Sistemas Distribuidos

Escuela Superior de Informática

DROBOTS: CROBOTS distribuido

David.Villa@uclm.es
profesor.jlsegura@uclm.es

ISRN ESI/--2015/SSDD-1516

versión 2015.12.09

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. Cuando la partida dispone del número de jugadores adecuado crea robots para cada jugador y les solicita controladores para los mismos. Todos ellos: servidor, jugador, robot y controlador son objetos distribuidos.

Después, el juego va indicando a cada controlador 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

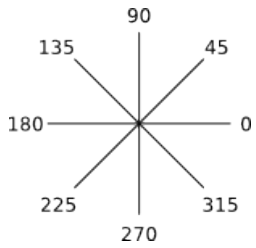
A continuación se describe el entorno y reglas que rigen el juego. La mayoría de lo explicado aquí es una traducción/resumen de las partes relevantes de la documentación original de CROBOTS [Poi85]. El texto hace especial énfasis en las discrepancias necesarias para permitir que funcione como un juego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

2.1. Campo de batalla

El campo de batalla es una cuadrícula de 1000 x 1000 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

[1] la primera versión es de 1985

tiene la coordenada (0,0) y la superior-derecha la (999,999). El sistema de brújula está orientado de modo que el ángulo de 0º 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 700 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 200 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 0-359º 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 (0-359º). La apertura mínima del escáner es de 1º y la máxima de 20º.

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 (0-359º), 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).

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 `Robot` (ver listado 1) con una breve explicación de cada método:

`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.

`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`.

`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`

```
1   interface RobotBase {
2     void drive(int angle, int speed) throws NoEnoughEnergy;
3     short damage() throws NoEnoughEnergy;
4     short speed() throws NoEnoughEnergy;
5     Point location() throws NoEnoughEnergy;
6     short energy() throws NoEnoughEnergy;
7   };
8
9   interface Attacker extends RobotBase {
```

```
10      bool cannon(int angle, int distance) throws NoEnoughEnergy;
11    };
12
13    interface Defender extends RobotBase {
14      int scan(int angle, int wide) throws NoEnoughEnergy;
15    };
16
17    interface Robot extends Attacker, Defender {};
```

LISTADO 2: Estructura `drobots::Point`

```
1     struct Point {
2       int x;
3       int y;
4     };
```

Existen en realidad tres tipos de robots:

Attacker
: Permite realizar todas las operaciones descritas anteriormente excepto `scan()`. Tu función por tanto en la de disparar a otros robots, pero no tiene por sí mismo capacidad de detectar enemigos.

Defender
: Permite realizar todas las operaciones descritas anteriormente excepto `cannon()`. Tu función por lo tanto es la de detectar enemigos, pero no tiene por sí mismo capacidad de ataque.

Robot
: Es una combinación de ambos y por ello soporta todas las operaciones.

3.2. Interfaz `RobotController`

La interfaz `RobotController` (ver listado 3) 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 3: Interfaz `drobots::RobotController`

```
1     interface RobotController {
2       void turn();
3       void robotDestroyed();
4     };
```

3.3. Interfaz `Player`

La interfaz `Player` (ver listado 4) 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`.

`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 4: Interfaz `drobots::Player`

```
1  interface Player {
2    RobotController* makeController(Robot* bot);
3    void win();
4    void lose();
5    void gameAbort();
6  };
```

3.4. Interfaz `Game`

La interfaz `Game` (ver listado 5) 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 5: Interfaz `drobots::Game`

```
1
2  interface Game {
3    void login(Player* p, string nick)
4      throws GameInProgress, InvalidProxy, InvalidName;
5  };
```

3.5. Interfaz completa

El fichero Slice completo es el siguiente:

LISTADO 6: Fichero `drobots.ice`

```
1  module drobots {
2
3    struct Point {
4      int x;
5      int y;
6    };
7
8    exception NoEnoughEnergy{};
9
10   interface RobotBase {
11     void drive(int angle, int speed) throws NoEnoughEnergy;
12     short damage() throws NoEnoughEnergy;
13     short speed() throws NoEnoughEnergy;
14     Point location() throws NoEnoughEnergy;
15     short energy() throws NoEnoughEnergy;
16   };
17
18   interface Attacker extends RobotBase {
19     bool cannon(int angle, int distance) throws NoEnoughEnergy;
20   };
21
22   interface Defender extends RobotBase {
```

```
23      int scan(int angle, int wide) throws NoEnoughEnergy;
24    };
25
26    interface Robot extends Attacker, Defender {};
27
28    interface RobotController {
29      void turn();
30      void robotDestroyed();
31    };
32
33    interface Player {
34      RobotController* makeController(Robot* bot);
35      void win();
36      void lose();
37      void gameAbort();
38    };
39
40    exception GameInProgress{};
41    exception InvalidProxy{};
42    exception InvalidName{
43      string reason;
44    };
45
46    interface Game {
47      void login(Player* p, string nick)
48        throws GameInProgress, InvalidProxy, InvalidName;
49    };
50  };
```

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 `attach()`.

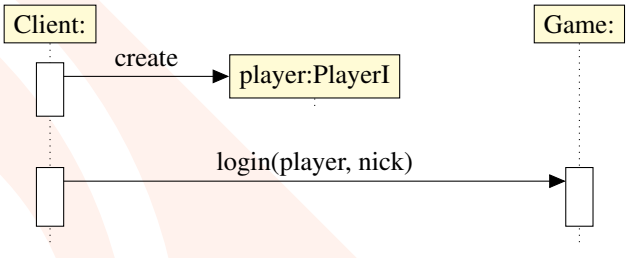


FIGURA 1: Registro del jugador en la partida

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()`.

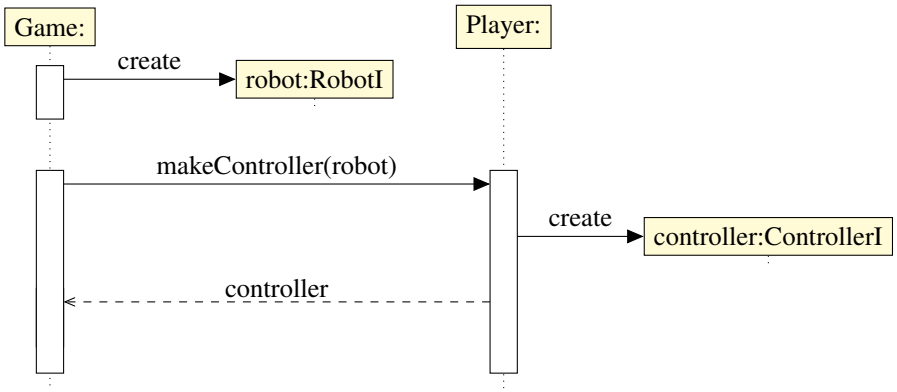


FIGURA 2: Petición de creación de un controlador por parte del servidor

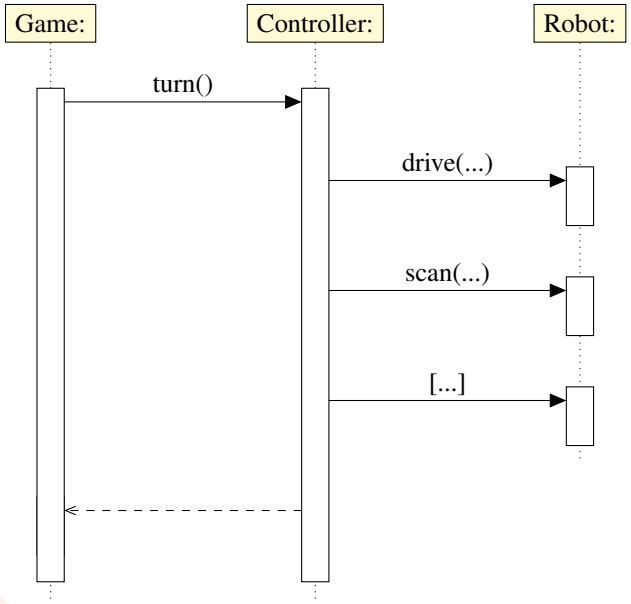


FIGURA 3: Diagrama de secuencia de un turno de juego. Las llamadas de `Controller` a `Robot` son un mero ejemplo

5. 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.

6. Entregables

La realización de este trabajo implica tres entregables, aunque por el momento solo los dos primeros está especificados. Por ese motivo, las interfaces explicadas anteriormente (y también este documento)

pueden sufrir cambios de cara a cubrir las necesidades de los entregables 2 y 3.

Entregable 1
El alumno debe escribir un jugador tal como se indica en la sección anterior. El objetivo de dicho jugador es mover un único robot desde su posición de aparición (aleatoria) hasta el centro del tablero (posición 500,500).

Entregable 2
El alumno debe escribir un jugador capaz de destruir a los robots de sus oponentes. Para este entregable el servidor del juego aceptará entre 2 y 4 jugadores. Es posible enfrentar entre sí dos instancias del programa del mismo jugador si no hubiera otros oponentes conectados.

Entregable 3
En este caso cada jugador contará con un equipo de cuatro robots: dos *attacker* y dos *defender*. 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 interfaz remota adicional que permita comunicarse a los robots para realizar una estrategia conjunta. Se valorará especialmente que las responsabilidades de todos los robots sean similares, es decir, que sea un enfoque más distribuido y menos centralizado. Además, los controladores deben crearse en nodos diferentes de un grid (3 nodos al menos). La gestión de este grid debe realizarse con IceGrid.

7. Evaluación y condiciones

La elaboración de la aplicación se compone de tres entregables. Todos ellos se realizarán de forma individual, se entregarán mediante una tarea moodle y serán defendidos por el alumno en una sesión de laboratorio.

En todos los casos las entregas se realizarán por medio de 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. El alumno deberá realizar una defensa presencial individual. La ejecución se efectuará en el computador del alumno.

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

7.1. Nivel básico

Corresponde con la realización satisfactoria de los tres entregables indicados en la sección anterior. Se aplican las siguientes simplificaciones:

- Todos los elementos de la aplicación se ejecutarán en el portátil del alumno. No se requiere el uso del IceGrid.

Este nivel dará lugar a la obtención de la calificación mínima para superar la asignatura[2], con una puntuación en el rango [8–13) sobre los 25 puntos de la actividad «Realización de prácticas de laboratorio». Además podrá obtener hasta 5 puntos adicionales en la actividad «Presentación oral de temas» por la defensa presencial (únicamente en la convocatoria ordinaria). Los alumnos que opten por este nivel deberán hacer su defensa en la última sesión de laboratorio.

[2] teniendo en consideración los 2 puntos adicionales correspondientes a la defensa del ejercicio de la sesión 3

7.2. 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.

Se valorará:

- Cualquier mecanismo que automatice las tareas de compilación, despliegue, ejecución y prueba. Se recomienda especialmente el uso de `icegridadmin` (evitando `icegrid-gui`) para lograrlo. La necesidad de ejecución manual de comandos o acciones superfluas de cualquier tipo penaliza la puntuación.
- Utilización de **nombres significativos** para clases, tipos, métodos y variables. La presencia de comentarios evidencia código poco expresivo. Elimina los comentarios y elige buenos nombres para tus abstracciones.
- Implementación de plantillas para la creación de los servidores en la aplicación IceGrid.
- 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.

Este nivel dará lugar a la obtención de una calificación de notable, con una puntuación en el rango [13–18) sobre los 25 puntos de la actividad «Realización de prácticas de laboratorio». Además podrá obtener hasta 7 puntos adicionales en la actividad «Presentación oral de temas» por la defensa presencial (únicamente en la convocatoria ordinaria). Los alumnos que opten por este nivel harán su defensa en una fecha que se anunciará convenientemente una vez acabado el período de clases.

7.3. Nivel avanzado

En este nivel, además de los requisitos y criterios de evaluación del nivel medio, se aplican los siguientes:

- La aplicación del alumno es capaz de ganar al menos dos de tres partidas, jugando contra las aplicaciones de otros compañeros que opten también por presentar el nivel avanzado.

Se valorará:

- Grado de descentralización de la estrategia de los robots.
- Uso de más de un lenguaje de programación.

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». Además podrá obtener hasta 10 puntos adicionales en la actividad «Presentación oral de temas» por la defensa presencial (únicamente en la convocatoria ordinaria). 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.