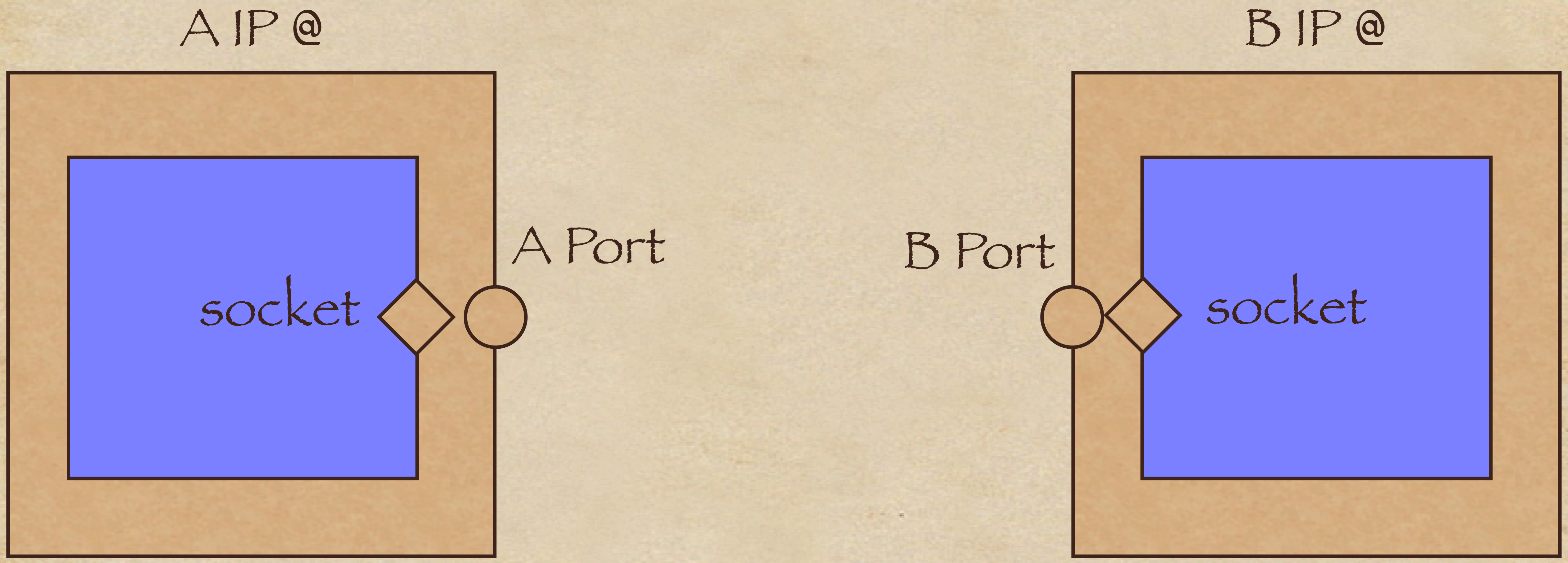
Close the socket

Create and bind the socket

Receiving data

Close the socket

# Create + bind socket



local @ = A IP @ ou Any

local port = A port

remote @ = Any

remote port = 0

proto = UDP

local @ = B IP @ ou Any

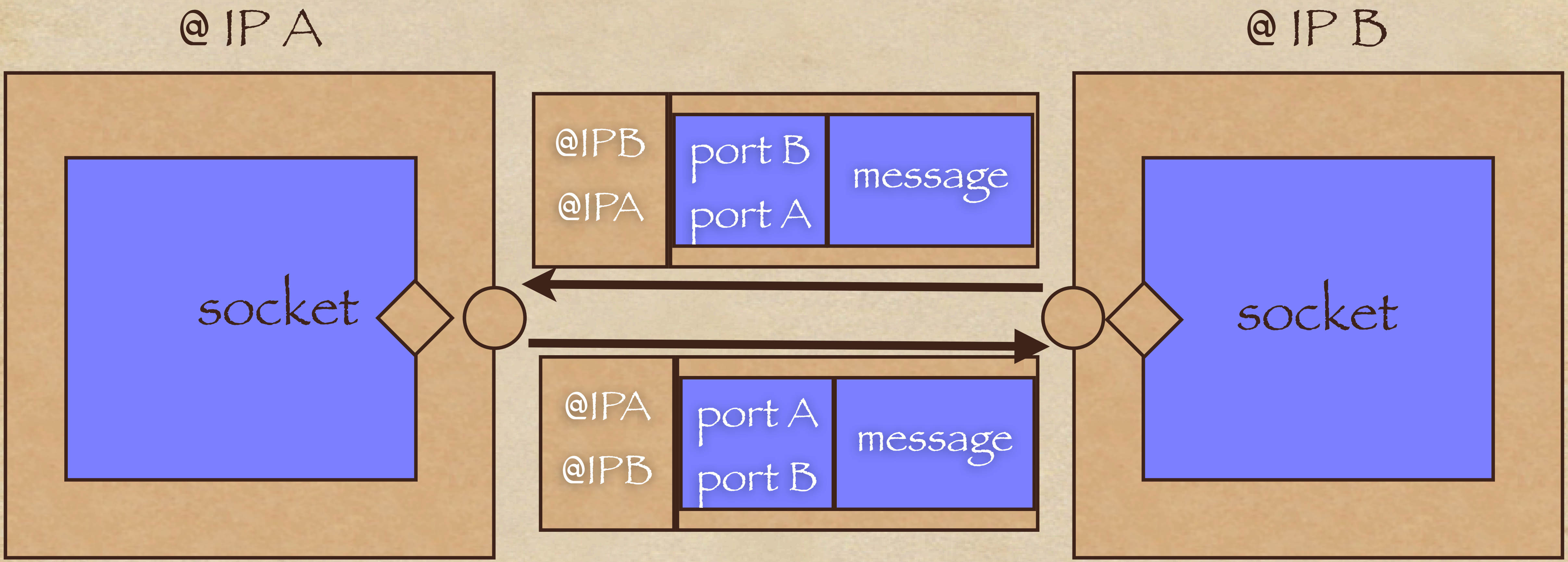
local port = B port

remote @ = Any

remote port = 0

proto = UDP

# Communication



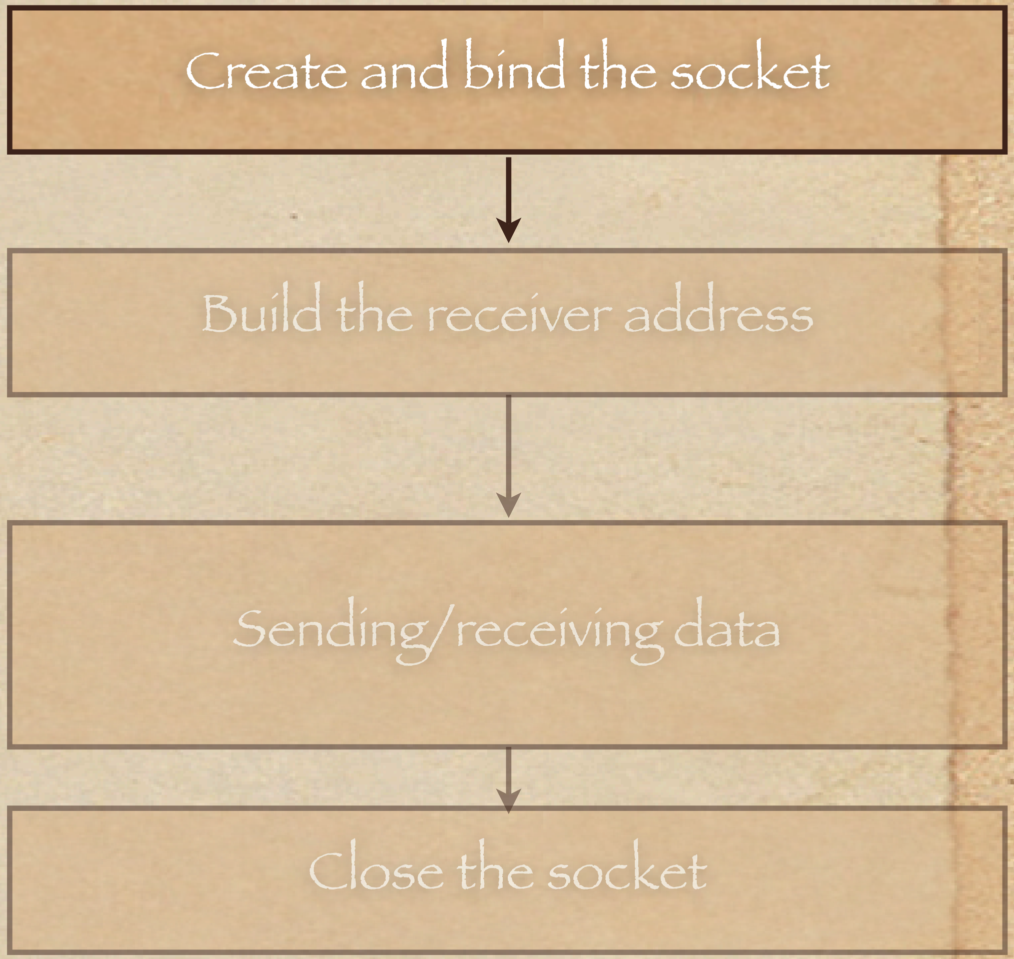
local @ = A IP @ ou Any
local port = A port
remote @ = Any
remote port = 0
proto = UDP

No connection
Usually you give the destination address and port each time you send something

local @ = B IP @ ou Any
local port = B port
remote @ = Any
remote port = 0
proto = UDP

# Sending/receiving with UDP

- DatagramSocket class
  - Creates a UDP socket and binds it to a local port (and IP @)
  - Constructors:
    - default constructor (OS choses the port)
    - choice of port
    - choice of port and local IP address (if the computer has several IP addresses)

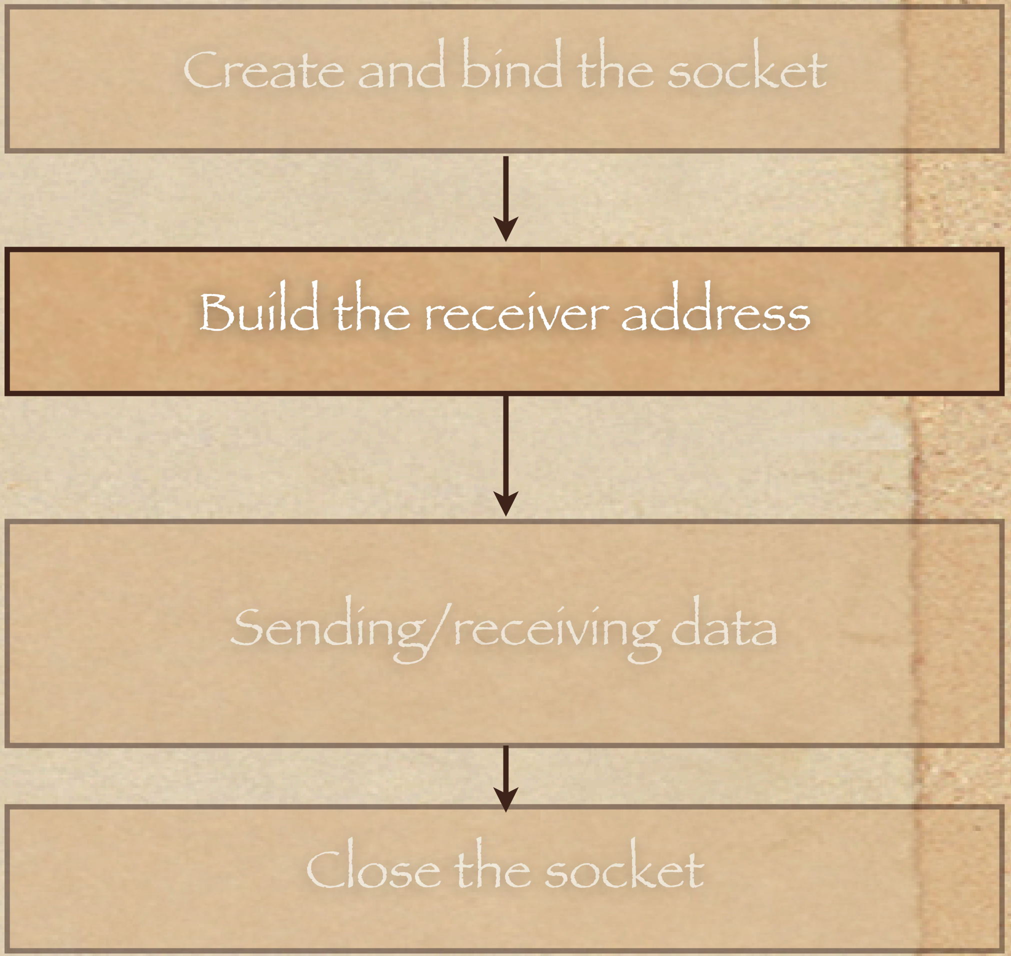


# Sending/receiving with UDP

```
import java.net.DatagramSocket;
import java.io.IOException;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.DataInputStream;
import java.io.DataOutputStream;

DatagramSocket sock;    // Datagram socket declaration

try {
    sock = new DatagramSocket(13214);     // Binds to UDP 13214 port
}

catch(IOException ioe) {
    System.out.println("Socket creation error: " + ioe.getMessage());
    return;
}
```

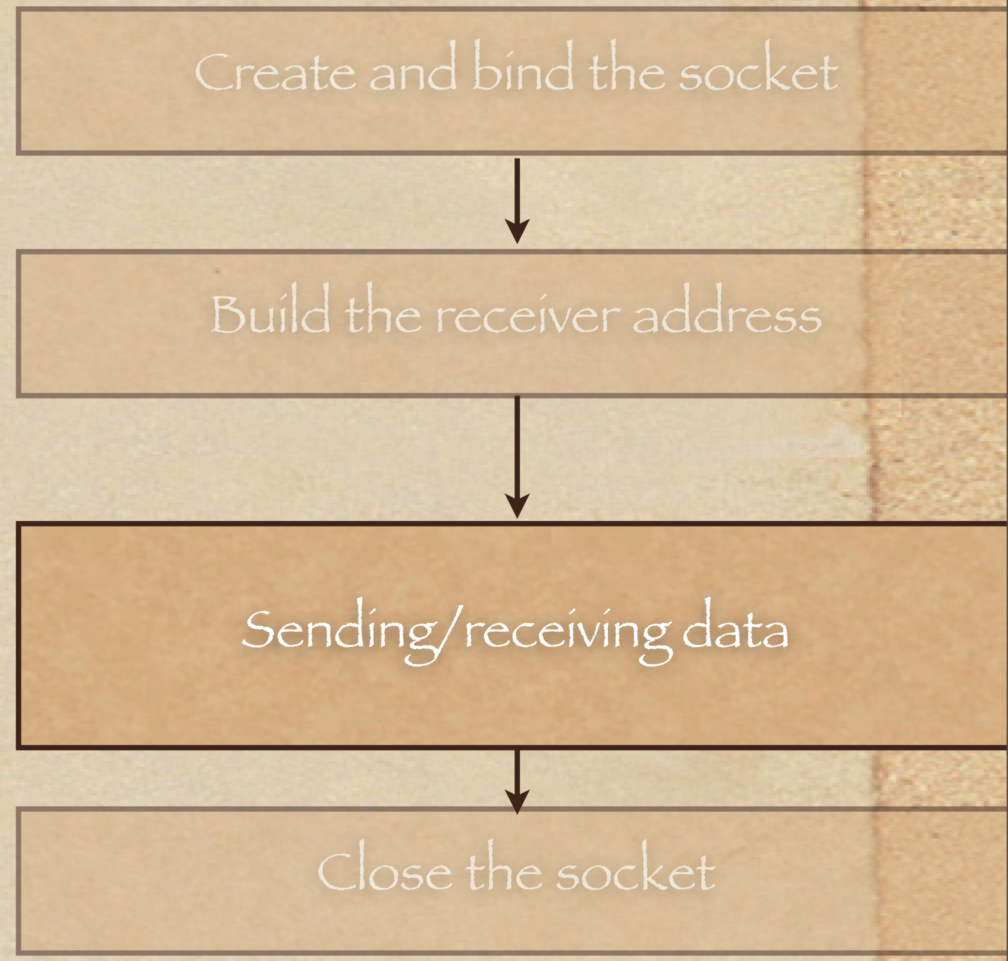
# Sending/receiving with UDP

- InetAddress class
  - Manages IP r4 and r6 addresses (using two derived classes Inet4Address and Inet6Address)
  - 3 static methods :
    - InetAddress getByName(String s) : name resolving or IP address parsing
    - InetAddress getLocalHost() : local IP address
    - InetAddress[] getAllByName(String name) : gives all addresses associated to a name



# Sending/receiving with UDP

- DatagramPacket class
  - Manages a UDP datagram that will be sent or received
  - 2 main constuctors
    - Sending: 4 parameters (data, length, IP@, port)
    - Receiving: 2 parameters (buffer, length of buffer)
  - Several get/set Methods for : data, length (of actually received data), remote IP @ and port, local IP @ and port (+ and offset)
- Methods send/receive of DatagramSocket



# Sending with UDP

```
try{
    //  Prepare IP @ and port
    InetAddress destAddr = InetAddress.getByName("10.25.43.9");
    int destPort = 13214;
    //  You can use a ByteArrayOutputStream to format data
    ByteArrayOutputStream boStream = new ByteArrayOutputStream();
    DataOutputStream oStream = new DataOutputStream(boStream);
    oStream.writeUTF("Hello!");                    // Write some data on the stream
    oStream.writeInt(3);
    byte[] dataBytes = boStream.getByteArray(); // Convert the stream as a byte array
    DatagramPacket dgram =                         // Create a DatagramPacket
        new DatagramPacket(dataBytes, dataBytes.length, destAddr, destPort)
    sock.send(dgram);
}
catch(IOException ioe) {
    System.out.println("Socket send error: " + ioe.getMessage());
}
```

# Receiving with UDP

```
try{
      // Build structures to hold incoming information
      byte[] buffer = new byte[255];
      DatagramPacket dgram = new DatagramPacket(buffer, buffer.length);

       // Receive the incoming datagram
       sock.receive(dgram);                              // Sender information available in
                                                         // dgram.getAddress() and dgram.getPort()

      //  Unpack the Datagram
      ByteArrayInputStream biStream = new ByteArrayOutputStream();
      DataInputStream iStream = new DataInputStream(biStream);
      String helloString = iStream.readUTF();
      int three = iStream.readInt();
}
catch(IOException ioe) {
      System.out.println("Socket receive error: " + ioe.getMessage());
}
```

# UDP Broadcasting

- Almost identical to UDP/IP unicast
    - But you must use a broadcast address as the destination address of the datagram.

```
InetAddress destAddr = InetAddress.getByName(“255.255.255.255”)
```

- 
    - Note: datagram sockets can receive both unicast and broadcast datagrams

# UDP Multicasting

- In order to multicast you should:
  - Use the MulticastSocket instead of the DatagramSocket class (MS extends DS)
  - Give a multicast IP address as the destination address of your datagram
    - Example (IPr4 address) 225.0.0.1

```
InetAddress destAddr = InetAddress.getByName(“225.0.0.1”)
```

- You can also choose the TTL to limit the multicast outreach

```
sock.setTimeToLive(1);
// then you send your datagram as before
sock.send(dgram);
```

# UDP multicast reception

- In order to receive you must subscribe to the IP multicast address like this:

```
sock.joinGroup(InetAddress.getByName(“225.0.0.1”));
```

- You can unsubscribe later on using:

```
sock.leaveGroup(InetAddress.getByName(“225.0.0.1”));
```

# Conclusion

- Network programming with sockets is easy
- But beware of asynchronism
  - Receiving is always a blocking call
  - For TCP, waiting for a connection and even sometimes sending are blocking calls
- Your solutions: threads or NIO select operations