Resumen e Estrategia de Integración de Jonoprotocol

Misión

El **Jonoprotocol** es un protocolo unificado diseñado para estandarizar datos de diversos dispositivos de seguimiento GPS en un formato único y consistente. Diferentes marcas y modelos de rastreadores utilizan sus propios "lenguajes" (protocolos propietarios), lo que hace que la integración sea costosa y propensa a errores.
Jonoprotocol resuelve esto actuando como un **traductor**, asegurando que todos los datos de los dispositivos puedan gestionarse, procesarse y escalarse de manera uniforme.

¿Por qué Jonoprotocol?

- **Interoperabilidad**: Maneja múltiples marcas/modelos sin necesidad de escribir lógica personalizada para cada servicio posterior.
- **Escalabilidad**: Una vez que los datos están estandarizados, pueden procesarse, almacenarse o analizarse sin consideraciones específicas de protocolo.
- **Mantenimiento**: Añadir un nuevo protocolo solo requiere escribir un nuevo intérprete → todo lo demás ya entiende Jonoprotocol.

Cómo Funciona

1. **El Dispositivo Envía Datos**: Cada oyente (en Jonobride) transmite mensajes utilizando su propio protocolo propietario a través de MQTT en formato de cadena hexadecimal.
2. **Módulo Intérprete**: Un intérprete específico del protocolo analiza el mensaje crudo recibido de MQTT.
3. **Conversión a Jonoprotocol**: Los datos analizados se mapean al esquema unificado de Jonoprotocol.
4. **Publicación**: El mensaje de Jonoprotocol se publica (por ejemplo, a MQTT) para su procesamiento posterior.

Esquema Central de Jonoprotocol

El esquema central está definido en `common/models` como estructuras de Go, con la estructura principal siendo `JonoModel`.
Este esquema asegura que **todos los intérpretes** produzcan una salida uniforme, estandarizando campos como la identificación del dispositivo, ubicación, eventos y telemetría extensible (por ejemplo, sensores, puertos de E/S, datos CAN). A continuación, se presenta una explicación detallada de `JonoModel` y sus estructuras asociadas.

Estructura JonoModel

La estructura `JonoModel` es la estructura raíz que encapsula todos los datos estandarizados de un dispositivo de seguimiento GPS.

```
type JonoModel struct {
    IMEI        string                `json:"IMEI"`
    Message     *string               `json:"Message"`
    DataPackets int                   `json:"DataPackets"`
    ListPackets map[string]DataPacket `json:"ListPackets"`
}
```

- **IMEI**: Una cadena que representa el identificador único del dispositivo.
- **Message**: Una cadena opcional para mensajes o notas específicas del dispositivo.
- **DataPackets**: Un entero que indica el número de paquetes de datos en el mensaje.
- **ListPackets**: Un mapa de estructuras `DataPacket`, indexado por un identificador único (por ejemplo, ID de paquete), que contiene datos de telemetría detallados.

Estructura DataPacket

La estructura `DataPacket` captura información detallada de telemetría y estado para cada paquete enviado por el dispositivo.

```
type DataPacket struct {
    Altitude                    int                         `json:"Altitude"`
    Datetime                    time.Time                   `json:"Datetime"`
    EventCode                   EventCode                   `json:"EventCode"`
    Latitude                    float64                     `json:"Latitude"`
    Longitude                   float64                     `json:"Longitude"`
    Speed                       int                         `json:"Speed"`
    RunTime                     int                         `json:"RunTime"`
    FuelPercentage              int                         `json:"FuelPercentage"`
    Direction                   int                         `json:"Direction"`
    HDOP                        float64                     `json:"HDOP"`
    Mileage                     int                         `json:"Mileage"`
    PositioningStatus           string                      `json:"PositioningStatus"`
    NumberOfSatellites          int                         `json:"NumberOfSatellites"`
    GSMSignalStrength           *int                        `json:"GSMSignalStrength"`
    AnalogInputs                *AnalogInputs               `json:"AnalogInputs"`
    IoPortStatus                *IoPortsStatus              `json:"IoPortStatus"`
    BaseStationInfo             *BaseStationInfo            `json:"BaseStationInfo"`
    OutputPortStatus            *OutputPortStatus           `json:"OutputPortStatus"`
    InputPortStatus             *InputPortStatus            `json:"InputPortStatus"`
    SystemFlag                  *SystemFlag                 `json:"SystemFlag"`
    TemperatureSensor           *TemperatureSensor          `json:"TemperatureSensor"`
    CameraStatus                *CameraStatus               `json:"CameraStatus"`
    CurrentNetworkInfo          *CurrentNetworkInfo         `json:"CurrentNetworkInfo"`
    FatigueDrivingInformation   *FatigueDrivingInformation  `json:"FatigueDrivingInformation"`
    AdditionalAlertInfoADASDMS  *AdditionalAlertInfoADASDMS `json:"AdditionalAlertInfoADASDMS"`
    BluetoothBeaconA            *BluetoothBeacon            `json:"BluetoothBeaconA"`
    BluetoothBeaconB            *BluetoothBeacon            `json:"BluetoothBeaconB"`
    TemperatureAndHumiditySensor *TemperatureAndHumidity    `json:"TemperatureAndHumiditySensor"`
}
```

- **Altitude**: Entero que representa la altitud en metros.
- **Datetime**: Marca de tiempo del paquete de datos utilizando `time.Time` de Go.
- **EventCode**: Una estructura que contiene un código y nombre de evento (ver `EventCode` a continuación).
- **Latitude** y **Longitude**: Valores de punto flotante para coordenadas geográficas.
- **Speed**: Entero que representa la velocidad del dispositivo en kilómetros por hora.
- **RunTime**: Entero que indica el tiempo operativo del dispositivo en segundos.
- **FuelPercentage**: Entero que representa el nivel de combustible como porcentaje.
- **Direction**: Entero que indica la dirección de viaje en grados (0–359).
- **HDOP**: Valor de punto flotante para la Dilución Horizontal de Precisión, que indica la precisión del GPS.
- **Mileage**: Entero que representa la distancia total recorrida en metros.
- **PositioningStatus**: Cadena que indica el estado de fijación del GPS (por ejemplo, "A" para válido, "V" para no válido).
- **NumberOfSatellites**: Entero que indica el número de satélites GPS visibles.
- **GSMSignalStrength**: Entero opcional para la intensidad de la señal GSM.
- **AnalogInputs**, **IoPortStatus**, **BaseStationInfo**, **OutputPortStatus**, **InputPortStatus**, **SystemFlag**, **TemperatureSensor**, **CameraStatus**, **CurrentNetworkInfo**, **FatigueDrivingInformation**, **AdditionalAlertInfoADASDMS**, **BluetoothBeaconA**, **BluetoothBeaconB**, **TemperatureAndHumiditySensor**: Estructuras opcionales para datos de telemetría adicionales (detallados a continuación).

Estructuras de Soporte

Las siguientes estructuras proporcionan información detallada de telemetría y estado, todas opcionales para adaptarse a las diversas capacidades de los dispositivos.

EventCode

```
type EventCode struct {
    Code int    `json:"Code"`
    Name string `json:"Name"`
}
```

- **Code**: Entero que representa el tipo de evento (por ejemplo, 1 para "encendido").
- **Name**: Cadena que describe el evento (por ejemplo, "Encendido").

BaseStationInfo

```
type BaseStationInfo struct {
    MCC    *string `json:"MCC"`
    MNC    *string `json:"MNC"`
    LAC    *string `json:"LAC"`
    CellID *string `json:"CellID"`
}
```

- **MCC**: Código de País Móvil (opcional).
- **MNC**: Código de Red Móvil (opcional).
- **LAC**: Código de Área de Ubicación (opcional).
- **CellID**: ID de la torre celular (opcional).

AnalogInputs

```
type AnalogInputs struct {
    AD1  *string `json:"AD1"`
    AD2  *string `json:"AD2"`
    AD3  *string `json:"AD3"`
    AD4  *string `json:"AD4"`
    AD5  *string `json:"AD5"`
    AD6  *string `json:"AD6"`
    AD7  *string `json:"AD7"`
    AD8  *string `json:"AD8"`
    AD9  *string `json:"AD9"`
    AD10 *string `json:"AD10"`
}
```

- **AD1–AD10**: Cadenas opcionales que representan valores de entrada analógica (por ejemplo, niveles de voltaje).

OutputPortStatus

```
type OutputPortStatus struct {
    Output1 *string `json:"Output1"`
    Output2 *string `json:"Output2"`
    Output3 *string `json:"Output3"`
    Output4 *string `json:"Output4"`
    Output5 *string `json:"Output5"`
    Output6 *string `json:"Output6"`
    Output7 *string `json:"Output7"`
    Output8 *string `json:"Output8"`
}
```

- **Output1–Output8**: Cadenas opcionales que indican el estado de los puertos de salida (por ejemplo, "ON" o "OFF").

InputPortStatus

```
type InputPortStatus struct {
    Input1 *string `json:"Input1"`
    Input2 *string `json:"Input2"`
    Input3 *string `json:"Input3"`
    Input4 *string `json:"Input4"`
    Input5 *string `json:"Input5"`
    Input6 *string `json:"Input6"`
    Input7 *string `json:"Input7"`
    Input8 *string `json:"Input8"`
}
```

- **Input1–Input8**: Cadenas opcionales que indican el estado de los puertos de entrada (por ejemplo, "ALTO" o "BAJO").

SystemFlag

```
type SystemFlag struct {
    EEP2                *string `json:"EEP2"`
    ACC                 *string `json:"ACC"`
    AntiTheft           *string `json:"AntiTheft"`
    VibrationFlag       *string `json:"VibrationFlag"`
    MovingFlag          *string `json:"MovingFlag"`
    ExternalPowerSupply *string `json:"ExternalPowerSupply"`
    Charging            *string `json:"Charging"`
    SleepMode           *string `json:"SleepMode"`
    FMS                 *string `json:"FMS"`
    FMSFunction         *string `json:"FMSFunction"`
    SystemFlagExtras    *string `json:"SystemFlagExtras"`
}
```

- **EEP2**, **ACC**, **AntiTheft**, **VibrationFlag**, **MovingFlag**, **ExternalPowerSupply**, **Charging**, **SleepMode**, **FMS**, **FMSFunction**, **SystemFlagExtras**: Cadenas opcionales que representan diversos estados del sistema (por ejemplo, "ON" para ACC indicando el estado de encendido).

TemperatureSensor

```
type TemperatureSensor struct {
    SensorNumber *string `json:"SensorNumber"`
    Value        *string `json:"Value"`
}
```

- **SensorNumber**: Cadena opcional que identifica el sensor.
- **Value**: Cadena opcional que representa la lectura de temperatura.

CameraStatus

```
type CameraStatus struct {
    CameraNumber *string `json:"CameraNumber"`
    Status       *string `json:"Status"`
}
```

- **CameraNumber**: Cadena opcional que identifica la cámara.
- **Status**: Cadena opcional que indica el estado de la cámara (por ejemplo, "ACTIVA").

CurrentNetworkInfo

```
type CurrentNetworkInfo struct {
    Version    *string `json:"Version"`
    Type       *string `json:"Type"`
    Descriptor *string `json:"Descriptor"`
}
```

- **Version**, **Type**, **Descriptor**: Cadenas opcionales que proporcionan información de red (por ejemplo, "4G", "LTE").

FatigueDrivingInformation

```
type FatigueDrivingInformation struct {
    Version    *string `json:"Version"`
    Type       *string `json:"Type"`
    Descriptor *string `json:"Descriptor"`
}
```

- **Version**, **Type**, **Descriptor**: Cadenas opcionales relacionadas con alertas de conducción por fatiga.

AdditionalAlertInfoADASDMS

```
type AdditionalAlertInfoADASDMS struct {
    AlarmProtocol *string `json:"AlarmProtocol"`
    AlarmType     *string `json:"AlarmType"`
    PhotoName     *string `json:"PhotoName"`
}
```

- **AlarmProtocol**, **AlarmType**, **PhotoName**: Cadenas opcionales para alertas de sistemas avanzados de asistencia al conductor (ADAS) o sistemas de monitoreo de conductores (DMS).

BluetoothBeacon

```
type BluetoothBeacon struct {
    Version        *string `json:"Version"`
    DeviceName     *string `json:"DeviceName"`
    MAC            *string `json:"MAC"`
    BatteryPower   *string `json:"BatteryPower"`
    SignalStrength *string `json:"SignalStrength"`
}
```

- **Version**, **DeviceName**, **MAC**, **BatteryPower**, **SignalStrength**: Cadenas opcionales para datos de balizas Bluetooth.

TemperatureAndHumidity

```
type TemperatureAndHumidity struct {
    DeviceName           *string `json:"DeviceName"`
    MAC                  *string `json:"MAC"`
    BatteryPower         *string `json:"BatteryPower"`
    Temperature          *string `json:"Temperature"`
    Humidity             *string `json:"Humidity"`
    AlertHighTemperature *string `json:"AlertHighTemperature"`
    AlertLowTemperature  *string `json:"AlertLowTemperature"`
    AlertHighHumidity    *string `json:"AlertHighHumidity"`
    AlertLowHumidity     *string `json:"AlertLowHumidity"`
}
```

- **DeviceName**, **MAC**, **BatteryPower**, **Temperature**, **Humidity**, **AlertHighTemperature**, **AlertLowTemperature**, **AlertHighHumidity**, **AlertLowHumidity**: Cadenas opcionales para datos y alertas de sensores de temperatura y humedad.

IoPortsStatus

```
type IoPortsStatus struct {
    Port1 int `json:"Port1"`
    Port2 int `json:"Port2"`
    Port3 int `json:"Port3"`
    Port4 int `json:"Port4"`
    Port5 int `json:"Port5"`
    Port6 int `json:"Port6"`
    Port7 int `json:"Port7"`
    Port8 int `json:"Port8"`
}
```

- **Port1–Port8**: Enteros que representan el estado de los puertos de E/S (por ejemplo, 0 para apagado, 1 para encendido).

Intérpretes de Protocolo

Cada protocolo soportado tiene su propio intérprete en `interpreters`.
Su misión: **analizar datos específicos del proveedor → generar Jonoprotocol**.

1. Huabao

- Analiza datos de GPS, DVR y alarmas de Huabao.
- Mapea directamente a los campos de `JonoModel` (por ejemplo, `Latitude`, `Longitude`, `EventCode`).

2. Meitrackprotocol

- Decodifica paquetes de Meitrack (ubicación, E/S, eventos).
- Soporta múltiples tipos de dispositivos/firmwares.
- Se convierte en `JonoModel` para un procesamiento uniforme.

3. Pinoprotocol

- Para dispositivos Pino (BSJ-EG01, GT06).
- Mapea alarmas, latidos y datos GPS a `JonoModel`.

4. Queclinkprotocol

- Decodifica paquetes de dispositivos Queclink (GPS, E/S, eventos).
- Soporta múltiples modelos de Queclink, mapeados a `JonoModel`.

5. Ruptelaprotocol

- Maneja mensajes de Ruptela (ubicación, CAN, eventos).
- Manejo de errores y extensiones mapeadas a `JonoModel`.

6. Skywaveprotocol

- Analiza tramas satelitales/terrestres de Skywave.
- Convertido a `JonoModel` sin problemas.

7. Suntech

- Soporta modelos/versiones de protocolo de Suntech.
- Mapea GPS, E/S, eventos a `JonoModel`.

8. Xpot

- Maneja tramas personalizadas/estándar de Xpot.
- Unificado a través del traductor `JonoModel`.

¿Por qué MQTT?

MQTT es la columna vertebral de Jonobridge para:

- Recibir datos del dispositivo: Cada intérprete se suscribe a temas MQTT específicos para mensajes de dispositivos crudos.
- Publicar datos procesados: Después de la conversión a Jonoprotocol, los intérpretes publican los resultados en temas MQTT dedicados.
- Escalabilidad: El modelo de publicación/suscripción asíncrono de MQTT asegura que los intérpretes no se bloqueen entre sí.

Resumen del Flujo de Datos

1. El dispositivo envía datos al corredor MQTT en un tema específico del protocolo.
2. El intérprete se suscribe al tema, recibe y analiza el mensaje.
3. El intérprete convierte el mensaje al formato Jonoprotocol.
4. El intérprete publica el mensaje Jonoprotocol en un tema MQTT de salida para un procesamiento posterior.

Protocolos y Temas MQTT Reales

Protocolo	Tema(s) de Entrada	Tema(s) de Salida	Prevención de Bloqueos y Estructura
Huabao	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Cada mensaje manejado en un goroutine; sesión persistente, reconexión automática.
Meitrackprotocol	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Grupo de goroutines, cortacircuitos, monitor de salud, canales amortiguados.
Pinoprotocol	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Sync.Map para caché de dispositivos, RWMutex para operaciones críticas, no bloqueante.

Protocolo	Tema(s) de Entrada	Tema(s) de Salida	Prevención de Bloqueos y Estructura
Queclinkprotocol	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Grupo de goroutines, cortacircuitos, salud.
Ruptelaprotocol	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Goroutine por mensaje, sesión persistente, reconexión automática.
Skywaveprotocol	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Goroutine por mensaje, sesión persistente, reconexión automática.
Suntech	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Goroutine por mensaje, sesión persistente, reconexión automática.
Xpot	`http/get`	(varía, ver implementación)	Cliente MQTT con sesión persistente, sin estado.

Arquitectura de Prevención de Bloqueos

- **Goroutines**: Cada mensaje MQTT entrante se procesa en su propio goroutine, asegurando que los intérpretes no se bloqueen entre sí.
- **Canales Amortiguados y Cortacircuitos**: Utilizados en algunos intérpretes (por ejemplo, Meitrack) para procesamiento asíncrono y tolerancia a fallos.
- **Diseño sin Estado**: La mayoría de los intérpretes no comparten estado, evitando contenciones y bloqueos.
- **Contrapresión de MQTT**: El corredor gestiona el flujo de mensajes, por lo que los consumidores lentos no bloquean a los rápidos.
- **Sync.Map y Mutexes**: Utilizados en Pinoprotocol para la caché de datos del dispositivo, pero solo para secciones críticas.

Conclusiones Clave para Arquitectos de Software

- **Piensa en Jonoprotocol, no en protocolos de dispositivos** → los servicios posteriores están aislados de las diferencias de los proveedores.
- **Diseño Extensible** → nuevos campos y dispositivos pueden integrarse sin romper los consumidores existentes al aprovechar los campos opcionales en `JonoModel`.
- **Canalización Unificada** → una vez que los datos se traducen a `JonoModel`, el análisis, almacenamiento y alertas se simplifican.

Al imponer Jonoprotocol como la capa estándar, tu arquitectura gana **claridad, extensibilidad y mantenibilidad a largo plazo**.