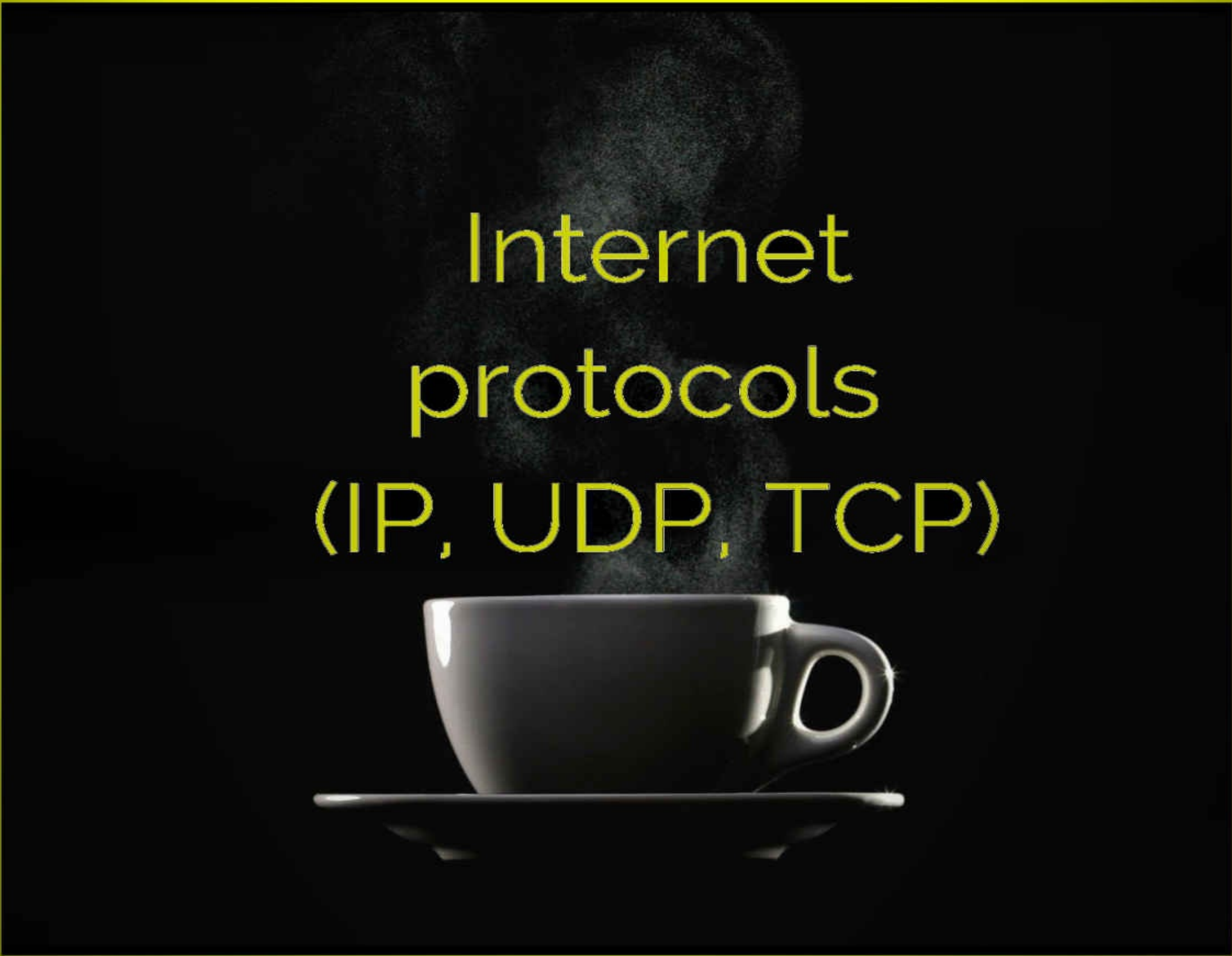
Network Programming in Java



Internet protocols (IP, UDP, TCP)

Alan MOUHLI

Network Programming in Java Internet protocols (IP, UDP, TCP)

By Alan MOUHLI

Content

1. Introduction to Java network API
2. Discovering package java.net
3. essential components
4. compatible locations
5. OSI model: it is possible to do (or not) with the standard Java API
6. internal functioning of the Network API
7. Competition
8. multi-threaded approach to blocking
9. non-blocking approach Event
10. Some generalities about IPv4 and IPv6
11. Two IP (Internet Protocol) co-existing on the net
12. IPv4
13. IPv6
14. IPv4 address blocks
15. RFC 2373 IPv6 addresses
16. Some address ranges IPv6 unicast special
17. IP addresses with the API java.net
18. java.net.InetAddress: an IP address in Java
19. What is this mysterious address?
20. Converting an octal IPv4 address text
21. lazy method
22. more laborious method
23. Converting an IPv4 address text octal
24. lazy method
25. more laborious method
26. an IP address connectivity test
27. How it works?
28. If negative responses
29. Name Resolution
30. Domain Name System (DNS)
31. DNS records
32. Name Resolution
33. Using InetAddress
34. IP address from a name
35. Implementation of a DNS resolver in Java
36. Network interfaces in Java
37. java.net.NetworkInterface
38. Get a network interface
39. Viewing the tree of network interfaces
40. UDP datagrams for communication in Java

41. UDP
42. Introducing UDP (User Datagram Protocol)
43. UDP datagram format
44. UDP sockets in Java
45. sockets
46. On POSIX systems
47. Network interfaces
48. DatagramSocket
49. Socket UDP network: DatagramSocket class
50. UDP datagram: DatagramPacket class
51. Characters and bytes
52. Characters to bytes (and vice versa)
53. Send and receive packets with DatagramSocket
54. Send UDP packets containing text
55. Java example: send a packet containing the time an address range
56. Some tips for the exchange of packets
57. Mechanism of pseudo-connection
58. Some tests with Netcat
59. Java example: receive datagrams on a port with a deadline
60. multicast
61. UDP multicast
62. multicast protocols
63. java.net.MulticastSocket
64. TCP communication in Java
65. TCP
66. TCP (Transport Control Protocol)
67. Halfway between UDP and TCP: SCTP (Stream Control Transmission Protocol)
68. TCP Operation: states
69. TCP operation: sending and acknowledgment packets
70. TCP segment format
71. TCP sockets
72. TCP client socket
73. Boot a client socket
74. Waves of Socket
75. Socket configuration
76. Closing Socket
77. Example: A TCP client that sends a file
78. TCP socket server
79. Operating principle of a server socket
80. java.net.ServerSocket
81. Construction
82. connection acceptance
83. iterative model of using ServerSocket
84. Using a service socket
85. Example: a TCP server that receives files

Introduction

The Java language largely reproduces the syntax of the C ++ language, widely used by computer. Nevertheless, Java has been stripped of most subtle concepts of C ++ and simultaneously the most confusing, such as pointers and references, or multiple inheritance circumvented by implementing interfaces. The designers have favored the object-oriented approach so that in Java, everything is an object except primitive types (integer, floating point numbers, etc.).

Java enables the development of client-server applications. On the client side, applets are responsible for awareness of language. This is especially server-side Java has established itself in the middle of the company through the servlets, for the server applets, and more recently the Java Server Pages (JSP) which can substitute for PHP, ASP and ASP.NET.

Java has created an operating system (Java OS) in development environments (Eclipse / JDK) Virtual Machine (MSJVM (in) JRE) application cross-platform (JVM), a variation for mobile devices / embedded (J2ME), a graphical interface design library (AWT / Swing), heavy applications (Jude, Oracle SQL Worksheet, etc.), web technologies (servlets, applets) and declination for the company (J2EE). Java bytecode portability is provided by the Java virtual machine, and possibly included in a standard JRE libraries. This virtual machine can interpret the bytecode or compile it on the fly into machine language. Portability is dependent on the JVM porting quality on each OS.

Discovering package java.net

Essential components

IP addresses: Inet Address class
Network interfaces: Network Interface class
UDP sockets: Datagram Socket classes and Multicast Socket (UDP packet:

Datagram Packet class)
TCP sockets: Server Socket and Socket classes
application level connections: URL Connection class (to manage such an HTTP

connection)
Compatible locations
Open JDK (reference implementation of JDK) Android APIs

OSI model: it is possible to do (or not) with the standard Java API

Physical
Link: send raw Ethernet frames possible
Network: ICMP connectivity test possible, sending raw IP packets cannot (no

support raw sockets)
Transport: support UDP, TCP, SCTP (unofficial)
Session, presentation: TLS support TCP (SSL Server Socket, SSL Socket) Application: DNS resolution, HTTP connections …

Note: the model of the last OSI layers is no longer relevant now (tendency to

create new protocols managing both transportation issues, session, presentation and application such as HTTP / 2.0)

Internal functioning of the Network API

Based on calls POSIX systems: getaddrinfo, socket, recv, send, accept …

native compiled code in conjunction with system calls for each supported platform (Windows, Linux, FreeBSD …)
Interfacing with native code with Java Native Interface (JNI)
The Java API is a wrapper for calls network systems (Java does not implement the UDP, TCP, DNS … is the core of the system that handles)

Competition

Two approaches to manage multiple network communications.
Approach blocking multi thread

A flood of communication = 1 thread
Operations blocking, I / O (hand given back to another thread while blocking) non-blocking approach Event

One thread
Using a selector indicating available communication waves

Operations I / O non-blocking: immediate return if nothing can be read or written
Approach established since Java 1.4 in java.nio

Some generalities about IPv4 and IPv6

Two IP (Internet Protocol) co-existing on the net

IPv4
Historic first version described in RFC 791 in 1981
size of address space 2 ^ {32}

IPv6
New version introduced in 1996 to solve the shortage of IPv4 addresses size of address space 2 ^ {128}
Version in adoption

IPv4 and IPv6 addresses are allocated by the Internet Assigned Numbers Authority (IANA).
IPv4 address blocks

5 classes of publicly addressable historical addresses (from the 4 bits of the first byte) with some exceptional beaches:
0 …: Class A addresses with network $$ 1 $ byte (from 0.0.0.0 to 127.255.255.255 …)
joker address 0.0.0.0: for the socket configuration

10.0.0.0/8 private beach (2 ^ {24} addressable machines)
127.0.0.0/8 beach localhost (local host)
… 10: Class B addresses with network of 2 bytes (128.0.0.0 … to 191,255,255,255)
address range 169.254.0.0/16 local link
172.16.0.0/12 private beach (2 ^ {20} addressable machines)
… 110: Class C addresses with network 3 bytes (192.0.0.0 to 223.255.255.255 to …)
192.168.0.0/16 private beach (2 ^ {16} addressable machines)
1110 …: D class addresses used for multicast (224.0.0.0 to 239.255.255.255 to …)
… 1111: Class E addresses reserved for future use (from 240.0.0.0 to 255.255.255.255 …)
address 255.255.255.255 reserved for broadcast on a local network
Note: in practice obsolete notions of class (supplanted by CIDR).

RFC 2373 IPv6 addresses

Coding on $ 128 $ bits expressed by two groups of hex bytes
simplification rules:
The zeros are optional group head
A zero sequence groups may be elliptical
The address must be enclosed in brackets in a URL
Example of IPv6 address (W3C website): 2001: 200: 1C0: 3601 :: 53: 1⇔ 2001

0200: 01C0: 3601: 0000: 0000: 0053: 0001
URL to access the web server: http: // [2001: 200: 1C0: 3601 :: 53: 1] Address Families:
Unicast (or anycast): 64-bit network (network, subnet), 64-bit interface The interface bits identify a machine on the LAN (@MAC, DHCPv6 …)

Multicast: 1 ff byte, second byte indicating properties on Multicast (permanent address, temporary scope of multicast).
Some address ranges IPv6 unicast special

null address (: :): joker address used for socket configuration :: 1 localhost address (local loop)
fe80 :: 10/10: link local address range
FC00 :: / 8: Private scope address range
:: Ffff: 0: 0/96: transitional IPv4 address range.

IP addresses with the API java.net

java.net.InetAddress: an IP address in Java

Two derived classes for IPv4 addresses (Inet4Address) and IPv6 (Inet6Address) No constructors to initialize an IP address but static methods:
1. Since a text form at: InetAddress.getByName (String addr)
2. Since an address in octal form: InetAddress.getByAddress (byte [] addr) 3. If one wishes to obtain the address localhost: InetAddress.getLocalHost ()

What is this mysterious address?

```
public static void d(String s) { System.out.println(s); }
public static void displayAddressInfo(byte[] addr) {
// D’abord on récupère un objet InetAddress
InetAddress ia = InetAddress.getByAddress(addr); // On affiche l’adresse sous forme textuelle
d(“Adresse texte: ” + ia.getHostAddress());
// On rappelle l’adresse octale
d(“Adresse octale: ” + Arrays.toString(ia.getAddress())); // Ensuite on affiche les informations associées d(“Adresse joker ? ” +
ia.isAnyLocalAddress()); d(“Adresse de lien local ? ” + ia.isLinkLocalAddress()); d(“Adresse de boucle locale ? ” + ia.isLoopbackAddress()); d(“Adresse
de réseau privé ? ” + ia.isSiteLocalAddress()); d(“Adresse de multicast ? ” + ia.isMulticastAddress()); if (ia.isMulticastAddress())
{
// Testons la portée du multicast d(“Portée globale ? ” + ia.isMCGlobal()); d(“Portée organisationnelle ? ” +
ia.isMCOrgLocal());
d(“Portée site ? ” + ia.isMCSiteLocal()); d(“Portée lien ? ” + ia.isMCLinkLocal()); d(“Portée noeud ? ” + ia.isMCNodeLocal()); }
}
```

Converting an octal IPv4 address text

Lazy method

```
public String convert(byte[] addr) throws UnknownHostException
{
return InetAddress.getByAddress(addr).getHostAddress();
}
```

More laborious method

```
public int toUnsigned(byte b) { return (int)b & 0xff; } public String convert(byte[] addr) throws UnknownHostException {
return String.format(“%d.%d.%d.%d”,
toUnsigned(addr[0]), toUnsigned(addr[1]), toUnsigned(addr[2]), toUnsigned(addr[3]));
}
```

Converting an IPv4 address text octal

Lazy method

```
public byte[] convert(String textAddr) throws UnknownHostException
{
return InetAddress.getByName(textAddr).getAddress(); }
\end{lstlisting}
```

More laborious method

```
public byte[] convert(String textAddr) {
String[] split = textAddr.split(“\.”); byte[] addr = new byte[split.length];
for (int i = 0; i < addr.length; i++)
addr[i] = (byte)Integer.parseInt(split[i]); return addr;
}
```

An IP address connectivity test

The method boolean isReachable (int timeout) throws IOException an instance of InetAddress to test the connectivity of the address.
How it works?

 We send either ICMP echo request (ping) or attempting to open a TCP connection on port 7 (Echo service) Host
 The host is considered reachable iff a response arrives before timeout ms

If negative responses

 Machine offline
 Machine not responding to pings and without TCP echo service Firewall filtering met on the routing path

Name Resolution

Domain Name System (DNS)
 Remember IPv4 addresses is difficult, IPv6 addresses torture → need for indirection

system linking easy to remember names to IP addresses

→ proposed solution: Domain Name System (DNS) with RFC 1035 hierarchical naming system overseen by the ICANN (ICANN)

→ each DNS zone delegated the administration of sub-names to other authorities

 Example delegation igm.univ-mlv.fr. Management of . ICANN .com management. by AFNIC .univ-mlv.fr management. by the University Paris-Est Marne-la-Vallée .igm.univ-mlv.fr management. by Gaspard-Monge Institute

 Converting a name to IP address resolution by DNS It interrogates the servers of the broader authorities to more specialized Cache servers allowing recursive resolution (ISP's DNS server)

 reverse lookup (IP @ $ \ rightarrow $ name) Query on the domain to an IPv4 address z.y.x.w.in-addr.arpa w.x.y.z

X.ip6.arpa interrogation on the field for an IPv6 address (X is the sequence of hexadecimal digits in reverse order separated by dots)

DNS records

 Key / value pairs for a resource on a DNS server:

1. A: IPv4 address
2. AAAA: IPv6 address
3. CNAME: canonical name (the requested resource is an alias)
4. MX: mail server managing incoming messages
5. SRV: server handling a specific service area
6. NS: Delegation server a DNS zone
7. SOA: information on the authority managing the DNS zone
8. PTR: pointer field to a canonical name used for reverse resolution
9. TXT: various text fields
10. Some fields may exist in multiple copies (multiple IP addresses for load balancing, mail servers for redundancy)
11. There are other specialized fields used by DNS Extensions (DNSSEC) or other related protocols (with SPF email authentication, SSH footprint with SSHFP …).

Name Resolution

 host name: displays the IP addresses related to name using the system server and DNS cache
 dig @dnsserver name: retrieves the DNS entries from the name server dnsserver
 POSIX system call} getaddrinfo () (netdb.h)

Using InetAddress

IP address from a name

The static method `InetAddress.getByName(String nom)` also works well for a text @IP a domain name: it returns solved uneInetAddress (locking resolution).

`InetAddress.getAllByName(String nom)` returns all IP addresses corresponding to the domain name given in the form of an array of InetAddress resolved.

`InetAddress.getByAddress(String nom, byte[] addr)` returns a InetAddress associating the name and address specified without referencing. UnknownHostException an exception is raised in case of problems (host name not resolvable).
useful method of InetAddress:

`String getHostName()`returns the host name (may also allow reverse lookups)
`String getCanonicalHostName()`provides the canonical name (CNAME)

Implementation of a DNS resolver in Java

```
public static void main(String[] args) throws
UnknownHostException
{
if (args.length < 1)
throw new IndexOutOfBoundsException("A domain must be provided");
String name = args[0];
InetAddress[] addrs = InetAddress.getAllByName(name); for (InetAddress addr: addrs)
System.out.println(addr.getHostAddress()); }
```

java.net.NetworkInterface

In existence since Java 1.4
Organization as a tree of interfaces
Interface properties:

local loopback interface `(isLooopback ())`
interface supports multicasting`(supportsMulticast ())` virtual interface `(IsVirtual ()`), e.g., VLAN interface tunnel … link interface point to point`(isPointToPoint ())`

Get a network interface

Identifying a network interface:
by an interface name: `getName ()`, e.g. eth0, wlan0, ppp0 … by physical addresses (Ethernet MAC address): `getInterfaceAddresses ()`
by logical address (IP address):`getInetAddresses ()`
static interface collecting:
by interface name: `NetworkInterface.getByName (String)` IP Address: `NetworkInterface.getByInetAddress (InetAddress)`

Viewing the tree of network interfaces

```
public static void displayTree(NetworkInterface interface, PrintStream out)
{
public static void displayTree(NetworkInterface iface, PrintStream out) throws SocketException
{
Enumeration<NetworkInterface> ifaces = (iface ==
null)?NetworkInterface.getNetworkInterfaces() // We get the root interfaces
:iface.getSubInterfaces(); // or the child interfaces
if (iface != null) {
out.println(iface.getName() + ":");
Enumeration<InetAddress> addresses = iface.getInetAddresses();
while (addresses.hasMoreElements()) out.print(addresses.nextElement() + ",");
out.print("(");
}
while (ifaces.hasMoreElements())
displayTree(ifaces.nextElement(), out); if (iface != null) out.println(“)”); }
}
```

UDP

Introducing UDP (User Datagram Protocol)
 Sends independent IP packets

 The source and destination of a packet is defined by an IP address and a port number (socket)
 Simple protocol implementation (low latency) but no guarantee on the receipt of the package (as well as the delivery time): no flow control and congestion
 Vulnerable to spoofing source IP @
 Supports multicast and broadcast unlike TCP
 Used for applications requiring low latency with tolerance to packet loss: Sending small informational messages on a local network
AV streaming
Voice over IP
network games
…

UDP datagram format

 Source port 16-bit
 Port 16-bit destination
 Length of 16 bits (65535 bytes max)
 16-bit checksum (calculated on the header including the end of the IP header

and data)
 Field data

UDP sockets in Java

Sockets

 IP stacks implant a socket mechanism (IP address association and communication port opened on a machine)
 Making a connection by sending a packet to a socket to another

On POSIX systems
 `int socket(int domain, int type, int protocol)` to open a socket

 `int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen)` to bind a socket to an address and port
 `int send(int socket, const void *buf, size\_t len, int flags)` to send a datagram
 `ssize_t recv(int s, void *buf, size\_t len, int flags)` to extract a datagram receiving queue
 `man 7 socket`, find out more

Network interfaces
 For each active network interface of a machine: at least one @IP

 A socket must bind to one or more network interfaces (to focus only on the data from this interface):

Choosing a network interface by its @IP (e.g. lo 127.0.0.1 for IPv4)
Choice of all network interfaces with the wildcard address (no address)

DatagramSocket

Socket UDP network: class`DatagramSocket`
Association of an address and a port to receive or send UDP datagrams Builders available:

`DatagramSocket ():` create a socket on the wildcard address on one of the available ports
`DatagramSocket (int port):` used to select the port, use of the wildcard address
`DatagramSocket (int port, InetAddress addr):` to focus on a specific address of the machine (not listening on all interfaces)

These constructors can throw SocketException if the socket cannot be allocated. After use, the socket can be closed with close () to free system resources associated.
UDP datagram: class`DatagramPacket`

This class allows to handle UDP datagrams. They can be created by manufacturers using a byte array `(byte [])` or to receive data or to send.

To receive data, the byte press table should be of sufficient size to receive data (if the packet is truncated!).
An instance can be used to send multiple packets, or to receive. Ability to change:

The destination address with `setAddress (InetAddress)` and the harbor with `setPort (int).`
The data buffer with `setData (byte [] buffer [offset int, int length]).`

The data can be accessed with `byte [] getData ()` and`int getLength ()` (size of the received packet).
Characters and bytes
Data is exchanged over the network byte sequence To send a string from A to B:

Encoding the string into bytes in A Transmitting a packet with the byte array package receiving with byte array B Decoding byte chain B

Characters to bytes (and vice versa)

byte character-conversion agreement implanted class Charset
Some character sets ASCII, iso-8859-15, UTF-8, UTF-16LE, UTF-16BE, … A character can be encoded by one or more bytes
The string to bytes:`byte [] String.getBytes (String charset)` Bytes to`: new String (byte [] tableauOctets, departing int, int length, String charset)`
If the character set is not specified: use the default game `(static Charset Charset.defaultCharset ())`
For all available games`: static SortedMap <String, Charset> Charset.availableCharsets ()`

Send and receive packets with `DatagramSocket`

1. Method`send (DatagramPacket)` to send a package: package added in the transmit queue
2. Method`receive (DatagramPacket)` to receive a packet reception queue Blocking call until receiving a packet
`setSoTimeout (int timeoutInMillis)` sets a packet reception deadline (beyond, lifting a`java.net.SocketTimeoutException`)
1. These methods can throw a `java.io.IOException` that must be managed.

2. The size of the queue is available and configurable with:
`int get {Send, Receive BufferSize} ()`
and`set {void Send, Receive} `BufferSize ().`

Java example: send a packet containing the time an address range

```
import java.util.*; import java.io.*; import java.net.*;
public class TimeSender implements AutoCloseable {
private final DatagramSocket socket; private final DatagramPacket packet;
public TimeSender(int port) throws SocketException {
this.socket = new DatagramSocket();
this.packet = new DatagramPacket(new byte[0], 0); // Packet with empty array
this.packet.setPort(port);
}
public void sendTimePacket(InetAddress a) throws IOException
{
// We reuse the same datagram packet but modify its content and addressee
byte[] timeData = new
Date().toString().getBytes();
packet.setData(timeData);
packet.setAddress(a);
// this.packet.setPort is useless since the port does not change
socket.send(packet); // May throw an IOException }
public static InetAddress getSuccessor(InetAddress a) {
// Exercise: write the implementation of getSuccessor(InetAddress) returning the successor of an IP address.
// Example: the successor of 10.1.2.3 is 10.1.2.4 (for last byte 0 and 255 are forbidden)
}
public void close() throws IOException
{
socket.close();
}
public static void main(String[] args) throws IOException {
if (args.length < 3) throw new
IllegalArgumentException("Not enough arguments"); InetAddress firstA =
InetAddress.getByName(args[0]); // Start address InetAddress lastA =
InetAddress.getByName(args[1]); // Stop address int port = Integer.parseInt(args[2]); try (TimeSender ts = new TimeSender(port)) {
for (InetAddress currentA = firstA; !
currentA.equals(lastA); currentA = getSuccessor(currentA)) ts.sendTimePacket(currentA); }
}
}
```

Some tips for the exchange of packets

Recycle DatagramPacket already created (by changing the hosted data)

Do not create unnecessary DatagramSocket, release them when no longer needed
Always check that the receipt size is sufficient

classic mistake: send k byte packet and receive on the same package without changing table or size; it is then limited to k bytes for reception

Do not forget to use the size of the data received: the end of the table is not exploitable
Check the IP address / transmitter port of a received packet (even if it makes only a small safety)

Mechanism of pseudo-connection

Possibility of pseudo-connect two UDP sockets with void
`DatagramSocket.connect (InetAddress addr, int port):` automatic filtering of packets sent and received

Checking the connection with`isConnected boolean ()`
Disconnection with `void disconnect ()`

Some tests with Netcat

Netcat: simple application to communicate with UDP (and TCP)

We run the server on port UDP N listening on all interfaces (wildcard address): `nc -l -u N`
It runs a client connecting to the port N of the machine:
`n -u N MachineName`
The client sends each line of standard input in a UDP datagram: the server receives and displays it on its standard output
server datagram sending to the customer also possible

Java example: receive datagrams on a port with a deadline

```
public class PacketReceiver {
public static final int BUFFER_LENGTH = 1024;
public static void main(String[] args) throws IOException {
if (args.length < 3) throw new
IllegalArgumentException("Not enough arguments");
InetAddress a = InetAddress.getByName(args[0]); int port = Integer.parseInt(args[1]); int timeout = Integer.parseInt(args[2]); DatagramSocket s = new
DatagramSocket(port, a); try {
s.setSoTimeout(timeout);
DatagramPacket p = new DatagramPacket(new byte[BUFFER_LENGTH], BUFFER_LENGTH);
for (boolean go = true; go; )
{
try { s.receive(p);
} catch (SocketTimeoutException e) {
}
if (go) {
System.out.println("Has received data from " + p.getAddress() + ":" + p.getPort()); System.out.println(new String(p.getData(), 0, p.getLength(), "UTF-
8"));
p.setLength(BUFFER_LENGTH); // Reset the length to the size of the array
}
}
} finally
{
s.close(); // Don't forget to liberate the socket resource
} }
}
System.err.println("Sorry, no packet received before the timeout of " + timeout + " ms");
go = false;
```

Multicast

Multicast UDP

Some Background on multicast addresses:
224.0.0.0/4 IPv4 (prefix 1110)
FF00 :: 0/8 IPv6 (\ rfc {4291})
Some predefined use addresses (host group, routers, uPnP …)

principle:
Guests subscribe / unsubscribe to a multicast address A. A frame sent to A is addressed to all members. **Advantages:**
In theory, pools the packets exchanged on network segments unlike unicast. Avoids sending packets to all hosts as broadcast (broadcast incompatible with an extensive network).
multicast protocols

IGMP (Internet Group Management Protocol) to report to the local network gateway members of each group. Some Layer 2 switches use.
On the Internet, a multicast distribution tree is built with PIM (Protocol Independent Multicast) in Sparse Mode, Dense Mode, Mixed Sparse-Dense Mode or Source Specific.

`java.net.MulticastSocket`

improved version of `DatagramSocket` with multicast support classic sending a datagram to send`(DatagramPacket)` (an IP unicast or multicast)

multicast datagrams reception ducted by adding a prior subscription to the multicast group
`void joinGroup (InetAddress ia)` to join a group
`void LeaveGroup (InetAddress ia)` to leave a group Some useful parameters for multicast

`void setInterface (InetAddress ia)` or `void setNetworkInterface (NetworkInterface netif)` to secure the interface used for multicast communication

`setTimeToLive void (int ttl)` to indicate the scope of multicast packets sent

TCP

TCP (Transport Control Protocol)

Unlike UDP, TCP (\ rfc {793}) has the following characteristics:
 Bidirectional unicast transmission in online mode (stream) packet retransmission mechanism not paid by the addressee **Consequences:**

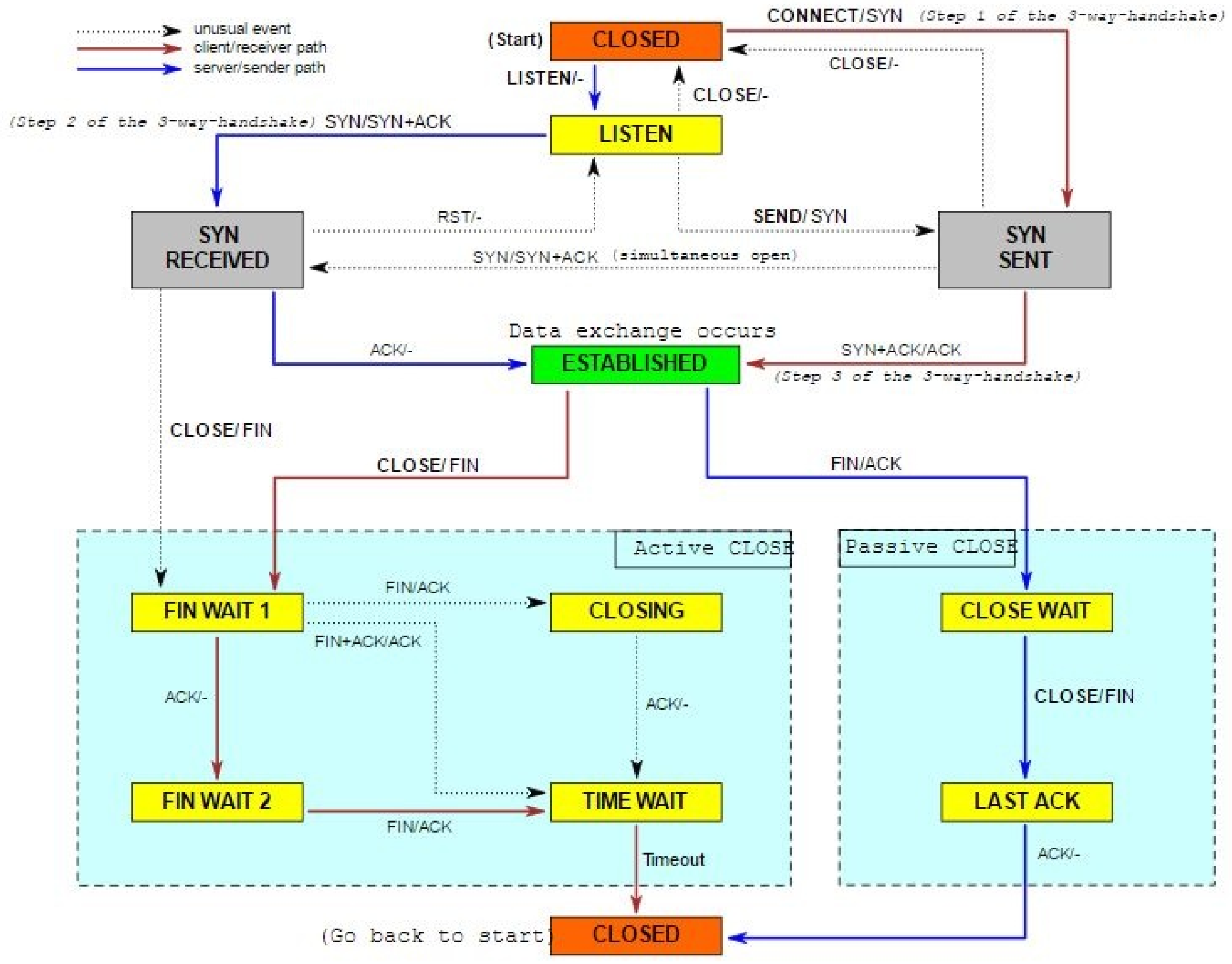
 heavier implantation with management of corresponding states Poorly adapted to real time (receive shift due to lost packets) Low performance on wireless networks (rate reduction during transient

problems)

Halfway between UDP and TCP: SCTP (Stream Control Transmission Protocol)

 Messaging (like UDP) but with preservation order
 Reliable routing with congestion management (such as TCP)
 Management of multi-stream
 multicast capabilities (such as UDP)
 Established standard with recent Linux and BSD kernels
 integrated Java implementation since JDK 1.7: Packaging `com.sun.nio.sctp`

TCP Operation: states



TCP operation: sending and acknowledgment packets

A sending window n segments is used: the peer should send at least one acknowledgment every n segments received
Acknowledgement sent to segment i: i until all segments have been received
Count-down for each sent segment: the expiration, the packet is returned
If too many packets lost: probable congestion and reducing the send window
Problem: a lost packet is not necessarily linked to the congestion links (without son networks)

TCP segment format

Source port (16 bits)
Port destination (16-bit)
Sequence Number (32 bits)
Acknowledgement number (32 bits) of the last received segment Size of the header in 32 bit words (32 bit)
various flags (28 bits): ECN (congestion), URG (urgent data), ACK

(acknowledgment), PSH (push), RST (reset), SYN (connection establishment), FIN (termination of the connection)

Window size (16 bits) required in bytes (beyond an acknowledgment should be sent)
checksum (16 bits) on the end of the IP header, the TCP header and data
urgent data pointer (16 bits)
Options (supplemented with a filler for a 32-bit multiple header length)

Sockets TCP

TCP client socket
Boot a client socket
In Java, TCP socket represented by an instance of `java.net.Socket` We want to listen on the wildcard address and let the system choose a free port of:

Socket `(InetAddress addr, int port):` connects to the remote socket addr: port
Socket `(String host, int port):` equivalent to socket
`(InetAddress.getByName (host) port)`

the address of the interface is desired and specify the local port attachment: Socket
`(InetAddress addr, int port, InetAddress localaddr, int localPort)`
We wish not to connect the socket for now:

`Socket ()`
Then, we must bind the socket with locally bind (its`SocketAddress)` and then connect (with a time limit if timeout ≠ 0) to the remote socket to

connect (`SocketAddress` remote, `int timeoutInMillis`) Waves of Socket
To communicate among TCP sockets (local and remote), using binary streams.

`InputStream getInputStream()` stream which read binary data from the corresponding
`OutputStream getOutputStream()`stream where write data to the corresponding

Some notes (and reminders)

The I / O operations on these waves are blocking.
IOException should be managed.
The scriptures do not necessarily cause the immediate sending of TCP packet

(can buffering)
Flush () is used to request the immediate dispatch (use TCP No Delay)) Close firm afloat the underlying socket

Socket configuration

`void setKeepAlive (boolean keepAlive)` regularly sends packets to keep alive the connection
`void setOOBInline (boolean oobinline)` integrates urgent data OutOf-Band in the incoming stream (otherwise they are ignored)

`sendUrgentData void (int data`): Sends an urgent byte (8 bits of low weight)
`setReceiveBufferSize void (int size):` Still the size of reception buffer
`setSendBufferSize void (int size): fixed size transmission buffer`
`void setSoLinger (boolean on, int lingerTimeInSeconds`): Sets up a closing blocking confirmation of receipt of the last data sent or timeout reached
`setSoTimeout void (int timeoutInMillis`) sets a time limit for the readings `(SocketTimeoutException` exercisable)
`void setTcpNoDelay (boolean on):` to disable the Nagle algorithm (useful for interactive protocols: immediate sending small segments)

`void setTrafficClass (tc int`): Sets the byte Type-Of-Service for packages (useful for QoS)
`void setReuseAddress (boolean on):` allows to reuse a local socket address recently closed (TIME_WAIT state); must be called before bind.

Note: existence of getters for these methods (for the current configuration)
Closing Socket
`shutdownInput ():` allows you to ignore all data subsequently entering the socket $ \ Longrightarrow $ the remote socket is not informed and can continue to send data
`shutdownOutput ():` closes the outgoing stream of the socket `close ():` frees resources associated with the socket
Example: A TCP client that sends a file

```
public static void main(String[] args)
{
if (args.length < 3) throw new
IllegalArgumentException("Not enough arguments");
try ( Socket s = new Socket(args[0],
Integer.parseInt(args[1]));
InputStream is = new FileInputStream(args[2]); OutputStream os = new
BufferedOutputStream(s.getOutputStream()) )
{
s.shutdownInput(); // We are not interested in the answer of the server
transfer(is, os);
os.flush();
}
}
```

TCP socket server

Operating principle of a server socket
The server socket listens for incoming packets
Upon receiving a SYN packet:
return a SYN / ACK to the client socket to continue the 3-way handshake return a RST packet to refuse the connection (if too many connections pending for example)
The client socket confirms opening by an ACK packet to the server socket The socket server created a service socket to communicate with the client socket and adds in a queue to be retrieved by the server program (accept)
java.net.ServerSocket
Construction

`ServerSocket (int port, int backlog, InetAddress addr):` set up a TCP socket server listening on the specified local IP, the port indicated with the backlog (maximum number of queued connections) given (if not indicated use of addr the wildcard address, if not stated backlog use of a default)

`ServerSocket ():` created a socket server locally unattached subsequent call to bind `(SocketAddress saddr, int backlog)` necessary
connection acceptance `Socket accept ()` returns the oldest socket service socket on hold. method is set arbitrarily `bloquante.Timeout` with `setSoTimeout (int timeoutInMillis)`
Iterative model of using ServerSocket

We construct a ServerSocket
Recovering a socket connected with the next customer waiting with accept () One dialog using the socket connected to the client
The communication ended with the customer, we close the socket The process returns to step 2 to get the next client
Limitation of the model: 1 single client processed at a time (if multiple threads required)

Using a service socket

This is an instance of `java.net.Socket`: same use as client-side `getInputStream()`for a stream to receive client data `getOutputStream()`for a stream to send data to the client possible use of configuration setters

How to know the details of the customer?

`getInetAddress ()` to obtain the IP address `getPort ()` to get the port

Example: a TCP server that receives files

```
public class FileReceiverServer
{
// After one minute with no new-connection, the server stops
public static final int ACCEPT_TIMEOUT = 60000;
private final File directory; private final int port;
public FileReceiverServer(File directory, int port) {
this.directory = directory; this.port = port;
}
public void launch() throws IOException {
try (ServerSocket ss = new ServerSocket(port)) {
ss.setSoTimeout(ACCEPT_TIMEOUT); while (true)
{
try (Socket s = ss.accept(); OutputStream os = new BufferedOutputStream(new FileOutputStream(
new File(directory, s.getInetAddress() + "_" + s.getPort())) )
{
// Save the data in a file named with the address and port of the client
transfer(s.getInputStream(), os);
} catch (SocketTimeoutException e) {
System.err.println("The server will shutdown because no new connections are available");
break;
} catch (IOException e) {
e.printStackTrace();
continue; // Treat the next incoming client
}
} }
}
public static void main(String[] args) {
new FileReceiverServer( new File(args[0]),
Integer.parseInt(args[1])).launch(); }
}
```