Concurrencia
Prácticas 1 y 2

Grado en Ingeniería Informática/ Grado en Matemáticas e Informática/ 2ble. grado en Ing. Informática y ADE
Convocatoria de Semestre feb–jun 2017–2018

Normas

- La fecha límite de la entrega preliminar (no obligatoria) de la Práctica 1 es el **27 de mayo de 2018** a las 23:59:59. Los resultados de someter a más pruebas y revisión las entregas anteriores a dicha fecha se publicarán el día **31 de mayo**.
- La fecha límite de la entrega preliminar (no obligatoria) de la Práctica 2 es el **3 de junio de 2018** a las 23:59:59 y los resultados se publicarán el día **7 de junio**.
- Después de estas entregas preliminares revisaremos los sistemas de entrega y publicaremo nuevas pruebas en el sistemas de entrega para ambas prácticas.
- **Todas prácticas**, incluso las en que no se han detectado problemas, deben ser entregadas de nuevo antes de la fecha límite que será el **15 de junio de 2018** a las 23:59:59.
- La práctica se realiza en grupos de dos alumnos. Los ficheros entregados (`QuePasaMonitor.java` y `QuePasaCSP.java`) deberán tener un comentario

  ```
  //Grupo: Nombre Alumno 1 (Matrícula Alumno 1), Nombre Alumno 2 (Matrícula Alumno 2)
  ```

  en el principio de los ficheros, como definición del grupo. **Nota: sólo entrega un miembro por grupo**, no deben entregar ambos alumnos de un grupo.
- Deberá mencionarse explícitamente el uso de recursos (código, algoritmos específicos, esquemas de implementación, etc.) que no hayan sido desarrollados por el alumno o proporcionados como parte de asignaturas de la carrera.
- Os recordamos que **todas** las prácticas entregadas pasan por un proceso automático de detección de copias.

1. ¿QuéPasa?

Después de haber gastado demasiado tiempo usando *WhatsApp*, hemos decidido que es un momento oportuno para desarrollar una aplicación parecida, pero con un nombre infinitamente mejor: *¿QuéPasa?*. Al igual que WhatsApp, la aplicación ¿QuéPasa? permite formar grupos de usuarios y mandar mensajes a los miembros de un grupo.

1.1. Diseño

El sistema de gestión de ¿QuéPasa? se ha diseñado como un único recurso compartido con operaciones para:

- crear un grupo nuevo,

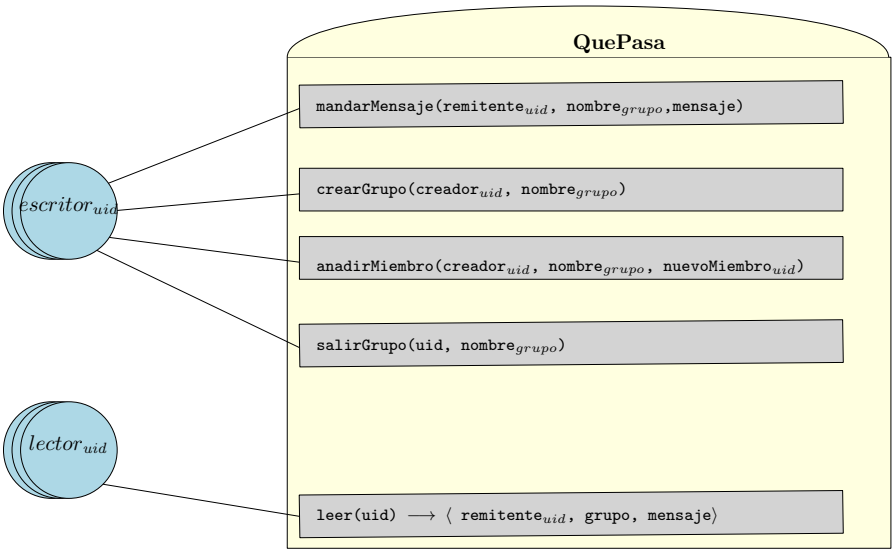

Figura 1: Grafo de procesos/recursos.

- añadir un miembro nuevo a un grupo,
- salir de un grupo,
- mandar un mensaje a un grupo y
- leer mensajes.

Existen algunas diferencias no muy importantes entre WhatsApp y ¿QuéPasa?:

- No se permiten diferentes grupos con el mismo nombre.
- El creador de un grupo es el único usuario que puede añadir más usuarios.
- El creador de un grupo siempre será miembro del grupo, es decir, nunca podrá salir del grupo.

La aplicación ¿QuéPasa? consta de un recurso compartido — que debéis implementar — y, por cada usario hay dos procesos (o hilos de ejecución): un *lector* que intenta leer mensajes y un *escritor* que maneja el resto de la funcionalidad de ¿QuéPasa?: crear grupos, añadir un miembro a un grupo, mandar mensajes, etc. La arquitectura de ¿QuéPasa? queda reflejada en el grafo de procesos y recursos de la figura 1.

1.2. Especificación formal del recurso

La especificación formal del recurso **¿QuePasa?** se muestra en la figura 2.

1.2.1. Explicaciones

Los identificadores de usuarios se representan con números naturales: $UID = \mathbb{N}$. Los nombres de los grupos son cadenas de caracteres ($Grupo = String$), al igual que el contenido de los mensajes. El dominio del recurso compartido se ha representado con una tupla de tres componentes: *miembros*, *creador* y *mensajes*.

C-TAD QuePasa
OPERACIONES
ACCIÓN crearGrupo: *UID[e]* × *Grupo[e]*
ACCIÓN anadirMiembro: *UID[e]* × *Grupo[e]* × *UID[e]*
ACCIÓN salirGrupo: *UID[e]* × *Grupo[e]*
ACCIÓN mandarMensaje: *UID[e]* × *Grupo[e]* × *Contenido[e]*
ACCIÓN leer: *UID[e]* × *(UID*× *Grupo*× *Contenido)[s]*

SEMÁNTICA
DOMINIO:
TIPO: ***QuePasa*** = ($creador : Grupo \nrightarrow UID \times$
$miembros : Grupo \nrightarrow \mathsf{Conjunto}(UID) \times$
$mensajes : UID \nrightarrow \mathsf{Secuencia}(UID \times Grupo \times Contenido)$)
DONDE: UID = $\mathbb{N}$
Grupo = String

INICIAL: self.*creador* = {} $\wedge$ self.*miembros* = {} $\wedge$ self.*mensajes* = {}
INVARIANTE: *dom*(self.creador) = *dom*(self.miembros)

PRE: $grupo \notin dom(\text{self}.creador)$
CPRE: Cierto
crearGrupo(uid, grupo)
POST: $\text{self}^{pre} = (c, m, s) \wedge$
$\text{self} = (c \cup \{grupo \mapsto uid\}, m \cup \{grupo \mapsto \{uid\}\}, s)$

PRE: $\text{self}.creador(grupo) = creadoruid \wedge uid \notin \text{self}.miembros(grupo)$
CPRE: Cierto
anadirMiembro(creadoruid, grupo, uid)
POST: $\text{self}^{pre} = (c, m, s) \wedge$
$\text{self} = (c, m \oplus \{grupo \mapsto m(grupo) \cup \{uid\}\}, s)$

PRE: $uid \in \text{self}.miembros(grupo) \wedge uid \neq \text{self}.creador(grupo)$
CPRE: Cierto
salirGrupo(uid, grupo)
POST: $\text{self}^{pre} = (c, m, s) \wedge$
$borrados = \{(u, g, x) \in s(uid) \mid g = grupo\} \wedge$
$\text{self} = (c, m \oplus \{grupo \mapsto m(grupo) \setminus \{uid\}\}, s \oplus \{uid \mapsto s(uid) \upharpoonright borrados\})$

PRE: $uid \in \text{self}.miembros(grupo)$
CPRE: Cierto
mandarMensaje(uid, grupo, contenido)
POST: $self^{pre} = (c, m, s) \wedge self = (c, m, s') \wedge$
$\forall u \in UID \bullet u \notin m(grupo) \Rightarrow s'(u) = s(u) \wedge$
$u \in m(grupo) \Rightarrow s'(u) = s(u) + \langle (uid, grupo, contenido) \rangle)$

CPRE: $\text{self}.mensajes(uid) \neq \langle\rangle$
leer(uid, msg)
POST: $\text{self}^{pre} = (c, m, s) \wedge pendientes = s(uid) \wedge msg = pendientes(1)$
$\text{self} = (c, m, s \oplus \{uid \mapsto s(2..\mathsf{Longitud}(pendientes))\})$

Figura 2: Especificación formal del recurso.

- El componente *miembros* : *Grupo* $\nrightarrow$ Conjunto(*UID*) es una funcion parcial (*map*) de nombres de grupos en conjuntos de identificadores usuario. Al ser una *función parcial*, no asocia necesariamente todos los posibles nombres de grupos a un conjunto de UID. Informalmente, *miembros* devuelve los miembros de un grupo. Como funcion parcial, necesitamos comprobar si está "definida" para un grupo usando la sintaxis $grupo \in dom(miembros)$ (véase por ejemplo la precondición de la operación **crearGrupo**).

- El componente *creador* : *Grupo* $\nrightarrow$ *UID* asocia un grupo con el usuario que lo ha creado.

- El componente *mensajes* : *UID* $\nrightarrow$ Secuencia(*UID* $\times$ *Grupo* $\times$ *Contenido*) declara una funcion parcial que asocia un identificador de usuario con una sequencia de tuplas compuesta por otro identificador de usuario (quién envió el mensaje), un grupo (dónde se publicó el mensaje) y el contenido del mensaje.

Existen algunas operaciones con precondiciones (**PRE**) que consideran tanto los parámetros como el estado del recurso. Si vuestra implementación detecta que no se cumple una precondición, el recurso **debe** lanzar la excepción `PreconditionFailedExcepcion`. Dicha excepción se encuentra disponible en el código auxiliar de la práctica.

Las postcondiciones son similares. Por ejemplo, en la postcondición para la operación **crearGrupo** se define que la función *creador*(*grupo*) debería devolver *uid* despues de la operación, y que *miembros*(*grupo*) devuelve el conjunto {*uid*}, y *mensajes* no cambia.

Todas las operaciones del recurso deberían respetar la invariante, es decir, si se cumple antes de la invocación, debería cumplirse después de la ejecución. Notad que *no es obligatorio* implementar la invariante en vuestra implementación del recurso, pero puede ser útil para comprobar que el programa es correcto.

Notad que la operación de recibir mensajes (**leer**) tiene un parámetro de entrada *UID*[*e*] y un parámetro de salida que resulta ser una tupla *UID* $\times$ *Grupo* $\times$ *Contenido*. En la implementación en Java se representa dicha operación como un método **Mensaje leer(int uid)** que dado un `uid` devuelve un objeto de tipo `Mensaje`, que contiene los tres datos de salida.

En la **POST** de **salirGrupo** usamos la sintaxis $S \upharpoonright R$, donde S es una sequencia y R un conjunto, para construir una nueva sequencia derivada de S donde los elementos que *están en el conjunto R han sido borrados*. En la práctica significa que **salirGrupo(uid,grupo)** borra los mensajes con origen en el grupo **grupo** para el usario **uid**.

2. Prácticas

2.1. Primera práctica

La entrega consistirá en una implementación del recurso compartido en Java usando la clase `Monitor` de la librería `cclib`. La implementación a realizar debe estar contenida en un fichero llamado `QuePasaMonitor.java` que implementará la interfaz `QuePasa`. (ver sec. 3). **Nótese** que la clase `QuePasaMonitor` debe estar en el paquete `cc.qp`, es decir, el fichero `QuePasaMonitor` debe empezar con la declaración "`package cc.qp;`".

2.2. Segunda práctica

La entrega consistirá en una implementación del recurso compartido en Java mediante paso de mensajes síncrono, usando la librería JCSP. La implementación deberá estar contenida en un fichero llamado `QuePasaCSP.java` que implementará la interfaz `QuePasa`. (ver sec. 3). **Notad** que la clase `QuePasaCSP`

deberia estar en el paquete `cc.qp`, es decir el fichero `QuePasaCSP` deberia empezar con la declaración "`package cc.qp;`".

3. Información general

La entrega de las prácticas se realizará **vía WWW** en la dirección `http://lml.ls.fi.upm.es/entrega`.

El código que finalmente entreguéis (tanto para memoria compartida como para paso de mensajes) no debe realizar **ninguna** operación de entrada/salida.

Para facilitar la realización de la práctica están disponibles en `http://babel.upm.es/teaching/concurrencia` varias unidades de compilación:

- `QuePasa.java`: define la interfaz común a las distintas implementaciones del recurso compartido.
- `Mensaje.java`: define los mensajes creados en el metodo `mandarMensaje` en vuestra implementación del recurso, y devuelto por el metodo `leer`.
- `PreconditionFailedException.java`: representa una excepcion que vuestra implementacion del recurso deberia lanzar cuando no se cumple una precondition.
- Para poder simular vuestra implementación está disponible los ficheros:
  - `SimulateQuePasa.java` – para ejecutar una simulacro con un numero de usarios simulados
  - `SimulatedLector.java` – implementa un proceso usario que lee mensajes
  - `SimulatedEscritor.java` – implementa un proceso usario que manda mensajes, y tambien crea grupos, sale grupos, y anadé miembros a grupos.
- `QuePasaCSP.java`: esqueleto para la práctica 2, que incluirá la mayor parte del código *vernacular* de JCSP.

Por supuesto durante el desarrollo podéis cambiar el código que os entreguemos para hacer diferentes pruebas, depuración, etc., pero el código que entreguéis debe poder compilarse y ejecutar sin errores junto con el resto de los paquetes entregados **sin modificar estos últimos**. Podéis utilizar las librerías estandar de Java y las librerias auxiliares que estén disponibles para asignaturas previas (p.ej. la librería aedlib.jar de Algoritmos y Estructuras de Datos disponible en `http://lml.ls.fi.upm.es/~entrega/aed/aedlib`), para la definición de estructuras de datos.

El programa de recepción de prácticas podrá rechazar entregas que:

- Tengan errores de compilación.
- Utilicen otras librerías o aparte de las estándar de Java y las que se han mencionado anteriormente.
- No estén suficientemente comentadas. Alrededor de un tercio de las líneas deben ser comentarios **significativos**. No se tomarán en consideración para su evaluación prácticas que tengan comentarios ficticios con el único propósito de rellenar espacio.
- No superen unas pruebas mínimas de ejecución, integradas en el sistema de entrega.