IBD – Intergiciels et Bases de Données

Socket-based distributed systems

Fabien Gaud, Fabien.Gaud@inrialpes.fr

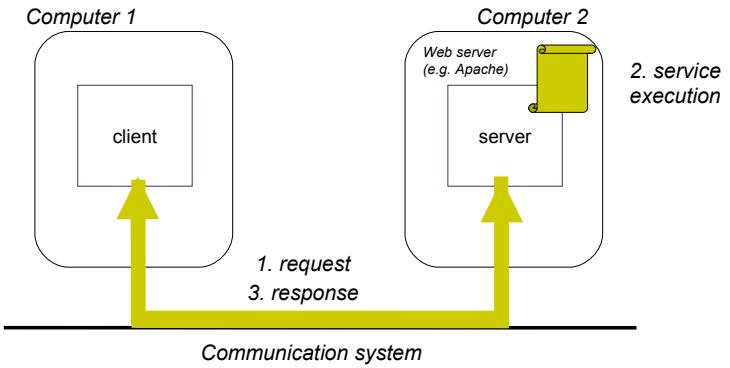
http://www-ufrima.imag.fr/ ⇨ Placard électronique ⇨ M1 Info ⇨ IBD

Client/server applications

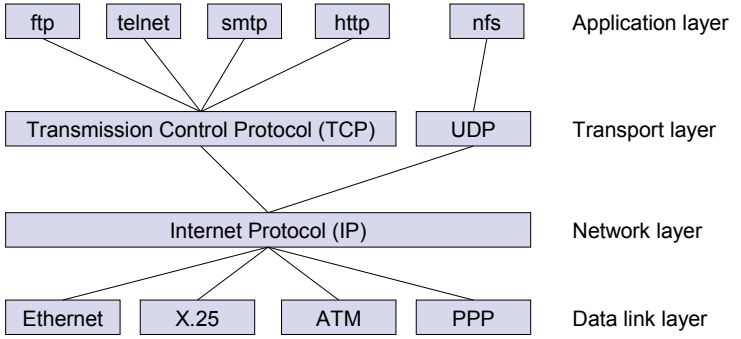
- A server provides some services
  - Examples
    - processing database queries
    - sending out current stock prices
- A client uses some services provided by a server
  - Examples
    - displaying database query results to the user
    - making stock purchase recommendations to an investor

Client/server applications (2)

- Synchronous communication



Protocol layers

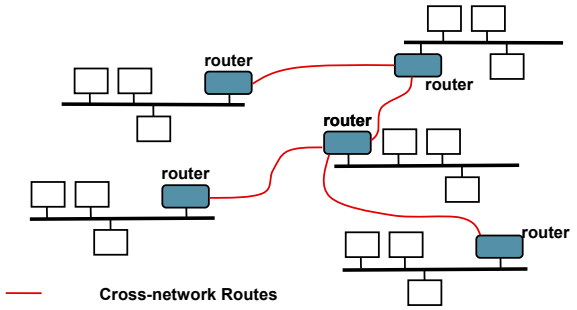


Internet Protocol

- IP (Internet Protocol)
  - The Internet 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, naming a host machine (e.g. 192.168.2.100)
  - IP addresses are location dependent
- IP Routing on LANs
  - Physical layer directly provides this

The Internet

- IP Routing on WANs
  - It is a collaborative and distributed protocol between routers
  - Routers exchange their routing tables to build up their routing knowledge

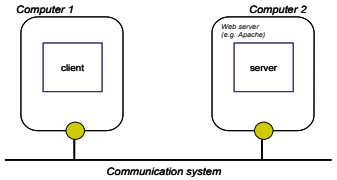


Ports

- Ports are end points
  - For communication channels over IP
  - An IP address and a port names an end point
- Port numbers are managed by the operating system
  - Many important services have a standardized port
  - Example: port 25 for telnet service
  - Port between 1 and 1023 are reserved
- Port numbers are allocated on-demand to processes
  - A telnet service provider will be allocated the port 25
  - Only one process may be allocated the port 25

Ports (2)

- Communication Channel
  - Between two ports, allocated to two processes
  - Two processes may be on the same machine or not
- Client-Server Pattern
  - The server waits for data from the channel
  - A client process is allocated a port when establishing the communication channel to a server

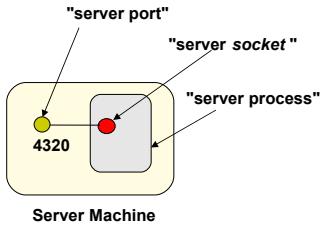


Sockets

- Basic Middleware
  - A programming model based on streams
  - A behavior semantics (UDP,TCP, ...)
- Programming Model
  - A socket is the endpoint of a communication channel
  - A socket represents a port allocated to a process on a machine
  - Through a stream interface, one may send/receive bytes through a socket

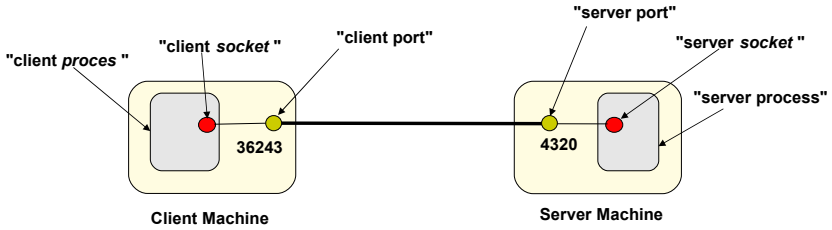
Sockets – Steps Involved

- Server side
  - Creation of the server process
    - Request a socket on a port
    - The local port is granted by the operating system
  - Server waits for incoming data



Sockets – Steps Involved

- Client side
  - Creation of the client process
  - Request a channel to the remote port
    - A local port is allocated
    - The communication channel is established
  - The two sockets are connected to each other
    - Client sends/receives data



User Datagram Protocol

- UDP (User Datagram Protocol)
  - Over IP that routes data packets
    - Data packets are of fixed size
    - But different sizes for 100Mbps or 1Gbps Ethernet
  - **WARNING**
    - Output stream is automatically sliced into data packets
    - Data packets may be lost or reordered
    - But very efficient (just an IP++)

Transmission Control Protocol

- TCP (Transmission Control Protocol)
  - Also above IP, but it is lossless and FIFO
    - Lossless: data packets are not lost
    - FIFO: data packets are delivered in the order they were sent
  - Can be used as a real stream
    - Bytes can be streamed through
    - Bytes will not be lost and arrive in the same order
  - A connection-oriented protocol
    - An actual point-to-point connection needs to be opened and closed
    - Connections introduce an overhead

UDP and TCP

- Typical applications
  - 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
  - TCP
    - Transferring files (ftp for instance)
    - Downloading web pages or images

Sockets – A bit of history

- Originally
  - Sockets were developed for BSD Unix, in the 1980s
  - Sockets used to be part of the operating system; they had to be invoked via system-specific libraries for C/C++
  - Programming distributed applications was hard (access was different from one OS to another, programs were not portable)
- Today
  - Sockets are available on all platforms and represent the most fundamental communication mechanism
  - Example: the Java programming interface for sockets abstracts them from the underlying OS, making them easier to use

Outline

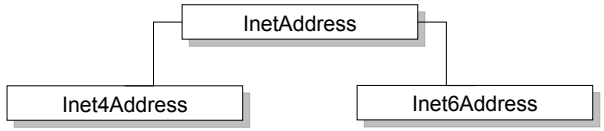
- Introduction to sockets
- **Addressing**
- Point-to-point communication with TCP sockets
- Point-to-point communication with UDP sockets
- Group communication with multicast
- Locating network resources
- Threads in JAVA

Addressing in Java

- The *java.net* package provides the following addressing-related classes:
  - InetAddress
  - Inet4Address
  - Inet6Address
  - SocketAddress
  - InetSocketAddress

IP addressing in Java

- For IP addressing, three classes are provided:
  - ***InetAddress*** represents an IP address
  - ***Inet4Address*** represents a 32-bit IPv4 address
  - ***Inet6Addresss*** represents a 128-bit IPv6 address



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 string in textual presentation.
  - String **getHostName**()
    - Gets the host name for this IP address.
    - E.g. *hoff.e.ejf-grenoble.fr*

IP addressing in Java

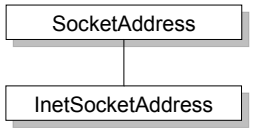
- ***Inet4Address***
  - Represents a 32-bit IPv4 address
- Some API elements
  - byte[] **getAddress**()
    - Returns the raw IP address of this InetAddress object
  - String **getHostAddress**()
    - Returns the IP address string in textual presentation form
    - Familiar form (a.b.c.d), e.g. 129.250.35.250

IP addressing in Java

- Host name to IP address resolution
  - Using a network naming service
    - DNS (Domain Name System)
    - NIS (Network Information Service)
  - Discussing DNS
    - A fairly complex world-wide distributed system
      - Associate names to IP addresses
      - Allows aliases (one IP address, several names)
    - Organized as hierarchical zones across the world
      - A zone is managed by a DNS server
    - Replicated servers for high-availability

Socket addressing in Java

- SocketAddress
  - Abstract class for a socket address
  - Independent of any specific addressing protocol
- InetSocketAddress
  - It 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)



Outline

- Introduction to sockets
- Addressing
- **Point-to-point communication with TCP sockets**
- Point-to-point communication with UDP sockets
- Group communication with multicast
- Locating network resources
- Threads in JAVA

Java Sockets over TCP

- General Schema
  - A connection is opened between a client and a server,
  - A series of request/response (i.e. messages) are exchanged
  - The connection is closed
- Server-side Session
  - Servers often maintain a session per client
    - Each session maintains the state of client-server interaction
    - A typical example is HTTP sessions
  - Remember
    - TCP is a loss-less and FIFO protocol

Java classes related to TCP sockets

- The *java.net* package provides
  - ServerSocket class
  - Socket class
- *ServerSocket*
  - It represents the socket on a server
  - Servers accept incoming connections
- *Socket*
  - It represents the endpoints
  - Both on server and client sides

Example of Java sockets/TCP

```
import java.net.*;

int port = 1234;

// Create a server socket associated with port 1234

ServerSocket server = new ServerSocket(port);

// 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.");
    // Server receives a message from client
    ...
}
```


Example of Java sockets/TCP

```
import java.net.*;

int port = 1234;

// Create a server socket associated with port 1234

ServerSocket server = new ServerSocket(port);

// 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.");
     // Server receives a message from client
     ...
}
```

```
import java.net.*;

String serverHost = "sun";

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 a message to 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 respectively represent input streams of bytes and output streams of bytes
  - *ObjectInputStream* / *ObjectOutputStream*
    - classes respectively representing input streams of Java objects and output streams of Java objects
  - *FileInputStream* / *FileOutputStream*
    - classes representing input streams for respectively reading data from a file ou writing data to a file
  - *FilterInputStream* / *FilterOutputStream*
    - classes that respectively contain other input streams or output streams, which it uses to possibly transform data along the way

Java Sockets and Streams

```
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();

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

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


Java Sockets and Streams

```
import java.io.*;

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

    // Get the server's output stream
    OutputStream os = client.getOutput

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

    // 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);
```

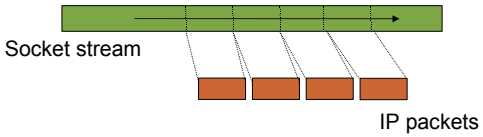

```
// Get the client's input stream
InputStream is = server.getInputStream();
byte[] b = new byte[100];
int num = is.read(b);
String date = new String(b);
```

Java Sockets and Streams

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

```
// 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:
    - Nothing is left in the input stream before reading
    - The entire string has been received when reading
  - Just not always true
    - When sending variable length data structures, **send the length**
    - How?
      - Length is an **integer**
      - We can send **bytes**

```
// 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
  - Extends FilterOutputStream
  - Provides type-aware operations
  - **Endianness proof !**
  - But introduces an overhead
- Solution
  - Sending the length

```
// 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);
```

Correct?

```
// 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);
```


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);
```

Correct?

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

Java Sockets and Streams

We could also use Java Object streams...

```
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 Object Streams

Looking at both server and client sides...

```
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.get
    // Get the server's output stream
    OutputStream os =   client.getOutputStrea
    // Get the associated object output //str
    ObjectOutputStream oos = new
        ObjectOutputStream(os);
    // Write object in output stream
    Date date = new Date();
    oos.writeObject(date);
    // Close output stream
    oos.close();
}
```

```
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 with multicast
- Locating network resources
- Threads in JAVA

Java sockets over UDP

- Networking in the unconnected mode (using UDP)
  - 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
  - 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.

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

```
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

```
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
    // Build a response
    initialize buf ...
    // Build a datagram packet for response
    packet = new DatagramPacket(buf, buf.length, cli
    // Send a response
    serverSock.send(packet);
}
```

```
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);

// Build a datagram packet for response
packet = new DatagramPacket(buf, buf.length);
// Receive response
clientSock.receive(packet);
String received = new String(packet.getData(), 0,
    packet.getLength());
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 with multicast**
- Locating network resources
- Threads in JAVA

Java Multicast

- Based on UDP Sockets
  - DatagramPacket (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 a port

Java Multicast

- Multicasting to a group
  - Needs to create a DatagramPacket
  - Destination is the group InetAddress and port
  - Normal UDP send
- Joining a multicast group
  - Create the MulticastSocket with the group port
  - Needs to join the multicast group, use the group InetAddress
  - Receives messages multicasted to the group
- Leaving a multicast group
  - Explicit departure from a multicast group

Java Multicast – Receiving

```
// Import some needed classes
import sun.net.*;
import java.net.*;

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

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

// Create a DatagramPacket and do a receive
DatagramPacket pack;
byte buf[] = byte[1024];
pack = new DatagramPacket(buf, buf.length);

s.receive(pack);

// When done...
// leave the multicast group and close the socket
s.leaveGroup(InetAddress.getByName(group));
s.close();
```

Java Multicast – Sending

```
import sun.net.*;
import java.net.*;

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

// Create the socket
// but we don't bind it and we don't join the multicast
group
// as we are only going to send data
MulticastSocket s = new MulticastSocket();

byte buf[] = byte[10];
for (int i=0; i<buf.length; i++) buf[i] = (byte)i;

// Create a DatagramPacket
DatagramPacket pack = new DatagramPacket(buf, buf.length,
                                         groupAddr, groupPort);
// Do a send.
byte ttl = 1;
s.send(pack,ttl);

// And when we have finished sending data close the socket
s.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 LANs however
- Middleware Multicast
  - Multicast can be built above IP or UDP
    - Using point-to-point messages
  - Middleware provides
    - Group management and membership to groups
      - From simple groups to publish-subscribe topics
    - Other multicast properties
      - Totally-ordered multicast, reliable multicast

Outline

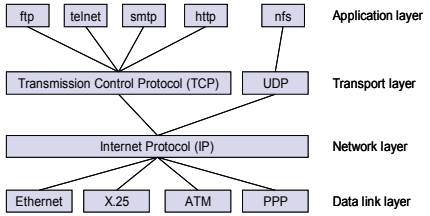
- Introduction to sockets
- Addressing
- Point-to-point communication with TCP sockets
- Point-to-point communication with UDP sockets
- Group communication with multicast
- **Locating network resources**
- Threads in JAVA

Network Resources

- Examples of network resources
  - An image, audio file available on the network
  - A program file, such as a Servlet, available on the network
- Java classes related to locating or identifying network resources (c.f. package java.net)
  - URI
  - URL
  - URLClassLoader
  - URLConnection
  - URLStreamHandler
  - HttpURLConnection
  - JarURLConnection

Network Resources

- URI – Uniform Resource Identifier
  - It is an identifier for a resource
  - But not necessarily a locator for that resource
- URL – Uniform Resource Locator
  - A URL tells how to access the resource
  - The protocol used to locate the resource is known from the URL



Network Resources

- A URL is a pointer to a "resource" on the World Wide Web
  - Example of a URL:
    - http : //java.sun.com
  - A URL has two main components:
    - Protocol identifier
      - Hypertext Transfer Protocol (HTTP)
      - File Transfer Protocol (FTP)
    - Resource name
      - The name format depends entirely on the protocol used

Network Resources

- More URL Names
  - The format of the resource name depends entirely on the protocol used
  - For many protocols, including HTTP, the resource name contains one or more of the components listed below:
    - **Host Name:** the name of the machine on which the resource lives.
    - **Port Number: t**he port number to which to connect (typically optional).
    - **Filename:** the pathname to the file on the machine.
    - **Reference:** a reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

Network Resources

- URLConnection
  - Abstract superclass of all connection classes
  - Used by applications to identify and locate network resources
  - Usage:
    - Obtain a URLConnection from a URL
      - static URLConnection URL.openConnection()
    - Connect
      - URLConnection.connect() actually opens the connection
      - URLConneciton.getInputStream() to read the resource
- HttpURLConnection
  - It is the most commonly used implementation
  - It uses the HTTP protocol

Outline

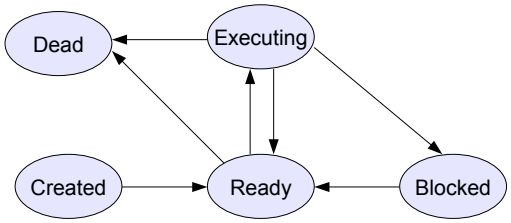
- Introduction to sockets
- Addressing
- Point-to-point communication with TCP sockets
- Point-to-point communication with UDP sockets
- Group communication with multicast
- Locating network resources
- **Threads in JAVA**

Threads in JAVA

- Integrated into the language
  - Creation / Execution
  - Synchronization mechanisms
- One process, the JVM
- User vs Kernel threads

Thread states

- A thread can be in different states



See Thread.State for details

Thread objects

- Two ways
  - Extends Thread
  - Implements Runnable

```
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 taken from Java Thread tutorial

Thread objects (2)

- Another way is to use the Runnable interface

```
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

Another Example

```
public class Counter {
          private int counter;
          public Counter (int c){
                counter = c;
          }
          void increment(){
                counter++;
          }
          void decrement(){
                counter--;
          }
          int getCounter(){
                return counter;
          }
}
```

Correct ?

```
public class Example extends Thread {
      private int no;
      private Counter c;

      public Example(int no, Counter c) {
            this.no = no; this.c = c;
      }
      public void run() {
            try {
                  Thread.sleep(100);
                  if((no % 2) == 0)
                        c.increment();
                  else
                        c.decrement();
            }
            catch (InterruptedException e) {
                  // Handling
            }
    }
    public static void main(String args[])
      throws InterruptedException {
      Counter c = new Counter(0);
            Thread T1 = new Example(0,c);
            Thread T2 = new Example(1,c);
            T1.start();
            T2.start();
            T1.join();
            T2.join();
         System.out.println( "Counter = " + c.getCounter() );
    }
}
```


Another Example (2)

- Results on 10 iterations

```
Counter = -1
Counter = 0
Counter = 0
Counter = 0
Counter = -1
Counter = 0
Counter = 0
Counter = 1
Counter = 0
Counter = 0
```

- ++ / -- are not atomic operations
- Concurrent accesses to a shared memory need protecting it

The synchronized keyword

- Synchronization is based on monitors
- The synchronized keyword permits to set critical sections
- The synchronized keyword applies on an object
- Possibility to wait() on a condition and to notify() a waiter

Synchronized example

```
public class Counter {
        private int counter;
        public Counter (int c){
                counter = c;
        }
        synchronized void increment(){
                counter++;
        }
        synchronized void decrement(){
                counter--;
        }
        synchronized int getCounter(){
                return counter;
        }
}
```

```
[ ... ]
        if((no % 2) == 0){
                synchronized (c) {
                        c.increment();
                }
        }
        else{
                synchronized (c) {
                        c.decrement();
                }
        }
[ ... ]
```

```
public class Counter {
        private int counter;
        public Counter (int c){
                counter = c;
        }
        void increment(){
                synchronized (this) {
                        counter++;
                }
        }
        void decrement(){
                synchronized (this) {
                        counter--;
                }
        }
        int getCounter(){
                synchronized (this) {
                        return counter;
                }
        }
}
```


Another synchronization example

```
public class Example2 extends Thread {
     final static int nbThreads = 4; static int counter; static Object barrier;
     private int no;

     public Example2(int no) {
          this.no = no;
     }

     public void run() {
          synchronized(barrier){
               counter--;
               if(counter == 0){
                    System.out.println("Thread "+no+", releasing"); barrier.notifyAll();
               }
               else{
                    System.out.println("Thread "+no+", waiting"); barrier.wait();
               }
          }
     }

    public static void main(String args[]) throws InterruptedException {
     Thread threads[] = new Thread[nbThreads];
     counter = nbThreads;  barrier = new Object();

     for(int i = 0; i<nbThreads; i++){
          threads[i] = new Example2(i);
          threads[i].start();
     }

     for(int i = 0; i<threads.length; i++)
          threads[i].join();
    }
}
```

Remember

Concurrency is a really hard problem. Be careful when using shared objects among threads.

Future Outline

- Lectures
  - *Introduction to distributed systems and middleware*
  - Socket-based distributed systems
  - RMI -based distributed systems
  - Servlet-based distributed systems
  - Introduction to multi-tier distributed Internet services
- Practical work
  - Programming distributed systems with Sockets
  - Programming distributed systems with RMI
  - Programming distributed systems with Servlets
  - Project on multi-tier Internet services

References

This lecture is built from:

- Sun Microsystems. *Java Tutorial on Networking*. http://java.sun.com/docs/books/tutorial/networking/
- Sun Microsystems. *Java Tutorial on Concurrency*. http://java.sun.com/docs/books/tutorial/essential/concurrency/index.html
- M. Boger. *Java in Distributed Systems: Concurrency, Distribution and Persistence*. Wiley, 2001.
- This lecture is mostly based on lectures given by Sara Bouchenak, http://sardes.inrialpes.fr/~bouchena/