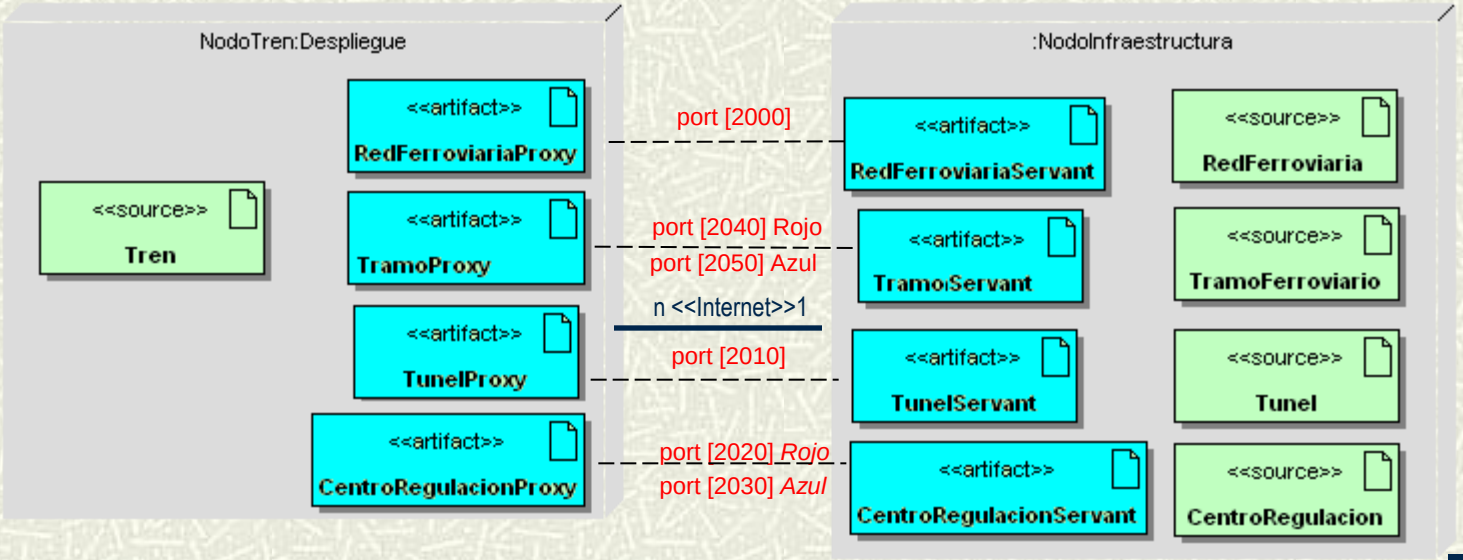
# PROGRAMACION CONCURRENTE Y DISTRIBUIDA

## VI.2: Red Ferroviaria ConSocket


Laura Barros

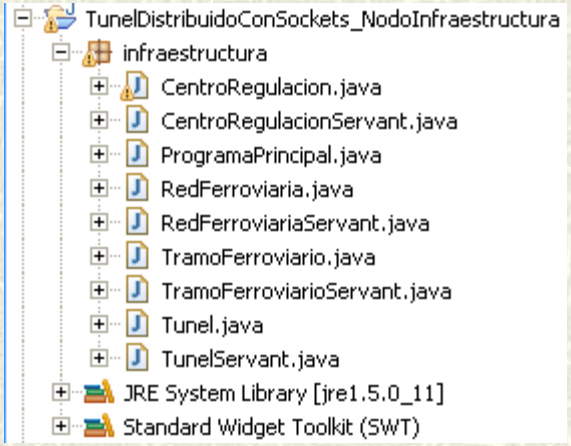
# Objetivo

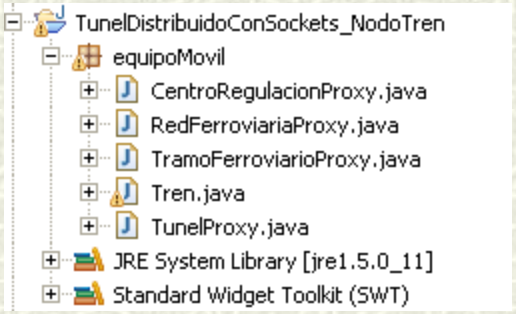
- En esta versión del proyecto Red Ferroviaria se distribuye el código en varios procesadores.
  - El procesador **NodoInfraestructura** ejecuta el código relativo a los elementos situados en tierra (infraestructura) (***RedFerroviaria, Tunel, TramoFerroviario y CentroRegulación***).
  - Hay un procesador **NodoTren** por cada uno de los trenes que circulan que ejecuta el código relativo al control de su marcha (***Tren***). La interacción entre los trenes y la infraestructura se realiza utilizando el paradigma Proxy-Servant, y la comunicación entre estos se realiza utilizando sockets.



# Punto de partida I

- Partimos de la aplicación **RedFerroviariaLockJava**, a la que hemos cambiado su estructura para poder ser distribuida en varios procesadores.
- El código se ha separado en dos particiones (paquetes):
  - El paquete ***infraestructura***: contiene las clases que se instancian en el procesador **NodoInfraestructura**.
  - El paquete ***equipoMovil***: contiene las clases que se embarcan en el procesador **NodoTren**.



# Punto de partida II

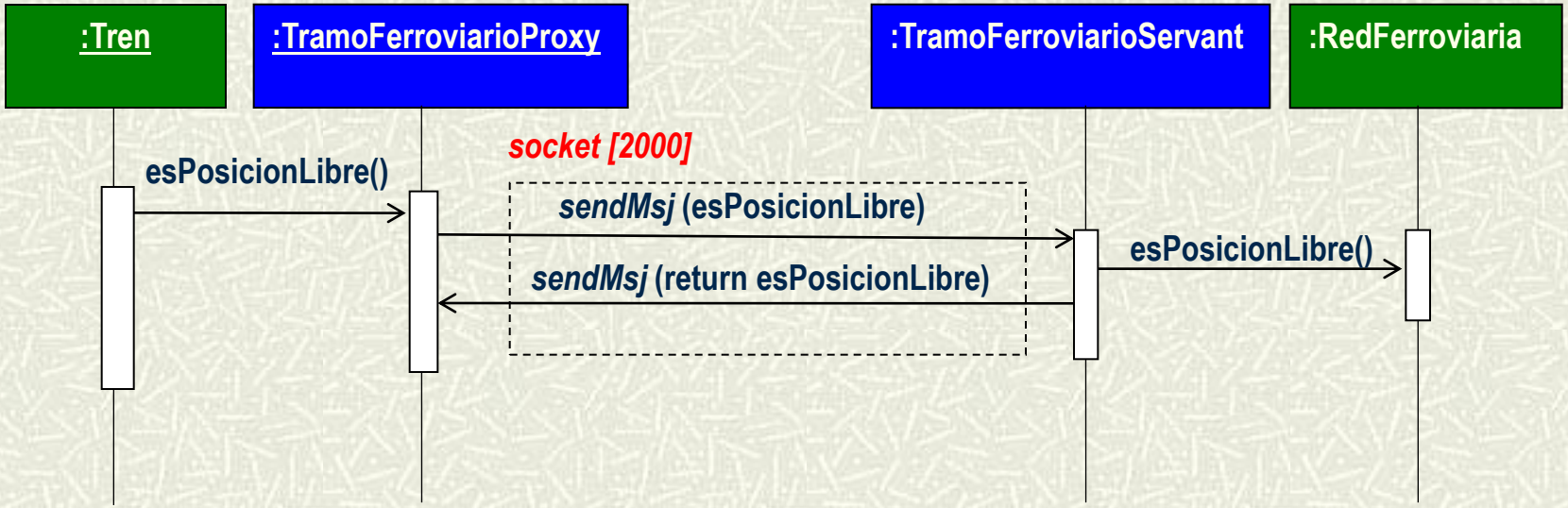
- Las modificaciones que se han hecho respecto de RedFerroviariaLockJava son:
  - **Finalización de la actividad de los trenes**: El método ***entro()*** de la clase CentroRegulación retorna un **booleano**. Lo obtiene el *tren* cuando se accede al CentroRegulación para anunciar que entra en el centro de regulación.
    - si es ***true*** :actividad del tren debe finalizar definitivamente.
    - si es ***false***: el tren debe continuar.
  - **Proxys y Servants**: La **conexión** de cada ***tren*** con las clases de la ***infraestructura*** se ha realizado a través de parejas complementarias Proxy-servant.
    - La clase ***Tren*** tiene instancias de las clases: ***RedFerroviariaProxy***, ***TramoFerroviarioProxy, TunelProxy***, ***CentroRegulacionProxy***.
      - Cada ****Proxy*** ofrece localmente al tren, el conjunto de operaciones que éste requiere del objeto que representa (***RedFerroviaria***,***TramoFerroviario***, ***Tunel***, ***CentroRegulacion***).

# Punto de partida III

- **...y Servants**:
  - La clase ***RedFerroviaria*** tiene agregada una instancia de la clase ***RedFerroviariaServant***.
  - La clase ***Tunel*** tiene agregado una instancia de la clase ***TunelServant.***
  - La clase ***CentroRegulacion*** tiene agregado una instancia de la clase ***CentroRegulacionServant***.
  - La clase ***TramoFerroviario*** tiene agregado una instancia de la clase ***TramoFerroviarioServant***.
  - Los ****Servant*** ejecutan por delegación en el elemento al que pertenecen los requerimientos que se reciben de los trenes (a través de los proxies).
- ¿Cómo interacciona un ***tren*** con un elemento de la ***infraestructura***?
  1. Invoca un método del correspondiente ****Proxy*** que es local.
  2. El ****Proxy*** , a través de un **socket**, se comunica con el correspondiente ****Servant***.
  3. El ****Servant*** ejecuta la interacción con el elemento.

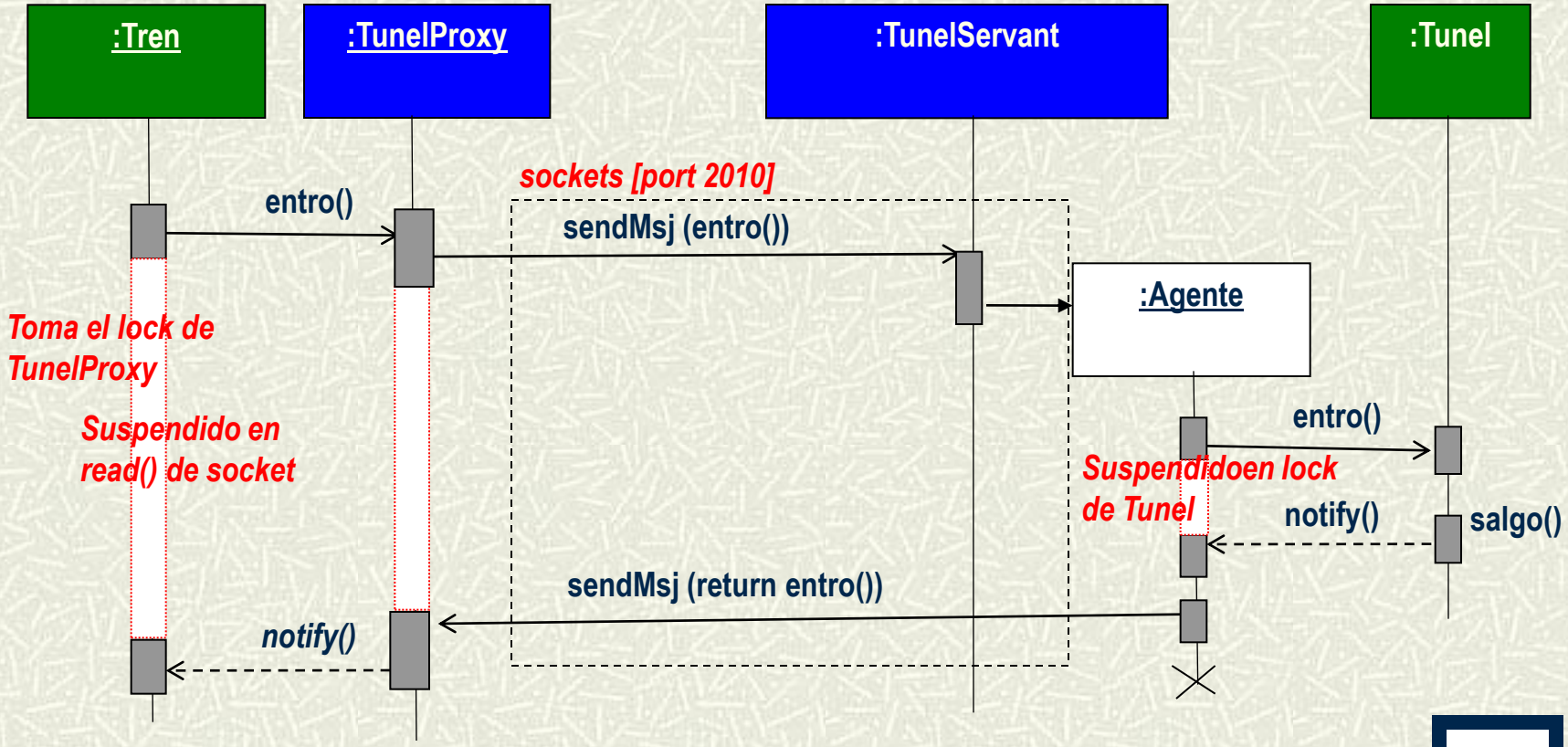
# Punto de partida III

- Las interacciones que los ***trenes*** realizan sobre los elementos de la ***infraestructura*** pueden ser de dos tipos:
  - **No bloqueantes**: No implica ningún bloqueo prolongado del thread del tren. Ejemplos de este tipo de interacción son:
    - salgo():void de la clase *Tunel*
    - esPosicionLibre(posición,tramo):boolean de la clase *TramoFerroviario*.



# Punto de partida IV

- **Bloqueantes**: El thread del tren se suspende hasta que en el elemento de la *infraestructura* se alcanza el estado que responde. Estas son:
  - entro():boolean de la clase *Tunel*.
  - entro(tramo):void de la clase *CentroRegulacion*.



# Arquitectura de la aplicación I

- **Clases incluidas en la partición *infraestructura*:**
  - Las clases ***RedFerroviaria, TramoFerroviario y Tunel*** son básicamente iguales a las utilizadas en la aplicación **RedFerroviariaLockJava**.
  - La clase ***CentroRegulación*** es también la misma, salvo que incorpora el mecanismo de finalización de los trenes. Los cambios son:

    Atributo nuevo:

    *terminado:boolean=false =>* Vale True si la actividad de los trenes debe finalizar.

    Métodos nuevos:

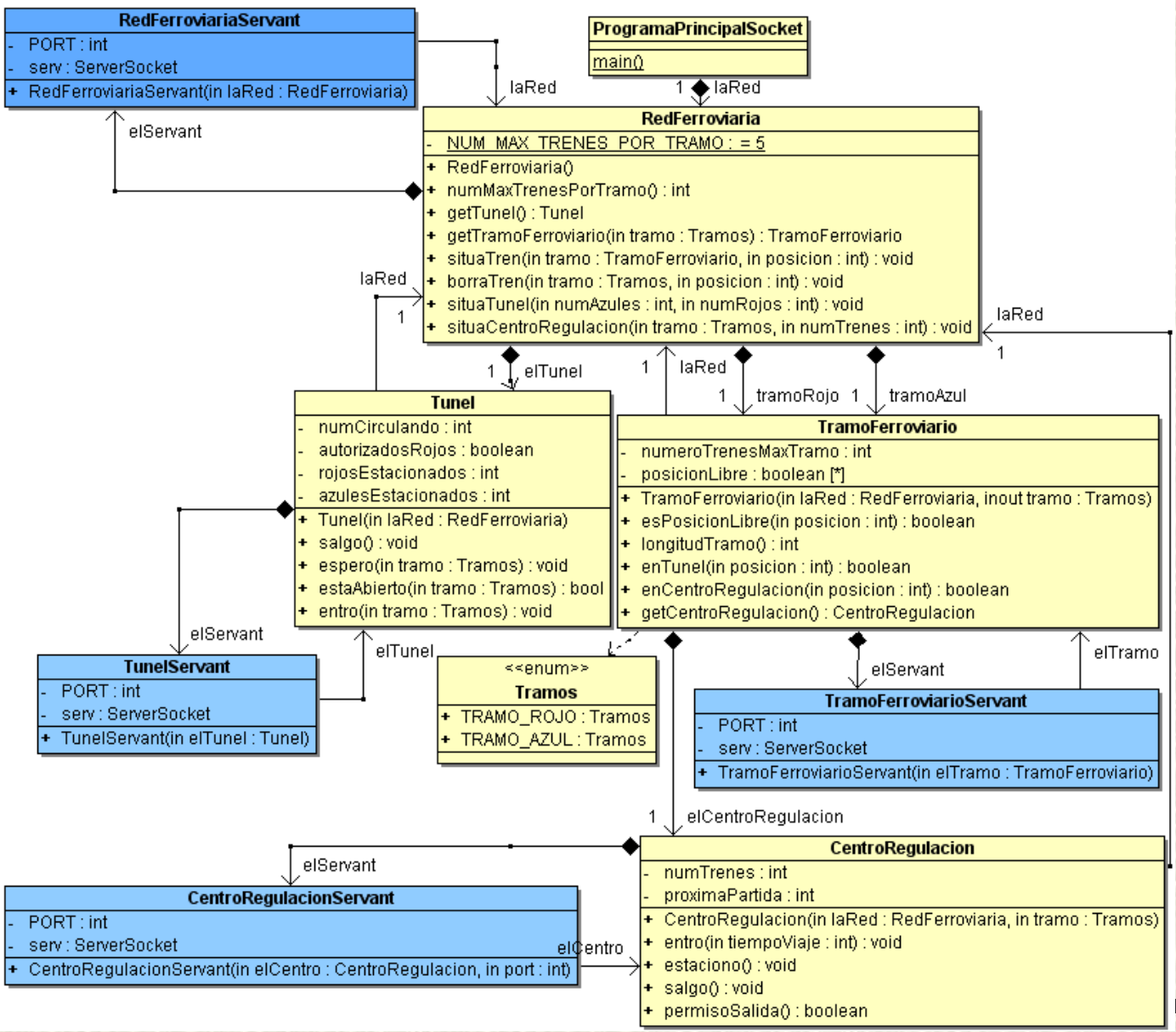
    *termina() =>* Establece que la actividad de los trenes del tramo ferroviario debe finalizar.

    *entro():boolean =>*
    - Bloquea el thread entrante hasta que llegue la hora de salida de tren.
    - Retorna true si la actividad del tren que invoca debe terminar.

# Centro de Regulacion

```
/*Un tren informa de que entra en el centro de regulación. */
 public synchronized boolean entro(){
      numTrenes=numTrenes+1;
      laRed.muestraCentroRegulacion(tramo, numTrenes);
      while(proximaPartida>System.currentTimeMillis()){
            try{
                  wait(100);
            }catch(InterruptedException e){};
      }
       proximaPartida=System.currentTimeMillis()+100*laRed.LAST_POSITION/
                        (laRed.numMaxTrenesPorTramo()-2);
       return terminado;
 }
 /**
  * Se establece que la actividad de los trenes del tramo debe concluir.
  */
 public void termina(){
      terminado=true;
 }
```

# Arquitectura de la aplicación II



# Nuevas Clases I

- Clase ***RedFerroviariaServant***:
  - Servant que ejecuta los metodos de la *RedFerroviaria* que requiere cualquier tren a través del **Proxy* complementario utilizando el socket que se establece entre ellos.
    - Recibe por el socket de puerto 2000 como mensajes los métodos que los trenes realizan sobre la RedFerroviaria.
    - Invoca el método requerido en *RedFerroviaria.*
    - Retorna el resultado.

Constructor:

*RedFerroviariaServant(laRed:RedFerroviaria)*=> Recibe como parámetro la referencia a la *RedFerroviaria* de la que es servant.

Atributos:

*laRed:RedFerroviaria* => Referencia a la *RedFerroviaria* sobre el que opera. Se establece en su constructor

*<<final>>PORT:int=2000* => Puerto del socket por el que el **Servant* queda a la espera de recibir mensajes de los **Proxies* complementarios. Es constante [2000]

*ServerSocket* serv => Server que atiende los requerimientos que recibe de los **Proxies* complementarios por su puerto.

# Nuevas Clases II

- Clase ***TunelServant:***
  - Servant que ejecuta los métodos del *Tunel* que requiere cualquier tren a través del **Proxy* complementario utilizando el socket que se establece entre ellos.
    - Recibe por el socket de puerto 2010 como mensajes las invocaciones de los métodos que los trenes realizan sobre el *Tunel.*
    - Invoca por el método requerido en Tunel.
    - Retorna el resultado.

Constructor:

*TunelServant(elTunel:Tunel)*=> Recibe como parámetro la referencia al *Tunel* del que es servant.

Atributos:

*elTunel:Tunel* => Referencia el *Tunel* sobre el que opera. Se establece en su constructor.

*<<final>>PORT:int=2010* => Puerto del socket por el que el **Servant* queda a la espera de recibir mensajes de los **Proxies* complementarios. Es constante [2010]

ServerSocket serv => Server que atiende los requerimientos que recibe de los **Proxies* complementarios por su puerto.

# Nuevas Clases III

- Clase ***CentroRegulacionServant***:
  - Servant que ejecuta los métodos del ***CentroRegulacion*** que requiere cualquier tren a través del ****Proxy*** complementario utilizando el socket que se establece entre ellos.
  - Recibe por el socket de puertos 2020 (tramo Rojo) y 2030 (tramo Azul) como mensajes los métodos que los trenes realizan sobre el correspondiente ***CentroRegulacion.***
  - Invoca el método requerido en ***CentroRegulacion.***
  - Retorna el resultado.

Constructor:

*CentroRegulacionServant(elCentroRegulacion:CentroRegulacion, port:int)*=> Recibe como parámetro la referencia al CentroRegulación del que es servant, y el puerto (2020 - tramo Rojo ,2030 - tramo Azul).

Atributos:

*elCentroRegulacion:CentroRegulacion* => Referencia el *CentroRegulacion* sobre el que opera. Se establece en su constructor.

*<<final>>PORT:int* => Puerto del socket por el que el **Servant* queda a la espera de recibir mensajes de los **Proxies* complementarios. Se establece en el constructor. Puede tomar los valores [2020] tramo rojo y [2030] tramo azul.

ServerSocket serv => Server que atiende los requerimientos que recibe de los **Proxies* complementarios por su puerto.

# Nuevas Clases IV

- Clase ***TramoFerroviarioServant***:
  - Servant que ejecuta los métodos del ***TramoFerroviario*** que requiere cualquier tren a través del ****Proxy*** complementario utilizando el socket que se establece entre ellos.
  - Recibe por el socket de puertos 2040 (tramo Rojo) y 2050 (tramo Azul) como mensajes los métodos que los trenes realizan sobre el correspondiente ***TramoFerroviario.***
  - Invoca el método requerido en ***TramoFerroviario.***
  - Retorna el resultado.

Constructor:

*TramoFerroviarioServant(elTramo:TramoFerroviario,port:int)*=> Recibe como parámetro la referencia al TramoFerroviario del que es servant, y el puerto (2040 - tramo Rojo ,2050 - tramo Azul).

Atributos:

*elTramo:TramoFerroviario* => Referencia el *TramoFerroviario* sobre el que opera. Se establece en su constructor.

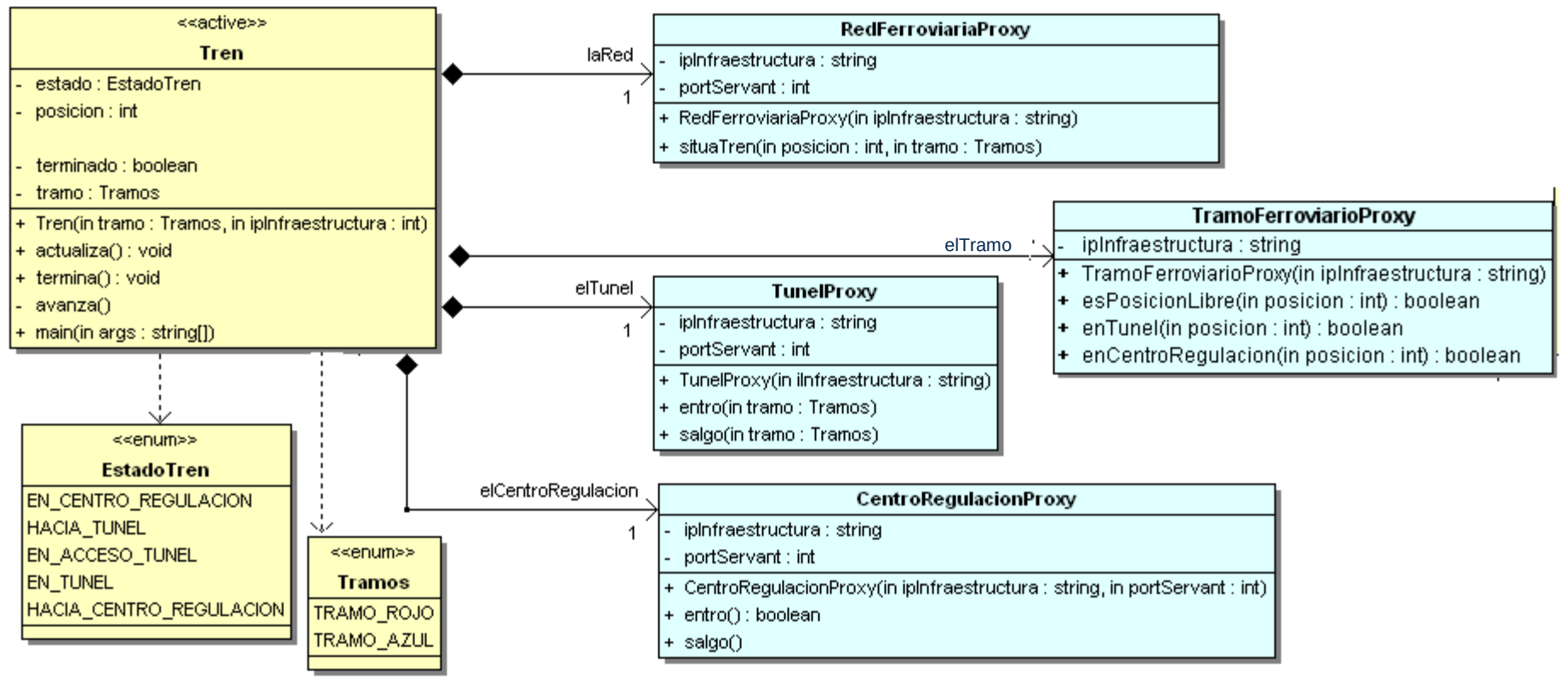
*<<final>>PORT:int* => Puerto del socket por el que el **Servant* queda a la espera de recibir mensajes de los **Proxies* complementarios. Se establece en el constructor. Puede tomar los valores [2040] tramo rojo y [2050] tramo azul.

ServerSocket serv => Server que atiende los requerimientos que recibe de los **Proxies* complementarios por su puerto.

# Nuevas Clases IV

- **Clases incluidas en la partición *equipoMovil*:**

  La clase ***Tren*** es básicamente igual a las utilizadas en la aplicación no distribuida, salvo que se comunica con los elementos incluidos en el paquete *infraestructura* a través de las nuevas clases ***RedFerroviariaProxy, TunelProxy y CentroRegulacionProxy*.**



# Nuevas Clases V

- Cambios en la clase ***Tren***:
  - Se ha eliminado el acceso a la clase *RedFerroviaria*.
  - Se le agregan los **Proxies*:
    - laRed:**RedFerroviariaProxy** => *Proxy que representa localmente y sirve de acceso a la *RedFerroviaria* de **NodoInfraestructura**.
    - elTunel: **TunelProxy** => *Proxy que representa localmente y sirve de acceso al *Tunel* de NodoInfraestructura.
    - elCentroRegulacion:**CentroRegulacionProxy** => *Proxy que representa localmente y sirve de acceso al *CentroRegulacion* de **NodoInfraestructura**, del *TramoFerroviario* sobre el que circula el tren.
    - elTramo:**TramoFerroviarioProxy** => *Proxy que representa localmente y sirve de acceso al *TramoFerroviario*de **NodoInfraestructura**.
  - El constructor contiene como parámetro el IP en la red del procesador **NodoInfraestructura**.
  - Contiene el procedimiento main() que lanza de forma autónoma un tren en el procesador **NodoTren**. Main requiere tres argumentos tipo string:
    - Tramo es uno de los strings “TRAMO_ROJO” o “TRAMO_AZUL”
    - IP de nodo de infraestructura. Es un string con un formato del tipo: “199.155.255.15”
    - Velocidad del tren. Es un entero.

# Tren

```
/**
 *Constructor de la clase Tren. Hay que pasar el identificador del tramo en que circula, el ip del  procesador
 *NodoInfraestructura  y la velocidad a la que circula.
 */
public Tren(Tramos tramo, String ipInfraestructura, int velocidad) {
        this.tramo=tramo;
        this.velocidad=velocidad;
        estado=EstadoTren.EN_CENTRO_REGULACION;
        posicion=0;
        horaPartida=0;
        laRed=new RedFerroviariaProxy(ipInfraestructura);
        elTunel=new TunelProxy(ipInfraestructura);
        switch (tramo){
        case TRAMO_ROJO:
                elCentroRegulacion=new CentroRegulacionProxy(ipInfraestructura,2020);
                elTramo=new TramoFerroviarioProxy(ipInfraestructura,2040);
                break;
        case TRAMO_AZUL:
                elCentroRegulacion=new CentroRegulacionProxy(ipInfraestructura,2030);
                elTramo=new TramoFerroviarioProxy(ipInfraestructura,2050);
                break;
        }
        this.start();
 }
```

# Tren

```
/**
 *  Método que lanza un tren como sistema autónomo. Requiere tres parametros:
 *  args[1] => Tramo en el que circula el tren que se crea ("TRAMO_ROJO" o"TRAMO_AZUL")
 *  args[2] => IP del procesador NodoInfraestructura del que el tren requiere información
 *          para circular. Es un string con el formato "199.255.255.15"
 *  args[3] => Velocida en pasos por segundo con que se mueve el tren (valor típico "5")
 */
static public void main (String[] args){
    int velocidad= Integer.parseInt(args[2]);
    Tramos tramo;
    if (args[0].equals("TRAMO_ROJO")) tramo=Tramos.TRAMO_ROJO;
    else tramo=Tramos.TRAMO_AZUL;
    Tren elTren=new Tren(tramo,args[1],velocidad);
}
```

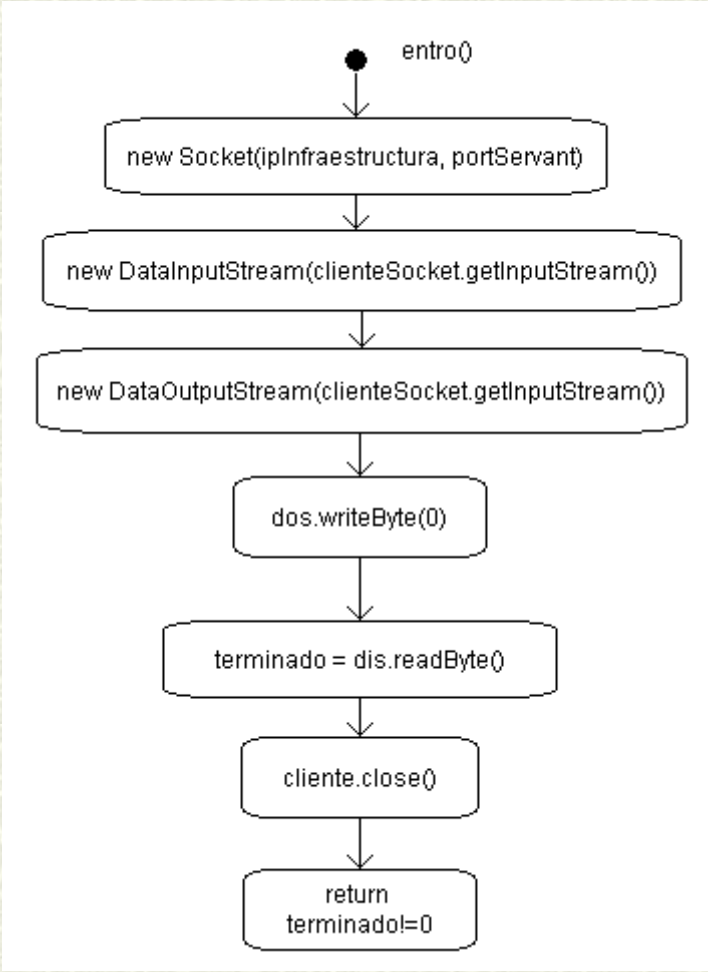
# Formatos de los mensajes entre Proxies y Servants

- Todos los formatos de invocación tiene un primer byte que identifica la operación, y un byte adicional por cada argumento que tenga la operación.
- El retorno puede no existir, o consistir en un byte que es siempre booleano.
- Representación de los valores en los mensajes:
  - Valores boleanos:  false => 0=$00 y true=-1=$FF.
  - Valores Tramos: TRAMO_ROJO => 0 y TRAMO_AZUL=>1
  - posición: 0-255 (octeto sin signo)
- Contenidos de los mensajes de invocación (de *Proxy a *Servant) y de retorno (de *Servant a *Proxy).

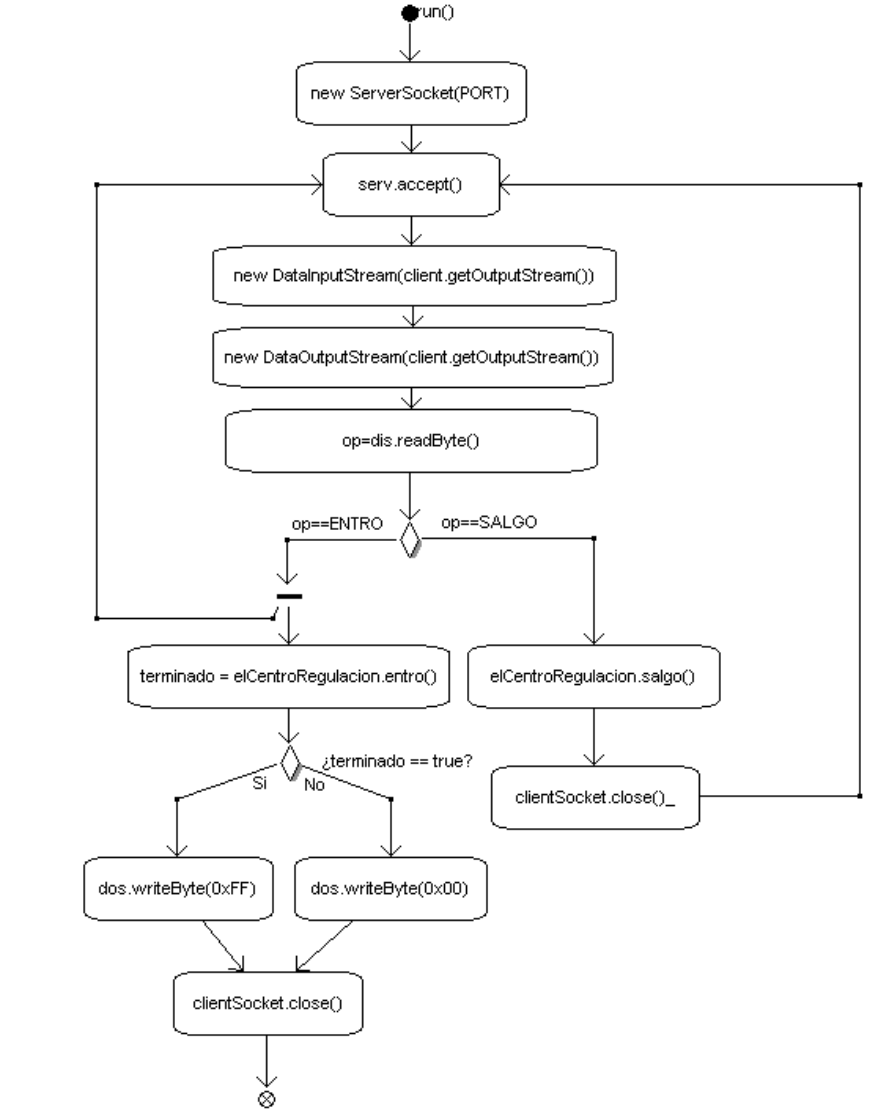
# Codificación de los mensajes entre Proxies y Servants

| Operación | Op | Param1 | Param2 | Return |
|---|---|---|---|---|
| RedFerroviaria | | | | |
| **situaTren(posición:int, tramo: Tramos)** | 0 | Posición | Tramo | No hay |
| **borraTren(posición:int,tramo:Tramos)** | 1 | Posicion | Tramo | No hay |
| TramoFerroviario | | | | |
| **esPosicionLibre(posición):boolean** | 0 | Posición | No hay | bool |
| **enCentroRegulacion(posición:int):boolean** | 1 | Posicion | No hay | bool |
| **enTunel(posición:int):boolean** | 2 | Poscion | No hay | bool |
| Tunel | | | | |
| **entro(tramo:tramos):boolean** | 0 | Tramo | No hay | bool |
| **salgo()** | 1 | No hay | No hay | No hay |
| CentroRegulacion | | | | |
| **entro: boolean** | 0 | Tramo | No hay | bool |
| **salgo()** | 1 | No hay | No hay | No hay |

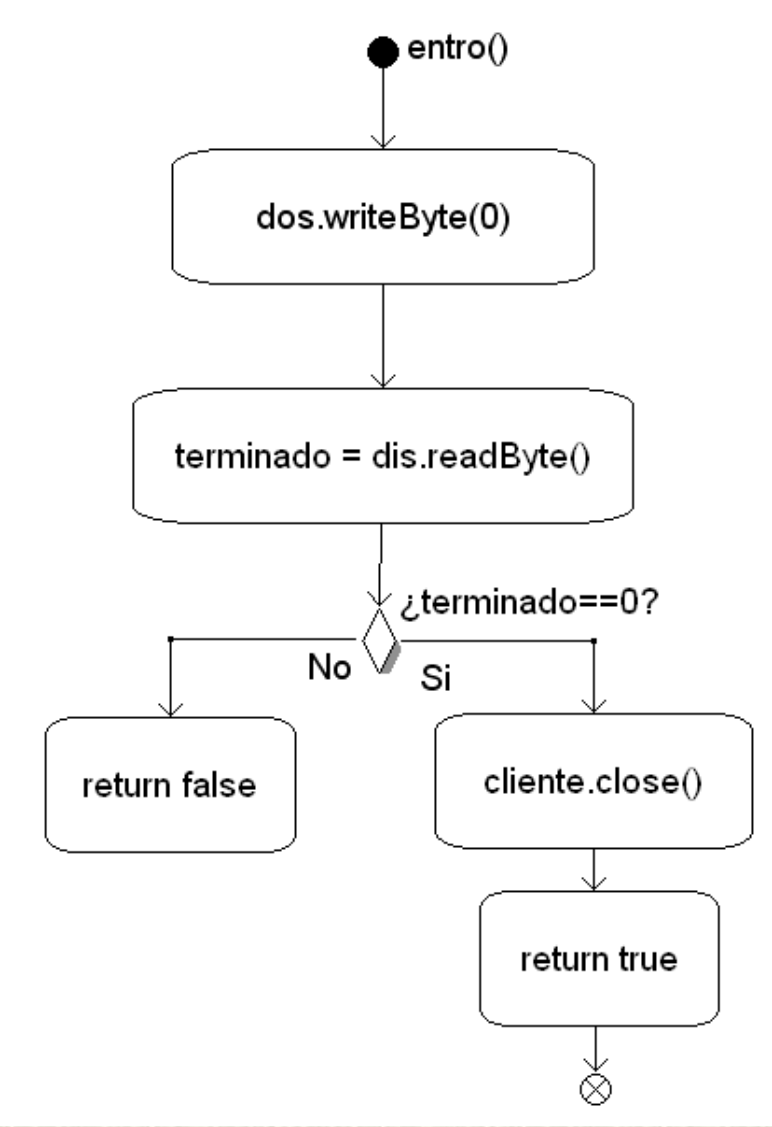
# CentroRegulacionProxy Implementación I



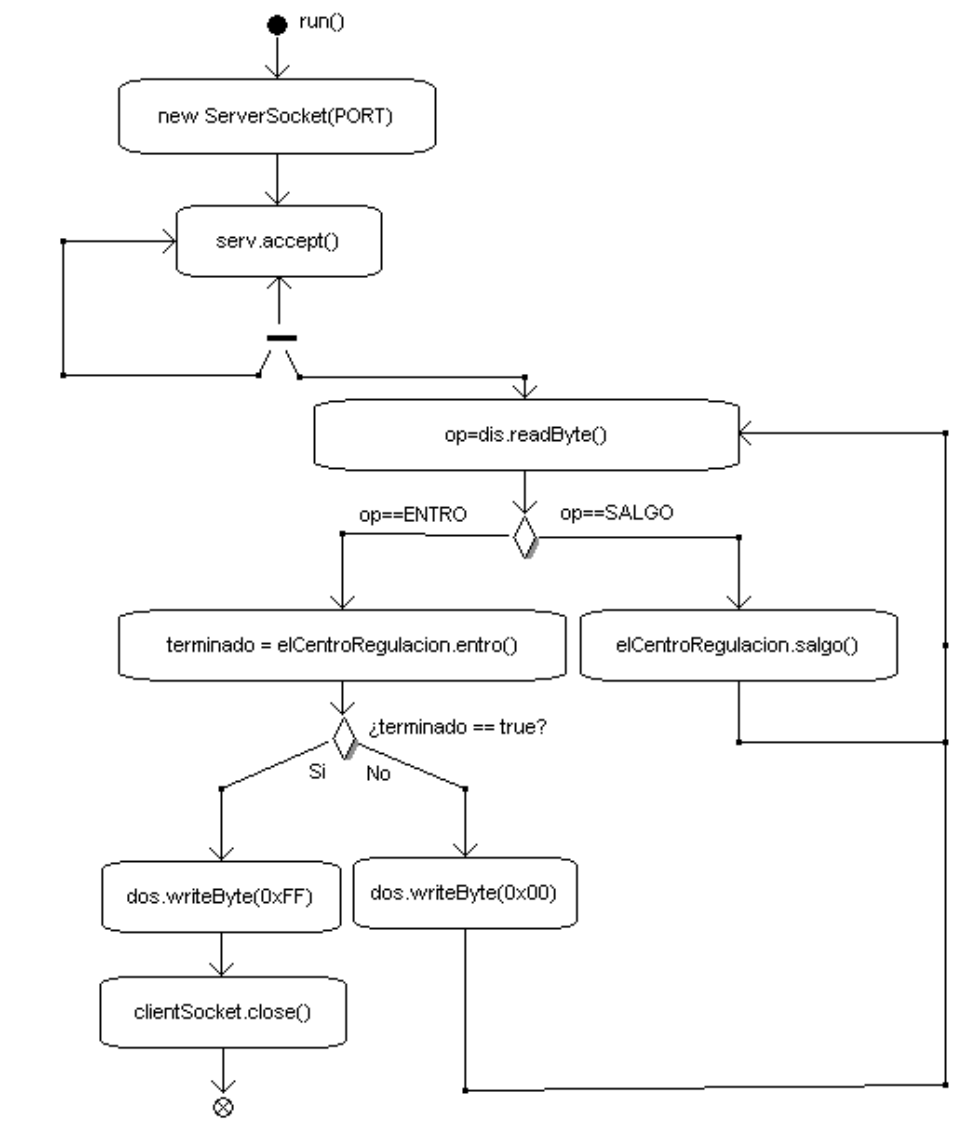
# CentroRegulacionServant Implementación I



# CentroRegulacionProxy Implementación II



# CentroRegulacionServant Implementación II



# TunelProxy Código I

```
// Constituye el representatnte local en el procesador del tren del tunel.
// Ofrece los mismos metodos que el tunel (sólo los que puede requerir un
// tren. Se encarga de enviar los mensajes para que el servant
// complementario haga la invocación solicitada, y recibe la resuesta que
//se obtiene.
public class TunelProxy {

    // Puerto a través del que el proxy se comunica con el servant que se
    // encuentra asociado al tunel. Es constante [2010]
    final int portServant=2010;

    // IP del procesador en el que se ejecutan los elementos de la infraestructura
    // en la que circula el tren.
    String ipInfraestructura;

    // Canales de entrada y salida del socket
    private DataInputStream dis = null;
    private DataOutputStream dos = null;

    // Constructor: Recibe como parámetros el IP del procesador de la infraestructura
    public TunelProxy(String ipInfraestructura){
        this.ipInfraestructura=ipInfraestructura;
        try {
            Socket cliente = new Socket(ipInfraestructura, portServant);
            dis = new DataInputStream(cliente.getInputStream());
            dos = new DataOutputStream(cliente.getOutputStream());
        } catch (UnknownHostException e) {}
        catch (IOException e) {}
    }
```

# TunelProxy Código II

```
    // Invocación remota de la operación salgo del tunel.
    public synchronized void salgo(){
        try {
            dos.writeByte(1);
            dos.writeByte(0); // no se usa
        } catch (IOException e) {}
    }

    // Invocación remota de la operación salgo del tunel. El thread del tren queda
    // suspendido en el read() del socket hasta que se recibe la respuesta.
    public synchronized void entro(Tren.Tramos tramo){
        try {
            dos.writeByte(0);
            if (tramo == Tren.Tramos.TRAMO_AZUL)
                dos.writeByte(1);
            else
                dos.writeByte(0);
            dis.readByte();
        } catch (IOException e) {}
    }
}
```

# Otra implementación.

- En esta versión del proyecto Red Ferroviaria la interacción entre los trenes y la infraestructura se realiza utilizando un solo Proxy y un Servant que engloba todas las operaciones que se necesitan para la comunicación de los trenes con la Red Ferroviaria (teniendo en cuenta que el Tren puede, a través de la clase Red Ferroviara, obtener el acceso al resto de elementos de la misma).
- En este caso, las comunicaciones se hacen a través de un solo socket.