Práctica 3: "Gugel Vehicles" para búsqueda distribuida

Eila Gómez Hidalgo, Manuel Lafuente Aranda,
Elías Méndez García, Baltasar Ruiz Hernández
y Daniel Soto del Ojo

1. El diseño de nuestra idea

a. Esqueleto de la práctica, diagrama de clases:

Nuestro proyecto ha sido dividido en tres subconjuntos de clases o paquetes:

- **Agentes:** es el paquete principal. Contiene el agente central asociado a los vehículos y el agente encargado de controlar el envío de mensajes entre los distintos vehículos existentes, además de un pequeño grupo de clases auxiliares que encapsulan el envío y recepción de mensajes ACL y que controlan el funcionamiento de la interfaz. Dentro de este paquete encontramos además otros cuatro subconjuntos de clases:
  - **Data:** contiene clases que facilitan el tratamiento de los nombres, capacidades, percepciones, posiciones y tipos de los agentes, de los movimientos a ejecutar, de las tomas de decisiones en los conflictos, de los datos de los vehículos a introducir en la BD, y del formato de los mensajes a enviar al Contable.
  - **Old:** que contiene:
    - Las versiones antiguas del agente central, en el cual el planteamiento de las comunicaciones entre agentes estaba mal realizado.
    - El agente Contable, encargado de gestionar la decisión de recarga de los vehículos en la versión inicial en red de nuestra sociedad.
  - **Shenron:** contiene las clases para levantar el servidor y reiniciar las conexiones en caso de problemas con el controlador, desestructuración del paso de mensajes o comunicaciones incompletas producidas por errores.
  - **Utilities:** contiene clases que proporcionan apoyo al proyecto, facilitando el control de los nombres de los agentes, el uso de JSON y SQL, el cálculo de los costes para las casillas y de los valores de sus posiciones en el mapa según cómo lo gestionamos en la BD.
- **GUI:** es el paquete principal de la interfaz de usuario. Contiene las clases que gestionan el comportamiento de las ventanas, los paneles y los botones, además de las clases que constituyen el patrón Observable - Observador encargado de actualizar los mensajes y valores a mostrar en pantalla. Dentro de este paquete se encuentra también el paquete Observables que contiene las clases de los diversos Observables que se usan en el patrón antes mencionado.
- **Representacion3D:** contiene toda la lógica 3D del mapa: fondo, cámara, texturas, iluminación, límites, etc. Dentro de este paquete también tenemos el paquete PrimitiveShape3D que contiene las clases encargadas de crear las dos figuras tridimensionales que se verán en la interfaz: la esfera representando los vehículos, y las cajas representando las paredes.

Diagrama de clases

b. Comportamiento global, diagrama de secuencia:

El comportamiento general de los tres actores diferentes (**Bellatrix**, **AgenteMovil** y **Controlador**) que se coordinan en nuestro proyecto es el siguiente:

- El primer agente móvil creado realiza el proceso de subscripción y obtención de capacidades mediante el cual se inicia la conexión con el servidor en un mapa, convirtiendo al agente en uno de los tres tipos diferentes de agentes móviles (dron, coche o camión). Sin embargo, en esta primera fase se fuerza al agente móvil inicial a ser un dron, ya que vuela y no necesita sortear obstáculos. Por tanto, en caso de obtener un agente terrestre al realizar por primera vez el proceso mencionado, este se repite tantas veces como sean necesarias hasta conseguir el dron.
- Una vez conseguido el dron o agente volador, este se encarga de divulgar el *conversation_id* al resto de agentes, necesario para continuar las comunicaciones con el controlador del servidor. Cuando estos agentes reciben el mensaje con el *conversation_id*, simplemente han de enviar el mensaje de solicitud *checkin* para dar por finalizada esta primera fase de logueo.
- A continuación, los agentes creados y logueados envían un mensaje de información al controlador local con el cual se comienza un proceso repetitivo y secuencial marcado por las siguientes fases:
    - PEDIR_SENSORES: el controlador da luz verde a los agentes móviles para realizar la solicitud de información del entorno al servidor. Cuando estos agentes reenvían al controlador local el mensaje de información antes mencionado, se pasa al siguiente estado.
    - ACTUALIZAR_BASE_DATOS: el controlador da luz verde a los agentes móviles para que estos invoquen los métodos correspondientes de actualización de variables. Es en este preciso momento cuando un agente comprueba si ha llegado al destino, en cuyo caso el agente manda un mensaje de finalización al controlador (no sin comprobar antes además que no se encuentra en la fase de exploración).
    - COMPROBAR_RECARGA: el controlador pide a los agentes móviles que comprueben su surtido de combustible:
        - En caso de necesitar una recarga de batería, un agente móvil concreto realizará dos acciones ordenadas por el controlador dependiendo de si queda suficiente energía global en el mundo (en cuyo caso puede recargar) o no (en cuyo caso el agente muere). En caso de realizar la primera acción, el agente solicitará de nuevo información del entorno al servidor.
        - En caso de no necesitar una recarga, el agente simplemente envía un mensaje de disconformidad al controlador local.

      El controlador local comprueba a continuación si siguen quedando agentes pendientes en la cola de recargas, informando al primero en dicha cola que puede recargar e informando a los agentes restantes que sigan esperando.
    - SIGUIENTE_COMANDO: el controlador pide a los agentes móviles que ejecuten su método de obtención del siguiente movimiento a realizar, valor que usa para solucionar posibles conflictos de movimiento, de la siguiente manera:
        - Si no existe ningún tipo de conflicto de movimiento con el resto de agentes, el controlador permite al agente solicitante que envíe el comando de movimiento al servidor.
        - En caso de existir conflicto, el controlador introduce a los agentes problemáticos en una cola, dando paso al que tenga mayor prioridad y teniendo que esperar el resto hasta el siguiente ciclo.

      Una vez solucionados todos los conflictos de movimiento existentes, el controlador da paso de nuevo al estado PEDIR_SENSORES con el cual se inicia el proceso

secuencial una vez más, y así hasta que la lista del controlador local queda vacía, momento en el que el controlador envía un mensaje de cancelación al servidor, cerrando la sesión.

Diagrama de secuencia

c. Comportamiento de cada agente, diagramas de actividad:

- Agente AgenteMovil: inicialmente, al comenzar su ejecución, el agente comprueba la variable lógica de estado *first* mediante la cual se define si dicho agente fue el primero en ser creado o no. En caso afirmativo, el agente realiza el proceso antes comentado mediante el cual fuerza al servidor a otorgarle las capacidades de un dron, divulgando el *conversation_id* de dicha comunicación al resto de agentes, para que también puedan enviarle mensajes al servidor. Una vez todos los agentes creados han obtenido sus correspondientes capacidades, se inicia el proceso secuencial marcado por el controlador local en el cual el agente móvil solicita la percepción del mundo al servidor, actualizando la BD y los observables. A continuación, el agente actualiza sus propias variables con la percepción obtenida, finalizando inmediatamente en caso de haber logrado llegar al destino (y estar en la fase de llegada al destino). Si el agente, por desgracia, no ha logrado llegar todavía a su destino, tendrá que comprobar si dispone de combustible suficiente para avanzar, de manera que en caso contrario tiene que solicitar la aprobación del controlador local para poder recargar. Cuando esto logra suceder, el agente llama a su método interno de obtención del siguiente movimiento a realizar, enviando dicho comando al servidor, no sin antes realizar las comprobaciones oportunas de conflictos, de nuevo con la ayuda y permiso del controlador local. Cuando el agente ha dado el paso deseado, pide una nueva percepción del mundo al servidor, reiniciando el proceso secuencial.

- Agente Controlador: al iniciarse, el controlador local espera los mensajes de confirmación de *checkin* de los agentes móviles, momento en el que inicia el ya comentado proceso iterativo y secuencial: En primer lugar, ordena a los agentes que perciban la información del mundo que les rodea, lo cual le sirve para actualizar el porcentaje de exploración y llamar al cálculo de superficies de costes si se ha completado o llegado al límite de la exploración. A continuación, el controlador pide a los agentes que realicen la fase de actualización de variables, descartando de sus variables internas propias a los agentes que hayan finalizado por llegar al destino. Después, el controlador pide a los agentes que le informen de si necesitan recargar o no, en cuyo caso afirmativo tendrá que valorar la concesión de recarga, teniendo en cuenta el nivel global de batería y las características del agente concreto. Si no ha podido llegar a una conclusión, el controlador introducirá al agente en su cola interna de recargas pendientes, de la cual tomará a los agentes y los informará según su prioridad. Por último, el controlador pide a los agentes móviles que le informen del siguiente movimiento que realizarán cada uno de ellos, teniendo que valorar ahora a cuál de ellos ha de cederle el paso en caso de conflicto, y a cuáles ha de meter en la lista de agentes en conflicto, de forma similar a como funciona y se comporta para conceder la recarga de combustible. Una vez el controlador local ha solventado todos los conflictos de movimiento, reinicia el proceso secuencial indicando a todos los agentes móviles que inicien una nueva petición de información del mundo al servidor, repitiéndose todos los pasos mencionados hasta que el controlador se queda sin agentes en su lista interna, momento en el que cierra la sesión con el servidor.

Diagrama de actividad de Controlador

Diagrama de actividad de AgenteMovil

2. Nuestros agentes

a. Diseño de los agentes principales: dichos agentes han sido diseñados de manera que su cuerpo principal de ejecución consta de la recepción de un mensaje y una estructura de control que viaja a una parte del flujo del proceso en función del contenido y performativa de dicho mensaje.

i. AgenteMovil: contiene variables para controlar la finalización de su ejecución en función del caso que se dé, además de todas las necesarias para controlar el entorno y la comunicación con los agentes que lo acompañan en él, el envío de mensajes, sus capacidades necesarias para el proceso de ejecución y para interactuar con la base de datos.

Y además de los métodos convencionales de inicialización, ejecución y finalización, el agente móvil dispone de varios métodos para envío y recepción de mensajes concretos, divulgación de mensajes, actualización de variables del entorno y atributos, cálculo del siguiente movimiento a realizar (tanto para exploración como llegada al destino), análisis del mapa, evasión de otros agentes, entre otros.

ii. Controlador: contiene variables para la finalización de su ejecución cuando se termine la ejecución de todos los vehículos, para controlar cada flujo de acción que se podría desencadenar según la heurística utilizada, para comunicarse con la interfaz gráfica diseñada y para interaccionar con la base de datos.

Y además de los métodos convencionales de inicialización, ejecución y finalización, el controlador local dispone de un método de divulgación de mensajes y un segundo método de control encargado de eliminar de la lista de agentes interna a todos aquellos que por una razón u otra han muerto.

b. Diseño de otras entidades secundarias:

i. Agente: esta es la clase encargada de encapsular el envío y recepción de los mensajes ACL. Para ello dispone de diversos métodos: uno de envío a un único receptor, uno de envío con un *conversation_id* a un único receptor, uno de envío con un *conversation_id* y un *in_reply_to* a un único receptor, y uno de recepción.

ii. AgenteInterfaz: esta clase gestiona el funcionamiento de un agente asociado a la interfaz de usuario. Se encarga únicamente de mandar un mensaje al agente Controlador indicándole si debe continuar sin cesar hasta el destino, avanzar un número de pasos determinado, o cerrar la sesión.

iii. JsonWrapper: esta clase es un *wrapper* o clase empaquetadora Singleton en la cual se encapsula el tratamiento (generación e interpretación) de los mensajes JSON que se envían los agentes entre sí (incluyendo al controlador del servidor). Tiene como atributos la única instancia existente de sí mismo y el parser de JSON para interpretar los mensajes.

iv. SQLWrapper: esta clase es un *wrapper* o clase empaquetadora en la cual se encapsula el tratamiento de las actualizaciones realizadas por parte de los agentes sobre la base de datos en SQLite que usamos para almacenar los diversos mapas entre distintas ejecuciones. Tiene como atributos la url de la base de datos y los posibles valores a indicar que tiene una casilla.

3. Estructura social

a. Sociedad elegida

En nuestra solución adoptada, la sociedad de agentes se caracteriza por ser **jerárquica**, ya que los agentes no se comunican en exclusiva entre sí, sino que necesitan el consentimiento y son dirigidos por los mensajes enviados por un agente central, llamado controlador central, que se encarga de manejar el flujo y realizar la toma de decisiones. Este tipo de organización ha sido su simplicidad para coordinar los movimientos de los vehículos entre sí y que no colisionen.

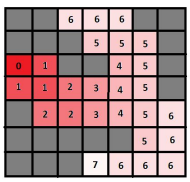
b. Diagrama de clase de los tipos de agentes

Diagrama de clases de los tipos de agentes

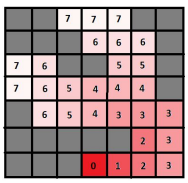
c. Estrategia de búsqueda y comunicación entre agentes

i. Estrategia de búsqueda

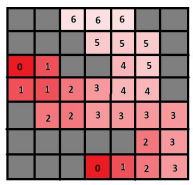
Como estrategia para encontrar la solución, una vez explorado todo el mapa, generamos 2 superficies de costes(una para vehículos aéreos y otra para vehículos por tierra) en la que en cada casilla se guarda el coste en pasos para llegar a la solución. Se genera un mapa de costes por cada casilla solución y después se mezclan en uno sólo quedándose siempre con el menor coste de cada casilla. Una vez los costes están calculados y almacenados cuando un vehículo aparezca en el mapa sabrá inmediatamente a qué casilla debe moverse para llegar a la solución por el camino más corto, sólo debe moverse a la casilla adyacente a su posición que tenga el menor coste. (Como se ve abajo Costes1 + Costes2 = Costes3)



Img. Costes1



Img. Costes2



Img. Costes3

ii. Estrategia de exploración

La exploración es llevada por un dron. El dron en primer lugar lo que hace es moverse hacia el norte o el sur hasta alcanzar un borde. A medida que se mueve, insertará las nuevas casillas dentro de la base de datos y registrará aquellas sobre las que haya pasado para quitarles prioridad frente a otras casillas que no haya pisado aún. Cuando alcance el borde, comprobará si existen en la base de datos todas las casillas pertenecientes a la columna sobre la que se encuentra. En caso afirmativo, avanzará hacia la siguiente columna y comprobará si existen. Cada vez que esta comprobación falle, se moverá hasta alcanzar el borde opuesto y se repetirá este proceso hasta haber explorado todo el mapa.

iii. Comunicación entre agentes

Inicialmente, se pensó en una sociedad de agentes compuesta por los agentes móviles, ya comentados a lo largo de este informe, así como un agente Contable encargado de solucionar los conflictos surgidos por la falta de batería. El comportamiento base de los agentes seguía un orden semisecuencial, actuando de una manera u otra en función del mensaje que recibían en un momento determinado. Sin embargo, esta idea fue desechada casi al completo ya que había desincronización total entre los agentes, realizándose una segunda versión de los agentes móviles en la que se incluía un mayor control de estados y envíos y recepciones de confirmaciones entre agentes, manteniendo al agente Contable. A pesar de ello, todavía existía una importante desincronización, así que el equipo optó por realizar una remodelación casi completa del proceso de ejecución de los agentes móviles,

implementando la versión final del proceso secuencial explicada en los puntos anteriores de este informe, eliminando al agente Contable e incorporando en su lugar al agente Controlador.

d. Resultados de nuestra solución.
Nuestras trazas no pueden ser tomadas como las mejores ya que se probó a resolver los mapas una primera vez y esas trazas
Resultado mapa 1 (CONV-u2szwtw7), Resultado mapa 2 (CONV-4hc3ogu0), Resultado mapa 3 (CONV-c6rfo9z2), Resultado mapa 4 (CONV-cpjb1r9u), Resultado mapa 5 (CONV-8trawd39), Resultado mapa 6 (CONV-g3qm3qmd), Resultado mapa 7 (CONV-m2q1cqqg), Resultado mapa 8 (CONV-x2igz6e8), Resultado mapa 9 (CONV-vjumevit)

4. Interfaces gráficas:

El proyecto consta de dos interfaces: una con Java Swing a modo de ventana de control y otra implementada en Java3D que nos va mostrando un mapa con el recorrido de los vehículos.

La ventana del panel de control nos muestra información sobre los distintos vehículos (concretamente su batería y su localización en el GPS) en cada decisión que tomamos además de algunos mensajes que nos indican qué decisión se ha tomado. Este panel dispone de 4 botones:

- Avanzar hasta el final: le indica a los vehículos (que sigan vivos) que avancen en la ejecución de la heurística planteada hasta que todos lleguen a un caso en el que deben terminarla, bien por haber llegado a la solución o haber tenido que enviar un CANCEL al servidor.
- Avanzar un paso: le indica a los vehículos (que sigan vivos) que tomen una única decisión.
- Avanzar: se les indica un número de pasos que tienen que avanzar. Si todos los vehículos vivos encuentran la solución antes de dar el total de pasos, el programa se para.
- Salir: se le indica a los vehículos que paren la búsqueda o avancen hacia la solución si ésta ya fue encontrada, finalizando a los agentes.

La otra interfaz nos muestra cómo se va generando en tiempo real el mapa a medida que el dron de exploración lo analiza en la fase de exploración, y cuando dicha fase ya está finalizada, nos muestra el avance de todos los agentes en tiempo real hacia la solución hallada. En dicho proceso, se marca en blanco las zonas descubiertas por las que no se ha pasado, en verde las zonas por las que se ha pasado, en gris las paredes u obstáculos, en amarillo los bordes y en rojo el objetivo. La posición actual de un vehículo se indica con una esfera de color azul.

Los componentes de la interfaz actúan como Observadores, es decir, cuando los agentes reciben nueva información del servidor un Observable se encarga de notificar a todos sus Observadores, que necesitan saber de este cambio, y les envían la nueva información. En el caso de las interfaces reciben la información, la tratan y la muestran. La ventana de control recibe información de todos los agentes, mientras que el gestor de la escena (encargado de la visualización del mapa) necesita la información guardada en la BD (que va almacenando para poder dibujar el mapa).

Las comunicaciones entre la ventana de control y los vehículos para indicarle qué es lo siguiente que deben hacer (si avanzar *n* pasos, un único paso, avanzar hasta el final o salirse) se realizan mediante paso de mensajes entre agentes. Cada vez que la interfaz necesita mandar un mensaje crea un agente que haga esta función y que inmediatamente después finaliza.

Anexo

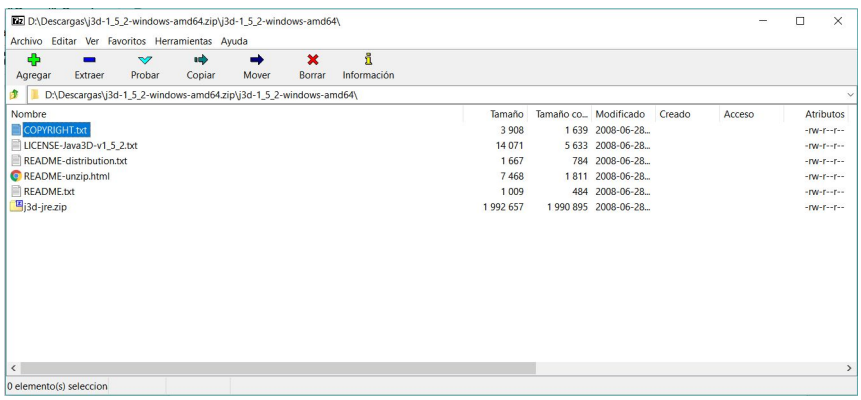
NOTA: El rendimiento de la práctica depende mucho del almacenamiento donde se guarden las bases de datos, se recomienda ejecutar el proyecto desde un ssd, en caso negativo pueden saltar excepciones porque el disco duro no sea lo suficientemente rápido al leer. Saltan excepciones pero nos hemos asegurado de que las tareas se realizan cuando el disco puede. Es un problema de la librería de la base de datos. Hemos intentado tener en cuenta todas las excepciones que se puedan dar pero puede haberse escapado alguna, en ese caso lo recomendable es lanzar shenron y volver a lanzar la práctica.

- **Dependencias.**
    a. JDK 8.
    b. SQLite-jdbc
    c. Gson 2.8.0
    d. Java3D 1.5.2.
    e. NetBeans 8.2 (en caso de usar NetBeans para compilar).

Para la compilación del proyecto se **necesita** JDK 8. Las librerías JAR necesarias se encuentran incluidas dentro del proyecto de NetBeans, sólo es necesaria la instalación de Java3D como se indica a continuación.

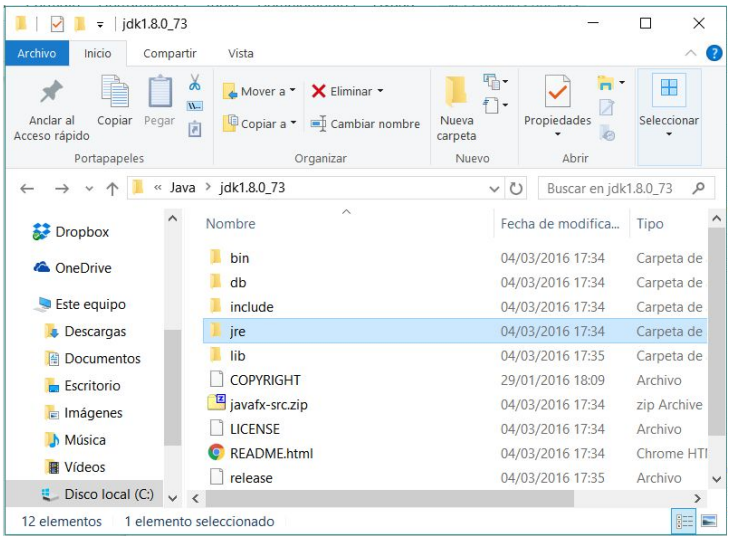
- **Instalación de Java3D.**

Para la instalación de Java3D se debe bajar el binario de la página enlazada anteriormente, una vez descargada y abierto el zip encontramos una carpeta, entramos en ella y nos encontramos la siguiente jerarquía de archivos.



Extraemos el zip llamado *j3d-jre.zip* que tenemos y dentro de él tenemos dos carpetas “lib” y “bin” en el caso de Windows, en Linux solo tenemos la carpeta “lib”. En ambos casos lo que tenemos que hacer es copiar esas carpetas a la carpeta “jre” del JDK que vamos a utilizar. A continuación tenemos el

directorio de instalación de JDK 8 resaltando la carpeta donde deben copiarse las carpetas del archivo *j3d-jre.zip*.



La ruta del JDK puede variar dependiendo del sistema operativo y la arquitectura. Una vez copiadas las carpetas ya está todo listo para usar Java3D con los JAR's incluidos en el proyecto.