Socket-based Distributed Systems

Fabienne Boyer, fabienne.boyer@imag.fr

Adaptation du cours d’Olivier Gruber, LIG
Outline

- Introduction to sockets
- Point-to-point communication with TCP sockets
- Point-to-point communication with UDP sockets
- Group communication with sockets
- Client-server programming with sockets
Internet Protocol (IP)

- IP (Internet Protocol)
  - The Internet protocol is not the Web
  - Corresponds to the network layer of the OSI model
  - Addressing, routing and transport of data packets

- IP addresses are names
  - 4 bytes (IPv4), naming a host machine (e.g. 192.168.2.100)
  - IP addresses are location dependent
DNS

- Host name to IP Address resolution
  - Using a network naming service
  - DNS (Domain Name System)

- Discussing DNS
  - A fairly complex world-wide distributed system
  - Allows name aliases (multiple names for an address)
  - And the reverse (multiple addresses for a name)
  - Organized as hierarchical zones across the world
    - A zone is managed by a DNS server
    - Replicated servers for high availability
IP addressing in Java

- **InetAddress class**
  - Represents an IP address
  - Either a 32- or 128-bit unsigned number used by IP
  - No public constructor
    - `static InetAddress getLocalHost()`
      - Returns the local host
    - `static InetAddress getByName(String hostname);`
      - Lookup a host machine by name

- **Some API elements**
  - `String getHostAddress()`
    - Returns the IP address in textual presentation.
    - E.g., 129.250.35.250
  - `String getHostName()`
    - Gets the host name for this IP address.
    - E.g. `hoff.e.efj-grenoble.fr`
Ports

- Ports are end points
  - For communication channels over IP
  - An IP address and a port names an end point

- Communication Channel
  - Between two ports, allocated to two processes

- Port numbers are managed by the operating system
  - Many important services have a standardized port
  - Example: port 513 for rlogin, 25 for telnet
  - Ports between 1 and 1023 are *well-known*
  - Ports between 1024 and 49151 are *registered*

- Port numbers are allocated on-demand to processes
  - Only one process may be allocated a port
Ports in Java

- **SocketAddress**
  - Abstract class for a socket address
  - Independent of any specific addressing protocol

- **InetSocketAddress**
  - Represents a socket address over IP
  - Essentially an IP address and a port
    - For example 129.250.35.250 and port 80
  - A hostname can be used instead of an IP address
    - It will be looked up using InetAddress.getByName(String)
Sockets

- A distributed programming model based on messages
- Two protocols over IP
  - TCP
    - Stream oriented
    - Lossless
    - Ordered
    - Connection-oriented
  - UDP
    - Packet based
    - Packets may be lost or reordered
    - Efficient
Sockets

- Typical applications
  - TCP
    - Do not accept loss or reordering
    - Examples:
      - Transferring files (ftp for instance)
      - Downloading web pages
  - UDP
    - Require high bandwidth, can accept loss or reordering
    - Examples:
      - Transmission of video/sound in real time
        - Out of sequence or incomplete frames are just dropped
      - Other more complex communication protocols
        - Such as totally-ordered multicast using Lamport's logical clocks
Outline

- Introduction to sockets
- **Point-to-point communication with TCP sockets**
- Point-to-point communication with UDP sockets
- Group communication
- Client-server programming based on sockets
TCP Sockets – Steps Involved

- **Server side**
  - Creation of the server process
    - Allocate a port
    - The port is granted by the operating system
    - Associate the port with a socket
  - Server waits for incoming connections
TCP Sockets – Steps Involved

- **Client side**
  - Creation of the client process
  - Request a connection to the remote port
    - A port is allocated
    - A socket is associated with the port
    - The connection is requested to the server

![Diagram of TCP Sockets](Illustration)

Client Machine

- "client process"
- "client socket"
- "client port"

Server Machine

- "server process"
- "server socket"
- "server port"

Connection request: 36243 -> 4320
TCP Sockets – Steps Involved

- **Server side**
  - Accept the connection from the client
    - A port is allocated
    - A socket is associated with the port
    - The connection is established between the client and the server
TCP Sockets – Steps Involved

Client 1
- client socket
- client port: 36243

Client 2
- client port: 56789

Server
- server port: 4320
- server socket

connection request
Java classes related to TCP sockets

- The java.net package provides
  - ServerSocket class
  - Socket class

- ServerSocket class
  - Represent a socket on a server
  - Accept incoming TCP connections

- Socket class
  - Represent the end points
  - Both on server and client sides
Example of Java sockets/TCP

```java
import java.net.*;

int port = 1234;

// Create a server socket associated with port 1234
ServerSocket server = new ServerSocket(port);

// End-less loop
while (true) {
    // Server waits for a connection
    Socket client = server.accept();
    // A client connected
    System.out.println("Client " + client.getInetAddress() + "connected.");
    // Server receives bytes from client
    ...
}
```
import java.net.*;

int port = 1234;

// Create a server socket associated with port
ServerSocket server = new ServerSocket(port);

// End-less loop
while (true) {
    // Server waits for a connection
    Socket client = server.accept();
    // A client connected
    System.out.println("Client " + client.getInetAddress() + " connected.");
    // Server receives bytes from client
    ...
}

Example of Java sockets/TCP

import java.net.*;

String serverHost = "goedel.imag.fr";
int serverPort = 1234;

// Client connects to server
Socket server;

server = new Socket(serverHost, serverPort);
// Client connected
System.out.println("Connected to " + server.getInetAddress());
// Client sends bytes to server
...

// Client receives bytes from server
...
Streams

- **Definition**
  - Streams are an abstraction for arbitrary data streams
  - Input streams used to receive (i.e. read) bytes
  - Output streams used to send (i.e. write) bytes

- **Examples**
  - Streams from/to a socket
  - Streams from/to a file
  - Streams from/to the console
Streams in Java

- The *java.io* package contains the following classes related to streams

  - *InputStream / OutputStream*
    abstract classes that represent streams of bytes

  - *DataInputStream / DataOutputStream*
    To manipulate streams of Java primary types

  - *ObjectInputStream / ObjectOutputStream*
    To manipulate streams of Java objects

  - *FileInputStream / FileOutputStream*
    To read data from a file or write data to a file

  - *FilterInputStream / FilterOutputStream*
    To transform data along the way
import java.io.*;

... // End-less loop
while (true) {
    // Server waits for a connection
    Socket client = server.accept();

    // A client connected
    System.out.println("Client " + client.getInetAddress() + " connected.");
    // Get the server’s output stream
    OutputStream os = client.getOutputStream();

    // Build data and transform it to bytes
    Date date = new Date();
    byte[] b = date.toString().getBytes();

    // Send the date to the client (write in output stream)
    os.write(b);
}
import java.io.*;

...  
// End-less loop  
while (true) {
    System.out.println("Waiting for client…");  
    // Server waits for a connection  
    Socket client = server.accept();  

    // Get the server’s output stream  
    OutputStream os = client.getOutputStream();  

    // Build data and transform it to bytes  
    Date date = new Date();  
    byte[] b = date.toString().getBytes();  

    // Write in output stream  
    os.write(b);  
}

import java.io.*;

...  
// Client connects to server  
Socket server = new Socket(serverHost, serverPort);  

// Get the client’s input stream  
InputStream is = server.getInputStream();  

// Read data from input stream  
byte[] b = new byte[100];  
int num = is.read(b);  

// Transform data from bytes to String  
String date = new String(b);  
System.out.println("Server said: " + date);
Java Sockets and Streams

// Get the server's output stream
OutputStream os =
client.getOutputStream();
Date date = new Date();
byte[] b = date.toString().getBytes();
os.write(b);

Correct?

// Get the client's input stream
InputStream is =
server.getInputStream();
byte[] b = new byte[100];
int num = is.read(b);
String date = new String(b);
Java Sockets and Streams

- Maybe, Maybe Not
  - Assumes:
    - Nothing is left in the input stream before reading
  - Just not always true
    - Some previous read might have left something...
  - Beware of IP data packets...

```java
// Get the server's output stream
OutputStream os = client.getOutputStream();
Date date = new Date();
byte[] b = date.toString().getBytes();
os.write(b);

// Get the client's input stream
InputStream is = server.getInputStream();
byte[] b = new byte[100];
int num = is.read(b);
String date = new String(b);
```
Java Sockets and Streams

- Maybe, Maybe Not
  - Assumes:
    - The entire string has been received when reading
  - Just not always true
  - When sending variable length data structures, **send the length**
  - How can we do that?
    - Length is an **integer**
    - We can send **bytes**

```java
// Get the server's output stream
OutputStream os = client.getOutputStream();
Date date = new Date();
byte[] b = date.toString().getBytes();
os.write(b);  Correct?

// Get the client's input stream
InputStream is = server.getInputStream();
byte[] b = new byte[100];
int num = is.read(b);
String date = new String(b);
```
Java Sockets and Streams

- **DataOutputStream**
  - Provides type-aware operations
  - Endianness proof!
  - But introduces an overhead

```java
// Get the server's output stream
OutputStream os;
DataOutputStream dos;
os = client.getOutputStream();
dos = new DataOutputStream(os);
Date date = new Date();
byte[] b =
date.toString().getBytes();
dos.writeInt(b.length);
dos.write(b);

// Get the client's input stream
InputStream is;
DataInputStream dis;
is = server.getInputStream();
dis = new DataInputStream(is);
int length = dis.readInt();
byte[] b = new byte[length];
int num = dis.read(b,0,length);
String date = new String(b);
```

Correct?
void write(OutputStream os, byte[] bytes, int offset, int len) {
    int offset = 0;
    int remaining = len;
    while (remaining!=0) {
        int n = os.write(bytes,offset,remaining);
        if (n==-1) {
            ... // handle EOF
            offset += n;
            remaining -= n;
        }
    }
}

byte[] read(InputStream in, int len) {
    byte[] bytes = new byte[len];
    int offset = 0;
    int remaining = len;
    while (remaining!=0) {
        int n = in.read(bytes,offset,remaining);
        if (n==-1) {
            ... // handle EOF
            offset += n;
            remaining -= n;
        }
    }
    return bytes;
}
Java Sockets and Streams

- String Encoding...
  - Depends on your string encoding...
    - Better to use UTF-8
    - `DataOutputStream.writeUTF(b)`
  - How about now?
    - Yes, but keep in mind that any variable length data structure needs to be prefixed with its length...

```
// Get the server's output stream
OutputStream os;
DataOutputStream dos;
os = client.getOutputStream();
dos = new DataOutputStream(os);
Date date = new Date();
String str = date.toString();
dos.writeUTF(str);

// Get the client's input stream
InputStream is;
DataInputStream dis;
is = server.getInputStream();
dis = new DataInputStream(is);
String date = dis.readUTF();
```

Correct?
Java Sockets and Streams

We could also use Java Object streams...

```java
import java.io.*;
...
while (true) {

    // Server waits for a connection
    Socket client = server.accept();
    // A client connected
    System.out.println("Client " + client.getInetAddress() + " connected.");

    // Get the server’s output stream
    OutputStream os = client.getOutputStream();

    // Get the associated object output stream
    ObjectOutputStream oos = new ObjectOutputStream(os);

    // Write object in output stream
    Date date = new Date();
    oos.writeObject(date);

    // Close output stream
    oos.close();
}
```
Java Object Streams

Looking at both server and client sides...

```java
import java.io.*;
...
// End-less loop
while (true) {
    System.out.println("Waiting for client…");
    // Server waits for a connection
    Socket client = server.accept();
    // A client connected
    System.out.println("Client " + client.getInetAddress() + " connected.");
    // Get the server’s output stream
    OutputStream os = client.getOutputStream();
    // Get the associated object output stream
    ObjectOutputStream oos = new ObjectOutputStream(os);
    // Write object in output stream
    Date date = new Date();
    oos.writeObject(date);
    // Close output stream
    oos.close();
}
```

```java
import java.io.*;
...
// Client connects to server
Socket server = new Socket(serverHost, serverPort);
// Client connected
System.out.println("Connected to " + server.getInetAddress());
// Get the client’s input stream
InputStream is = server.getInputStream();
// Get the associated object input stream
ObjectInputStream ois = new ObjectInputStream(is);
// Read object from input stream
Date date = (Date) ois.readObject();
System.out.println("Server said: " + date);
// Close input stream
ois.close();
```
Java Object Streams

- Based on the Java Serialization framework
  - Any class may implement the Serializable interface
  - If it does, instances of that class can be serialized
  - Serialization is a deep copy
    - Recursive serialization along object references
    - Sharing is respected
  - Warning
    - Deep copy does not stop by itself
      - If any object encountered is not serializable, an exception is thrown
      - Most JRE classes are serializable
  - Possible control point
    - References in classes may be declared transient (easy)
    - Redefine how instances of a class are serialized (harder)
Outline

- Introduction to sockets
- Addressing
- Point-to-point communication with TCP sockets
- Point-to-point communication with UDP sockets
- Group communication
Java sockets over UDP

- Networking in the unconnected mode
  - UDP protocol provides a mode of network communication whereby applications send packets of data, called datagrams, to one another.
  
  - A datagram is an independent self-contained message sent over the network whose arrival, arrival time, and order are not guaranteed.
  
  - Some applications that communicate over the network do not require reliable, point-to-point channel provided by TCP
  
  - Applications might benefit from a mode of communication that delivers independent packages of information whose arrival and order of arrival are not guaranteed
Java classes related to UDP

- **java.net.DatagramPacket**
  - Represents a data packet
    - Essentially a byte buffer
    - Maximum buffer length is known by calling `DatagramSocket.getReceiveBufferSize()`
  - Includes an InetAddress and port number

- **java.net.DatagramSocket**
  - Used for sending and receiving datagram packets
  - Communication happens over UDP
    - Send a DatagramPacket by calling `send` on a DatagramSocket
    - Receive a DatagramPacket by calling `receive` on a DatagramSocket
Example of Java sockets/UDP

```java
import java.net.*;

int port = 1234;
// Create a datagram socket associated with the server port
DatagramSocket serverSock = new DatagramSocket(port);

// End-less loop
while (true) {
    System.out.println("Waiting for client packet…");
    byte[] buf = new byte[256];
    // Create a datagram packet
    DatagramPacket packet = new DatagramPacket(buf, buf.length);
    // Wait for a packet
    serverSock.receive(packet);

    // Get client IP address and port number
    InetAddress clientAddr = packet.getAddress();
    int clientPort = packet.getPort();
    // Build a response
    initialize buf ...
    // Build a datagram packet for response
    packet = new DatagramPacket(buf, buf.length, clientAddr, clientPort);
    // Send a response
    serverSock.send(packet);
}
```
Example of Java sockets/UDP

```java
import java.net.*;

int port = 1234;
// Create a datagram socket with the server port
DatagramSocket serverSock = new DatagramSocket(port);
// End-less loop
while (true) {
    System.out.println("Waiting for client packet…");
    byte[] buf = new byte[256];
    // Create a datagram packet
    DatagramPacket packet = new DatagramPacket(buf, buf.length);
    // Wait for a packet
    serverSock.receive(packet);
    // Get client IP address and port number
    InetAddress clientAddr = packet.getAddress();
    int clientPort = packet.getPort();
    // Build a response
    initialize buf ...
    // Build a datagram packet for response
    packet = new DatagramPacket(buf, buf.length, clientAddr, clientPort);
    // Send a response
    serverSock.send(packet);
}
```

```java
import java.net.*;

int serverPort = 1234;
String serverHost = ...;

// Create a datagram socket
DatagramSocket clientSock = new DatagramSocket();
byte[] buf = new byte[256];
// Get server’s IP address
InetAddress serverAddr = InetAddress.getByName(serverHost);
// Build a request
initialize buf ...
// Create a datagram packet destined for the server
DatagramPacket packet = new DatagramPacket(buf, buf.length, serverAddr, serverPort);
// Send datagram packet to server
clientSock.send(packet);
// Get client IP address
InetAddress clientAddr = packet.getAddress();
// Build a response
initialize buf ...
// Build a datagram packet for response
packet = new DatagramPacket(buf, buf.length);
// Send a response
serverSock.send(packet);
```

System.out.println("Response: " + received);
```
Outline

- Introduction to sockets
- Addressing
- Point-to-point communication with TCP sockets
- Point-to-point communication with UDP sockets
- Group communication
Java Multicast

- Based on UDP sockets
  - Datagram packets (same as before)
  - MulticastSocket extends DatagramSocket

- Relies on IP-level multicast
  - Multicast IP addresses
  - Class D addresses are reserved for multicast
  - In the range 224.0.0.0 to 239.255.255.255
  - A multicast group
    - A multicast address and port
Java Multicast

- Multicasting to a group
  - Need to create a datagram packet
  - Destination is the group InetAddress and port
  - Normal UDP send

- Joigning a multicast group
  - Create the multicast socket with the group port
  - Need to join the multicast group, use the multicast address
  - Receive messages multicasted to the group

- Leaving a multicast group
  - Explicit departure
import java.net.*;

// Multicast group
int groupPort = 5000;
InetAddress groupAddr = InetAddress.getName(“225.4.5.6”);

// Create the socket
MulticastSocket sock = new MulticastSocket(groupPort);
sock.joinGroup(groupAddr);

// Create a datagram packet and do a receive
byte buf[] = new byte[1234];
DatagramPacket packet = new DatagramPacket(buf, buf.length);

// Wait for a packet
sock.receive(packet);

    // When done, leave the multicast group and close the socket
sock.leaveGroup(groupAddr);
sock.close();

}
import java.net.*;

// Multicast group
int groupPort = 5000;
InetAddress groupAddr = InetAddress.getName("225.4.5.6");

// Create the socket
// but we don’t bind it and we don’t join the multicast group
MulticastSocket sock = new MulticastSocket();
byte buf[] = new byte[10];
For (int i=0; i<buf.length; i++)
    buf[i] = (byte)i;

// Create a datagram packet and do a receive
DatagramPacket packet = new DatagramPacket(buf, buf.length, groupAddr, groupPort);

// Send the packet
Byte ttl = 1;
sock.send(packet, ttl);

// When done close the socket
sock.close();
}
Java Multicast

- **Caveats**
  - IP multicast is supported by many routers
    - But most Internet providers forbid IP multicast
    - Over the public Internet, multicast is simply not available
  - Usable on local LAN
  - Normal UDP send

- **Middleware multicast**
  - Multicast can be built above IP or UDP or TCP/IP
    - Using point to point messages
  - Middleware provides
    - Group management and memberships
      - From simple groups to publish-subscribe topics
    - Other multicast properties
      - Totally ordered multicast, reliable multicast
Client-Server based on sockets

Client socket: 36243
Client port: 4320
Server port: 4320
Server socket: 4320

Server:
- Request queue
- Request selection
- Request processing

Client requests
Server response
Server Design

- Request processing model

Sequential / Parallel / Replicated
May cache request results
Sequential processing

while (true) {
    receive(client_id, message);
    extract(message, service_id, params);
    results = do_service(service_id, params);
    send(client_id, results);
}
Design choices for a parallel server

Thread-based

- Advantages
  - Light-weight resources
  - Memory sharing
- Drawbacks
  - Limited isolation

Process-based

- Advantages
  - Strong isolation
  - Real parallelism (also with kernel threads)
- Drawbacks
  - Heavy resources
Multi-threaded server

Client requests → Request queue → Request selection → Worker thread

Main thread

Response
Multi-threaded server UDP-based

```c
while (true) {
    receive(client_id, message);
    extract(message, service_id, params);
    thr = create_thread(client_id, service_id, params);
}
```

Program executed by thread thr:

```c
results = do_service(service_id, params);
send(client_id, results);
exit()
```
Multi-threaded server – TCP-based

Program executed by thread thr:

```java
Socket s;
s.receive(client_id, message);
extract(message, service_id, params);
results = do_service(
    service_id, params);
send(client_id, results);
close()
exit()
```

while (true) {
    Socket s;
    s = accept(client_id, message);
    thr = create_thread(s);
}

client requests

main thread

request queue

request selection

worker thread

request processing

response
Multi-threaded server (thread-pool)
Multi-threaded server (pool) – UDP-based

while (true) {
    receive(client_id, message);
    extract(message, service_id, params);
    work_to_do.put(client_id, service_id, params);
}

Pool of threads:
while (true) {
    work_to_do.get(
        client_id, service_id, params);
    results = do_service(
        service_id, params);
    send(client_id, results);
}
Multi-threaded server (pool) – TCP-based

while (true) {
    Socket s;
    s = accept(client_id, message);
    work_to_do.put(s);
}

Pool of threads:
Socket s;
while (true) {
    s = work_to_do.get();
    s.receive(client_id, message);
    extract(message, service_id, params);
    results = do_service(    service_id, params);
    send(client_id, results);
    close();
}
Multi-threaded clients

- Some clients are single-threaded programs but rarely
- Let’s discuss a simple GUI-based client
- Graphical User Interface
  - Most widget toolkits are single-threaded
  - Imposes that clients are multi-threaded
  - Otherwise, blocking on a socket freezes the GUI
- Examples
  - Eclipse Rich Platform, Google Android
    - Use background threads to carry out potentially blocking invocations such as socket read / write
    - Post events on the GUI thread
Multi-threaded servers with Java

- Integrated into the language
  - Creation / Execution
  - Synchronization mechanisms

- One process, the JVM

- User vs Kernel threads
Thread (Reminder)

- Two ways
  - Extends Thread
  - Implements Runnable

```java
public class HelloThread extends Thread {
    public void run() {
        System.out.println("Hello from a thread!");
    }

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

This example is taken from Java Thread tutorial
Thread (Reminder)

- Another way is to use the Runnable interface

```java
public class HelloRunnable implements Runnable {
    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }
}
```

- Be careful, directly calling the run method does not create a thread
Multi-process servers

- Parallelize the processing of requests on a pool of processes
  - Static vs dynamic pool
  - Request dispatching through a load-balancer

- Load-balancer needs to communicate with the slaves servers
  - Use socket or higher level communication layer
  - Need to detect and manage failures
References

This lecture is built from:

