Universidad de
Castilla-La Mancha

MANUAL DE MÓDULO:
Comportamiento Base

Autores:	G1 y G8
Colaboradores:	G2, G3, G4 y G7
Grado:	Ingeniería Informática
Curso:	4º
Asignatura:	Sistemas Multiagentes
Profesor:	Miguel Ángel Fernández Graciani
Fecha:	7 de noviembre de 2022

Contents

Introducción

Este módulo desarrolla el comportamiento base del agente *ACC*. La modificación de sus clases permite variar el comportamiento de los partos, la búsqueda de puertos vacíos para establecerse, y la localización de otros agentes ya alojados.

a) Tutorial

Para cada método se redactará una explicación inicial sobre su propósito. Acto seguido, se mostrará su código, y se explicará parte por parte su comportamiento.

Se añadirán, junto a las explicaciones, enlaces directos como este a los diferentes apartados del documento, para acceder de manera más rápida a las clases y métodos a los que las descripciones se refieren.

b) Ejemplo

Este método, funciona como *main* principal de la clase. Se encarga de escribir por pantalla *"Hello, World!"* y de incrementar una variable:

```
1 public static void main(String[] args) {
2     /* SECCION 1 */
3     int x = 0;
4     /* SECCION 2 */
5     System.out.println("Hello, World!");
6     /* SECCION 3 */
7     x = x+1;
8 }
```

- Sección 1: Se inicializa y declara una variable local.
- Sección 2: Se escribe por pantalla la frase *"Hello, World!"*.
- Sección 3: Se le suma una cifra al valor de la variable x.

¡¡¡ADVERTENCIA!!!

Los miembros encargados de desarrollar este módulo han programado, y explicado en este documento, las clases Acc y comportamientoBase. Las clases **MensajeAEnviar**, **MensajeRecibido**, **accLocalizado**, **funcionDeAgente** y **gestorDeMensajes**, a las cuales se referencia, se programaron como prototipos para comprobar el correcto funcionamiento del comportamiento base y de los partos.

CLASE 1 - Acc

a) Variables globales

Esta clase presenta una serie de variables globales, declaradas como *static*, que se utilizarán en todo el sistema:

```
    /* ENUMERADOS */
    enum Estado_del_ACC { VIVO, MUERTO }
    /* CADENAS DE TEXTO */
    static String Ip_Monitor, Inicio_rango_IPs,
        Ip_Propia, ID_propio;
    /* NUMEROS ENTEROS */
    static int Puerto_Monitor, Rango_IPs, Tiempo_de_vida,
        Numero_de_generaciones, Estado_Actual,
        Puerto_Inicio, Rango_Puertos, UDPport, TCPport,
        tiempo_espera_comportamiento_base;
    /* NUMEROS DECIMALES */
    static double Frecuencia_partos, Frecuencia_rastreo_puertos;
    /* SOCKET */
    static Socket monitorComunicacion;
    /* DATAOUTPUTSTREAM */
    static DataOutputStream out;
```

- enum Estado_del_ACC: Un enumerado que genera la estructura de estados del agente. Esto permitirá informarle al código sobre el estado vital actual del *ACC*. Presenta dos modos: *VIVO* y *MUERTO*. La variable *Estado_Actual* almacena dicho estado.
- static String Ip_Monitor: Almacenará la *IP* de la máquina que monitorizará todo el sistema multiagente.
- static String Inicio_rango_IPs: Existirá un rango de *IPs* de entre los que buscar dentro de la red universitaria. Esta variable marca la primera *IP* de este.
- static String Ip_Propia: *IP* de la máquina en la que se aloja el agente.
- static String ID_Propio: *ID* que representa al agente, el cual será único.
- static int Puerto_Monitor: Almacenará el puerto en el que se situará el monitor dentro de la máquina con la *IP* almacenada en *IP_Monitor*.
- static int Rango_IPs: Cantidad de *IPs*, a partir del principio del rango, *Inicio_rango_IPs*, de entre los que buscar máquinas que almacenan agentes de nuestro sistema.
- static int Tiempo_de_vida: Tiempo vital, expresado en segundos, que el agente vivirá e interactuará con los demás agentes, antes de perecer.
- static int Numero_de_generaciones: Número de generaciones, contándole a él, que un agente tendrá dentro de su árbol genealógico. Si el número de generaciones es 3, el agente solo podrá llegar a ser abuelo.

- static int Estado_Actual: Guarda, como número entero, un estado de entre los disponibles en el enumerado *Estado_del_ACC*.
- static int Puerto_Inicio: Existirá un rango de puertos de entre los que buscar dentro de una máquina. Esta variable marca el primer puerto de este.
- static int Rango_Puertos: Cantidad de puertos, a partir del principio del rango, *Puerto_Inicio*, de entre los que buscar agentes alojados en ellos.
- static int UDPport: Variable que guarda el puerto del agente encargado de las conexiones *UDP*.
- static int TCPport: Variable que guarda el puerto del agente encargado de las conexiones *TCP*.
- static int tiempo_espera_comportamiento_base: Un periodo de tiempo de espera, expresado en segundos, para limitar el acceso a las clases GenerarNuevoAcc() y GestorDeDirectorio().
- static float Frecuencia_partos: Porcentaje de probabilidad, del 0 a 1, de que un agente consiga tener un hijo en un intento.
- static float Frecuencia_rastreo_puertos: Porcentaje de probabilidad, del 0 a 1, de que un agente se ponga a buscar a otros agentes en un intento.
- static Socket monitorComunicacion: *Socket* que servirá como vía de comunicación entre el agente y el monitor del sistema.
- static DataOutputStream out: Asociado a *monitorComunicacion*, nos permitirá enviar cadenas de caracteres al monitor.

b) main()

Autores: G1 y G8

Este método es el utilizado como *main* principal de la clase Acc. Se encarga de iniciar cada una de las funciones del agente como hilos dentro del proceso. Los parámetros que recibe son aquellos que le sean adjuntados al llamarlo. Su estructura es la siguiente:

```
public static void main(String[] args) {

    /* LOCALIZACION DE PUERTOS */
    Puerto_Inicio = 50000;
    Rango_Puertos = 100;

    int ports[] = buscaNido();
    UDPport = ports[0];
    TCPport = ports[1];

    /* CONFIGURACION INICIAL */
    generaConfiguracionInicial(args);

    /* GESTOR DE MENSAJES */
    gestorDeMensajes gm = new gestorDeMensajes();

    /* COMPORTAMIENTO BASE */
    comportamientoBase cb =
        new comportamientoBase(ID_propio,
        Numero_de_generaciones, Puerto_Inicio, Rango_Puertos,
        Tiempo_de_vida*1000,
        tiempo_espera_comportamiento_base*1000,
        Frecuencia_partos, Frecuencia_rastreo_puertos, gm);

    /* FUNCION DEL AGENTE */
    funcionDeAgente fa = new funcionDeAgente();

    /* NOTIFICACION DE NACIMIENTO */
    notificaNacimiento();
    Estado_Actual = Estado_del_ACC.VIVO.ordinal();

}
```

- Localización de Puertos: Tras inicializar un puerto de inicio y un rango de puertos, la función buscaNido() devolverá una cadena con dos puertos libres, donde podrá alojarse el *ACC* que va a ser creado. Acto seguido, asignamos el primero (par) para conexiones *UDP*, y el segundo (impar) para aquellas *TCP*.
- Configuración Inicial: generaConfiguracionInicial() es un método que se encarga de dotar al agente con valores para sus variables, es decir, de dejarlo preparado para su arranque.
- Gestor de Mensajes: Apartado prototipo, que en la versión final del sistema se encargará de enviar mensajes como cliente, y de recibirlos como servidor.

- Comportamiento Base: Se inicia la clase comportamientoBase, encargada de crear hijos con el método GenerarNuevoAcc(), de hallar otros agentes con los que establecer conexión con GestorDeDirectorio(), y de gestionar recursos de ingeniería social.
- Función del Agente: Apartado prototipo, que en la versión final del sistema se encargará de generar archivos *XML*, de interpetar los recibidos, y de realizar las funciones propias del agente.
- Notificación de Nacimiento: La función notificaNacimiento() trata de enviarle al monitor, para que lo recopile, información completa sobre cada nacimiento que va sucediendo. Tras todo esto, se puede dar por sentado que el agente ya está vivo, como se indica en su estado.

c) buscaNido()

Autores: G8

Este método está encargado de buscar un par de puertos a los que el agente se va a conectar para instalarse. Devuelve una cadena de enteros, conteniendo dichos puertos libres:

```
static int[] buscaNido() {

    /* SECCION 1 */
    ServerSocket socket = null;
    ServerSocket socket2 = null;
    int[] ports = { 0, 0 };

    Random r = new Random();
    int n = r.nextInt(Puerto_Inicio + Rango_Puertos)
        + Puerto_Inicio;

    /* SECCION 2 */
    while (true){

        while (n% 2 != 0){ n = r.nextInt
            (Puerto_Inicio + Rango_Puertos) + Puerto_Inicio; }

        try {
            socket = new ServerSocket(n);
            socket2 = new ServerSocket(n + 1);
            assert socket != null;

                /* SECCION 3 */
                try {
                    socket.close();
                    socket2.close();
                } catch (IOException e) { e.printStackTrace(); }

                ports[0] = n;
                ports[1] = n + 1;
        } catch (IOException e) { continue; }

        /* SECCION 4 */
        return ports;
    }

}
```

- Sección 1: Inicializamos a *null* un *ServerSocket* para cada puerto: uno encargado de la localización, *UDP*, y otro para la negociación, *TCP*. Consigo, inicializamos un vector de dos elementos que guardará los puertos disponibles para el agente.
- Sección 2: Trataremos de buscar estos dos puertos:
    1. Se buscará de manera aleatoria un número par dentro del rango de puertos disponibles, para después intentar abrir dos *sockets* con los que conectarse a esos puertos.

2. En caso de encontrarse *sockets* ocupados justo en esos puertos, saltará una excepción. De ser así, se repite la misma iteración del bucle, hasta encontrar un par de *sockets* disponibles.

- Sección 3: Una vez abiertos los *sockets*, y comprobado que no causan errores ni excepciones, se cierran para que más tarde el agente pueda utilizarlos.

- Sección 4: Al final, devolveremos un vector de 2 elementos, en la que su primera posición guarda el puerto *UDP*, y la segunda el puerto *TCP*.

d) generaConfiguracionInicial()

Autores: G1 y G8

Este método se encarga de generar una configuración para el agente una vez es generado, con la inicialización de sus variables propias. Su estructura es la siguiente:

```
static void generaConfiguracionInicial(String[] args) {

    /* SECCION 1 */
    try {
        Ip_Propia = InetAddress.getLocalHost().getHostAddress();
    } catch (UnknownHostException e)
        { throw new RuntimeException(e); }

    /* SECCION 2 */
    ID_propio = Ip_Propia + "-" +
        UDPport + "-" + TCPport + args[0];

    /* SECCION 3 */
    Numero_de_generaciones = Integer.parseInt(args[1]) - 1;
    Tiempo_de_vida = 10; // <- Segundos
    tiempo_espera_comportamiento_base = 3; // <- Segundos
    Puerto_Monitor = 40000;
    Frecuencia_partos = 0.5;
    Frecuencia_rastreo_puertos = 0.7;

    /* SECCION 4 */
    try {
        System.setOut(new PrintStream(new File
            ("C:/ACC-Multiagentes/salida" + ID_propio + ".txt")));
    } catch (FileNotFoundException e)
        { throw new RuntimeException(e); }

    System.out.println("generaConfiguracionInicial");

}
```

- Sección 1: Se le asigna a la variable *Ip_Propia* la *IP* de la máquina. Acto seguido, y en caso de no obtenerse dicha *IP* del host, salta una excepción.
- Sección 2: Se genera una *ID* única para cada agente, que consiste en la concatenación de la *IP* de la máquina a la que pertenece el agente, su puerto *UDP* y su puerto *TCP*, y un número adicional que los padres pasan a los hijos. Esta cadena se asigna a la variable *ID_propio*.
- Sección 3: En esta sección, se inicializan varias variables:
    1. Se le resta una cifra al valor de la variable *Numero_de_generaciones*, de cara al correcto funcionamiento del código.
    2. Se le asigna al agente un tiempo vital en segundos.

3. Se define un lapso de espera en segundos para arrancar las funciones GenerarNuevoAcc() y GestorDeDirectorio(). Esta prórroga se produce en cada bucle de la clase comportamientoBase.
4. Se especifica el puerto donde hallar al monitor, con intención de que el agente se comunique con él.
5. Por último, se establece un porcentaje de éxito que debe tener el agente para conseguir parir un hijo, y otro para ponerse a buscar a agentes con los que intercambiar cromos entre sí.

- Sección 4: Se establece un directorio de salida donde generar un archivo de texto plano en el que cada agente podrá dejar por escrito cada una de sus operaciones. Si esta dirección no es variada, el archivo se guardará por defecto en *"C:/ACC-Multiagentes/salida"*. En caso de que no se pueda crear dicho archivo, salta una excepción.

e) notificaNacimiento()

Autores: G3 y G4

La esencia de este método reside en la de trasmitir información completa sobre cada nacimiento que va dando lugar. Toda esta quedará recopilada dentro del monitor que hemos especificado:

```
static void notificaNacimiento() {

    try {

        /* SECCION 1 */
        monitorComunicacion =
            new Socket(Ip_Propia, Puerto_Monitor);

        /* SECCION 2 */
        out = new DataOutputStream
            (monitorComunicacion.getOutputStream());

        /* SECCION 3 */
        out.writeUTF("El agente con id: " + ID_propio + "\nIp "
            + Ip_Propia + "\nPuerto \textit{UDP} " + UDPport
            + "\nPuerto TCP " + TCPport
            + "\nha creado un nuevo agente");

        System.out.println("Se ha enviado la notificacion de
    nacimiento");

    /* SECCION 4 */
    } catch (IOException e) {
        System.out.println(" No se ha podido enviar la
            notificacion de nacimiento");
    }

}
```

- Sección 1: Creamos un *socket* que nos permitirá enviar información al monitor, en este caso una notificación de que el agente ha nacido. Para ello, introducimos como parámetros la *IP* de la máquina y el puerto del monitor, para crear la conexión.
- Sección 2: Asignamos a la variable *out*, de tipo *DataOutputStream*, el *socket* recién creado. *out* nos permitirá escribir datos de tipo primitivo, en esta caso cadenas de caracteres, de forma portable.
- Sección 3: Escribimos la notificación de nacimiento del agente al monitor. Esta dejará plasmado tanto su *ID* y puertos asignados, como la *IP* del host al que pertenece.
- Sección 4: En caso de que ocurriese algún error en la transmisión de dicho mensaje, se indicaría por terminal.

CLASE 2 - comportamientoBase

a) Variables globales

La mayoría de variables son heredaras de la clase Acc, las cuales serán construidas en el método comportamientoBase(). Las demás se declaran por primera vez:

```
1     String id, direccionJar;
2     long horaDeMuerte;
3     int generaciones, Puerto_Inicio, Rango_Puertos, tiempoDeVida,
4         tiempo_espera_comportamiento_base;
5     double Frecuencia_partos, Frecuencia_rastreo_puertos;
6     ArrayList<accLocalizado> contenedor_directorio_ACCs;
7     gestorDeMensajes gm;
8     Random random = new Random();
```

- String id: *ID* que representa al agente, el cual será único.
- String direccionJar: Directorio donde se encontrará el ejecutable *ACC.jar*.
- long horaDeMuerte: Variable que almacena la hora, en milisegundos, en la que el agente cesará su acción. Será comparada con la hora interna de la máquina para matar el proceso cuando esta haya sido alcanzada.
- int generaciones: Número de generaciones, contándole a él, que un agente tendrá dentro de su árbol genealógico. Si el número de generaciones es 3, el agente solo podrá llegar a ser abuelo.
- int Puerto_Inicio: Existirá un rango de puertos de entre los que buscar dentro de una máquina. Esta variable marca el primer puerto de este.
- int Rango_Puertos: Cantidad de puertos, a partir del principio del rango, *Puerto_Inicio*, de entre los que buscar agentes alojados en ellos.
- int tiempoDeVida: Tiempo vital, expresado en segundos, que el agente vivirá e interactuará con los demás agentes, antes de perecer.
- int tiempo_espera_comportamiento_base: Un periodo de tiempo de espera, expresado en segundos, para limitar el acceso a las clases GenerarNuevoAcc() y GestorDeDirectorio().
- double Frecuencia_partos: Porcentaje de probabilidad, del 0 a 1, de que un agente consiga tener un hijo en un intento.
- double Frecuencia_rastreo_puertos: Porcentaje de probabilidad, del 0 a 1, de que un agente se ponga a buscar a otros agentes en un intento
- ArrayList<accLocalizado> contenedor_directorio_ACCs: Cadena de tipo **accLocalizado** que almacena *ID*, *IP* y puerto de todos los agentes localizados.

- gestorDeMensajes gm: Variable de tipo clase **gestorDeMensajes**, utilizado en la función GestorDeDirectorio() para poder ejecutar métodos de las clases **gestorDeMensajes** y **MensajeAEnviar**.
- Random random: Se utilizará para generar números aleatorios y simular la proactividad del agente.

b) comportamientoBase()

Autores: G1

A pesar del nombre, este método actúa como constructor, de cara a la ejecución satisfactoria de la clase comportamientoBase, donde nos encontramos. Recibe como parámetros varias variables declaradas en la clase Acc, e inicializadas en generaConfiguracionInicial():

```
comportamientoBase(String id, int generaciones, int puerto_Inicio,
    int rango_Puertos, int tiempoDeVida,
    int tiempo_espera_comportamiento_base,
    double frecuencia_partos, double frecuencia_rastreo_puertos,
    gestorDeMensajes gm) {

    /* SECCION 1 */
    this.id = id;
    this.generaciones = generaciones;
    this.Puerto_Inicio = puerto_Inicio;
    this.Rango_Puertos = rango_Puertos;
    this.tiempoDeVida = tiempoDeVida;
    this.tiempo_espera_comportamiento_base =
        tiempo_espera_comportamiento_base;

    this.Frecuencia_partos = frecuencia_partos;
    this.Frecuencia_rastreo_puertos = frecuencia_rastreo_puertos;

    /* SECCION 2 */
    this.horaDeMuerte =
        System.currentTimeMillis() + (long) tiempoDeVida;

    /* SECCION 3 */
    this.direccionJar = "C:/ACC-Multiagentes/ACC.jar";

    this.gm = gm;

    /* SECCION 4 */
    new Thread(this, "comportamiento_base").start();

}
```

- Sección 1: Actúa como constructor, asignando a las variables del agente los valores pasados por parámetro.
- Sección 2: Calcula la variable *horaDeMuerte* a partir de la hora que indica el reloj del host, sumándole el tiempo de vida del agente, ambos en milisegundos.
- Sección 3: Se establece la dirección del archivo *JAR*, denominado *ACC.jar*, que utilizará el agente. Por defecto, *"C:/ACC-Multiagentes/ACC.jar"*.
- Sección 4: Se crea un nuevo hilo que ejecutará los métodos de esta clase, comenzando con el inicio de run().

c) run()

Autores: G1

Tras el constructor comportamientoBase(), este método actúa como función principal de la clase comportamientoBase. Este se encarga de llamar a las funciones GenerarNuevoAcc() y GestorDeDirectorio() en caso de cumplirse las condiciones necesarias expuestas a continuación:

```
@Override
public void run() {

    /* SECCION 1*/
    int i = 0;
    while (horaDeMuerte > System.currentTimeMillis()) {

        /* SECCION 2 */
        i++;
        GenerarNuevoAcc(id, i);

        /* SECCION 3 */
        if(this.Frecuencia_rastreo_puertos >=
            this.random.nextDouble())
            this.GestorDeDirectorio();

        /* SECCION 4 */
        try { Thread.sleep(tiempo_espera_comportamiento_base);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    /* SECCION 5 */
    System.out.println("fin");
    // Detiene al agente
    System.exit(0);

}
```

- Sección 1: Se repetirá el bucle, siempre y cuando el reloj de la máquina no haya alcanzado la hora de la muerte del agente.
- Sección 2: Se da inicio al método GenerarNuevoAcc(), introduciéndole el *ID* del padre y el número de intentos de parir que han sido realizados.
- Sección 3: En caso de que se supere de manera aleatoria el porcentaje de éxito de rastrear puertos, se ejecutará el método GestorDeDirectorio()
- Sección 4: Antes de terminar el bucle, habrá un periodo de espera, previamente establecido, para controlar las iteraciones del bucle que el agente realiza en este método. En caso de que esto genere alguna interrupción, se dará fin al programa.
- Sección 5: Con solo ejecutar *System.exit(0)*, todo el proceso, junto a sus hilos, dará fin. Es decir, el agente metafóricamente morirá.

d) GenerarNuevoAcc()

Autores: G1

Este método se encargará de crear procesos, entendidos como hijos del agente, siempre y cuando se satisfagan una serie de condiciones necesarias. De lo contrario, no realiza ninguna acción:

```
void GenerarNuevoAcc(String id, int i) {

    /* SECCION 1 */
    try {
        if (Frecuencia_partos < random.nextDouble()
            && generaciones > 0) {
            /* SECCION 2 */
            ProcessBuilder pb = new ProcessBuilder
                ("C:/Program Files/Java/jdk-17.0.4.1/bin/java.exe",
                "-jar", direccionJar, "" + i, "" + generaciones);
            pb.start();
        }
    } catch (Exception e) { throw new RuntimeException(e); }

}
```

- Sección 1: Esta condición se ejecutará y, por lo tanto, se generará un hijo, solo si se supera el porcentaje de éxito de parir de manera aleatoria, y si no se ha alcanzado la generación 0. Alcanzar dicha generación implicaría que el árbol genealógico ha llegado a su límite.
- Sección 2: Superada la condición, inicializaremos un ProcessBuilder llamado *pb*, que ejecutará con *JDK 17.0.4.1* el ejecutable *ACC.jar*, situado en el directorio introducido en *direccionJar*. En adición, incluiremos en la directiva los intentos de parir del agente, y la generación a la que pertenecerá el hijo que se va a generar. Acto seguido, iniciamos *pb*, lo cual creará un proceso hijo.

¡¡¡ADVERTENCIA!!!

Para que este módulo funcione, debe ser instalado *JDK 17.0.4.1* en todas las máquinas participantes del sistema multiagente. De lo contrario, hallará errores y comportamientos inesperados. Puede descargar el instalador de *Microsoft Windows* haciendo clic aquí (153MB), o descargarlo para su sistema y arquitectura específicos en esta página.

Para confirmar que el *JDK* ha sido instalado, asegúrese de que aparece en el siguiente directorio: *"C:/Program Files/Java/jdk-17.0.4.1/bin/java.exe"*.

e) GestorDeDirectorio()

Autores: G2 y G7

Este método se encarga de comprobar el tipo de búsqueda de *IPs* y puertos, y de enviar mensajes de localización a agentes, con tal de encontrar alguno con el que intercambiar cromos:

```
void GestorDeDirectorio() throws InterruptedException {

    /* BUSQUEDA ALEATORIA */
    if(this.puertos_aleatorios){
        for (int i = 0; i < 254; i++) {

            String host = "172.19.154." + i;
            int puertos_a_buscar = 5;

            for(int j = 0; j < puertos_a_buscar ; j++) {

                int puerto = this.Puerto_Inicio +
                    this.random.nextInt(this.Rango_Puertos);

                if (puerto % 2 != 0) { puerto++; }

                System.out.println("envia mensaje");

                try { this.gm.EnviarMensaje();
                } // <- Gestion de excepciones

            }

        }

    /* BUSQUEDA NO ALEATORIA */
    } else {
        for (int i = 0; i < 254; i++) {

            String host = "172.19.154." + i;

            for (int puerto = Puerto_Inicio;
                puerto <= Puerto_Inicio + Rango_Puertos;
                puerto += 2) {

                try { this.gm.EnviarMensaje();
                } // <- Gestion de excepciones

            }

        }
    }
}
```

- Búsqueda aleatoria: Si la búsqueda es aleatoria, se recorrerían todas las *IPs* y, para cada una de ellas, en función del número de puertos *puertos_a_buscar_aleatorios*, obtendremos puertos aleatorios en el rango *[puerto_inicio, puerto_inicio + rango_puertos]*. Una vez que los

tengamos, comprobaremos que son pares ya que, si no lo fueran, serían puertos *TCP*, por lo que sumamos una cifra al puerto. Una vez hecho esto, se añadiría un mensaje a la cola de la clase **GestorMensajes**, *contenedor_de_mensajes_a_enviar*.

- Búsqueda no aleatoria: si no es aleatoria, recorremos todas las *IPs*, desde la del atributo *inicio_rango_ips* hasta la de *fin_rango_ips*. Además, recorremos todos los puertos de cada *IP*, correspondientes al rango establecido por *puerto_inicio* y *rango_puertos*, de dos en dos para solo hallar puertos *UDP*. Una vez hecho esto, se añadiría un mensaje a la cola de la clase **GestorMensajes**, *contenedor_de_mensajes_a_enviar*.

f) siguienteIP()

Autores: G2 y G7

Para buscar entre un rango de *IPs*, necesitaremos un último método para la clase comportamientoBase que calcule sistemáticamente cuál es la siguiente *IP* a comprobar desde aquella en la que nos encontramos.

```
public static String siguienteIP(String ip) {

    /* SECCION 1 */
    List<String> IP_split = Arrays.asList(ip.split("\\."));
    int[] IP_int = new int[4];

    for(int i = 0; i<4; i++) {
        IP_int[i] = Integer.parseInt(IP_split.get(i));
    }

    /* SECCION 2 */
    if(IP_int[3] < 255) { IP_int[3] += 1; }
    else {
        IP_int[3] = 0;
        if(IP_int[2] < 255) { IP_int[2] += 1; }
        else {
            IP_int[2] = 0;
            if(IP_int[1] < 255) { IP_int[1] += 1; }
            else {
                IP_int[1] = 0;
                IP_int[0] += 1;
            }
        }
    }

    /* SECCION 3 */
    return IP_int[0] + "." + IP_int[1]
        + "." + IP_int[2] + "." + IP_int[3];

}
```

- Sección 1: Separamos cada byte de la *IP* mandada como parámetro, y pasamos cada una de sus cuatro secciones a la cadena de números enteros *IP_int*.
- Sección 2: Vamos sumando una cifra al último campo de la *IP* para hallar la siguiente. En caso de que le sumemos 1 a un byte que tenga el valor 255, lo estableceremos a 0 y sumaremos al byte de la izquierda una cifra. Consecuentemente, se volvería a realizar esta comprobación con todos los bytes restantes de la cadena.
- Sección 3: Devolvemos la *IP* calculada como una cadena de caracteres.