Funcionalidades y Manejo de Eventos en Sistemas como Traccar

*

***Resumen*—La API `events` en sistemas de gestión de eventos, como el utilizado por Traccar, proporciona una solución integral para el manejo y procesamiento de eventos generados a partir de datos de dispositivos. Esta API se encarga de recibir datos de posición y otros eventos, procesarlos para detectar condiciones específicas como exceso de velocidad o mantenimiento, y manejar distintos tipos de eventos a través de manejadores especializados. Además, la API se integra con componentes del sistema como almacenamiento de datos, caché, y gestión de conexiones para una gestión eficiente de la información. Finalmente, notifica los eventos detectados a otros componentes del sistema para facilitar acciones como alertas y actualizaciones en paneles de control. Este documento detalla las funcionalidades clave de la API `events` y su rol en la optimización del monitoreo y gestión de dispositivos.**

I. Introducción

En el desarrollo de software, especialmente en aplicaciones basadas en arquitecturas como MVC (Modelo-Vista-Controlador), los modelos juegan un papel fundamental. Estos componentes son responsables de representar y gestionar los datos, así como la lógica de negocio de la aplicación. Un ejemplo claro de esto es la clase `Event`, utilizada en sistemas de rastreo y monitoreo para representar eventos como movimientos de dispositivos, detenciones, exceso de velocidad, o entradas y salidas de geovallas. Estos eventos son registrados en una base de datos y pueden ser procesados para generar alertas, informes o ser visualizados en la interfaz de usuario [1].

La clase RemoteAddressHandler es una implementación dentro de la infraestructura de manejo de canales en Netty, diseñada para enriquecer los mensajes procesados con información adicional sobre la dirección IP del cliente. En aplicaciones de red y sistemas distribuidos, es esencial no solo transmitir datos, sino también capturar información contextual que permita un mejor análisis y gestión de los mismos. Esta clase juega un papel crucial al añadir la dirección IP del origen al objeto Position, una tarea importante para rastrear la fuente de los datos y realizar auditorías o diagnósticos más precisos [1].

Implementada como un ChannelInboundHandlerAdapter, RemoteAddressHandler se activa solo si está habilitado a través de la configuración del sistema. Su funcionamiento se basa en interceptar mensajes de tipo Position y agregar la dirección IP del cliente al atributo correspondiente del mensaje. Este enfoque no solo mejora la calidad y el contexto de los datos recibidos, sino que también facilita la integración con otras partes del sistema que puedan necesitar esta información adicional para realizar análisis más profundos o tomar decisiones informadas [2].

Identify applicable funding agency here. If none, delete this.

II. Materiales y Métodos

Para el desarrollo y experimentación se utilizaron los siguientes materiales:

- **Sistema Operativo: Linux Debian.**
- **Contenedores: Docker y Docker Compose.**
- **Entorno de Desarrollo: Visual Studio Code.**

Métodos

Los métodos empleados en esta investigación incluyen:

- **Documentación API: Se revisaron y utilizaron las documentaciones de las API proporcionadas por las herramientas y servicios empleados.**
- **Experimentación: Se realizaron experimentos controlados para evaluar el comportamiento y rendimiento de los componentes desarrollados.**
- **Revisión bibliográfica: Se llevó a cabo una exhaustiva revisión de la literatura relevante para comprender las mejores prácticas y enfoques actuales en la implementación de sistemas de rastreo y monitoreo.**

III. Desarrollo

La API events en un sistema de gestión de eventos, como el que se describe en el contexto de Traccar, se encarga de manejar y procesar distintos tipos de eventos generados a partir de datos recibidos, como la posición de un dispositivo. En sistemas como Traccar, estos eventos pueden incluir cosas como mantenimiento, exceso de velocidad, actividad de medios (imágenes, videos, etc.) y movimiento.

Funcionalidades Clave de la API events

Generación y Procesamiento de Eventos

- **La API recibe datos de posición y otros eventos generados por los dispositivos.**
- **Analiza estos datos para detectar condiciones específicas que disparan eventos, como superar un límite de velocidad, entrar o salir de una geocerca, o realizar un mantenimiento programado.**

Manejo de Diferentes Tipos de Eventos

- **La API está diseñada para manejar diversos tipos de eventos, como eventos de mantenimiento (`MaintenanceEventHandler`), eventos de exceso de velocidad (`OverspeedEventHandler`), y eventos de medios (`MediaEventHandler`).**
- **Cada tipo de evento tiene un manejador específico que implementa la lógica de negocio para detectar y procesar el evento en función de los datos recibidos.**

Integración con Otros Componentes

- **La API se integra con otros componentes del sistema, como el almacenamiento de datos (`Storage`), el caché (`CacheManager`), y la gestión de conexiones (`ConnectionManager`), para almacenar y gestionar la información de eventos de manera efectiva.**

Notificación y Registro de Eventos

- **Una vez que se detecta un evento, la API lo notifica a través de un callback, lo que permite que otros componentes del sistema, como los sistemas de alerta o los paneles de control, tomen acción en función del evento.**

Modelos Events

En el contexto de desarrollo de software, especialmente en aplicaciones basadas en arquitecturas como MVC (Modelo-Vista-Controlador), los modelos son componentes clave que representan y manejan los datos y la lógica de negocio de la aplicación. A continuación, se presenta un desglose más detallado sobre el propósito y la utilidad de los modelos.

Funcionalidad: **La clase `Event` se utiliza para representar eventos que ocurren en el sistema de rastreo. Un evento podría ser, por ejemplo, un dispositivo que se mueve, se detiene, excede la velocidad, o entra/sale de una geovalla. Cada evento se guarda en la base de datos y puede ser procesado por otras partes del sistema para generar alertas, informes, o para ser visualizado en la interfaz de usuario.**

Este tipo de implementación es esencial en sistemas de rastreo y monitoreo donde es necesario registrar y responder a eventos en tiempo real o casi real.

```
package org.traccar.model;

import org.traccar.storage.StorageName;

import java.util.Date;

@StorageName("tc_events")
public class Event extends Message {

   public Event(String type, Position position) {
      setType(type);
      setPositionId(position.getId());
      setDeviceId(position.getDeviceId());
      eventTime = position.getDeviceTime();
   }

   public Event(String type, long deviceId) {
      setType(type);
      setDeviceId(deviceId);
      eventTime = new Date();
   }

   public Event() {
   }

   public static final String ALL_EVENTS = "
       allEvents";

   public static final String TYPE_COMMAND_RESULT =
       "commandResult";

   public static final String TYPE_DEVICE_ONLINE = "
       deviceOnline";
   public static final String TYPE_DEVICE_UNKNOWN =
       "deviceUnknown";
   public static final String TYPE_DEVICE_OFFLINE =
       "deviceOffline";
   public static final String TYPE_DEVICE_INACTIVE =
        "deviceInactive";
   public static final String
       TYPE_QUEUED_COMMAND_SENT = "queuedCommandSent
       ";

   public static final String TYPE_DEVICE_MOVING = "
       deviceMoving";
   public static final String TYPE_DEVICE_STOPPED =
       "deviceStopped";

   public static final String TYPE_DEVICE_OVERSPEED
       = "deviceOverspeed";
   public static final String TYPE_DEVICE_FUEL_DROP
       = "deviceFuelDrop";
   public static final String
       TYPE_DEVICE_FUEL_INCREASE = "
       deviceFuelIncrease";

   public static final String TYPE_GEOFENCE_ENTER =
       "geofenceEnter";
   public static final String TYPE_GEOFENCE_EXIT = "
       geofenceExit";

   public static final String TYPE_ALARM = "alarm";

   public static final String TYPE_IGNITION_ON = "
       ignitionOn";
   public static final String TYPE_IGNITION_OFF = "
       ignitionOff";

   public static final String TYPE_MAINTENANCE = "
       maintenance";
   public static final String TYPE_TEXT_MESSAGE = "
       textMessage";
   public static final String TYPE_DRIVER_CHANGED =
       "driverChanged";
   public static final String TYPE_MEDIA = "media";

   private Date eventTime;

   public Date getEventTime() {
      return eventTime;
   }

   public void setEventTime(Date eventTime) {
      this.eventTime = eventTime;
   }

   private long positionId;

   public long getPositionId() {
      return positionId;
   }

   public void setPositionId(long positionId) {
      this.positionId = positionId;
   }

   private long geofenceId = 0;

   public long getGeofenceId() {
      return geofenceId;
   }
```

```
    public void setGeofenceId(long geofenceId) {
        this.geofenceId = geofenceId;
    }

    private long maintenanceId = 0;

    public long getMaintenanceId() {
        return maintenanceId;
    }

    public void setMaintenanceId(long maintenanceId)
         {
        this.maintenanceId = maintenanceId;
    }

}
```

Handler:

El término *handler* se refiere a un componente o función que se encarga de manejar ciertos eventos o solicitudes en un programa. La función de un *handler* puede variar dependiendo del contexto, pero generalmente actúa como un intermediario que procesa un evento o una acción y ejecuta la lógica correspondiente.

Config:

El código que compartiste es una clase llamada `Config` en la plataforma *Traccar*, y su propósito principal es manejar la configuración de la aplicación. Esta clase proporciona una interfaz para cargar, acceder y manipular configuraciones de la aplicación, ya sea desde un archivo de configuración o desde variables de entorno.

Desglose de la Clase Config

Propiedades Internas:

- **`properties`: Un objeto `Properties` que almacena todas las configuraciones cargadas desde un archivo XML.**
- **`useEnvironmentVariables`: Un indicador booleano que determina si se deben usar variables de entorno para sobrescribir las configuraciones.**

Constructores:

- **Constructor sin parámetros: No realiza ninguna acción.**
- **Constructor con parámetro `file`: Carga las configuraciones desde un archivo XML especificado. Si se establece el uso de variables de entorno en la configuración, sobrescribe los valores con ellas.**

Métodos Principales:

Carga y Acceso a Configuraciones:

- **`getString`, `getBoolean`, `getInteger`, `getLong`, `getDouble`: Métodos para obtener valores de configuración de diferentes tipos (cadena, booleano, entero, largo, doble).**
- **`hasKey`: Verifica si una clave de configuración está presente.**
- **`setString`: Permite establecer una configuración durante las pruebas (anotado con `@VisibleForTesting`).**

Manejo de Variables de Entorno:

- **Si `useEnvironmentVariables` está activado, las configuraciones pueden ser sobrescritas por variables de entorno con nombres derivados de las claves de configuración.**
- **`getEnvironmentVariableName`: Convierte una clave de configuración en el nombre correspondiente de la variable de entorno (reemplaza puntos con guiones bajos y convierte a mayúsculas).**

Manejo de Excepciones:

- **Maneja problemas comunes como formatos inválidos en el archivo de configuración XML y errores generales de entrada/salida.**

¿Para Qué Sirve?

La clase `Config` es esencial en cualquier aplicación que necesite gestionar configuraciones externas, lo cual es casi siempre el caso en aplicaciones de servidor o de gran escala como *Traccar*. Algunas de las utilidades específicas de esta clase son:

- **Cargar Configuraciones: Permite cargar configuraciones desde un archivo XML, que puede ser fácilmente modificado sin necesidad de recompilar la aplicación.**
- **Soporte para Variables de Entorno: Facilita el despliegue en entornos donde es común sobrescribir configuraciones con variables de entorno, como en contenedores Docker o plataformas de nube.**
- **Acceso Consistente a Configuraciones: Proporciona un acceso centralizado y tipificado a las configuraciones, reduciendo la posibilidad de errores en el manejo de estas.**
- **Facilita Pruebas: La capacidad de modificar configuraciones a través del método `setString` durante las pruebas hace que sea más fácil testear diferentes escenarios de configuración.**

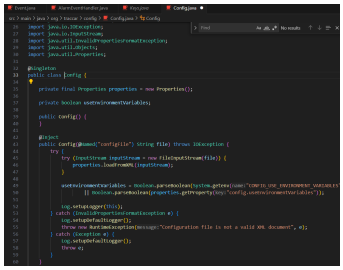

Fig. 1. Clase config

Events:

El *AlarmEventHandler* es una clase encargada de detectar y manejar eventos de alarma generados por dispositivos GPS en el sistema *Traccar*. Su función principal es analizar las posiciones reportadas, identificar alarmas, evitar duplicados si está configurado para hacerlo, y generar eventos que luego pueden ser manejados por otros componentes del sistema.

Este tipo de manejador es esencial en sistemas de rastreo para alertar a los usuarios sobre condiciones críticas o inusuales, como un exceso de velocidad o la entrada a una zona geográfica prohibida (*geofence*).

```
package org.traccar.handler.events;


import jakarta.inject.Inject;
import org.traccar.config.Config;
import org.traccar.config.Keys;
import org.traccar.model.Event;
import org.traccar.model.Position;
import org.traccar.session.cache.CacheManager;


import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;


public class AlarmEventHandler extends
    BaseEventHandler {


   private final CacheManager cacheManager;
   private final boolean ignoreDuplicates;
```

```
    @Inject
    public AlarmEventHandler(Config config,
        CacheManager cacheManager) {
       this.cacheManager = cacheManager;
       ignoreDuplicates = config.getBoolean(Keys.
           EVENT_IGNORE_DUPLICATE_ALERTS);
    }


    @Override
    public void analyzePosition(Position position,
        Callback callback) {
       String alarmString = position.getString(
           Position.KEY_ALARM);
       if (alarmString != null) {
          Set<String> alarms = new HashSet<>(Arrays.
              asList(alarmString.split(",")));
          if (ignoreDuplicates) {
             Position lastPosition = cacheManager.
                 getPosition(position.getDeviceId());
             if (lastPosition != null) {
                String lastAlarmString = lastPosition
                    .getString(Position.KEY_ALARM);
                if (lastAlarmString != null) {
                   Set<String> lastAlarms = new
                       HashSet<>(Arrays.asList(
                       lastAlarmString.split(",")));
                   alarms.removeAll(lastAlarms);
                }
             }
          }
          for (String alarm : alarms) {
             Event event = new Event(Event.TYPE_ALARM
                 , position);
             event.set(Position.KEY_ALARM, alarm);
             callback.eventDetected(event);
          }
       }
    }


}
```

BASEEVENTHANDLER

Este patrón de diseño es común en sistemas que necesitan manejar diferentes tipos de eventos basados en datos que se reciben (en este caso, posiciones de dispositivos). La clase BaseEventHandler establece un marco general para la detección de eventos, y permite que diferentes tipos de eventos sean manejados de manera específica por las clases que la extienden.

```
package org.traccar.handler.events;

import org.traccar.model.Event;
import org.traccar.model.Position;

public abstract class BaseEventHandler {

    public interface Callback {
       void eventDetected(Event event);
    }

    /**
     * Event handlers should be processed
         synchronously.
     */
    public abstract void analyzePosition(Position
        position, Callback callback);
}
```

BEHAVIOREVENTHANDLER

BehaviorEventHandler es una clase que se encarga de detectar eventos de comportamiento de conducción basado en las posiciones reportadas por un dispositivo, como un GPS instalado en un vehículo.

Uso Práctico

- **Detección de Comportamientos Peligrosos: Esta clase puede ser utilizada en sistemas de monitoreo de vehículos para detectar comportamientos peligrosos, como aceleraciones y frenadas bruscas, que podrían indicar conducción imprudente.**
- **Alertas y Reportes: Los eventos detectados pueden ser utilizados para generar alertas en tiempo real o para incluirse en reportes de comportamiento del conductor.**
- **Mejora de la Seguridad: Al identificar estos comportamientos, se pueden tomar medidas para mejorar la seguridad, ya sea notificando al conductor o analizando patrones para entrenamientos de seguridad.**

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
import org.traccar.config.Config;
import org.traccar.config.Keys;
import org.traccar.helper.UnitsConverter;
import org.traccar.model.Event;
import org.traccar.model.Position;
import org.traccar.session.cache.CacheManager;

public class BehaviorEventHandler extends
    BaseEventHandler {

   private final double accelerationThreshold;
   private final double brakingThreshold;

   private final CacheManager cacheManager;

   @Inject
   public BehaviorEventHandler(Config config,
       CacheManager cacheManager) {
      accelerationThreshold = config.getDouble(Keys.
          EVENT_BEHAVIOR_ACCELERATION_THRESHOLD);
      brakingThreshold = config.getDouble(Keys.
          EVENT_BEHAVIOR_BRAKING_THRESHOLD);
      this.cacheManager = cacheManager;
   }

   @Override
   public void analyzePosition(Position position,
       Callback callback) {

      Position lastPosition = cacheManager.
          getPosition(position.getDeviceId());
      if (lastPosition != null && position.
          getFixTime().equals(lastPosition.
          getFixTime())) {
         double acceleration = UnitsConverter.
             mpsFromKnots(position.getSpeed() -
             lastPosition.getSpeed()) * 1000
               / (position.getFixTime().getTime() -
                   lastPosition.getFixTime().getTime
                   ());
         if (accelerationThreshold != 0 &&
             acceleration >= accelerationThreshold)
             {
            Event event = new Event(Event.TYPE_ALARM
                , position);
            event.set(Position.KEY_ALARM, Position.
                ALARM_ACCELERATION);
            callback.eventDetected(event);
```

```
        } else if (brakingThreshold != 0 &&
            acceleration <= -brakingThreshold) {
           Event event = new Event(Event.TYPE_ALARM
               , position);
           event.set(Position.KEY_ALARM, Position.
               ALARM_BRAKING);
           callback.eventDetected(event);
        }
     }
  }

}
```

COMMANDRESULTEVENTHANDLER

CONSTRUCTOR

La clase tiene un constructor vacío anotado con `@Inject`, lo que sugiere que esta clase es parte de un sistema que utiliza inyección de dependencias para su creación y gestión. `@Inject` permite que el framework de inyección de dependencias (como Guice) cree instancias de esta clase automáticamente.

MÉTODO `ANALYZEPOSITION`

Este método es una implementación del método abstracto definido en `BaseEventHandler`. Se utiliza para analizar una posición (`Position position`) y determinar si contiene un resultado de comando (`commandResult`).

Pasos del Análisis

- **Obtención del Resultado del Comando: El método intenta obtener el atributo `KEY_RESULT` del objeto `Position`. Este atributo representa el resultado de un comando que fue ejecutado en el dispositivo.**
- **Verificación del Resultado: Si se encuentra un resultado (`commandResult != null`), se crea un nuevo evento (`Event`) de tipo `TYPE_COMMAND_RESULT`.**
- **Configuración del Evento: El evento recién creado tiene asociado el resultado del comando.**
- **Notificación del Evento: Finalmente, se llama al Callback para notificar que un evento ha sido detectado.**

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
import org.traccar.model.Event;
import org.traccar.model.Position;

public class CommandResultEventHandler extends
    BaseEventHandler {

   @Inject
   public CommandResultEventHandler() {
   }

   @Override
   public void analyzePosition(Position position,
       Callback callback) {
      Object commandResult = position.getAttributes
          ().get(Position.KEY_RESULT);
      if (commandResult != null) {
         Event event = new Event(Event.
             TYPE_COMMAND_RESULT, position);
         event.set(Position.KEY_RESULT, (String)
             commandResult);
         callback.eventDetected(event);
      }
   }

}
```

DRIVEREVENTHANDLER

Atributo `cacheManager`

`cacheManager` es una instancia de la clase `CacheManager`, que se inyecta en el constructor de `DriverEventHandler`. Este objeto se utiliza para acceder a la información almacenada en caché, como la última posición conocida de un dispositivo.

Constructor

El constructor de la clase toma un parámetro `cacheManager`, que se inyecta utilizando la anotación `@Inject`. Esto indica que `DriverEventHandler` depende de `CacheManager` para realizar su función, y permite que un framework de inyección de dependencias gestione la creación de esta clase.

Método `analyzePosition`

Este método implementa el método abstracto definido en `BaseEventHandler`. Su propósito es analizar una posición (`Position position`) para detectar si el conductor del dispositivo ha cambiado.

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
import org.traccar.helper.model.PositionUtil;
import org.traccar.model.Event;
import org.traccar.model.Position;
import org.traccar.session.cache.CacheManager;

public class DriverEventHandler extends
    BaseEventHandler {

   private final CacheManager cacheManager;

   @Inject
   public DriverEventHandler(CacheManager
       cacheManager) {
      this.cacheManager = cacheManager;
   }

   @Override
   public void analyzePosition(Position position,
       Callback callback) {
      if (!PositionUtil.isLatest(cacheManager,
          position)) {
         return;
      }
      String driverUniqueId = position.getString(
          Position.KEY_DRIVER_UNIQUE_ID);
      if (driverUniqueId != null) {
         String oldDriverUniqueId = null;
         Position lastPosition = cacheManager.
             getPosition(position.getDeviceId());
         if (lastPosition != null) {
            oldDriverUniqueId = lastPosition.
                getString(Position.
                KEY_DRIVER_UNIQUE_ID);
         }
         if (!driverUniqueId.equals(
             oldDriverUniqueId)) {
            Event event = new Event(Event.
                TYPE_DRIVER_CHANGED, position);
            event.set(Position.KEY_DRIVER_UNIQUE_ID,
                driverUniqueId);
            callback.eventDetected(event);
         }
      }
   }

}
```

FuelEventHandler

`FuelEventHandler` se utiliza en sistemas de monitoreo de vehículos para detectar automáticamente cambios significativos en el nivel de combustible. Esto es especialmente útil para:

- **Monitoreo de Consumo de Combustible: Permite a los administradores de flotas o propietarios de vehículos monitorear el consumo de combustible en tiempo real y detectar patrones de uso inusuales.**
- **Prevención de Robos de Combustible: Un descenso repentino en el nivel de combustible puede indicar un posible robo de combustible, lo que genera un evento que puede desencadenar una alerta.**
- **Registro de Reabastecimiento: Un aumento en el nivel de combustible puede indicar un reabastecimiento, lo que también se registra como un evento.**

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
import org.traccar.config.Keys;
import org.traccar.helper.model.AttributeUtil;
import org.traccar.helper.model.PositionUtil;
import org.traccar.model.Device;
import org.traccar.model.Event;
import org.traccar.model.Position;
import org.traccar.session.cache.CacheManager;

public class FuelEventHandler extends
    BaseEventHandler {

   private final CacheManager cacheManager;

   @Inject
   public FuelEventHandler(CacheManager cacheManager
       ) {
      this.cacheManager = cacheManager;
   }

   @Override
   public void analyzePosition(Position position,
       Callback callback) {

      Device device = cacheManager.getObject(Device.
          class, position.getDeviceId());
      if (device == null) {
         return;
      }
      if (!PositionUtil.isLatest(cacheManager,
          position)) {
         return;
      }

      if (position.hasAttribute(Position.
          KEY_FUEL_LEVEL)) {
         Position lastPosition = cacheManager.
             getPosition(position.getDeviceId());
         if (lastPosition != null && lastPosition.
             hasAttribute(Position.KEY_FUEL_LEVEL))
             {
            double before = lastPosition.getDouble(
                Position.KEY_FUEL_LEVEL);
            double after = position.getDouble(
                Position.KEY_FUEL_LEVEL);
            double change = after - before;

            if (change > 0) {
               double threshold = AttributeUtil.
                   lookup(
                     cacheManager, Keys.
                         EVENT_FUEL_INCREASE_THRESHOLD
                         , position.getDeviceId());
               if (threshold > 0 && change >=
                   threshold) {
                  callback.eventDetected(new Event(
                      Event.
                      TYPE_DEVICE_FUEL_INCREASE,
                      position));
               }
            } else if (change < 0) {
               double threshold = AttributeUtil.
                   lookup(
                     cacheManager, Keys.
                         EVENT_FUEL_DROP_THRESHOLD,
                         position.getDeviceId());
               if (threshold > 0 && Math.abs(change)
                    >= threshold) {
                  callback.eventDetected(new Event(
                      Event.TYPE_DEVICE_FUEL_DROP,
                      position));
               }
            }
         }
      }
   }

}
```

GeofenceEventHandler

Atributo `cacheManager`

Utiliza `cacheManager` para acceder a la información en caché, como la última posición del dispositivo, los datos de las geocercas y los calendarios asociados.

Constructor

El constructor toma `cacheManager` como parámetro y lo almacena en el atributo de instancia para que pueda ser utilizado en el análisis de posiciones.

Método `analyzePosition`

Verificación de Actualidad: Primero, se asegura de que la posición actual sea la más reciente mediante el uso de `PositionUtil.isLatest`. Si no es la posición más reciente, el método retorna sin hacer nada.

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
import org.traccar.helper.model.PositionUtil;
import org.traccar.model.Calendar;
import org.traccar.model.Event;
import org.traccar.model.Geofence;
import org.traccar.model.Position;
import org.traccar.session.cache.CacheManager;

import java.util.ArrayList;
import java.util.List;

public class GeofenceEventHandler extends
    BaseEventHandler {

   private final CacheManager cacheManager;

   @Inject
   public GeofenceEventHandler(CacheManager
       cacheManager) {
      this.cacheManager = cacheManager;
   }

   @Override
   public void analyzePosition(Position position,
       Callback callback) {
```

```
        if (!PositionUtil.isLatest(cacheManager,
            position)) {
            return;
        }

        List<Long> oldGeofences = new ArrayList<>();
        Position lastPosition = cacheManager.
            getPosition(position.getDeviceId());
        if (lastPosition != null && lastPosition.
            getGeofenceIds() != null) {
            oldGeofences.addAll(lastPosition.
                getGeofenceIds());
        }

        List<Long> newGeofences = new ArrayList<>();
        if (position.getGeofenceIds() != null) {
            newGeofences.addAll(position.getGeofenceIds
                ());
            newGeofences.removeAll(oldGeofences);
            oldGeofences.removeAll(position.
                getGeofenceIds());
        }

        for (long geofenceId : oldGeofences) {
            Geofence geofence = cacheManager.getObject(
                Geofence.class, geofenceId);
            if (geofence != null) {
                long calendarId = geofence.getCalendarId
                    ();
                Calendar calendar = calendarId != 0 ?
                    cacheManager.getObject(Calendar.
                    class, calendarId) : null;
                if (calendar == null || calendar.
                    checkMoment(position.getFixTime()))
                    {
                    Event event = new Event(Event.
                        TYPE_GEOFENCE_EXIT, position);
                    event.setGeofenceId(geofenceId);
                    callback.eventDetected(event);
                }
            }
        }
        for (long geofenceId : newGeofences) {
            long calendarId = cacheManager.getObject(
                Geofence.class, geofenceId).
                getCalendarId();
            Calendar calendar = calendarId != 0 ?
                cacheManager.getObject(Calendar.class,
                calendarId) : null;
            if (calendar == null || calendar.
                checkMoment(position.getFixTime())) {
                Event event = new Event(Event.
                    TYPE_GEOFENCE_ENTER, position);
                event.setGeofenceId(geofenceId);
                callback.eventDetected(event);
            }
        }
    }
}
```

IGNITIONEVENTHANDLER

Atributo `cacheManager`

Utiliza `cacheManager` para acceder a la información en caché, como la última posición del dispositivo y la información del dispositivo.

Constructor

El constructor toma `cacheManager` como parámetro y lo almacena en el atributo de instancia para que pueda ser utilizado en el análisis de posiciones.

Método `analyzePosition`

- **Verificación del Dispositivo y la Actualidad de la Posición: Primero, obtiene el dispositivo asociado a la posición actual. Si el dispositivo no existe o la posición no es la más reciente, el método retorna sin hacer nada.**
- **Detección de Cambios en el Encendido:**
  - **Verificación del Atributo de Encendido: Comprueba si la posición actual tiene el atributo de encendido (`Position.KEY_IGNITION`).**
  - **Comparación con la Última Posición: Obtiene la última posición conocida del dispositivo y verifica el estado del encendido en esa posición.**

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
import org.traccar.helper.model.PositionUtil;
import org.traccar.model.Device;
import org.traccar.model.Event;
import org.traccar.model.Position;
import org.traccar.session.cache.CacheManager;

public class IgnitionEventHandler extends
    BaseEventHandler {

    private final CacheManager cacheManager;

    @Inject
    public IgnitionEventHandler(CacheManager
        cacheManager) {
        this.cacheManager = cacheManager;
    }

    @Override
    public void analyzePosition(Position position,
        Callback callback) {
        Device device = cacheManager.getObject(Device.
            class, position.getDeviceId());
        if (device == null || !PositionUtil.isLatest(
            cacheManager, position)) {
            return;
        }

        if (position.hasAttribute(Position.
            KEY_IGNITION)) {
            boolean ignition = position.getBoolean(
                Position.KEY_IGNITION);

            Position lastPosition = cacheManager.
                getPosition(position.getDeviceId());
            if (lastPosition != null && lastPosition.
                hasAttribute(Position.KEY_IGNITION)) {
                boolean oldIgnition = lastPosition.
                    getBoolean(Position.KEY_IGNITION);

                if (ignition && !oldIgnition) {
                    callback.eventDetected(new Event(
                        Event.TYPE_IGNITION_ON, position)
                        );
                } else if (!ignition && oldIgnition) {
                    callback.eventDetected(new Event(
                        Event.TYPE_IGNITION_OFF, position
                        ));
                }
            }
        }
    }

}
```

MAINTENANCEEVENTHANDLER

Atributo `cacheManager`

Utiliza `cacheManager` para acceder a los datos de las posiciones anteriores y objetos de mantenimiento asociados con el dispositivo.

Constructor

El constructor toma `cacheManager` como parámetro y lo almacena en el atributo de instancia para su uso en el método `analyzePosition`.

Método `analyzePosition`

- **Verificación de la Última Posición:**
  - **Obtiene la última posición del dispositivo. Si la posición actual no es más reciente, el método retorna sin hacer nada.**
- **Análisis de Mantenimiento:**
  - **Obtiene la lista de objetos de mantenimiento para el dispositivo.**
  - **Para cada objeto de mantenimiento:**
    - **Verifica si el periodo de mantenimiento es distinto de cero.**
    - **Calcula los valores antiguos y nuevos para el tipo de mantenimiento especificado.**
    - **Compara estos valores para determinar si se ha alcanzado un nuevo ciclo de mantenimiento.**
    - **Si es así, crea un evento de mantenimiento (`Event.TYPE_MAINTENANCE`) y lo pasa al callback.**

Método `getValue`

- **Obtiene el Valor: Dependiendo del tipo especificado (por ejemplo, `"serverTime"`, `"deviceTime"`, `"fixTime"`), devuelve el valor correspondiente de la posición.**

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
import org.traccar.model.Event;
import org.traccar.model.Maintenance;
import org.traccar.model.Position;
import org.traccar.session.cache.CacheManager;

public class MaintenanceEventHandler extends
    BaseEventHandler {

   private final CacheManager cacheManager;

   @Inject
   public MaintenanceEventHandler(CacheManager
       cacheManager) {
      this.cacheManager = cacheManager;
   }

   @Override
   public void analyzePosition(Position position,
       Callback callback) {
      Position lastPosition = cacheManager.
          getPosition(position.getDeviceId());
      if (lastPosition == null || position.
          getFixTime().compareTo(lastPosition.
          getFixTime()) < 0) {
         return;
      }

      for (Maintenance maintenance : cacheManager.
          getDeviceObjects(position.getDeviceId(),
          Maintenance.class)) {
         if (maintenance.getPeriod() != 0) {
            double oldValue = getValue(lastPosition,
                maintenance.getType());
            double newValue = getValue(position,
                maintenance.getType());
            if (oldValue != 0.0 && newValue != 0.0
                && newValue >= maintenance.getStart
                ()) {
               if (oldValue < maintenance.getStart()
                   || (long) ((oldValue - maintenance
                       .getStart()) / maintenance.
                       getPeriod())
                   < (long) ((newValue - maintenance.
                       getStart()) / maintenance.
                       getPeriod())) {
                  Event event = new Event(Event.
                      TYPE_MAINTENANCE, position);
                  event.setMaintenanceId(maintenance
                      .getId());
                  event.set(maintenance.getType(),
                      newValue);
                  callback.eventDetected(event);
               }
            }
         }
      }
   }

   private double getValue(Position position, String
       type) {
      return switch (type) {
         case "serverTime" -> position.getServerTime
             ().getTime();
         case "deviceTime" -> position.getDeviceTime
             ().getTime();
         case "fixTime" -> position.getFixTime().
             getTime();
         default -> position.getDouble(type);
      };
   }

}
```

MEDIAEVENTHANDLER

Constructor

El constructor está vacío y simplemente se inyecta como un bean, lo que indica que la clase probablemente está diseñada para ser utilizada en un entorno de inyección de dependencias.

Método `analyzePosition`

- **Filtrado y Mapeo de Atributos Multimedia:**
  - **Usa un `Stream` para procesar los tipos de atributos multimedia que la posición puede tener (`Position.KEY_IMAGE`, `Position.KEY_VIDEO`, `Position.KEY_AUDIO`).**
  - **Filtra los atributos que están presentes en la posición actual (`position::hasAttribute`).**
  - **Para cada tipo de atributo multimedia presente:**
    - **Crea un nuevo `Event` de tipo `Event.TYPE_MEDIA`.**
    - **Establece el tipo de medio (`media`) y el archivo asociado (`file`) en el evento.**
    - **Finalmente, para cada evento creado, llama al `callback.eventDetected` para procesar el evento.**

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
```

```
import org.traccar.model.Event;
import org.traccar.model.Position;

import java.util.stream.Stream;

public class MediaEventHandler extends
    BaseEventHandler {

   @Inject
   public MediaEventHandler() {
   }

   @Override
   public void analyzePosition(Position position,
       Callback callback) {
      Stream.of(Position.KEY_IMAGE, Position.
          KEY_VIDEO, Position.KEY_AUDIO)
            .filter(position::hasAttribute)
            .map(type -> {
               Event event = new Event(Event.
                   TYPE_MEDIA, position);
               event.set("media", type);
               event.set("file", position.getString(
                   type));
               return event;
            })
            .forEach(callback::eventDetected);
   }

}
```

MOTIONEVENTHANDLER

Logger

`private static final Logger LOGGER = LoggerFactory.getLogger(MotionEventHandler.class);`
Se utiliza para registrar mensajes de advertencia en caso de errores durante el proceso de actualización del dispositivo.

Dependencias Inyectadas

- **CacheManager cacheManager: Para acceder a la caché de objetos.**
- **Storage storage: Para interactuar con el almacenamiento de datos, específicamente para actualizar la información del dispositivo.**

Constructor

`@Inject public MotionEventHandler(CacheManager cacheManager, Storage storage)`: Inyección de dependencias para `cacheManager` y `storage`.

Método `analyzePosition`

- **Obtener Información del Dispositivo:**
  - **Recupera el `Device` asociado con la `position` desde el `cacheManager`.**
  - **Verifica si el dispositivo existe y si la posición es la más reciente.**
- **Procesamiento de Posiciones Inválidas:**
  - **Usa la configuración para determinar si se deben procesar posiciones inválidas.**
  - **Si `processInvalid` es falso y la posición es inválida, se retorna sin procesar la posición.**

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.traccar.config.Keys;
import org.traccar.helper.model.AttributeUtil;
import org.traccar.helper.model.PositionUtil;
import org.traccar.model.Device;
import org.traccar.model.Position;
import org.traccar.reports.common.TripsConfig;
import org.traccar.session.cache.CacheManager;
import org.traccar.session.state.MotionProcessor;
import org.traccar.session.state.MotionState;
import org.traccar.storage.Storage;
import org.traccar.storage.StorageException;
import org.traccar.storage.query.Columns;
import org.traccar.storage.query.Condition;
import org.traccar.storage.query.Request;

public class MotionEventHandler extends
    BaseEventHandler {

   private static final Logger LOGGER =
       LoggerFactory.getLogger(MotionEventHandler.
       class);

   private final CacheManager cacheManager;
   private final Storage storage;

   @Inject
   public MotionEventHandler(CacheManager
       cacheManager, Storage storage) {
      this.cacheManager = cacheManager;
      this.storage = storage;
   }

   @Override
   public void analyzePosition(Position position,
       Callback callback) {

      long deviceId = position.getDeviceId();
      Device device = cacheManager.getObject(Device.
          class, deviceId);
      if (device == null || !PositionUtil.isLatest(
          cacheManager, position)) {
         return;
      }
      boolean processInvalid = AttributeUtil.lookup(
            cacheManager, Keys.
                EVENT_MOTION_PROCESS_INVALID_POSITIONS
                , deviceId);
      if (!processInvalid && !position.getValid()) {
         return;
      }

      TripsConfig tripsConfig = new TripsConfig(new
          AttributeUtil.CacheProvider(cacheManager,
          deviceId));
      MotionState state = MotionState.fromDevice(
          device);
      MotionProcessor.updateState(state, position,
          position.getBoolean(Position.KEY_MOTION),
          tripsConfig);
      if (state.isChanged()) {
         state.toDevice(device);
         try {
            storage.updateObject(device, new Request
                (
                  new Columns.Include("motionStreak"
                      , "motionState", "motionTime",
                       "motionDistance"),
                  new Condition.Equals("id", device.
                      getId())));
         } catch (StorageException e) {
            LOGGER.warn("Update device motion error"
                , e);
         }
      }
      if (state.getEvent() != null) {
```

```
            callback.eventDetected(state.getEvent());
        }
    }

}
```

OVERSPEEDEVENTHANDLER

Logger

`private static final Logger LOGGER = LoggerFactory.getLogger(OverspeedEventHandler.class);` Utilizado para registrar advertencias en caso de errores durante el proceso de actualización del dispositivo.

Dependencias Inyectadas

- **Config config: Configuración general para obtener parámetros de configuración.**
- **CacheManager cacheManager: Para acceder a la caché de objetos.**
- **Storage storage: Para interactuar con el almacenamiento de datos, específicamente para actualizar la información del dispositivo.**

Variables de Configuración

- **minimalDuration: Duración mínima en milisegundos para que se considere un evento de sobrevelocidad.**
- **preferLowest: Indica si se debe preferir el límite de velocidad más bajo de las geocercas.**
- **multiplier: Multiplicador para calcular el umbral de sobrevelocidad.**

Constructor

`@Inject public OverspeedEventHandler(Config config, CacheManager cacheManager, Storage storage)`: Inyección de dependencias para `config`, `cacheManager`, y `storage`. Inicializa las variables de configuración.

Método `analyzePosition`

- **Obtener Información del Dispositivo y Posición:**
  - **Recupera el `Device` asociado con la `position` desde el `cacheManager`.**
  - **Verifica si el dispositivo existe, si la posición es la más reciente y si es válida.**
- **Determinar Límite de Velocidad:**
  - **Obtiene el límite de velocidad configurado para el dispositivo.**
  - **Revisa los límites de velocidad definidos por las geocercas asociadas con la posición. Si alguna geocerca define un límite de velocidad, se actualiza el límite de velocidad.**

```
package org.traccar.handler.events;

import jakarta.inject.Inject;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.traccar.config.Config;
import org.traccar.config.Keys;
import org.traccar.helper.model.AttributeUtil;
import org.traccar.helper.model.PositionUtil;
import org.traccar.model.Device;
import org.traccar.model.Geofence;
import org.traccar.model.Position;
import org.traccar.session.cache.CacheManager;
import org.traccar.session.state.OverspeedProcessor;
import org.traccar.session.state.OverspeedState;
import org.traccar.storage.Storage;
import org.traccar.storage.StorageException;
import org.traccar.storage.query.Columns;
import org.traccar.storage.query.Condition;
import org.traccar.storage.query.Request;

public class OverspeedEventHandler extends
    BaseEventHandler {

   private static final Logger LOGGER =
       LoggerFactory.getLogger(OverspeedEventHandler
       .class);

   private final CacheManager cacheManager;
   private final Storage storage;

   private final long minimalDuration;
   private final boolean preferLowest;
   private final double multiplier;

   @Inject
   public OverspeedEventHandler(Config config,
       CacheManager cacheManager, Storage storage) {
      this.cacheManager = cacheManager;
      this.storage = storage;
      minimalDuration = config.getLong(Keys.
          EVENT_OVERSPEED_MINIMAL_DURATION) * 1000;
      preferLowest = config.getBoolean(Keys.
          EVENT_OVERSPEED_PREFER_LOWEST);
      multiplier = config.getDouble(Keys.
          EVENT_OVERSPEED_THRESHOLD_MULTIPLIER);
   }

   @Override
   public void analyzePosition(Position position,
       Callback callback) {

      long deviceId = position.getDeviceId();
      Device device = cacheManager.getObject(Device.
          class, position.getDeviceId());
      if (device == null) {
         return;
      }
      if (!PositionUtil.isLatest(cacheManager,
          position) || !position.getValid()) {
         return;
      }

      double speedLimit = AttributeUtil.lookup(
          cacheManager, Keys.EVENT_OVERSPEED_LIMIT,
          deviceId);

      double positionSpeedLimit = position.getDouble
          (Position.KEY_SPEED_LIMIT);
      if (positionSpeedLimit > 0) {
         speedLimit = positionSpeedLimit;
      }

      double geofenceSpeedLimit = 0;
      long overspeedGeofenceId = 0;

      if (position.getGeofenceIds() != null) {
         for (long geofenceId : position.
             getGeofenceIds()) {
            Geofence geofence = cacheManager.
                getObject(Geofence.class, geofenceId
                );
            if (geofence != null) {
               double currentSpeedLimit = geofence.
                   getDouble(Keys.
                   EVENT_OVERSPEED_LIMIT.getKey());
               if (currentSpeedLimit > 0 &&
                   geofenceSpeedLimit == 0
                     || preferLowest &&
```

```
                        currentSpeedLimit <
                        geofenceSpeedLimit
                    || !preferLowest &&
                        currentSpeedLimit >
                        geofenceSpeedLimit) {
                geofenceSpeedLimit =
                    currentSpeedLimit;
                overspeedGeofenceId = geofenceId;
            }
        }
    }
}
if (geofenceSpeedLimit > 0) {
    speedLimit = geofenceSpeedLimit;
}

if (speedLimit == 0) {
    return;
}

OverspeedState state = OverspeedState.
    fromDevice(device);
OverspeedProcessor.updateState(state, position
    , speedLimit, multiplier, minimalDuration,
     overspeedGeofenceId);
if (state.isChanged()) {
    state.toDevice(device);
    try {
        storage.updateObject(device, new Request
            (
              new Columns.Include("
                  overspeedState", "
                  overspeedTime", "
                  overspeedGeofenceId"),
              new Condition.Equals("id", device.
                  getId())));
    } catch (StorageException e) {
        LOGGER.warn("Update device overspeed
            error", e);
    }
}
if (state.getEvent() != null) {
    callback.eventDetected(state.getEvent());
}
}

}
```

A. Clase AcknowledgementHandler

La clase `AcknowledgementHandler` es responsable de:

- **Sincronización de Mensajes: Maneja la sincronización de mensajes con eventos para asegurar que los mensajes se envíen en el orden correcto y no se pierdan.**
- **Gestión de Estado de Eventos: Utiliza una cola para almacenar mensajes cuando hay eventos pendientes, garantizando que los mensajes se envíen solo cuando todos los eventos relevantes han sido manejados.**

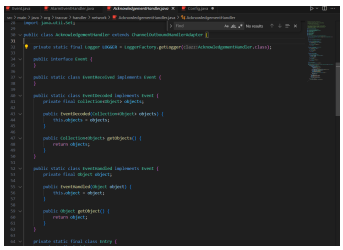

Fig. 2. Clase AcknowledgementHandler

IV. Clase MainEventHandler

`MainEventHandler` extiende `ChannelInboundHandlerAdapter`, lo que indica que es un manejador de entrada en el pipeline de Netty. Su propósito es gestionar eventos relacionados con el estado de los canales de red y la conexión de dispositivos.

A. Componentes Principales

1) Dependencias Inyectadas:

- **`ConnectionManager`: Utilizado para manejar la conexión y desconexión de dispositivos.**
- **`Config`: Configuración de la aplicación para determinar protocolos sin conexión.**

2) Atributos:

- **`connectionManager`: Maneja el estado de las conexiones de los dispositivos.**
- **`connectionlessProtocols`: Conjunto de protocolos que no requieren una conexión persistente.**

3) Constructor:

```
@Inject
public MainEventHandler(Config config,
    ConnectionManager connectionManager) {
   this.connectionManager = connectionManager;
   String connectionlessProtocolList = config.
       getString(Keys.STATUS_IGNORE_OFFLINE);
   if (connectionlessProtocolList != null) {
      connectionlessProtocols.addAll(Arrays.asList(
          connectionlessProtocolList.split("[, ]")))
          ;
   }
}
```

Inyecta `Config` y `ConnectionManager`. Inicializa `connectionlessProtocols` con los protocolos especificados en la configuración que no requieren conexión persistente.

4) Métodos Sobrescritos:

- **`channelActive(ChannelHandlerContext ctx)`**
  Registra cuando un canal se activa (se conecta). Utiliza `NetworkUtil.session(ctx.channel())` para obtener información de sesión y loguear la conexión.
- **`channelInactive(ChannelHandlerContext ctx)`**
  Registra cuando un canal se desactiva (se desconecta). Cierra el canal y llama a `connectionManager.deviceDisconnected()`. Verifica si el protocolo soporta el estado offline y actualiza el `connectionManager`.
- **`exceptionCaught(ChannelHandlerContext ctx, Throwable cause)`**
  Registra excepciones y errores que ocurren en el canal. Busca la causa raíz de la excepción y la registra. Cierra el canal en caso de error.
- **`userEventTriggered(ChannelHandlerContext ctx, Object evt)`**
  Maneja eventos de tiempo de inactividad (`IdleStateEvent`). Registra cuando un canal ha superado el tiempo de espera y cierra el canal.

5) Método Privado:

```
private void closeChannel(Channel channel) {
   if (!(channel instanceof DatagramChannel)) {
      channel.close();
   }
}
```

Cierra el canal si no es una instancia de `DatagramChannel`. Utilizado para cerrar conexiones no deseadas o cuando ocurre un error.

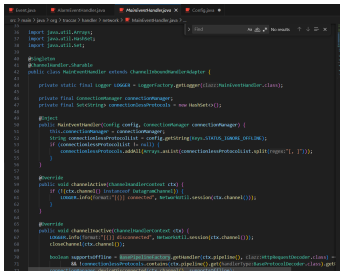

Fig. 3. Clase MainEventHandler

CLASE NETWORKFORWARDERHANDLER

Campos

- **`private final int port;`**
  Almacena el puerto al que se reenviarán los datos.
- **`private NetworkForwarder networkForwarder;`**
  Instancia de `NetworkForwarder` que se inyecta y se usa para reenviar los datos.

Constructor

```
public NetworkForwarderHandler(int port) {
   this.port = port;
}
```

Inicializa el puerto.

Método de Inyección

```
@Inject
public void setNetworkForwarder(NetworkForwarder
     networkForwarder) {
   this.networkForwarder = networkForwarder;
}
```

Permite inyectar una instancia de `NetworkForwarder` después de la creación del manejador.

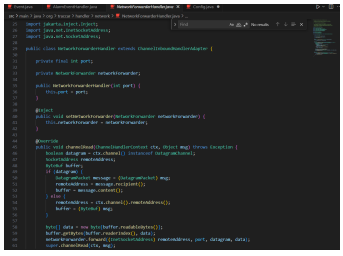

Fig. 4. Clase NetworkForwarderHandler

MÉTODOS DE NETWORKMESSAGEHANDLER

Método `channelRead(ChannelHandlerContext ctx, Object msg)`

Descripción: Se llama cuando se lee un mensaje del canal.
Acción:

- **Para DatagramChannel: Si el canal es un `DatagramChannel`, convierte el `DatagramPacket` en un `NetworkMessage` y lo pasa al siguiente manejador en el pipeline.**
- **Para otros canales: Si el mensaje es una instancia de `ByteBuf`, lo envuelve en un `NetworkMessage` junto con la dirección remota del canal y lo pasa al siguiente manejador.**

Código relevante:

```
@Override
public void channelRead(ChannelHandlerContext ctx,
     Object msg) {
   if (ctx.channel() instanceof DatagramChannel) {
      DatagramPacket packet = (DatagramPacket) msg;
      ctx.fireChannelRead(new NetworkMessage(packet.
          content(), packet.sender()));
   } else if (msg instanceof ByteBuf buffer) {
      ctx.fireChannelRead(new NetworkMessage(buffer,
           ctx.channel().remoteAddress()));
   }
}
```

Método `write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)`

Descripción: Se llama cuando se escribe un mensaje en el canal.
Acción:

- **Para NetworkMessage: Si el mensaje es una instancia de `NetworkMessage`, verifica el tipo de canal:**
  - **Para DatagramChannel: Crea un `DatagramPacket` con la dirección del destinatario y la dirección local del canal, luego lo escribe en el canal.**
  - **Para otros canales: Escribe el mensaje directamente en el canal.**
- **Para otros tipos de mensajes: Simplemente los escribe en el canal.**

Código relevante:

```
@Override
public void write(ChannelHandlerContext ctx, Object
     msg, ChannelPromise promise) {
   if (msg instanceof NetworkMessage message) {
      if (ctx.channel() instanceof DatagramChannel)
           {
         InetSocketAddress recipient = (
              InetSocketAddress) message.
              getRemoteAddress();
         InetSocketAddress sender = (
              InetSocketAddress) ctx.channel().
              localAddress();
         ctx.write(new DatagramPacket((ByteBuf)
              message.getMessage(), recipient, sender
              ), promise);
      } else {
         ctx.write(message.getMessage(), promise);
      }
   } else {
      ctx.write(msg, promise);
   }
}
```

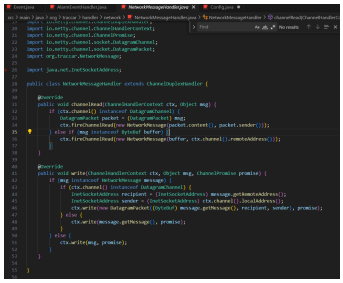

Fig. 5. NetworkMessageHandler

CLASE OPENCHANNELHANDLER

Constructor

OpenChannelHandler(TrackerConnector connector)

Descripción: Inicializa el manejador con una instancia de `TrackerConnector`.

Acción: Almacena el `TrackerConnector` en un campo final para que pueda ser utilizado en los métodos de manejo de eventos.

Código relevante:

```
public OpenChannelHandler(TrackerConnector connector
    ) {
   this.connector = connector;
}
```

Método `channelActive(ChannelHandlerContext ctx)`

Descripción: Se llama cuando un canal se activa (es decir, se ha establecido una conexión).

Acción: Agrega el canal al grupo de canales del `TrackerConnector`. Esto permite que el `TrackerConnector` mantenga un registro de todos los canales activos.

Código relevante:

```
@Override
public void channelActive(ChannelHandlerContext ctx)
     throws Exception {
   super.channelActive(ctx);
   connector.getChannelGroup().add(ctx.channel());
}
```

Método `channelInactive(ChannelHandlerContext ctx)`

Descripción: Se llama cuando un canal se desactiva (es decir, se ha cerrado la conexión).

Acción: Elimina el canal del grupo de canales del `TrackerConnector`. Esto asegura que el `TrackerConnector` no mantenga referencias a canales que ya no están activos.

Código relevante:

```
@Override
public void channelInactive(ChannelHandlerContext
    ctx) throws Exception {
   super.channelInactive(ctx);
   connector.getChannelGroup().remove(ctx.channel())
       ;
}
```

Fig. 6. Clase OpenChannelHandler

CLASE REMOTEADDRESSHANDLER

Campo `enabled`

Descripción: Determina si el procesamiento de la dirección IP está habilitado o no.

Código relevante:

```
private final boolean enabled;

@Inject
public RemoteAddressHandler(Config config) {
   enabled = config.getBoolean(Keys.
       PROCESSING_REMOTE_ADDRESS_ENABLE);
}
```

Método `channelRead(ChannelHandlerContext ctx, Object msg)`

Descripción: Maneja la lectura de mensajes del canal.

Acción:

- **Si el procesamiento está habilitado (`enabled` es `true`), obtiene la dirección IP del cliente desde la dirección remota del canal.**

Propósito: Añadir información adicional (la dirección IP del cliente) a los objetos `Position` para su posterior procesamiento o almacenamiento.

Código relevante:

```
@Override
public void channelRead(ChannelHandlerContext ctx,
    Object msg) {
   if (enabled) {
      InetSocketAddress remoteAddress = (
          InetSocketAddress) ctx.channel().
          remoteAddress();
      String hostAddress = remoteAddress != null ?
          remoteAddress.getAddress().getHostAddress
          () : null;

      if (msg instanceof Position position) {
         position.set(Position.KEY_IP, hostAddress);
      }
   }
   ctx.fireChannelRead(msg);
}
```

Funcionamiento:

- **Activación Condicional: El procesamiento de la dirección IP solo se realiza si está habilitado en la configuración (`enabled` es `true`).**
- **Adición de IP a Position: Si el mensaje es un objeto `Position`, se establece un atributo en el objeto que contiene la dirección IP del cliente. Esto permite asociar la dirección IP con el objeto `Position` para su uso posterior.**

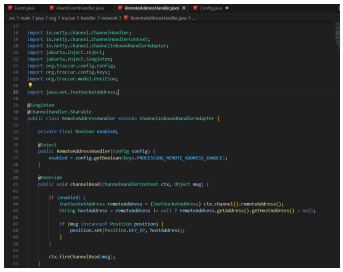

Fig. 7. Clase RemoteAddressHandler

CLASE STANDARDLOGGINGHANDLER

Dependencias Inyectadas

- **Config: Proporciona la configuración para determinar si se deben decodificar los datos de texto.**
- **ConnectionManager: Utilizado para actualizar los registros de log relacionados con las conexiones.**

Atributos

- **protocol: Nombre del protocolo asociado con el `StandardLoggingHandler`.**
- **connectionManager: Maneja actualizaciones de registros de log.**
- **decodeTextData: Indica si se deben decodificar los datos de texto.**

Constructor

Constructor:

```
public StandardLoggingHandler(String protocol) {
    this.protocol = protocol;
}
```

Descripción: Inicializa el nombre del protocolo.

Métodos Inyectados

Método `setConnectionManager(ConnectionManager connectionManager)`

Descripción: Establece el `ConnectionManager` para gestionar actualizaciones de registros de log.

Métodos Sobrescritos

Método `channelRead(ChannelHandlerContext ctx, Object msg)`

Descripción: Crea un `LogRecord` basado en el mensaje recibido (`msg`) y el contexto del canal (`ctx`).

Acción:

- **Registra la información del mensaje y actualiza el `ConnectionManager` con el registro.**
- **Llama a `super.channelRead(ctx, msg)` para pasar el mensaje al siguiente manejador en el pipeline.**

Código relevante:

```
@Override
public void channelRead(ChannelHandlerContext ctx,
     Object msg) {
   LogRecord record = createLogRecord(ctx, msg);
   log(ctx, true, record);
   super.channelRead(ctx, msg);
}
```

Método `write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise)`

Descripción: Crea un `LogRecord` basado en el mensaje que se va a enviar (`msg`) y el contexto del canal (`ctx`).

Acción:

- **Registra la información del mensaje y luego llama a `super.write(ctx, msg, promise)` para enviar el mensaje.**

Código relevante:

```
@Override
public void write(ChannelHandlerContext ctx, Object
     msg, ChannelPromise promise) {
   LogRecord record = createLogRecord(ctx, msg);
   log(ctx, false, record);
   super.write(ctx, msg, promise);
}
```

Métodos Privados

Método `createLogRecord(ChannelHandlerContext ctx, Object msg)`

Descripción: Crea un `LogRecord` a partir del mensaje.

Acción:

- **Si el mensaje es una instancia de `NetworkMessage` y contiene un `ByteBuf` con datos, se crea un `LogRecord` con la dirección local y remota.**
- **Si `decodeTextData` es verdadero y los datos son imprimibles, los datos se convierten a una cadena de texto; de lo contrario, se convierte a un formato hexadecimal.**

Código relevante:

```
private LogRecord createLogRecord(
     ChannelHandlerContext ctx, Object msg) {

}
```

Método `log(ChannelHandlerContext ctx, boolean downstream, LogRecord record)`

Descripción: Registra la información contenida en `LogRecord`.

Acción: Formatea el mensaje de registro con la dirección del canal, el protocolo y los datos del registro.

Código relevante:

```
private void log(ChannelHandlerContext ctx, boolean
     downstream, LogRecord record) {

}
```

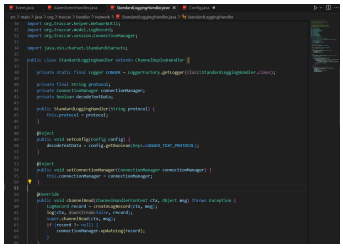

Fig. 8. Clase StandardLoggingHandler

V. Resultados

Para implementar Traccar en Linux, primero debes instalar Java Runtime Environment (JRE) y luego descargar el archivo de instalación de Traccar desde su sitio web. Después de descomprimir el archivo, configura la base de datos en el archivo traccar.xml, inicia el servicio con el script incluido y accede a la interfaz web en http://localhost:8082 para gestionar y monitorear el sistema. Asegúrate de configurar los puertos necesarios y revisa los logs para mantenimiento y depuración.

Fig. 9. Inicio Traccar

Estos eventos se pueden manejar mediante una API de eventos que permita a otros componentes del sistema:

- **Recibir los datos de eventos generados.**
- **Procesar y analizar estos datos para detectar condiciones específicas.**
- **Notificar a otros sistemas o usuarios mediante alertas o informes.**
- **Registrar los eventos en una base de datos para auditoría y análisis posterior.**

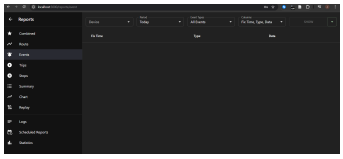

Fig. 10. Events

VI. Discusión y Conclusiones

Optimización en el Manejo de Eventos: Las clases de manejo de red, como AcknowledgementHandler y MainEventHandler, juegan un papel crucial en la optimización del manejo de eventos en el sistema. Estas clases se encargan de gestionar la cola de mensajes y las respuestas del sistema de manera eficiente, garantizando que los eventos sean procesados en el orden correcto y que se mantenga la integridad de los datos transmitidos. Al manejar la conexión y el estado de los dispositivos de manera efectiva, aseguran que los eventos relevantes se reconozcan y se registren apropiadamente, minimizando la pérdida de datos y mejorando la precisión en la gestión de eventos.

Mejora en la Gestión de Errores y Conexiones: Las clases de manejo de red también son esenciales para la gestión de errores y la administración de conexiones. Por ejemplo, MainEventHandler se encarga de detectar desconexiones y manejar excepciones, proporcionando una respuesta adecuada ante problemas de comunicación. Esto es fundamental para mantener la estabilidad del sistema, ya que asegura que las conexiones se cierren de manera ordenada y que los errores se registren y gestionen correctamente, lo que contribuye a una mayor resiliencia del sistema ante fallos y errores.

Monitoreo y Registro Detallado de la Comunicación: La clase StandardLoggingHandler añade un nivel adicional de visibilidad al sistema mediante el registro detallado de la comunicación de red. Este registro incluye tanto los datos recibidos como los enviados, lo que facilita el monitoreo y la depuración. Al proporcionar una representación clara de los mensajes de red y sus contenidos, permite a los desarrolladores y administradores del sistema identificar y resolver problemas de manera más eficiente. Este nivel de detalle en el registro es vital para el diagnóstico y para mantener un control preciso sobre la interacción entre los componentes del sistema y los eventos que se generan a lo largo del tiempo.

Referencias

[1] M. M. Sharom, M. A. Fauzi, and A. Sipit, "Live location sharing system (l2s2): Empowering management of unmanned aircraft systems (uas) operations," *ASM Sc. J.*, vol. 19, 2024, malaysian Space Agency (MYSA), Ministry of Science, Technology and Innovation (MOSTI), Malaysia. [Online]. Available: https://doi.org/10.32802/asmscj.2023.1657

[2] K. Beiglböck, *GPS Tracking with Java EE Components: Challenges of Connected Cars*, 1st ed. Chapman and Hall/CRC, 2018. [Online]. Available: https://doi.org/10.1201/9781315166322