Sistemas Distribuidos

Escuela Superior de Informática

DROBOTS: CROBOTS distribuido

David.Villa@uclm.es
joseluis.segura@uclm.es

ISRN ESI/--2017/SSDD-1617

versión 2017.02.27

Índice

Este documento constituye la especificación del entregable para la evaluación del laboratorio de la asignatura de Sistemas Distribuidos. El alumno debe construir una aplicación distribuida conforme a la siguiente especificación. La realización es individual y la entrega obligatoria.

1. Introducción

La tarea del alumno consiste en desarrollar algunos componentes de la aplicación DROBOTS. Se trata de una modalidad distribuida de CROBOTS. El CROBOTS [Poi85] original es un juego muy antiguo[1] en el que compiten varios robots en un campo de batalla acotado. En este juego no hay interacción directa entre el usuario y el juego. El «jugador» es un programador que escribe la lógica de comportamiento de un robot, con el objetivo de destruir al resto de los robot presentes en el campo. El programa original consistía en un compilador y una máquina virtual que ejecutaba dichos programas. Según la descripción de la documentación original:

CROBOTS es un juego basado en programación de computadores. A diferencia de los juegos tipo arcade que requiere interacción con un humano para controlar algún objeto, toda la estrategia en CROBOTS está especificada antes del comienzo del juego. La estrategia del juego está condensada en un programa C que tú diseñas y escribes. Tu programa controla un robot cuya misión es buscar, perseguir y destruir otros robots, cada uno de ellos bajo el control de programas diferentes en ejecución. Cada robot está igualmente equipado, y hasta un máximo de cuatro robots pueden competir a la vez. CROBOTS es mejor si lo juegan varias personas, cada uno perfeccionando su propio programa, y después enfrentando a los programas entre sí.

En DROBOTS, la arquitectura y la mecánica de la aplicación es muy diferente. El juego está orquestado por un servidor que crea partidas a la que se conectan los jugadores. Los jugadores aportan controladores de robots y, opcionalmente, controladores de detectores. Cuando la partida dispone del número de jugadores adecuado, crea robots y detectores para cada jugador y les solicita los controladores para los mismos. Todos ellos: servidor, jugador, robot y controladores son objetos distribuidos. El detector no se considera un objeto distribuido, ya que no dispone de ninguna funcionalidad que ofrecer de manera directa, y por tanto no tiene interfaz.

Después, el juego va indicando a cada controlador de robot un turno en el que puede interaccionar con el robot asociado. En la modalidad más simple cada jugador tiene un único controlador y por tanto un único robot.

Salvo por los aspectos de comunicación entre programas, pasando de una máquina virtual y ejecución centralizada en CROBOTS, a un conjunto de programas que se comunican a través de la red. En DROBOTS, el juego trata de respetar siempre que sea posible las reglas y funcionamiento del CROBOTS original.

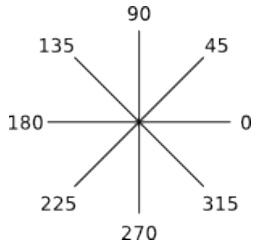
2. Funcionamiento

A continuación se describe el entorno y reglas que rigen el juego. La mayoría de lo explicado aquí es una traducción/resumen de las partes relevantes de la documentación original de CROBOTS [Poi85]. El texto hace especial énfasis en las discrepancias necesarias para permitir que funcione como un juego en red.

[1] la primera versión es de 1985

2.1. Campo de batalla

El campo de batalla es una cuadrícula de 400 x 400 metros. Existe un muro impenetrable alrededor del campo de batalla, de modo que si un robot choca contra él será dañado. La esquina inferior-izquierda tiene la coordenada (0,0) y la superior-derecha la (399,399). El sistema de brújula está orientado de modo que el ángulo de 0^o corresponde a la orientación este.



2.2. Equipamiento ofensivo

Los instrumentos ofensivos del robot son el lanzamisiles y el escáner. Puede lanzar misiles con un alcance máximo de 200 metros. No hay límite en la cantidad de misiles que se pueden disparar, pero hay que considerar el tiempo mínimo de recarga y la energía necesaria para realizar el disparo. La velocidad de los misiles es de 100 m/s. Los misiles vuelan por encima del campo de batalla, de modo que no impactarán con los robots que se encuentren entre el robot que dispara y el destino fijado. El lanzamisiles está montado en una torreta independiente, por tanto es posible disparar en cualquier dirección con un ángulo de 0-359^o sin importar la dirección en la que se mueve el robot.

El escáner es un dispositivo óptico que puede escanear de forma instantánea en la dirección deseada (0-359^o). La apertura mínima del escáner es de 1^o y la máxima de 20^o.

2.3. Equipamiento defensivo

La única defensa del robot son el motor y los registros de estado. El motor permite mover el robot en cualquier dirección (0-359^o), con una velocidad expresada entre 0 y 100 % de su potencia máxima. Una velocidad de 0 % indica que el robot debe parar. El robot puede girar solo si la velocidad es inferior al 50 %.

Los registros de estado permiten saber el porcentaje de daño, la velocidad actual y la posición del robot en coordenadas (x,y).

Además, en el campo de batalla se despliegan un conjunto de detectores. Dichos detectores no pueden ser destruidos, ni se puede chocar contra ellos, y tampoco pueden moverse. Si el jugador aporta los controladores necesarios, recibirá, en cada turno de ejecución, el número de robots que hay en 40 metros a la redonda de cada detector. Si no hubiera ninguno, no se recibirá ninguna notificación.

2.4. Destruyendo enemigos

Un robot se considera destruido si su daño alcanza el 100 %. Existen distintos eventos que provocan daño:

2 % Colisionar con otro robot o contra un muro. La colisión también provoca que el motor se pare.

5 % Un misil ha explotado en un radio de 80 metros.

20 % Un misil ha explotado en un radio de 40 metros.

40 % Un misil ha explotado en un radio de 20 metros.

El daño es acumulativo y no se puede reparar. El robot es perfectamente funcional siempre que el daño sea inferior al 100 %.

2.5. Consumo de energía

En cada nuevo turno el robot dispone de una cantidad limitada de energía (100 unidades). Cada vez que el controlador solicita una acción, se consume una parte. Si una acción requiere más energía de la disponible, no se producirá. Los consumos de energía para cada acción son los siguientes:

`drive()` Si se le indica una velocidad mayor que cero consume 60 unidades. Si la velocidad indicada es cero (es decir, parar el motor) consume 1 unidad.

`cannon()` Consume 50 unidades.

`scan()` Consume 10 unidades.

`damage()`, `location()`, `speed()` y `energy()` Consumen 1 unidad.

3. Interfaces

La aplicación está implementada con el middleware ZeroC Ice [ice15], de modo que todos los componentes son objetos distribuidos que implementan un interfaz descrita en lenguaje Slice. Dichas interfaces corresponden al fichero `drobots.ice` que se explica a continuación.

3.1. Interfaz `Robot`

Esta sección describe la interfaz del `RobotBase` (ver listado 1) con una breve explicación de cada método:

`void drive(int angle, int speed)`
: Pide al robot que se modifique su dirección al ángulo `angle` (en el rango 0-359) con una velocidad `speed` (en el rango 0-100). Al invocar este método, el robot se mantendrá en movimiento. No es necesario volver a invocarlo si no se desea cambiar la dirección y velocidad. Para parar al robot, se le debe indicar velocidad 0.

`short damage()`
: Pide al robot la cantidad de daño actual. El valor de retorno se encuentra en el rango 0-100.

`short speed()`
: Pide al robot su velocidad actual. El valor de retorno se encuentra en el rango 0-100.

`short energy()`
: Pide al robot la cantidad de energía restante en el turno en curso.

`Point location()`
: Pide al robot su posición actual. El valor de retorno es una estructura `Point` (ver listado 2) que expresa una coordenada 2D.

LISTADO 1: Interfaz `drobots::Robot`

```
interface RobotBase {
  void drive(int angle, int speed) throws NoEnoughEnergy;
  short damage() throws NoEnoughEnergy;
  short speed() throws NoEnoughEnergy;
```

```
  Point location() throws NoEnoughEnergy;
  short energy() throws NoEnoughEnergy;
};
```

LISTADO 2: Estructura `drobots::Point`

```
struct Point {
  int x;
  int y;
};
```

En realidad, nunca obtendremos un robot de tipo `RobotBase`, ya que es una interfaz base de la cuál heredan las dos interfaces que utilizaremos en realidad:

Attacker

Permite realizar todas las operaciones descritas anteriormente y, además, la operación `cannon()`. Su función por tanto en la de disparar a otros robots, pero no tiene por sí mismo capacidad de detectar enemigos.

- `bool cannon(int angle, int distance)` Pide al robot que dispare un misil en la dirección `angle` (en el rango 0-359) y distancia `distance`.

LISTADO 3: Interfaz `drobots::Attacker`

```
interface Attacker extends RobotBase {
  bool cannon(int angle, int distance) throws NoEnoughEnergy;
};
```

Defender

Permite realizar todas las operaciones de `RobotBase` y, además, la función `scan()`. Su función por lo tanto es la de detectar enemigos, pero no tiene por sí mismo capacidad de ataque.

- `int scan(int angle, int wide)` Pide al robot que haga un escaneado en la dirección `angle` (en el rango 0-359) con una amplitud `wide` (en el rango 1-20). El valor de retorno es el número de robots localizados.

LISTADO 4: Interfaz `drobots::Defender`

```
interface Defender extends RobotBase {
  int scan(int angle, int wide) throws NoEnoughEnergy;
};
```

Además, se define una cuarta interfaz llamada `Robot`, la cual es una combinación de las dos anteriores, y por tanto soporta todas las operaciones.

3.2. Interfaz `RobotController`

La interfaz `RobotController` (ver listado 5) recibe los mensajes desde el juego que controlan la partida. Sus métodos son:

void turn()

Indica al controlador el inicio de un nuevo turno. En este método, el controlador debería realizar sobre su robot las acciones que considere oportunas.

void robotDestroyed()

Indica el controlador que su robot ha sido destruido, es decir, su daño ha llegado al 100 %.

LISTADO 5: Interfaz `drobots::RobotController`

```
interface RobotController {
  void turn();
  void robotDestroyed();
};
```

3.3. Interfaz `DetectorController`

La interfaz `DetectorController` (ver listado 6) únicamente dispone de una función:

void alert(Point pos, int enemies)
: Indica al controlador que un detector ha detectado algún robot en 40 metros a la redonda de la posición indicada por la variable de tipo `Point`. También recibirá un entero con el número de robots detectados.

LISTADO 6: Interfaz `drobots::DetectorController`

```
interface DetectorController {
  void alert(Point pos, int enemies);
};
```

3.4. Interfaz `Player`

La interfaz `Player` (ver listado 7) corresponde al programa del jugador (el «cliente» en términos del juego). Su funcionalidad principal es crear controladores bajo demanda del juego. También incluye métodos con los que el juego notifica la finalización de la partida. Sus métodos son:

`RobotController* makeController(Robot* bot)`
: Es una función factoría con la que el juego le pide al jugador que cree (y le devuelva) un controlador para el robot `bot`.

`DetectorController* makeDetectorController()`
: Es una función factoría con la que el juego le pide al jugador que cree (y le devuelva) un controlador para un detector.

`void win()`
: Con este método el juego le indica al jugador que ha ganado la partida.

`void lose()`
: Con este método el juego le indica al jugador que ha perdido la partida.

LISTADO 7: Interfaz `drobots::Player`

```
interface Player {
  RobotController* makeController(Robot* bot);
  DetectorController* makeDetectorController();
  void win();
  void lose();
  void gameAbort();
};
```

3.5. Interfaz `Game`

La interfaz `Game` (ver listado 8) corresponde al servidor del juego. El jugador debe utilizarla para solicitar participar en una partida. Su único método es:

`void login(Player* p, string nick)`
: La utiliza un jugador para solicitar su participación en una partida. Si la partida ya ha comenzado,

este método elevará la excepción `GameInProgress`, y el jugador debería reintentarlo después.

LISTADO 8: Interfaz `drobots::Game`

```
interface Game {
  void login(Player* p, string nick)
    throws GameInProgress, InvalidProxy, InvalidName;
};
```

3.6. Interfaz `GameFactory`

La interfaz `GameFactory` (ver listado 9) se corresponse con una factoría de partidas. Estsa interfaz permite crear una partida privada, la cual será accesible únicamente para los conocedores del nombre de la partida.

Para crear una partida se debe invocar a:

`Game* makeGame(string gameName, int numPlayers)`
: Es utilizada para crear una partida que será identificada por la cadena pasada como primer argumento. Dicha partida alojará un máximo de jugadores indicado por el segundo argumento. El número mínimo de jugadores será 1, y el máximo, el definido por el tipo de partida. Para el ejemplo que nos ocupa, el máximo serán 4 jugadores.

LISTADO 9: Interfaz `drobots::GameFactory`

```
interface GameFactory {
  Game* makeGame(string gameName, int numPlayers)
    throws InvalidName, BadNumberOfPlayers;
```

3.7. Interfaz completa

El fichero Slice completo es el siguiente:

LISTADO 10: Fichero `drobots.ice`

```
module drobots {

  struct Point {
    int x;
    int y;
  };

  exception NoEnoughEnergy{};

  interface RobotBase {
    void drive(int angle, int speed) throws NoEnoughEnergy;
    short damage() throws NoEnoughEnergy;
    short speed() throws NoEnoughEnergy;
    Point location() throws NoEnoughEnergy;
    short energy() throws NoEnoughEnergy;
  };

  interface Attacker extends RobotBase {
    bool cannon(int angle, int distance) throws NoEnoughEnergy;
  };

  interface Defender extends RobotBase {
    int scan(int angle, int wide) throws NoEnoughEnergy;
  };

  interface Robot extends Attacker, Defender {};

  interface RobotController {
    void turn();
```

```
    void robotDestroyed();
  };

  interface DetectorController {
    void alert(Point pos, int enemies);
  };

  interface Player {
    RobotController* makeController(Robot* bot);
    DetectorController* makeDetectorController();
    void win();
    void lose();
    void gameAbort();
  };

  exception GameInProgress{};
  exception InvalidProxy{};
  exception InvalidName{
    string reason;
  };
  exception BadNumberOfPlayers{};

  interface Game {
    void login(Player* p, string nick)
      throws GameInProgress, InvalidProxy, InvalidName;
  };

  interface GameFactory {
    Game* makeGame(string gameName, int numPlayers)
      throws InvalidName, BadNumberOfPlayers;
  };
};
```

4. Mecánica de la partida

Cuando un cliente está listo para empezar a jugar una partida, debe avisar al servidor de partidas a través de una llamada a la función `login()`.

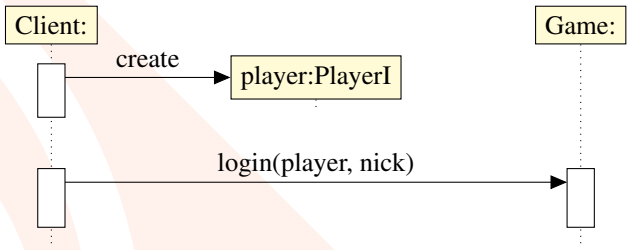


FIGURA 1: Registro del jugador en la partida

Una vez el servidor haya decidido que la partida puede dar comienzo, solicitará a cada jugador la creación de un controlador para cada nuevo robot llamando a la función `makeController()` de la interfaz `Player`.

DROBOTS es un juego por turnos: una vez el controlador reciba el aviso de que el turno comienza, podrá realizar cuantas acciones estime convenientes sobre su robot. Algunas acciones del robot se evaluarán una vez finalice el turno en el servidor, lo que implica que varias llamadas a los métodos `damage()`, `speed()` o `location()` en el mismo turno devolverán siempre el mismo valor.

Mientras dure la partida y el robot controlado por el jugador siga «vivo», el servidor llamará en cada turno a la función `turn()`.

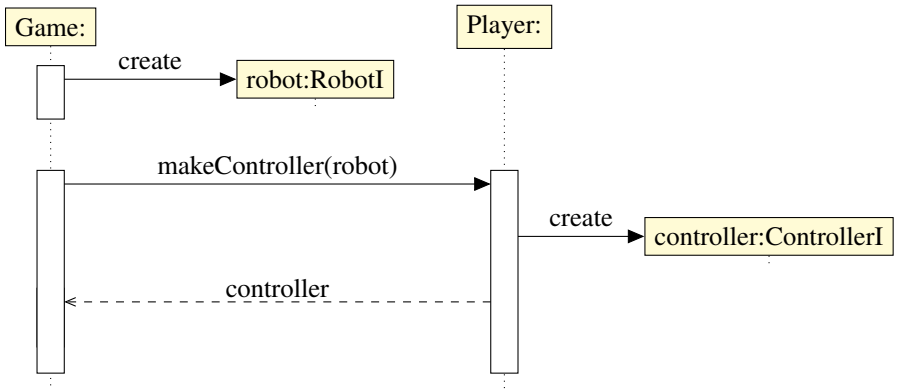


FIGURA 2: Petición de creación de un controlador por parte del servidor

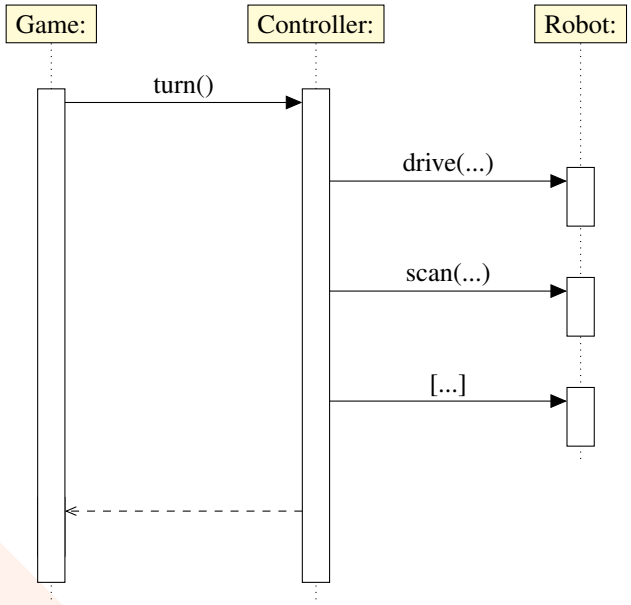


FIGURA 3: Diagrama de secuencia de un turno de juego. Las llamadas de `Controller` a `Robot` son un mero ejemplo

5. Uso de la factoría de partidas

Como novedad, el servidor implementará una factoría de partidas, que permitirá a los jugadores poder iniciar una partida sin necesidad de utilizar un segundo jugador falso o de esperar a que quede libre el servidor configurado.

La creación de la partida se realizará como se ha indicado anteriormente en la descripción de la interfaz `GameFactory` 9.

Para el segundo y sucesivos jugadores que quieran unirse a esta misma partida, deberán utilizar el mismo nombre identificativo pasado como primer argumento. En este contexto, el argumento de número de jugadores será ignorado por el servidor, que devolverá el proxy de la partida previamente creada.

Importante: si el número de jugadores especificado por el primer jugador es 1, la partida terminará terminado el primer turno (sólo quedaría un jugador). Esto puede ser útil para probar el mecanismo de conexión, pero totalmente inútil para probar las estrategias, por ejemplo.

El uso de esta funcionalidad del servidor es **totalmente optativa**, aunque será imprescindible para alcanzar el nivel avanzado.

6. Implementación

El alumno deberá implementar un programa utilizando ZeroC Ice que le permita tomar parte en las partidas de DROBOTS que se desarrollarán en un servidor.

El alumno deberá crear la implementación de un jugador de modo que:

- Realice el registro de su jugador (objeto `Player`) en la partida.
- Instancie un controlador de robot cuando el servidor se lo solicite.
- En cada turno (método `RobotController.turn()`), el controlador solicite al robot la información que necesite y le dé las órdenes adecuadas para conseguir el objetivo que se plantee para cada entregable.

7. Entregable

El entregable definitivo para las prácticas de la signatura debe tener las siguientes características:

Cada jugador contará con un equipo de cuatro robots, de los cuáles al menos uno será un “attacker”, otro un “defender” y otro un “robot” completo.

El alumno debería crear un controlador diferente en función del tipo de robot que se le asigne en la construcción de cada controlador.

El alumno debe diseñar e implementar una o varias interfaces remotas adicionales que permita a los robots comunicarse para realizar una estrategia conjunta.

Cada controlador debe ejecutarse en un programa diferente, o dicho de otro modo: los controladores de robots deben poder ejecutarse sin ninguna complicación en máquinas diferentes.

Se valorará positivamente:

Que las responsabilidades de todos los robots sean similares, es decir, que sea un enfoque más distribuido y menos centralizado.

La automatización del arranque y parada de los diferentes servidores.

La utilización de patrones de diseño.

La utilización de más de un lenguaje de programación.

Implementación de pruebas automáticas unitarias y de integración.

Cualquier esfuerzo por mejorar el rendimiento de la aplicación, por ejemplo, utilizando AMI/AMD para mejorar la paralelización de las invocaciones.

Se valorará negativamente:

La mala elección de nombres de programa, variables, clases.

Usar mecanismos de comunicación entre procesos diferentes a Ice o cualquier otro sistema de comunicación de sistemas distribuidos.

Se valorará de forma especialmente negativa la utilización de ficheros como mecanismo de comunicación para pasar *proxies*, parámetros o cualquier otra cosa.

La modificación del fichero de interfaz, aunque sea únicamente para añadir nuevas interfaces.

8. Evaluación y condiciones

La elaboración de este entregable se realizará en parejas. La entrega se realizará mediante un archivo `.zip` o `.tgz` que incluya todos los ficheros necesarios para la compilación de los distintos componentes, especificación de la aplicación (fichero XML), y otros scripts y ficheros `Makefile` necesarios para el despliegue, ejecución y prueba de la aplicación.

La evaluación se realizará en un sistema operativo GNU/Linux, de modo que el alumno debe comprobar que funciona correctamente en dicho entorno. Cada grupo deberá realizar una defensa presencial individual. La ejecución se efectuará en los computadores de los alumnos.

La calificación obtenida por los alumnos dependerá del nivel de complejidad que alcance dentro de los siguientes.

8.1. Nivel básico

Debe crearse una aplicación que sea capaz de jugar una partida de DROBOTS. Además, debe cumplir los siguientes requisitos:

> Los sirvientes de `RobotController` **no pueden** estar en el mismo programa que el sirviente de `Player`.
>
> El programa que aloje objetos de tipo `RobotController` podrá alojar, como máximo, 2. Idealmente debería haber 3 instancias del programa, de modo que cada instancia tenga al menos 1 sirviente.
>
> Implementar los controladores de los detectores. Puede utilizarse un único `DetectorController` para los 4 controladores, pero dicho controlador debe estar en un programa aparte de los `RobotController` y el `Player`. Para dar por válido este punto el controlador del detector debe ser capaz de enviar las alertas recibidas a los robots y estos obrar en consecuencia.
>
> No se requiere el uso de IceGrid

Este nivel dará lugar a la obtención de la calificación mínima para superar la asignatura con una puntuación en el rango [8–14) sobre los 25 puntos de la actividad «Realización de prácticas de laboratorio».

8.2. Nivel medio

Para lograr este nivel, además de la realización correcta de los entregables, se aplican los siguientes condicionantes:

- Debe crear una aplicación distribuida gestionada con IceGrid para el despliegue de los distintos componentes presentes en el entregable 3.
- La aplicación se deberá ejecutar en un grid de tres nodos. El alumno puede usar máquinas virtuales que se ejecutan en su propio portátil o ejecutar todos los nodos en la misma máquina.

Se valorará:

- Implementación de plantillas para la creación de los servidores en la aplicación IceGrid.

Este nivel dará lugar a la obtención de una calificación de notable, con una puntuación en el rango [14–19) sobre los 25 puntos de la actividad «Realización de prácticas de laboratorio». Los alumnos que opten por este nivel harán su defensa en una fecha que se anunciará convenientemente una vez acabado el período de clases.

8.3. Puntos extra

Para aquellos alumnos que quieran conseguir hasta un máximo de 2 puntos extra sobre la puntuación obtenida, deberán implementar lo siguiente:

Utilización de IcePatch. Para aquellos que utilicen IceGrid para el despliegue de la aplicación, podrán optar a un punto extra si utilizan IcePatch para el despliegue de su aplicación.

Utilización de la factoría de partidas.

8.4. Nivel avanzado

En este nivel, además de los requisitos y criterios de evaluación del nivel medio, deben implementarse los requisitos para obtener los puntos extras enunciados en el apartado anterior.

Se valorará:

- Grado de descentralización de la estrategia de los robots.

Este nivel dará lugar a la obtención de una calificación de sobresaliente, con una puntuación en el rango [19–23] correspondientes a la actividad «Realización de prácticas de laboratorio». Los alumnos que opten por este nivel harán su defensa en una fecha que se anunciará convenientemente una vez acabado el período de clases.

Referencias

[ice15] *Distributed Programming with Ice*. ZeroC, Inc, 2015. https://doc.zeroc.com/display/Ice36/Ice+Manual.

[Poi85] Tom Poindexter. *CROBOTS*, 1985. http://www.mit.edu/afs.new/sipb/user/sekullbe/crobots/crobots.doc.