Jonoprotocol Overview & Integration Strategy

Mission

The **Jonoprotocol** is a unified protocol designed to standardize data from diverse GPS tracking devices into a single, consistent format.
Different brands and models of trackers speak their own "languages" (proprietary protocols), which makes integration costly and error-prone.
Jonoprotocol solves this by acting as a **translator**, ensuring all device data can be managed, processed, and scaled in a uniform way.

Why Jonoprotocol?

- **Interoperability**: Handle multiple brands/models without writing custom logic for each downstream service.
- **Scalability**: Once data is standardized, it can be processed, stored, or analyzed without protocol-specific considerations.
- **Maintainability**: Adding a new protocol only requires writing a new interpreter → everything else already understands Jonoprotocol.

How It Works

1. **Device Sends Data**: Each listener (in Jonobride) transmits messages using its own proprietary protocol via MQTT in a hex string format.
2. **Interpreter Module**: A protocol-specific interpreter parses the raw message taked from the MQTT.
3. **Conversion to Jonoprotocol**: The parsed data is mapped to the unified Jonoprotocol schema.
4. **Publishing**: The Jonoprotocol message is published (e.g., to MQTT) for downstream processing.

Jonoprotocol Core Schema

The core schema is defined in common/models as Go structs, with the primary struct being JonoModel. This schema ensures **all interpreters** produce a uniform output, standardizing fields such as device identification, location, events, and extensible telemetry (e.g., sensors, IO ports, CAN data). Below is a detailed explanation of the JonoModel` and its associated structs.

JonoModel Struct

The `JonoModel` struct is the root structure that encapsulates all standardized data from a GPS tracking device.

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

- **IMEI**: A string representing the unique identifier of the device.
- **Message**: An optional string for additional device-specific messages or notes.
- **DataPackets**: An integer indicating the number of data packets in the message.
- **ListPackets**: A map of `DataPacket` structs, keyed by a unique identifier (e.g., packet ID), containing detailed telemetry data.

DataPacket Struct

The `DataPacket` struct captures detailed telemetry and status information for each packet sent by the device.

```
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**: Integer representing the altitude in meters.
- **Datetime**: Timestamp of the data packet using Go's `time.Time`.
- **EventCode**: A struct containing an event code and name (see `EventCode` below).
- **Latitude** and **Longitude**: Floating-point values for geographic coordinates.
- **Speed**: Integer representing the device's speed in kilometers per hour.
- **RunTime**: Integer indicating the device's operational time in seconds.
- **FuelPercentage**: Integer representing the fuel level as a percentage.
- **Direction**: Integer indicating the direction of travel in degrees (0–359).
- **HDOP**: Floating-point value for Horizontal Dilution of Precision, indicating GPS accuracy.
- **Mileage**: Integer representing the total distance traveled in meters.
- **PositioningStatus**: String indicating GPS fix status (e.g., "A" for valid, "V" for invalid).
- **NumberOfSatellites**: Integer indicating the number of GPS satellites in view.
- **GSMSignalStrength**: Optional integer for GSM signal strength.
- **AnalogInputs**, **IoPortStatus**, **BaseStationInfo**, **OutputPortStatus**, **InputPortStatus**, **SystemFlag**, **TemperatureSensor**, **CameraStatus**, **CurrentNetworkInfo**, **FatigueDrivingInformation**, **AdditionalAlertInfoADASDMS**, **BluetoothBeaconA**, **BluetoothBeaconB**, **TemperatureAndHumiditySensor**: Optional structs for additional telemetry data (detailed below).

Supporting Structs

The following structs provide detailed telemetry and status information, all optional to accommodate varying device capabilities.

EventCode

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

- **Code**: Integer representing the event type (e.g., 1 for "ignition on").
- **Name**: String describing the event (e.g., "Ignition On").

BaseStationInfo

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

- **MCC**: Mobile Country Code (optional).
- **MNC**: Mobile Network Code (optional).
- **LAC**: Location Area Code (optional).
- **CellID**: Cell tower ID (optional).

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**: Optional strings representing analog input values (e.g., voltage levels).

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**: Optional strings indicating the status of output ports (e.g., "ON" or "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**: Optional strings indicating the status of input ports (e.g., "HIGH" or "LOW").

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**: Optional strings representing various system states (e.g., "ON" for ACC indicating ignition status).

TemperatureSensor

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

- **SensorNumber**: Optional string identifying the sensor.
- **Value**: Optional string representing the temperature reading.

CameraStatus

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

- **CameraNumber**: Optional string identifying the camera.
- **Status**: Optional string indicating camera status (e.g., "ACTIVE").

CurrentNetworkInfo

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

- **Version**, **Type**, **Descriptor**: Optional strings providing network information (e.g., "4G", "LTE").

FatigueDrivingInformation

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

- **Version**, **Type**, **Descriptor**: Optional strings related to fatigue driving alerts.

AdditionalAlertInfoADASDMS

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

- **AlarmProtocol**, **AlarmType**, **PhotoName**: Optional strings for advanced driver-assistance system (ADAS) or driver monitoring system (DMS) alerts.

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**: Optional strings for Bluetooth beacon data.

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**: Optional strings for temperature and humidity sensor data and alerts.

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**: Integers representing the status of IO ports (e.g., 0 for off, 1 for on).

Protocol Interpreters

Each supported protocol has its own interpreter in `interpreters`.
Its mission: **parse vendor-specific data → output Jonoprotocol**.

1. Huabao

- Parses Huabao GPS, DVR, alarms.
- Maps directly into `JonoModel` fields (e.g., `Latitude`, `Longitude`, `EventCode`).

2. Meitrackprotocol

- Decodes Meitrack packets (location, IO, events).
- Supports multiple device types/firmwares.
- Converted into `JonoModel` for uniform processing.

3. Pinoprotocol

- For Pino devices (BSJ-EG01, GT06).
- Maps alarms, heartbeats, and GPS data into `JonoModel`.

4. Queclinkprotocol

- Decodes Queclink device packets (GPS, IO, events).
- Multiple Queclink models supported, mapped to `JonoModel`.

5. Ruptelaprotocol

- Handles Ruptela messages (location, CAN, events).
- Error handling and extensions mapped to `JonoModel`.

6. Skywaveprotocol

- Parses satellite/terrestrial Skywave frames.
- Converted to `JonoModel` seamlessly.

7. Suntech

- Supports Suntech models/protocol versions.
- Maps GPS, IO, events into `JonoModel`.

8. Xpot

- Handles custom/standard Xpot frames.
- Unified through `JonoModel` translator.

Why MQTT?

MQTT is the backbone of jonobridge for:

- Receiving device data: Each interpreter subscribes to specific MQTT topics for raw device messages.
- Publishing processed data: After conversion to jonoprotocol, interpreters publish results to dedicated MQTT topics.
- Scalability: MQTT's asynchronous publish/subscribe model ensures interpreters do not block each other.

Data Flow Overview

1. Device sends data to the MQTT broker on a protocol-specific topic.
2. Interpreter subscribes to the topic, receives and parses the message.
3. Interpreter converts the message to jonoprotocol format.
4. Interpreter publishes the jonoprotocol message to an output MQTT topic for further processing.

Protocols and Real MQTT Topics

Protocol	Input Topic(s)	Output Topic(s)	Lock Prevention & Structure
Huabao	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Each message handled in a goroutine; persistent session, auto-reconnect.
Meitrackprotocol	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Goroutine pool, circuit breaker, health monitor, buffered channels.
Pinoprotocol	`tracker/from-tcp`, `tracker/from-udp	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Sync.Map for device cache, RWMutex for critical ops, non-blocking.

Protocol	Input Topic(s)	Output Topic(s)	Lock Prevention & Structure
Queclinkprotocol	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Goroutine pool, circuit breaker, health
Ruptelaprotocol	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Goroutine per message, persistent session, auto-reconnect.
Skywaveprotocol	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Goroutine per message, persistent session, auto-reconnect.
Suntech	`tracker/from-tcp`, `tracker/from-udp`	`tracker/jonoprotocol`, `tracker/assign-imei2remoteaddr`	Goroutine per message, persistent session, auto-reconnect.
Xpot	`http/get`	(varies, see implementation)	MQTT client with persistent session, stateless.

Lock Prevention Architecture

- **Goroutines**: Each incoming MQTT message is processed in its own goroutine, ensuring interpreters do not block each other.
- **Buffered Channels & Circuit Breakers**: Used in some interpreters (e.g., Meitrack) for async processing and fault tolerance.
- **Stateless Design**: Most interpreters do not share state, preventing contention and locking.
- **MQTT Backpressure**: The broker manages message flow, so slow consumers do not block fast ones.
- **Sync.Map and Mutexes**: Used in Pinoprotocol for device data cache, but only for critical sections.

Key Takeaways for Software Architects

- **Think in Jonoprotocol, not device protocols** → downstream services are insulated from vendor differences.
- **Extensible Design** → new fields and devices can be integrated without breaking existing consumers by leveraging optional fields in `JonoModel`.
- **Unified Pipeline** → once data is translated into `JonoModel`, analytics, storage, and alerting are simplified.

By enforcing Jonoprotocol as the standard layer, your architecture gains **clarity, extensibility, and long-term maintainability**.