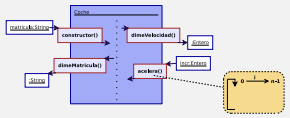
Grado en Ingeniería de Sistemas de Telecomunicación, Sonido e Imagen

Programación 2



Práctica 3

UNIVERSIDAD POLITECNICA DE VALENCIA

Escola Politècnica Superior de Gandia

DSIC

Departament de Sistemes Informàtics i Computació

Contenido

Sesiones

- Grupos lunes: abr-29, may-6, may-13, may-20
- Grupos viernes: abr-26, may-3, may-17, may-24

Examen escrito: jun-3 (10-12h)

Objetivos

- Utilización de clases de biblioteca.
- Ingeniería inversa del código de una clase.
- Implementación de métodos dados su diseño y su algoritmo.
- Desarrollo de programas cliente-sevidor.

¡ Atención !

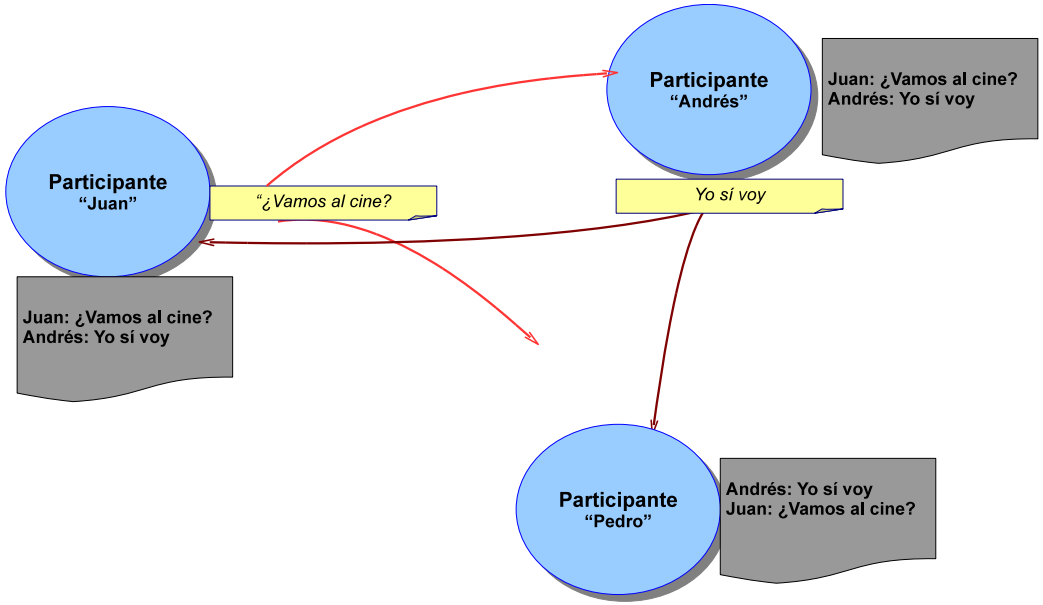
▷ Se recuerda que las prácticas deben prepararse antes de acudir al aula informática. Esto incluye leer el código proporcionado y probarlo.

▷ La realización de las prácticas es un trabajo individual y original. En caso de plagio se excluirá al alumno de la asignatura. Por tanto es preferible presentar el trabajo realizado por uno mismo aunque éste tenga errores.

1 Conceptos de aplicaciones distribuidas relacionadas con un chat.

Un chat es una aplicación distribuida que permite que varias personas mantengan una conversación intercambiando mensajes de texto cortos enviados simultáneamente desde distintos ordenadores. Estos mensajes son inmediatamente distribuidos a todos los participantes, por lo que, en principio, no son almacenados para su posterior consulta.

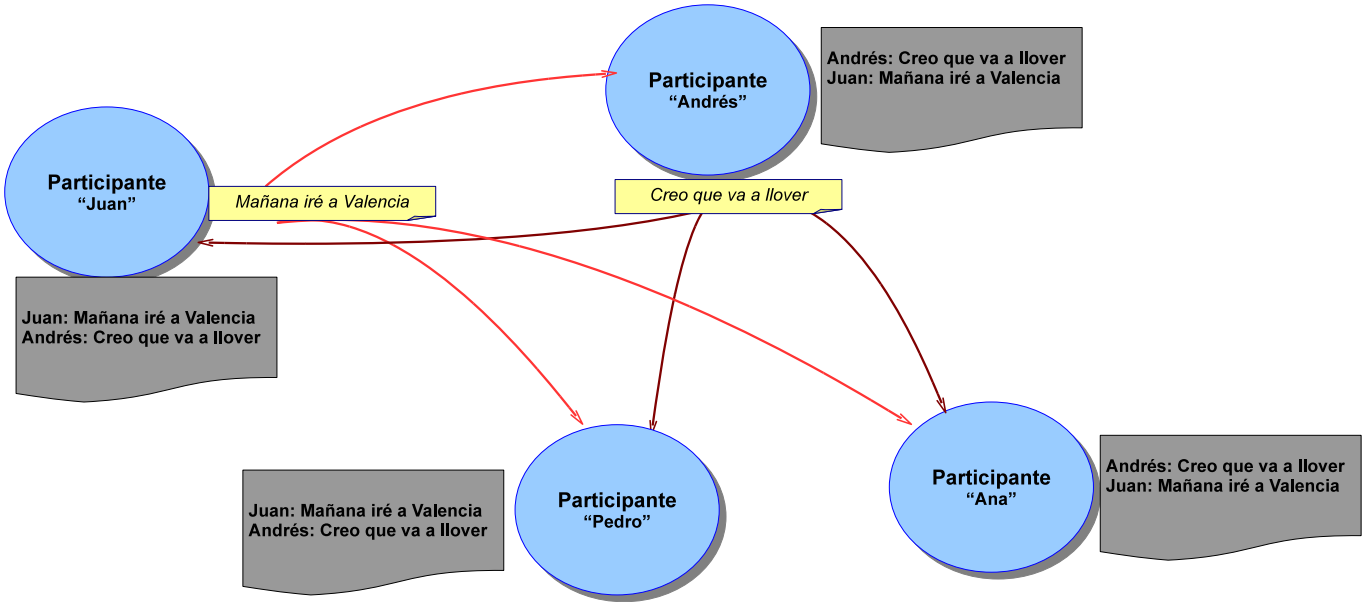
Un chat necesita un mecanismo de difusión (*broadcast*) que entregue los mensajes que envía un participante al resto de participantes. Para garantizar que una conversación es consistente, hay que considerar el orden en que el mecanismo de difusión entrega los mensajes. Por ejemplo, si el mecanismo de difusión no da ninguna garantía sobre el orden de entrega, podría ocurrir algo como lo siguiente, en donde uno de los participantes, "Pedro", recibe antes una respuesta que la pregunta corespondiente.



Evidentemente, la exigencia de entrega más estricta, conocida como difusión *atómica* es la que dicta que todos los mensajes se entregen *exactamente* en el mismo orden a todos los participantes.

Hay un nivel de exigencia menor que la atómica, que permite que los mensajes puedan, en algunos casos, entregarse en orden diferente a distintos participantes, pero sin afectar a la consistencia de la conversación. Se denomina difusión *causal* y consiste, resumidamente, en que si un participante p_e envía un mensaje m, entonces m no puede ser entregado a otro participante p_r hasta que todos los mensajes vistos o enviados por p_e *antes* de la emisión de m hayan sido entregados a p_r.

Con difusión causal no ocurrirá el problema expuesto en el diagrama anterior[1], pero sí el caso siguiente que, aún no siendo un problema, no se daría con difusión atómica[2].



[1] ¿Por qué?
[2] ¿Por qué?

Las consideraciones de consistencia son, como vemos, muy importantes a la hora de diseñar una aplicación distribuida. La difusión atómica, que es fácil de entender y utilizar, es difícil de resolver a no ser que se utilice un servidor central que reciba los todos los mensajes y después los reparta debidamente ordenados. En el campo teórico de las aplicaciones distribuidas, utilizar un servidor central es un último recurso que siempre se intenta evitar, porque es un punto de fallo importante para el sistema. Si el servidor central falla, la aplicación distribuida se detiene. Si no queda otra alternativa que utilizar un servidor y se necesita *tolerancia a fallos* hay que replicar dicho servidor.

Como ya expusimos en la práctica anterior, una aplicación distribuida en que no hay ningún servidor central[3] es claramente una aplicación p2p. Sin embargo, para la realización nuestro chat, por sencillez, sí utilizaremos un servidor. Aún así, nuestra aplicación no será estrictamente cliente-servidor porque, aunque tengamos un servidor, un participante en el chat no se comporta siempre como cliente.

Efectivamente, al enviar un mensaje, un participante es un cliente porque *inicia* la acción de envío. Pero cuando el servidor recibe un mensaje para distribuirlo, es él quien *inicia* el envío, sin que los participantes hayan pedido nada. Esta acción del servidor se conoce como *empuje* (*push*), y al realizarla, un servidor, se comporta más bien como cliente.

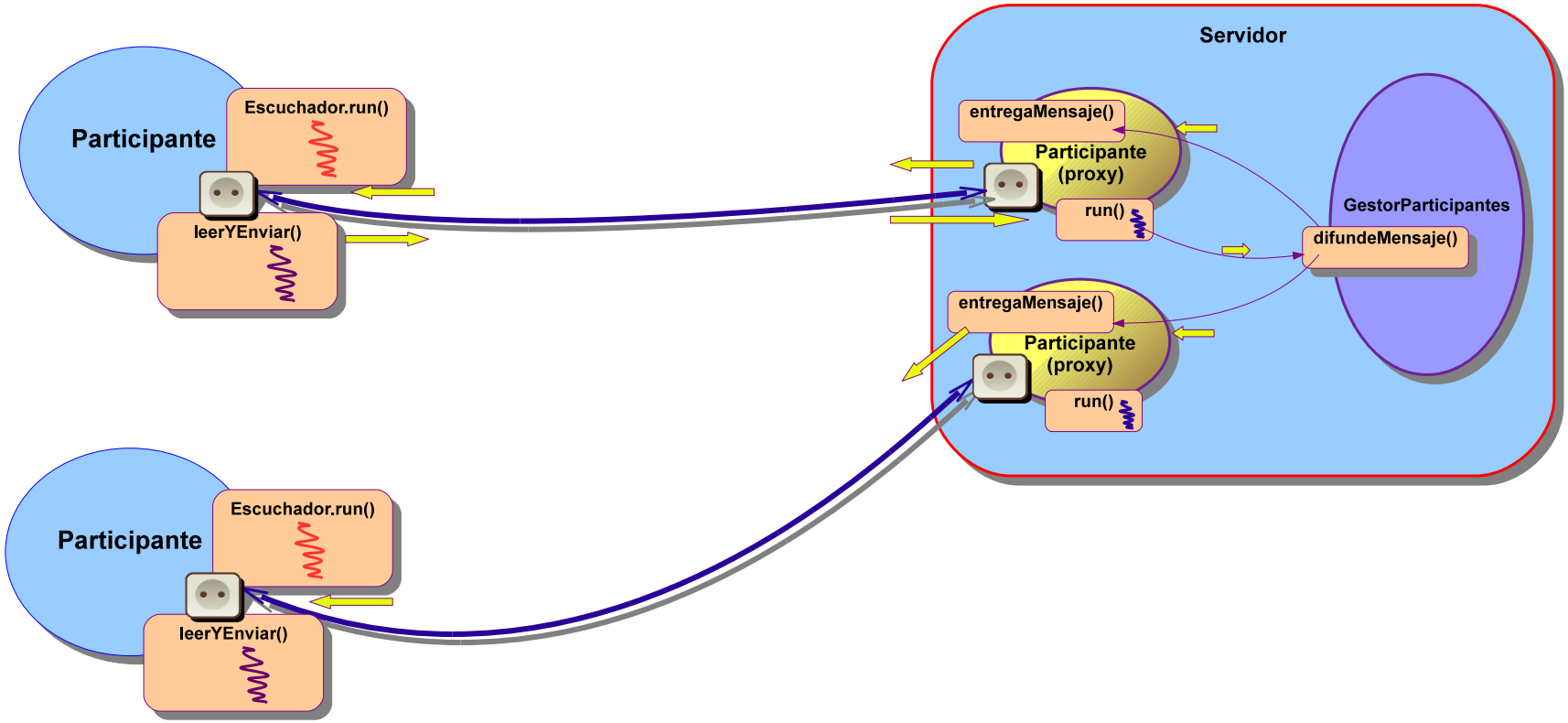
Si se quisiera conservar el modelo cliente-servidor, entonces, el servidor no puede realizar acciones push para distribuir mensajes. En este caso, la alternativa es que los participantes pregunten periódicamente al servidor si tiene mensajes pendientes para ellos y, en caso afirmativo, que se los devuelva como respuesta a la pregunta. Esta estrategia se conoce como *enquesta* (*polling*).

2 Desarrollo del chat en Java

Basaremos nuestro diseño en un servidor central que utilice push para distribuir mensajes. Como hemos dicho, esta decisión afecta a los participantes porque no son sólo clientes. Un participante debe debe tener dos threads. Un thread se encarga de leer los mensajes que el usuario escribe en el teclado y de remitirlos a continuación al servidor. Otro thread se encarga de leer los mensajes que el servidor le envia para

[3] En realidad, si no hay ningún proceso que sólo sea servidor.

mostrarlos en pantalla. Si como mecanismo de comunicación utilizamos sockets TCP, el esquema de la aplicación será el siguiente.

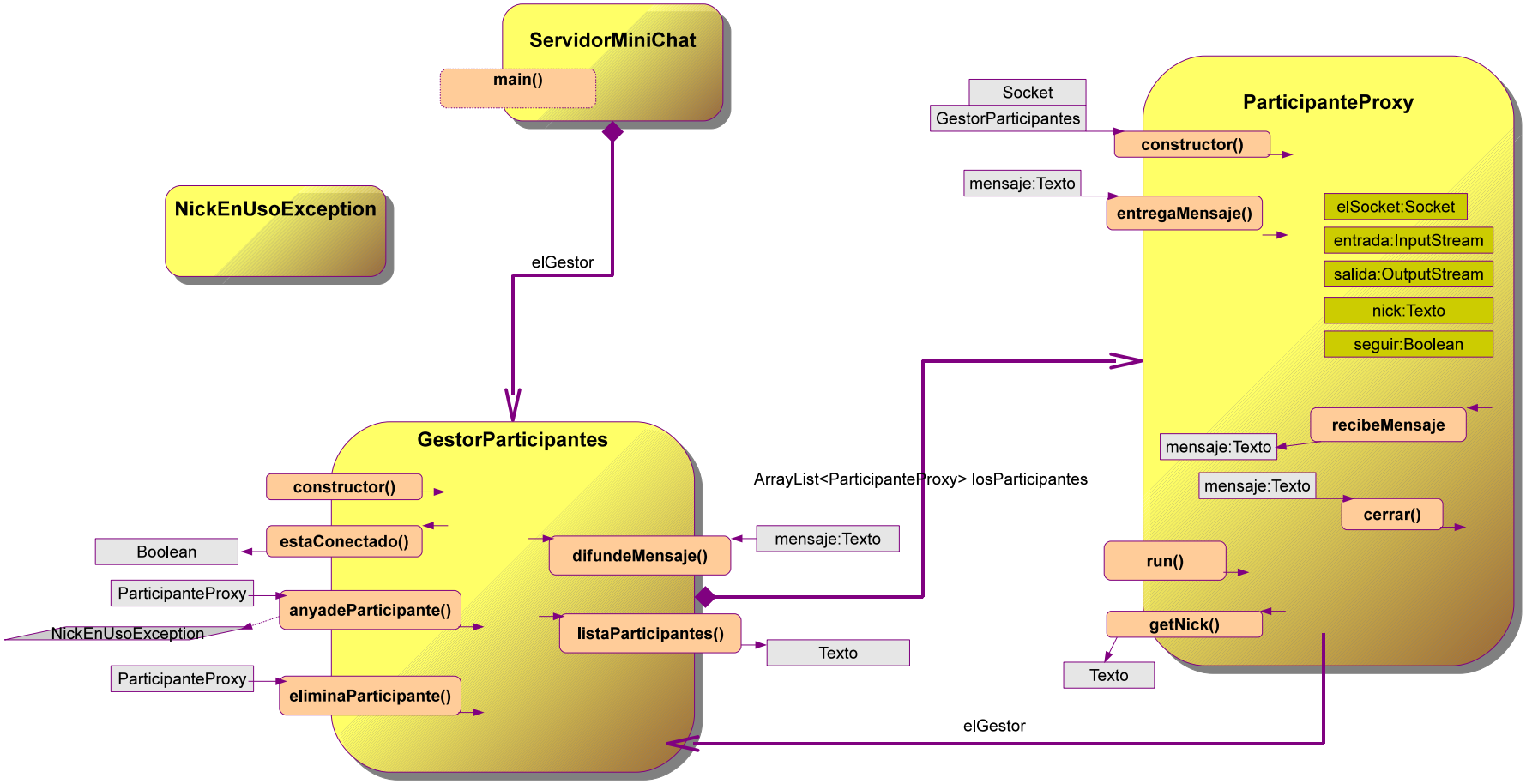


En el servidor tendremos un objeto "participante proxy". que represente a cada uno de los participantes en el chat, así como un objeto para gestionar dicha colección de representantes.

En cada proxy, un thread esperará que lleguen mensajes del participante remoto al que representa. Cuando un proxy recibe un mensaje, llama un método para difundir el mensaje quien a su vez, y para todos los proxies, llama a un método para que cada proxy entregue el mensaje a su representado.

2.1 Lado del servidor

El diagrama de clases correspondiente al proceso servidor es el siguiente[4].



[4] La punta de flecha que aparece en el diagrama no es de herencia. Sólo indica que una clase tiene una referencia a otra

2.1.1 Clase ServidorMiniChat

El algoritmo correspondiente a ServidorMiniChat.main() es este.

```
escribir la dirección IP y el número de puerto donde residirá el servicio
ss ← nuevo socket servidor
elGestor ← nuevo GestorParticipantes
repetir  por siempre
    Socket s ← conexión de un nuevo cliente aceptada en ss
    nuevo ParticipanteProxy(s, elGestor)
```

2.1.2 Clase ParticipanteProxy

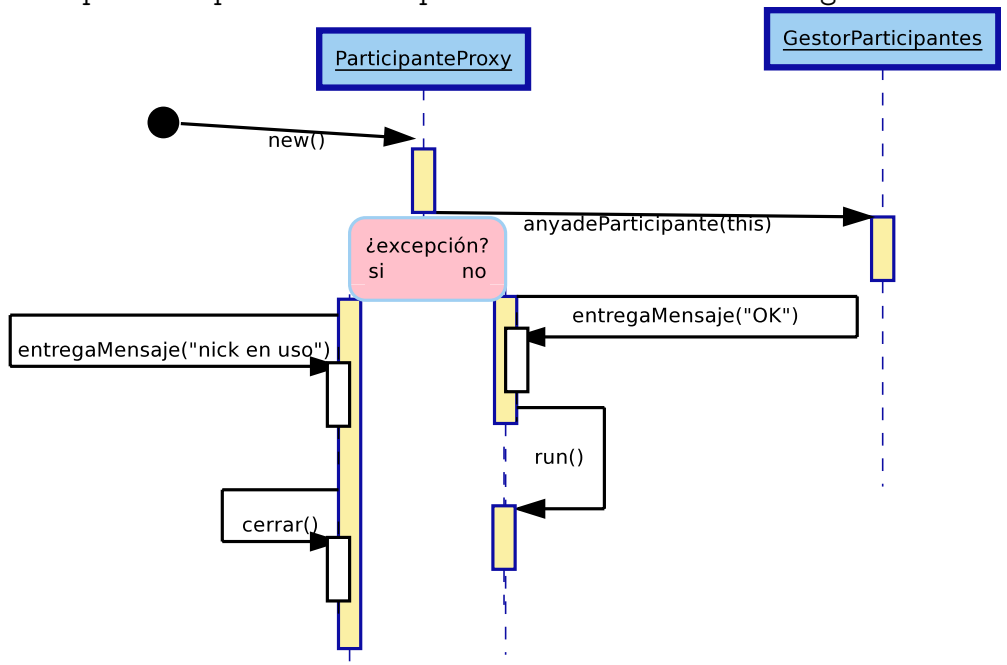
Ya sabemos que un objeto de tipo ParticipanteProxy representa a un participante remoto. Los algoritmos de sus métodos son los siguientes.

- Algoritmo de ParticipanteProxy.constructor(Socket con, GestorParticipantes ges).

```
en campos del objeto guardar ges y los streams (input y output) sacados de con

recibir el primer mensaje (línea de texto) que envia el participante a quien representamos. El mensaje es su nombre (nick): nick ← recibeMensaje()

probar (try) a añadirme (this) como participante a this.elGestor, capturando la excepción NickEnUsoException ex, en cuyo caso
```

```
    decir a mi participante que el nick ya se está usando: entregaMensaje("nick ya utilizado")
    llamar a cerrar() sin mensaje (null)
    terminar (return)

enviar a mi participante "OK"
hacer que arranque un thread en run()

si durante los pasos anteriores se disparara IOException, entonces llamar a cerrar() con mensaje
"se retira " + this.nick
```

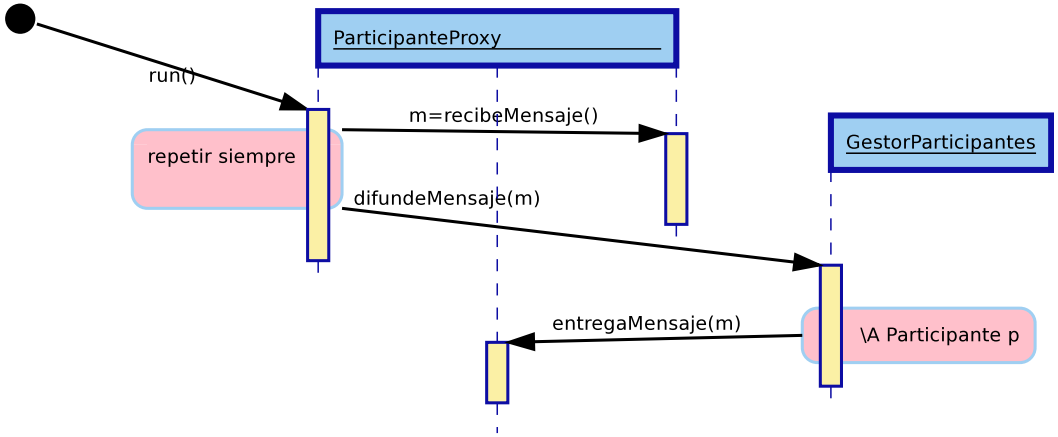
El anterior algoritmo se puede representar esquemáticamente con un diagrama de secuencia de UML.



- Algoritmo de ParticipanteProxy.run(), ejecutado por un thread nuevo, que espera mensajes de un participante para difundirlos a los demás.

```
llamar a elGestor para que difunda el mensaje "se conecta " + this.nick
enviar al participante al que representamos la lista de conectados al chat
repetir mientras this.seguir = verdadero
    mensaje ← recibir mensaje
    si mensaje es "SALIR" llamar a cerrar() con mensaje "se retira " + this.nick
    si no, difundir mensaje
```

El algoritmo, resumido en UML, es



- Para entregar un mensaje, el código es el siguiente:

```
public void entregaMensaje(String mensaje) {
 if (mensaje == null || mensaje.length() == 0) {
  return;
 }
 try {
```

```
        IO.escribeLinea(mensaje, this.salida);
       } catch (IOException ex) {
        this.cerrar("chat> se retira por fallos en la conexión (entregaMensaje()): " + this.nick);
       }
      }
```

- Para recibir un mensaje, el código es el siguiente:

```
     private String recibeMensaje() {
      try {
       return IO.leeLinea(this.entrada);
      } catch (IOException ex) {
       this.cerrar("chat> se retira por fallos en la conexión (recibeMensaje()): " + this.nick);
      }
      return "";
     } // ()
```

- Algoritmo de `ParticipanteProxy.cerrar(String mensaje)`.

```
detener el thread de run() (poniendo seguir a falso)
eliminarme como participante de elGestor
cerrar los streams de entrada y de salida, y el socket
si  mensaje ≠ null y longitud de mensaje no es 0
    |
    |  llamar a elGestor para que difunda mensaje
    |  llamar a elGestor para que difunda quién está conectado
    |
    |__
```

2.1.3 Clase GestorParticipantes

Esta clase se encarga de mantener una lista de los participantes actualmente conectados. Su código incompleto es el siguiente.

```
public class GestorParticipantes {

 private ArrayList<ParticipanteProxy> losParticipantes;

 public GestorParticipantes() {
  this.losParticipantes = new ArrayList<ParticipanteProxy> ();
 }

 private  synchronized  ParticipanteProxy  buscaParticipante(String nick)
    { ... }

 public synchronized boolean estaConectado (String nick)
    { ... }

 public synchronized void anyadeParticipante (ParticipanteProxy p)
  throws NickEnUsoException
    { ... }

 public synchronized void eliminaParticipante (ParticipanteProxy p)
    { ... }

 public synchronized void difundeMensaje (String s)
    { ... }
```

```
    public synchronized String listaParticipantes () {
     StringBuilder sb = new StringBuilder ();
     for (ParticipanteProxy p : this.losParticipantes) {
      sb.append(p.getNick()).append(" ");
     }
     return sb.toString();
    }
   } // class
```

Véase que los métodos tienen el adjetivo synchronized para que no pueda haber más de un thread (venido de objetos ParticipanteProxy) ejecutando algún método de este objeto, ya que podrían darse condiciones de carrera.

Hay que crear una clase NickEnUsoException para que public synchronized void anyadeParticipante (ParticipanteProxy p) throws NickEnUsoException lance una excepción en caso de que ya se haya dado de alta un participante con el mismo nick que p.

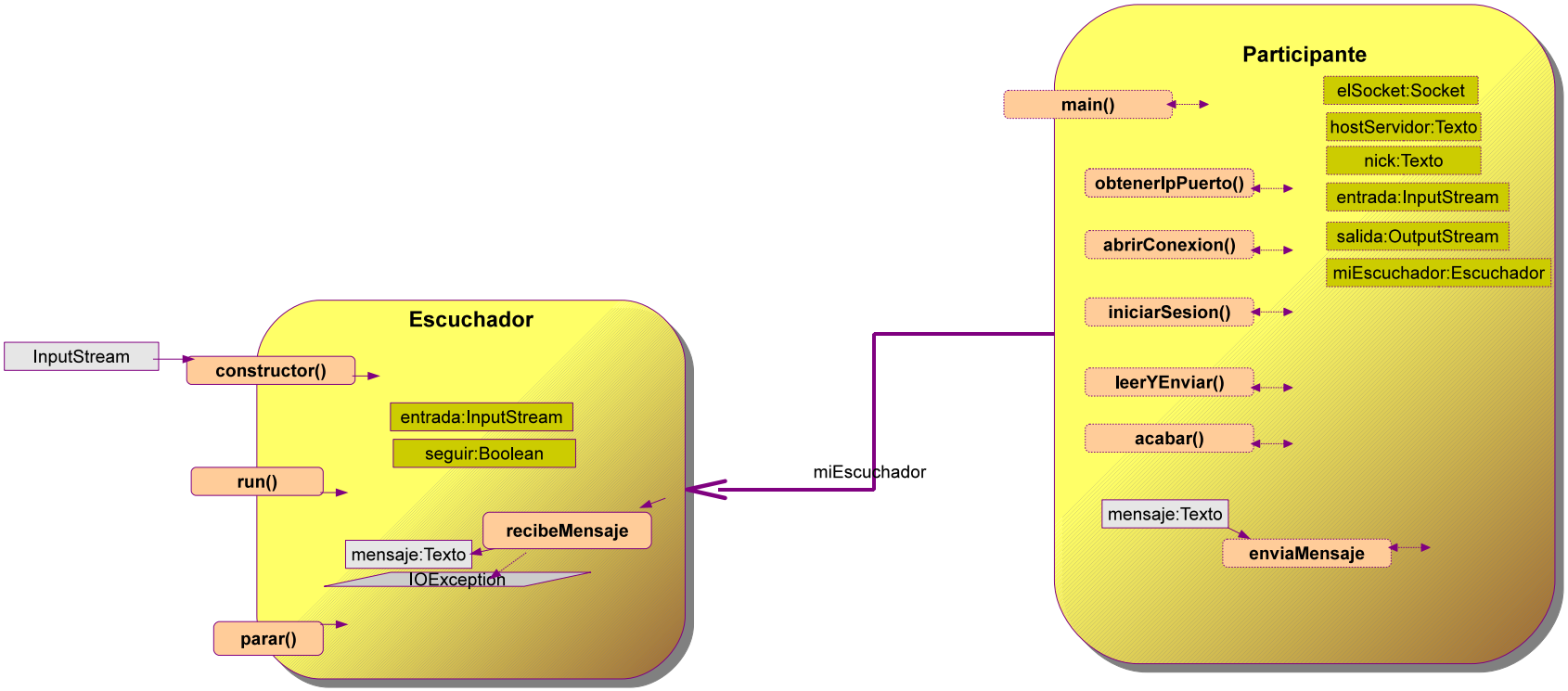
Finalmente, recordemos que difundeMensaje() debe llamar, para todo participante apuntando losParticipantes, el método entregaMensaje().

Ejercicio

Implementa las anteriores clases en el paquete chat.servidor.

2.2 Lado del participante

En el participante[5] encontramos las siguientes clases.



[5] Recordemos que no lo llamamos cliente porque estrictamente no lo es, ya que ha de recibir mensajes mediante push del servidor (sin que los haya pedido).

2.2.1 Clase Participante

Esta clase es la que tiene el main() del programa que utiliza el usuario. Básicamente tiene que preguntarle la dirección IP, el número de puerto y el nick que quiere utilizar. Con esos datos abre un socket al servidor y le envía el nick. Si el servidor acepta que el usuario entre en el chat (el único motivo para no hacerlo es que ya haya un usuario utilizando el mismo nick), creará un objeto Escuchador encargado de esperar mensajes del servidor. Después el thread de main() llama a leerYEnviar() que lee mensajes escritos por el usuario y los envía al servidor para su difusión. El código correspondiente al anterior algoritmo se ha repartido entre varias funciones[6] para que main() sea más legible. Hay sin embargo un mal menor que es el abuso de variables estáticas globales mediante las cuales, dichos métodos se comunican.

A continuación, damos algunos comentarios sobre las anteriores funciones.

- obtenerIpPuerto() simplemente ha de preguntar al usuario la dirección y el número de puerto donde está el servidor, mediante

  ```
  String  linea =  Utilidades.leerTextoG("dime localizador  del servidor (ip:puerto)");
  ```

  El usuario escribirá algo como

  ```
  192.165.2.8:1234
  ```

  por lo que tendremos que trocear linea para obtener String hostServidor y el int PUERTO.
  Como en todas las demás funciones, si se produce algún error u excepción hay que dar algún aviso, por ejemplo

  ```
  Utilidades.muestraMensajeG("no he entendido ip:puerto=" + linea);
  ```

  y luego llamar a acabar().
- abrirConexion() ha de crear un socket, guardándolo en elSocket, y extraer de él entrada y salida. Si se produce algún fallo, avisar y llamar a acabar().
- iniciarSesion() pregunta al usuario su nick

[6] Métodos estáticos.

```
    nick = Utilidades.leerTextoG("dime nick");
```

y lo envía (con enviaMensaje() al servidor. A continuación lee la respuesta (IO.leeLinea()) y en caso que no sea "OK" llama a acabar.

- leerYEnviar() tiene el siguiente pseudo-código.

```
String linea;
do {
        // lee un mensaje del usuario
        linea = Utilidades.leerTextoG("¡Escribe un mensaje, " + nick + " ! ");

         si linea no es null, la longitud de linea es mayor que 0 y linea no es "SALIR" {
            enviar linea
         }

} mientras linea no sea null ni contenga "SALIR";

enviar al servidor "SALIR"
} // ()
```

- El código completo de enviaMensaje() es

```
public static void enviaMensaje(String mensaje) {
 if (mensaje == null || mensaje.length() == 0) {
  return;
 }
 try {
  IO.escribeLinea(mensaje, salida);
 } catch (IOException ex) {
  // si no puedo enviar mensajes, acabo
  acabar();
```

```
      }
     }
```

- El código completo de acabar() es

```
    static void acabar () {
      if (elSocket == null) {
       // Si elSocket es null, simplemente acabo.
       // Lo compruebo, por si llaman
       // a acabar() antes haber abierto la conexión
       //
       System.exit(0);
      }
      try {
       // si el socket no es null  cierro todo y paro el
       // thread del escuchador
       miEscuchador.parar();
       entrada.close();
       salida.close();
       elSocket.close();
       elSocket = null;

       Utilidades.muestraMensajeC("* Hasta luego !");
       System.exit(0);
      } catch (Exception ex) {
      }

     } // ()
```

2.2.2 Clase Escuchador

Participante.main() crea un objeto Escuchador cuando el usario ha sido aceptado en el chat. En Escuchador.run() un thread esperará que lleguen mensajes para mostrarlos en pantalla.

- El constructor debe guardar el InputStream que recibe en entrada y arrancar un thread de run.
- recibeMensaje() es muy simple:

```
private String recibeMensaje() throws IOException {
  return IO.leeLinea(this.entrada);
 } // ()
```

- El pseudo-código de run() es

```
mientras seguir {
    String linea = recibir un mensaje
    Utilidades.muestraMensajeC(linea); // mostrarlo
}

si se produce alguna óexcepcin, llamar a Participante.acabar()
```

- parar() solamente es poner seguir a falso.

Ejercicio

Implementa las anteriores clases en el paquete chat.participante.

3 Entregas

3.1 Trabajo de lectura, pruebas, e implementación

- Antes de la primera sesión hay que leer el enunciado completo de la práctica. Y buscar en la práctica anterior el código Java para determinadas tareas, como por ejemplo crear threads, o trocear líneas de texto.
- Durante la primera y segunda sesión hay que implementar el lado del servidor.
- Durante la tercera y cuarta sesión hay que implementar el lado del participante y realizar las pruebas.

3.2 Código completo de la práctica

El fichero p3.zip con el código completo se debe entregar electrónicamente antes del 5 de junio.

Jordi Bataller Mascarell 22 abril 2013