UADY

FACULTAD DE MATEMÁTICAS

Sistemas Distribuidos

Unidad 2

Comunicación entre procesos

Sistemas Distribuidos

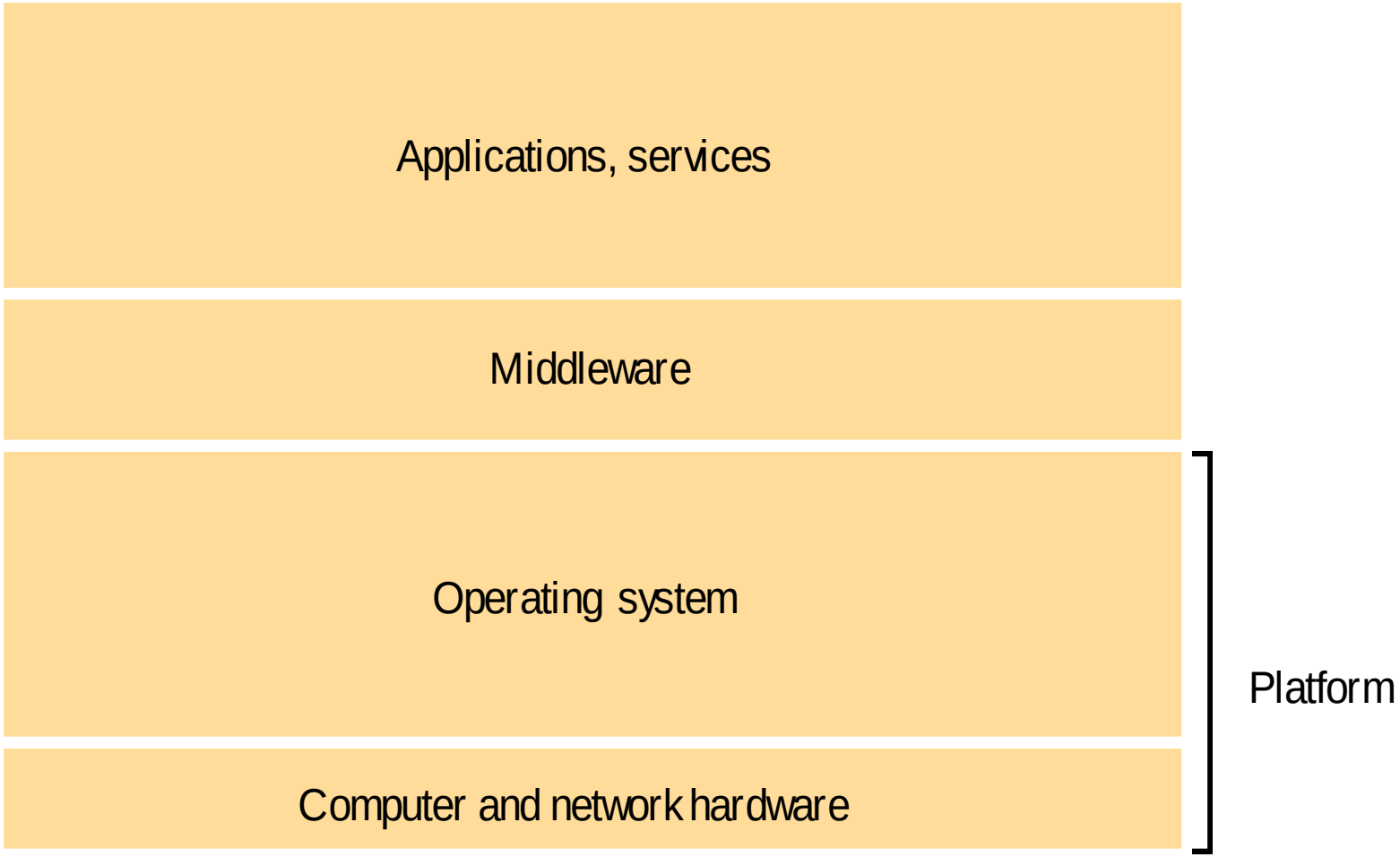
- **2. COMUNICACIÓN ENTRE PROCESOS**

  El alumno aplicará los modelos de comunicación entre procesos distribuidos que le permitan desarrollar aplicaciones acordes a las arquitecturas de sistemas distribuidos.

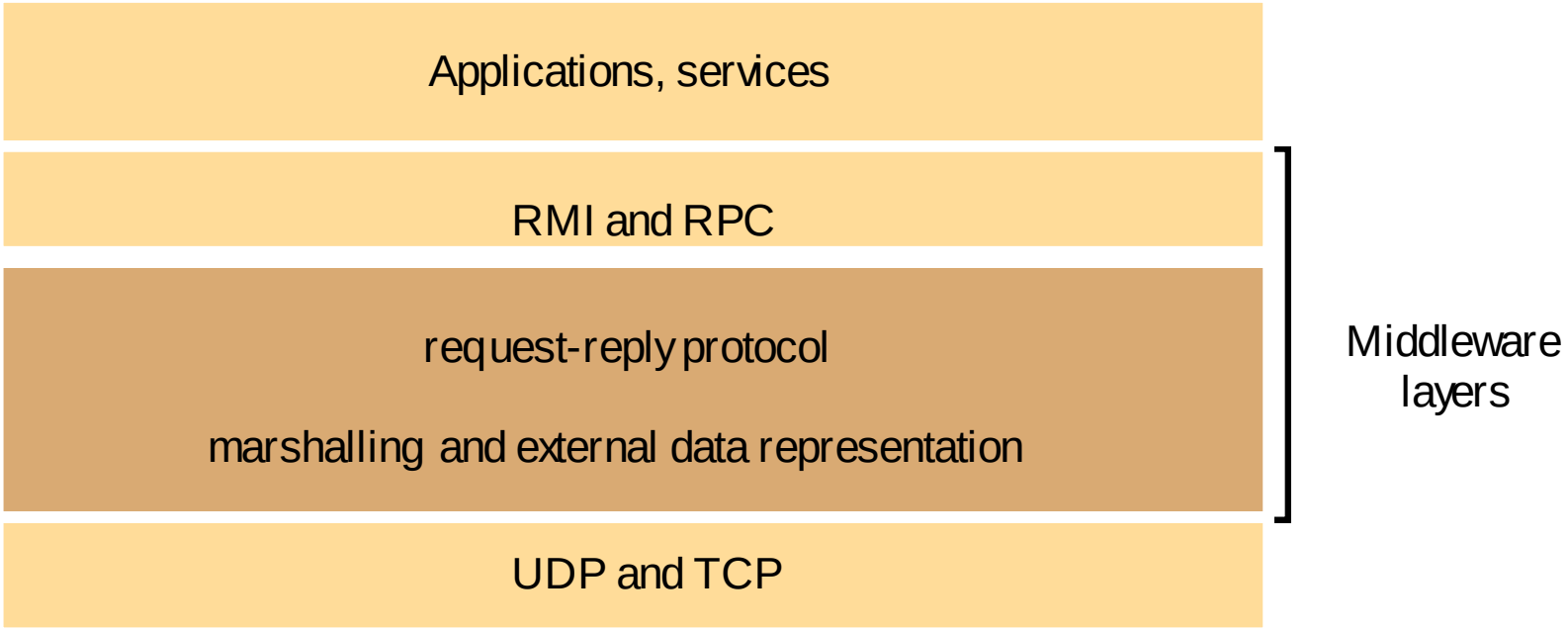
  - **Introducción.**
  - **Protocolos de Internet para la comunicación de procesos.**
    - **Sockets UDP.**
    - Sockets TCP.
    - Comunicación en grupo: multidifusión IP.
  - Representación externa de datos.
  - Comunicación entre procesos distribuidos.
    - RPC.
    - RMI.
    - Servicios Web

Introducción a la Comunicación entre Procesos

Capas de Hw y Sw en SD



Capas de Middleware



Introducción

- Muchas aplicaciones y servicios de red como, la transferencia de archivos, verificación de autenticidad, ingreso remoto, acceso a sistemas de archivos remoto, se basan en el modelo cliente servidor.
- Un proceso servidor puede estar en la misma máquina que el cliente o en otra, en tal caso se requiere comunicación a través de la red.
- Las dos principales clases de protocolos de comunicaciones de bajo nivel que manejan el paradigma cliente-servidor son:
  - protocolos orientados a conexión (TCP)
  - protocolos sin conexión (UDP)

Comunicación entre procesos

- La forma más simple de comunicación entre procesos es el paso de mensajes.
- Basada en dos operaciones
  - Enviar (*send*)
  - Recibir (*receive*)
- Esta actividad implica la comunicación de datos desde el proceso emisor al proceso receptor y puede implicar además la sincronización de los dos procesos.
- A cada destino se le asocia una cola de mensajes recibidos. Un emisor entrega un mensaje en una cola remota. Un receptor lee y elimina los mensajes de su cola local

Características de la Comunicación

La comunicación entre los procesos puede ser: síncrona y asíncrona.

- **Síncrona**
  - Los procesos emisor y receptor se sincronizan en cada mensaje.
  - En este caso, tanto **enviar** como **recibir** son operaciones **bloqueantes**.
  - A cada **enviar** producido, el proceso emisor se bloquea hasta que se produce el correspondiente recibe
  - Cuando se invoca un **recibir**, el proceso se bloquea hasta que llega un mensaje

Características de la Comunicación

- **Comunicación asíncrona**
  - La utilización de la operación **enviar no es bloqueante.**
  - La operación **recibir** puede ser bloqueante o no bloqueante.
  - El proceso emisor puede continuar tan pronto como el mensaje haya sido copiado en el búfer local
  - La transmisión del mensaje se lleva a cabo en paralelo con el proceso emisor
  - El proceso receptor sigue con su flujo de ejecución después de invocar la operación **recibir**

Comunicación entre procesos

- Destino de mensajes
  - (dirección IP, puerto local)
  - Los servidores hacen público el número de sus puertos para que sean usados por los clientes
- Fiabilidad
  - Garantiza que los mensajes se entregan a pesar de perder un número razonable de ellos
- Integridad
  - Los mensajes deben llegar a su destino sin corromperse ni duplicarse
- Ordenación
  - Los mensajes son entregados en el orden de su emisión

Sockets (conectores)

- Un socket es una abstracción de un canal de comunicación de datos entre un par de procesos, que esconde los detalles del proceso de comunicación. (conectar, leer, escribir, etc.)

- Permite tratar la conexión de red como un canal en el que los bytes pueden ser leídos o escritos de manera similar a la operación con archivos.

- Además, evita tener que preocuparse por detalles de bajo nivel como sería el tamaño de los paquetes, su retransmisión, el tipo de contenido, etc.

Sockets (Conectores)

- Un socket es un puerto que permite el enlace de otro puerto para establecer un canal de comunicación a través del cual se pueden transmitir paquetes de datos.

- Los sockets se encuentran implementados en la capa de transporte del modelo de redes.

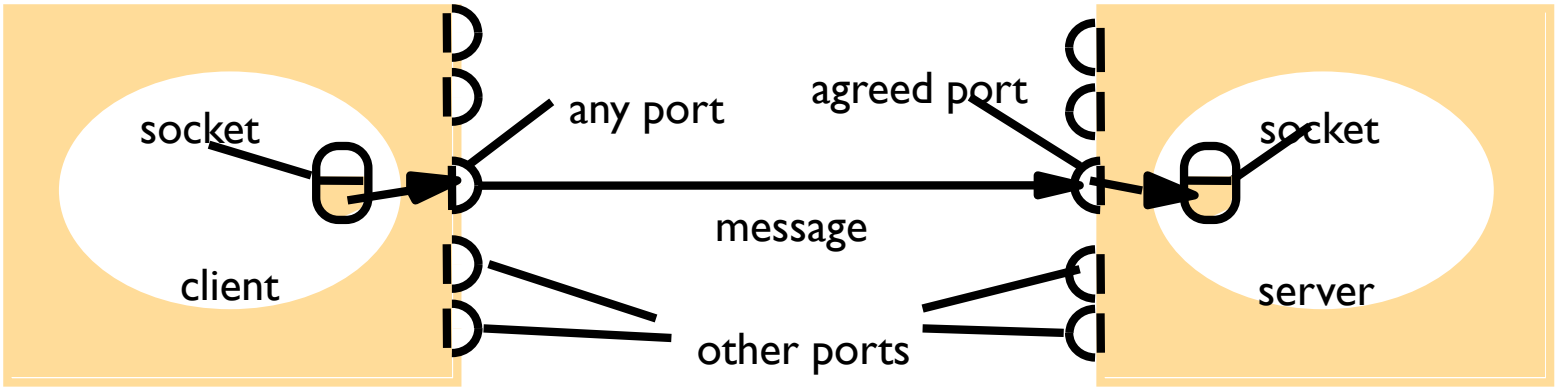
- Son identificados con un número de 16 bits, lo cual permite alrededor de 65,000 posibles puertos en una misma dirección.

Sockets (Conectores)

- Los tres recursos que originan el concepto de *socket son*:
  - Un **protocolo de comunicaciones**, que permite el intercambio de bytes.
  - Una **dirección del Protocolo de Red** (Dirección IP, si se utiliza el Protocolo TCP/IP), que identifica una computadora.
  - Un **número de puerto**, que identifica a un programa dentro de una computadora.

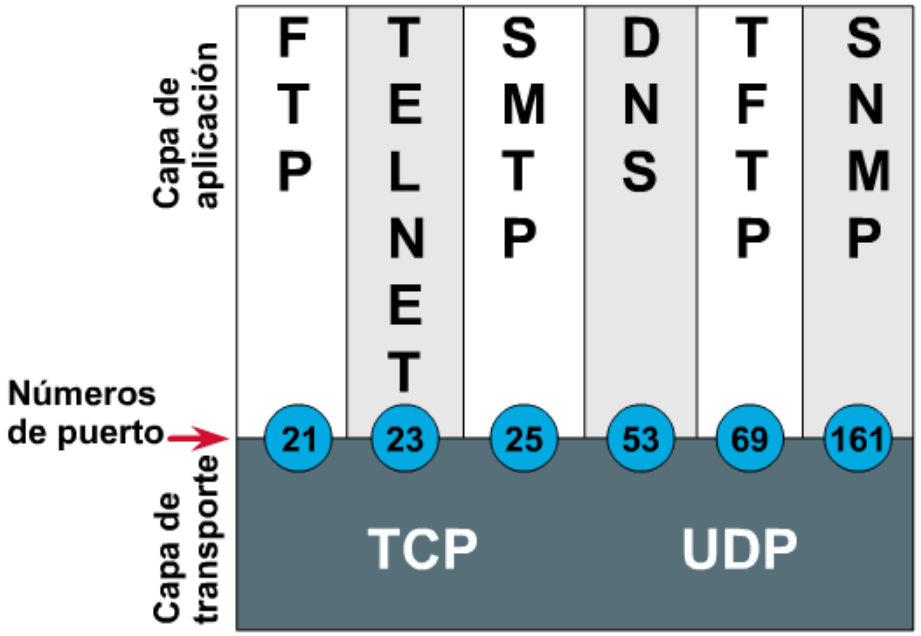
Conectores (*sockets*)

- Los procesos pueden utilizar un mismo conector para enviar y recibir mensajes.
- Cada computadora permite **2^{16} (65,536)puertos** posibles que pueden ser usados por los procesos locales para recibir mensajes
- Cada proceso puede utilizar varios puertos para recibir mensajes, pero no puede compartir puertos con otros procesos de la misma computadora
- Cada conector se asocia con un protocolo en concreto: UDP o TCP



Puertos y sockets

- Todas las aplicaciones TCP o UDP tienen un número de puerto que las identifica.



Conectores (*sockets*)

- Los primeros 1024 puertos se encuentran reservados para servicios específicos:

Puerto	**Protocolo**	**Referencia**
13	DayTime	RFC 867
7	Echo	RFC 862
25	SMTP (e-mail)	RFC 821,1869, 822, 1521
110	Post Office Protocol	RFC 1725
20	File Transfer Protocol	RFC 959
80	Hypertext Transfer Protocol	RFC 2616

Puertos y sockets

- El número de puerto identifica el **tipo de servicio** que un sistema TCP o UDP está solicitando.
- Asignación de números de puerto:
  - Los números **inferiores a 255** se utilizan para procesos de uso frecuente o "**aplicaciones públicas**".
  - Los números de puertos **mayores a 255** se reservan para el **uso privado** de la máquina local.
  - Entre los puertos mayores a 255, en particular **entre 255 y 1023** están asignados a las compañías con **aplicaciones comerciales**.
  - Los puertos **superiores al 1023 no están regulados** y se usan para asignarse a las aplicaciones de usuarios en general.

Sockets (conectores)

- Los *sockets* permiten implementar una arquitectura cliente-servidor.

- La comunicación ha de ser iniciada por uno de los programas que se denomina programa ***cliente***.

- El segundo programa espera a que otro inicie la comunicación (petición de servicio), por este motivo se denomina programa ***servidor***.

Conectores (*sockets*)

- Un *socket* puede realizar 7 operaciones básicas:
  - conectarse a una máquina remota
  - enviar datos
  - recibir datos
  - cerrar la conexión
  - *asignarse a un puerto*
  - *escuchar por peticiones*
  - *aceptar conexiones de máquinas remotas en el puerto asignado*

- Las últimas 3 operaciones sólo son utilizadas por los *ServerSockets*, que necesitan esperar a que los clientes se conecten.

Mecanismos de enlace de *sockets*

- **Orientados a conexión usando el protocolo TCP** (teléfono)
  - Se basa en la metodología de cliente/servidor donde se distinguen dos tipos de *sockets*:
    - *socket* cliente (clase Socket) es aquel que inicia la comunicación
    - *socket* servidor (clase ServerSocket) es aquel que espera por clientes

- **Orientados a no conexión usando el protocolo UDP** (correo)
  - Útil como medio de multidifusión (*broadcast)* donde se tienen dos componentes:
    - emisor/receptor (clase DatagramSocket)
    - información (clase DatagramPacket)

- Todas estas clases se encuentran dentro del paquete *java.net*.

Comunicación entre procesos

Sockets UDP. Comunicación por datagramas

Comunicación de datagramas (UDP)

- UDP no distingue entre clientes y servidores, usa el mismo tipo de *socket* para comunicar datos y escuchar por información
- UDP trata la conexión de red como un conjunto de paquetes individuales
- Un datagrama enviado por UDP se transmite desde un proceso emisor a uno receptor sin acuse de recibo ni reintentos.
- Cualquier proceso que necesite enviar o recibir mensajes debe crear un *socket* asociado a una dirección IP y un puerto local
- El método *receive* devolverá, además del mensaje, la dirección IP y el puerto del emisor para poder emitir una respuesta

Comunicación de datagramas

- Tamaño del mensaje
  - El receptor debe especificar el tamaño de la cadena sobre la que almacenará el mensaje recibido (8Kb). La aplicación establece el tamaño
- Bloqueo
  - Envío no bloqueante y recepción bloqueante
- Tiempo límite de espera
  - Uso de tiempos de espera (*timeouts*) en los *sockets*
- Recibe de cualquiera
  - El método *receive* no especifica el origen de los mensajes, acepta mensajes dirigidos a su conector desde cualquier origen y devuelve la dirección IP y el puerto del emisor

Comunicación de datagramas

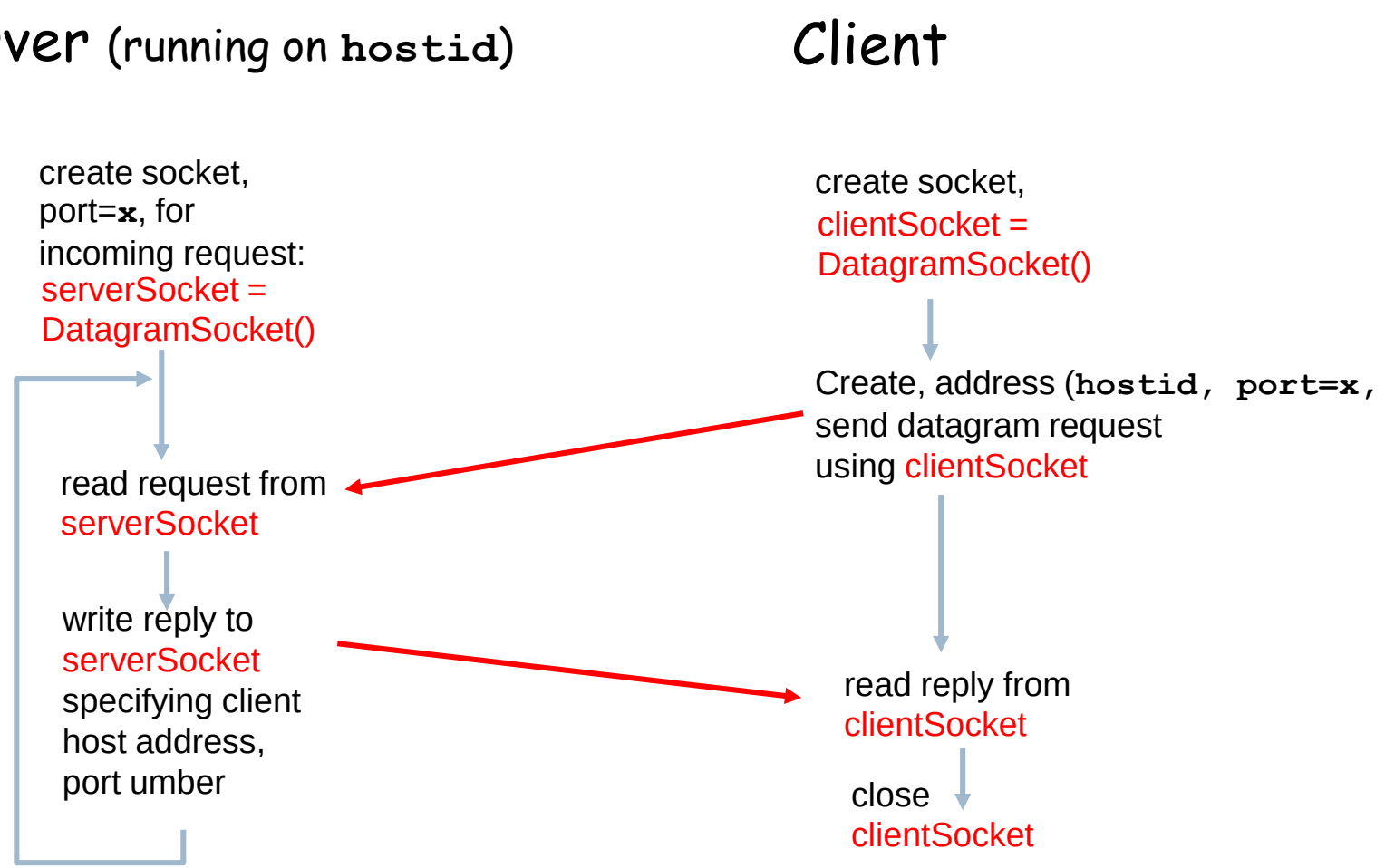
- Modelo de fallo
  - *Fallos de omisión*. Los mensajes pueden desecharse ocasionalmente, ya sea por un error producido en la suma de comprobación, o por falta de espacio en el búfer origen o destino.
  - *Ordenación*. Los mensajes pueden entregarse en desorden con respecto a su orden de emisión.
- Las aplicaciones que usan UDP dependen de sus propias comprobaciones para conseguir la fiabilidad de la comunicación.
- Adicionalmente se pueden implementar acuses de recibo y numeración de paquetes.

API Java para datagramas UDP

- Utiliza dos clases:
  - *DatagramPacket*
  - *DatagramSocket*

- Proceso:
  1. Colocar la información a enviar en un *DatagramPacket*
  2. Enviar el paquete usando un *DatagramSocket*
  3. Recibir el paquete a través de un *DatagramSocket*
  4. Acceder a la información recibida en un *DatagramPacket*

Interacción con socket Cliente/Servidor: UDP



Clase DatagramPacket

- Constructor para cuando se recibe información:
  - **DatagramPacket (byte buffer[], int length)**

- Se le proporciona un arreglo de bytes vacío para colocar la información y la longitud máxima que se puede acomodar en el mismo. El valor de longitud debe ser menor o igual al tamaño del arreglo, de lo contrario se lanzará una excepción en tiempo de ejecución que no es obligatorio atrapar pero sí recomendable.

- El límite superior es de 64 kilobytes (65,535 bytes) de donde 8 bytes son usados para el encabezado UDP y de 20 a 60 bytes para el encabezado IP. La longitud a escoger dependerá de factores como el tipo de red (paquetes pequeños para redes lentas o poco confiables). Un valor de 8 kilobytes (8192 bytes) es buena opción.

Clase DatagramPacket

- Constructor para cuando se envía información:
  - **DatagramPacket (byte buffer[], int length, InetAddress ia, int port)**

- Se le proporciona un arreglo de bytes que contiene la información, así como la longitud de la misma en el arreglo. También se puede generar una excepción en tiempo de ejecución si el valor de longitud es mayor que el tamaño del arreglo.

- Además se requiere la dirección y el puerto al que va dirigido el paquete. Estos datos se integran con la información del arreglo, por lo tanto hay que considerar que el total no sea mayor a 64 kilobytes.

Métodos get

- La información contenida en el paquete es la siguiente:
  - Encabezado IP: dirección del emisor y dirección del destinatario
  - Encabezado UDP: puerto del emisor, puerto del destinatario y longitud del paquete incluyendo este encabezado
  - Contenido del paquete
- **public InetAddress getAddress()**
  - Si el paquete es para enviarse, el método regresa la dirección del destinatario. Y si el paquete es recibido, regresa la dirección del emisor.
- **public int getPort()**
  - Se comporta de manera similar al método anterior para el puerto.
- **public int getLength()**
  - Regresa el número de bytes en el contenido del paquete.

Métodos get

- **public byte[] getData()**
  - Regresa en un arreglo de bytes el contenido del paquete. Usualmente es necesario convertir el valor de retorno en otra forma antes de utilizarlo.

- Conversión a un String en caso de texto ASCII:
  - ***public String (byte[] buffer, int high_byte, int start, int num_bytes)***

    Los argumentos son: el arreglo del método getData(), un cero para convertir cadacaracter a texto Unicode que utiliza Java, un cero para indicar donde inicia elcontenido del arreglo y el valor del método getLength().

- Conversión a un flujo de entrada:
  - ***public ByteArrayInputStream (byte[] buffer, int offset, int num_bytes)***

    Los argumentos son: el arreglo de getData(), un cero y el valor de getLength().

Clase DatagramSocket

- No hay distinción entre un *socket* servidor y un *socket* cliente, la única diferencia es si el puerto es conocido o es anónimo al momento de crear el *socket*. Los puertos TCP y UDP no están relacionados, así que se pueden utilizar los mismos puertos para dos protocolos diferentes.

- Constructores:
  - **DatagramSocket() throws SocketException**

  El sistema escoge un puerto disponible al azar para poder enviar información (iniciar una conversación). Las respuestas pueden ser devueltas a este mismo puerto gracias a que el encabezado de los paquetes indican el puerto de origen.
  - **DatagramSocket(int port) throws SocketException**

  Se asigna un puerto específico para poder recibir información (escuchar clientes). Las excepciones pueden deberse a que el puerto escogido ya está en uso o a que no se tienen privilegios para seleccionar ese puerto.
  - **DatagramSocket(int port, InetAddress intf) throws SocketException**

  Esta opción es para cuando se tienen múltiples *hosts*.

Clase DatagramSocket

- **public void send(DatagramPacket dp) throws IOException**

  Envía un paquete previamente construido a través de un *socket* previamente creado.
- **public void receive(DatagramPacket dp) throws IOException**

  El programa bloquea su ejecución hasta recibir un paquete. El tamaño del arreglo del DatagramPacket debe ser suficiente para almacenar la información recibida, porque de lo contrario se perderá el excedente.
- **public int getLocalPort()**

  Regresa el valor del puerto al que está ligado el *socket*.
- **public synchronized void close()**

  Libera el puerto al que está ligado el *socket*. El puerto también es liberado automáticamente cuando el programa termina su ejecución.
- **public synchronized void setSoTimeout(int timeout) throws SocketException**

  Un valor mayor que cero indica los milisegundos que debe esperar un *socket* para recibir información, ya que una vez transcurrido el tiempo se lanzará una excepción. En caso de ser cero el argumento, la espera nunca expira.

Ejemplo1. Cliente UDP

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
                // args = mensaje y nombre del servidor
                DatagramSocket aSocket = null;
                  try {
                                aSocket = new DatagramSocket();
                                byte [] m = args[0].getBytes();
                                InetAddress aHost = InetAddress.getByName(args[1]);
                                int serverPort = 6789;
                                DatagramPacket request = new DatagramPacket(m,  args[0].length(), aHost, serverPort);
                                aSocket.send(request);
                                byte[] buffer = new byte[1000];
                                DatagramPacket reply = new DatagramPacket(buffer, buffer.length);
                                aSocket.receive(reply);
                                System.out.println("Reply: " + new String(reply.getData()));
                  }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
                  }catch (IOException e){System.out.println("IO: " + e.getMessage());}
                }
}
```

Ejemplo1. Servidor UDP

```
import java.net.*;
import java.io.*;
public class UDPServer{
        public static void main(String args[]){
        DatagramSocket aSocket = null;
          try{
                aSocket = new DatagramSocket(6789);
                byte[] buffer = new byte[1000];
                while(true){
                  DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                  aSocket.receive(request);
                  DatagramPacket reply = new DatagramPacket(request.getData(),
                          request.getLength(), request.getAddress(), request.getPort());
                  aSocket.send(reply);
                }
           }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
           }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
        }
}
```

Ejemplo 2. UDPDiscardServer

```
import java.net.*;
import java.io.*;

public class UDPDiscardServer {

  public final static int discardPort = 9;
  static byte[] buffer = new byte[65507];

  public static void main(String[] args) {

    int port;

    try {
      port = Integer.parseInt(args[0]);
    }
    catch (Exception e) {
      port = discardPort;
    }

    try {
      DatagramSocket ds = new DatagramSocket(port);

      while (true) {
        try {
          DatagramPacket dp = new DatagramPacket(buffer,
        buffer.length);
          ds.receive(dp);
          String s = new String(dp.getData(), 0, dp.getLength());
          System.out.println(dp.getAddress() + " at port " +
        dp.getPort() + " says " + s);
        }
        catch (IOException e) {
          System.err.println(e);
        }
      } // end while
    }  // end try
    catch (SocketException se) {
      System.err.println(se);
    }  // end catch

  }  // end main

}
```

Ejemplo2. UDPDiscardClient

```
import java.net.*;
import java.io.*;

public class UDPDiscardClient {

  public final static int port = 9;

  public static void main(String[] args) {
    String hostname;

    if (args.length > 0) {
      hostname = args[0];
    }
    else {
      hostname = "localhost";
    }
    try {
      String theLine;
      DatagramPacket theOutput;
      InetAddress server = InetAddress.getByName(hostname);
      BufferedReader userInput = new BufferedReader(new
       InputStreamReader(System.in));
      DatagramSocket theSocket = new DatagramSocket();
      while (true) {
        theLine = userInput.readLine();
        if (theLine.equals(".")) break;
        byte[] data = new byte[theLine.length()];
        data = theLine.getBytes();
        theOutput = new DatagramPacket(data, data.length,
       server, port);
        theSocket.send(theOutput);
      }  // end while
    }  // end try
    catch (UnknownHostException e) {
      System.err.println(e);
    }
    catch (SocketException se) {
      System.err.println(se);
    }
    catch (IOException e) {
      System.err.println(e);
    }

  }  // end main
}
```

Práctica de programación (PP-01)

Para los ejemplos 1 y 2 de los Sockets UDP:

- Compila y ejecuta los códigos de los programas cliente y servidor.

- Entrega un reporte que incluya una descripción detallada de las operaciones que cada programa realiza y elabora un diagrama de secuencia que muestre la interacción entre cliente y servidor.

- Modifica los códigos del cliente y del servidor para implementar un conversor de temperaturas, donde el cliente envía una lectura en grados Celsius en el intervalo [-200,200] y el servidor responde con la temperatura equivalente en grados Fahrenheit.