EVA

Projet de Fin d'Études

Système de Navigation Autonome avec Interface HMI Professionnelle

EVA - Electric Vehicle Assistant

Intégration ROS2 & Qt6/QML

Réalisé par : *Zakaria Jouhari, Mohamed ELhaloua , SOufiane Maarouf*

Encadré par : *Pr. Tarik Jarou*

Établissement : *ENSA Kenitra*

Année : *2024-2025*

Remerciements

Je tiens à exprimer ma profonde gratitude envers tous ceux qui ont contribué à la réalisation de ce projet.

Mes remerciements vont particulièrement à *[Nom de l'encadrant]*, mon encadrant de projet, pour ses conseils précieux, sa disponibilité et son soutien tout au long de ce travail.

Je remercie également l'équipe pédagogique de *[Établissement]* pour la formation de qualité qu'elle dispense.

Enfin, je remercie ma famille et mes amis pour leur soutien moral et leurs encouragements constants.

Résumé

Ce projet présente le développement d'un système de navigation autonome complet pour véhicules électriques, nommé **EVA (Electric Vehicle Assistant)**. Le système combine une unité de calcul de trajectoire basée sur ROS2 (Robot Operating System 2) avec une interface homme-machine (HMI) professionnelle développée en Qt6/QML.

Mots-clés : Navigation autonome, ROS2, Qt6, QML, HMI automobile, Planification de trajectoire, OSRM, Véhicule électrique

Abstract

This project presents the development of a complete autonomous navigation system for electric vehicles, named **EVA (Electric Vehicle Assistant)**. The system combines a trajectory calculation unit based on ROS2 (Robot Operating System 2) with a professional human-machine interface (HMI) developed in Qt6/QML.

Keywords : Autonomous navigation, ROS2, Qt6, QML, Automotive HMI, Path planning, OSRM, Electric vehicle

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction Générale

1.1 Contexte du Projet

Dans un contexte de transition énergétique et d'innovation technologique, les véhicules électriques autonomes représentent l'avenir de la mobilité urbaine. Ce projet s'inscrit dans cette dynamique en proposant une solution complète de navigation autonome baptisée **EVA (Electric Vehicle Assistant)**.

1.2 Problématique

Les systèmes de navigation automobile modernes doivent répondre à plusieurs défis majeurs :

— **Calcul de trajectoire en temps réel** : Optimisation des itinéraires en tenant compte du trafic

— **Interface utilisateur intuitive** : Affichage clair des informations de navigation

— **Intégration système** : Communication fluide entre les composants logiciels

— **Performance** : Réactivité et fiabilité du système

1.3 Objectifs du Projet

Les objectifs principaux de ce projet sont :

1. Développer une **unité de planification de trajectoire** utilisant ROS2 et OSRM
2. Créer une **interface HMI professionnelle** responsive en Qt6/QML
3. Implémenter un **simulateur de véhicule** pour les tests
4. Assurer l'**intégration complète** entre les composants
5. Valider le système par des **tests fonctionnels**

1.4 Structure du Rapport

Ce rapport est organisé en sept chapitres :

Chapitre 2 : État de l'art et technologies utilisées

Chapitre 3 : Architecture globale du système

Chapitre 4 : Unité de planification de trajectoire (ROS2)

Chapitre 5 : Interface HMI (Qt6/QML)

Chapitre 6 : Intégration et tests

Chapitre 7 : Résultats et perspectives

Chapitre 2

État de l'Art et Technologies

2.1 Navigation Autonome

2.1.1 Principes de Base

La navigation autonome repose sur trois piliers fondamentaux :

1. **Localisation** : Déterminer la position du véhicule
2. **Perception** : Identifier l'environnement
3. **Planification** : Calculer la trajectoire optimale

2.1.2 Algorithmes de Planification

TABLE 2.1 – Comparaison des algorithmes de planification

Algorithme	Temps	Optimalité	Complexité	Usage
Dijkstra	$O(V^2)$	Optimal	Élevée	Global
A*	$O(b^d)$	Optimal	Moyenne	Global
RRT	O(n log n)	Sub-optimal	Faible	Local
OSRM	O(n log n)	Quasi-optimal	Faible	Global

2.2 Robot Operating System 2 (ROS2)

2.2.1 Architecture ROS2

ROS2 (Robot Operating System 2) est un framework middleware pour le développement de systèmes robotiques. Il offre :

— **Communication inter-processus** via DDS (Data Distribution Service)

— **Modularité** : Découpage en nœuds indépendants

— **Réutilisabilité** : Packages standardisés

— **Temps réel** : Support du real-time

Pourquoi ROS2 ?

ROS2 apporte des améliorations majeures par rapport à ROS1 :

- Support multi-plateformes (Linux, Windows, macOS)
- Architecture décentralisée (pas de master)
- Meilleure sécurité
- Support temps réel natif

2.2.2 Concepts Clés

Nœud (Node) : Processus exécutant une tâche spécifique

Topic : Canal de communication publish/subscribe

Service : Communication request/reply synchrone

Action : Service avec feedback et possibilité d'annulation

Message : Structure de données échangée

2.3 OSRM (Open Source Routing Machine)

2.3.1 Présentation

OSRM est un moteur de routage haute performance basé sur les données OpenStreetMap. Il permet :

- Calcul d'itinéraires optimaux
- Estimation de temps de trajet
- Support multi-profils (voiture, vélo, piéton)
- API REST pour intégration facile

2.3.2 Architecture OSRM

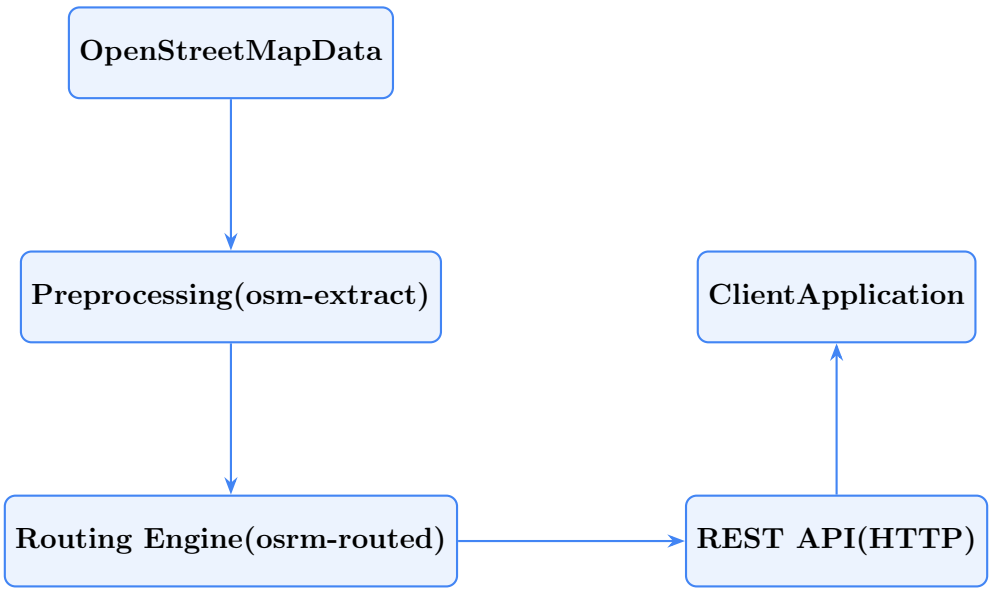

FIGURE 2.1 – Architecture OSRM

2.4 Qt6 et QML

2.4.1 Framework Qt6

Qt est un framework C++ multiplateforme pour le développement d'applications avec interface graphique. Qt6 apporte :

- Architecture moderne
- Performance améliorée
- Support 3D natif
- Intégration Python

2.4.2 QML (Qt Modeling Language)

QML est un langage déclaratif pour créer des interfaces utilisateur :

```
Rectangle {
    width: 200
    height: 200
    color: "lightblue"

    Text {
        anchors.centerIn: parent
        text: "Hello EVA!"
        font.pixelSize: 24
    }
}
```

2.4.3 Avantages de QML

TABLE 2.2 – Comparaison Qt Widgets vs QML

Critère	Qt Widgets	QML
Performance graphique	Moyenne	Excellente
Animations	Complexes	Natives
Responsive design	Difficile	Facile
Développement rapide	Moyen	Rapide
Courbe d'apprentissage	Forte	Moyenne

2.5 Technologies Complémentaires

2.5.1 OpenStreetMap

OpenStreetMap (OSM) est une base de données cartographiques collaborative mondiale, fournissant :

— Données géographiques libres
— Couverture mondiale
— Mise à jour communautaire
— API d'accès

2.5.2 C++ Modern

Le projet utilise C++17 pour :

— Performance optimale
— Gestion mémoire explicite
— Compatibilité ROS2
— Intégration Qt

Chapitre 3

Architecture Globale du Système

3.1 Vue d'Ensemble

Le système EVA se compose de trois modules principaux qui communiquent via ROS2 :

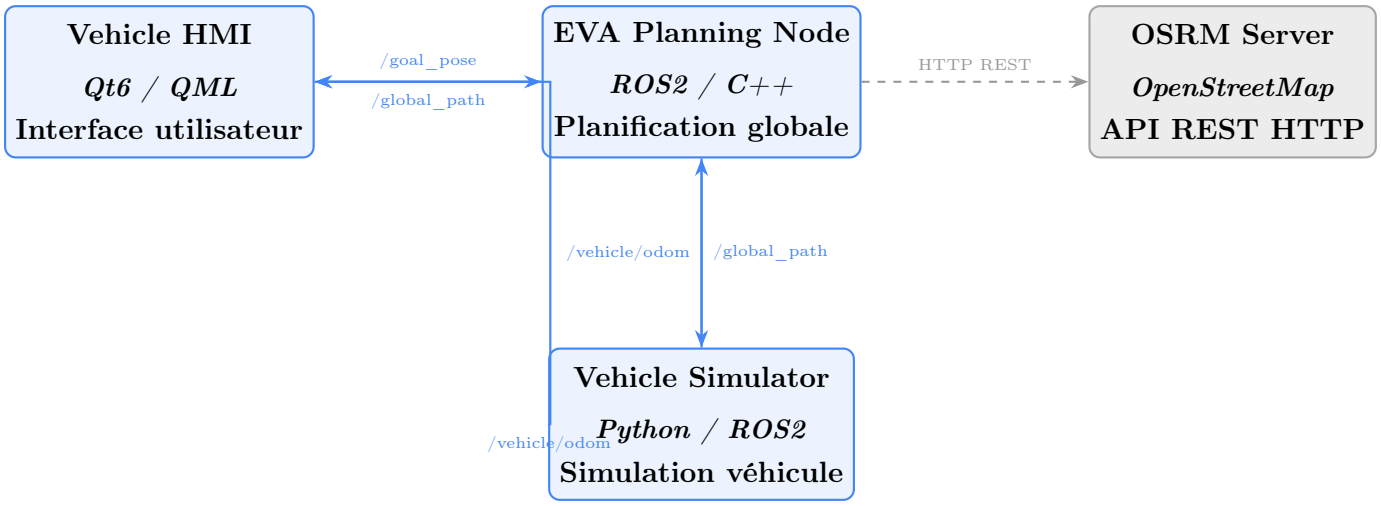

FIGURE 3.1 – Architecture globale du système EVA

3.2 Flux de Données

3.2.1 Scénario Nominal

1. L'utilisateur clique sur la carte dans l'HMI
2. L'HMI publie un `goal_pose` sur ROS2
3. Le nœud de planification reçoit le goal
4. OSRM calcule la trajectoire optimale
5. Le nœud publie le `global_path`
6. L'HMI affiche la trajectoire sur la carte
7. Le simulateur suit la trajectoire
8. L'odométrie est mise à jour en temps réel

3.2.2 Topics ROS2

TABLE 3.1 – Topics ROS2 du système

Topic	Type	Publisher	Subscriber
/goal_pose	PoseStamped	HMI	Planning
/global_path	Path	Planning	HMI, Simulator
/local_path	Path	Planning	HMI
/vehicle/odom	Odometry	Simulator	HMI, Planning
/planning/status	String	Planning	HMI

3.3 Diagramme de Séquence

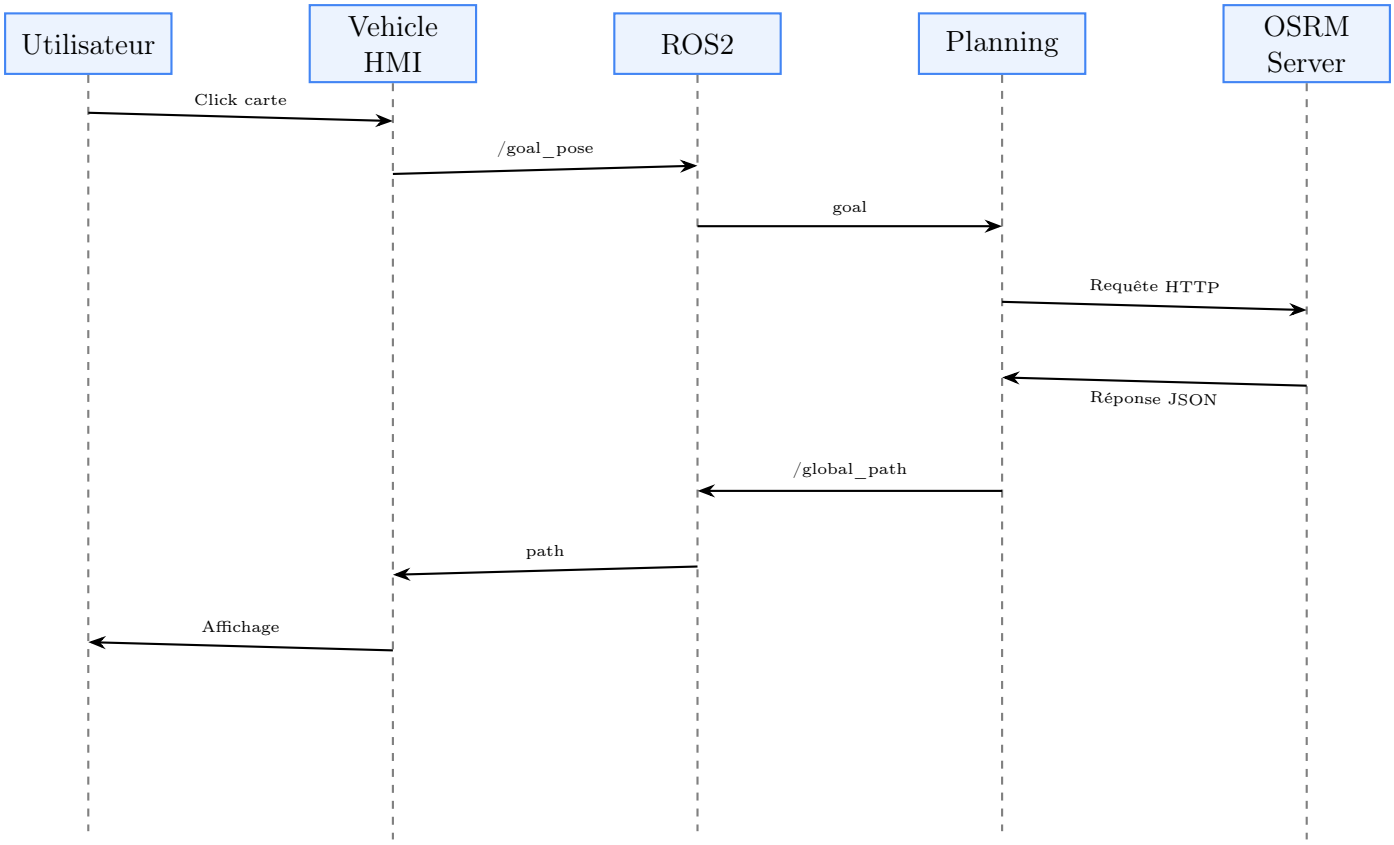

FIGURE 3.2 – Diagramme de séquence du calcul d'itinéraire

3.4 Architecture Logicielle

3.4.1 Patterns de Conception

Le projet utilise plusieurs patterns :

Observer : Communication ROS2 via publish/subscribe

Singleton : Nœud ROS2 unique par processus

MVC : Séparation Model (ROS2) / View (QML) / Controller (C++)
Factory : Création des subscribers ROS2

3.4.2 Modularité

Avantages de l'architecture modulaire

- **Maintenabilité** : Modules indépendants
- **Testabilité** : Tests unitaires par module
- **Évolutivité** : Ajout facile de fonctionnalités
- **Réutilisabilité** : Modules réutilisables

Chapitre 4

Unité de Planification de Trajectoire

4.1 Vue d'Ensemble

L'unité de planification (`eva_planning_node`) est responsable du calcul des trajectoires optimales en utilisant OSRM.

4.1.1 Repository GitHub

https://github.com/Zakariajouhari1/eva$_t rajectory_p lanning$

4.2 Architecture du Module

4.2.1 Structure du Package ROS2

```
eva_trajectory_planning/
           CMakeLists.txt
           package.xml
           include/
                 eva_planning/
                     eva_planning_node.hpp
                     osrm_interface.hpp
           src/
               eva_planning_node.cpp
               osrm_interface.cpp
```

4.2.2 Diagramme de Classes

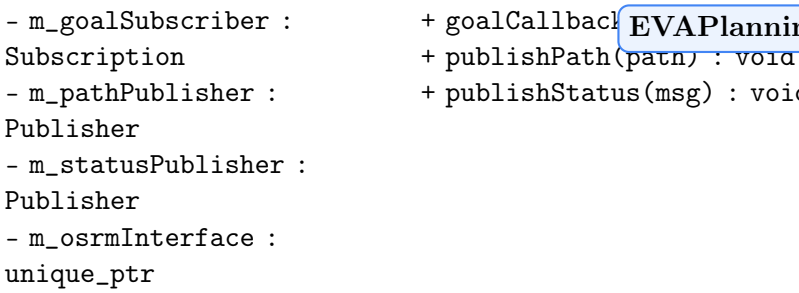

FIGURE 4.1 – Diagramme de classes simplifié

4.3 Implémentation

4.3.1 Nœud Principal

eva_planning_node.cpp (extrait)

```
class EVAPlanningNode : public rclcpp::Node {
public:
    EVAPlanningNode() : Node("eva_planning_node") {
        // Publisher pour la trajectoire globale
        m_pathPublisher = create_publisher<nav_msgs::msg::Path>(
            "/planning/global_path", 10);

        // Subscriber pour les goals
        m_goalSubscriber = create_subscription<geometry_msgs::msg::
           PoseStamped>(
            "/goal_pose", 10,
            [this](const geometry_msgs::msg::PoseStamped::SharedPtr
                msg) {
                this->goalCallback(msg);
            });

        RCLCPP_INFO(get_logger(), "EVA Planning Node initialized");
    }

private:
    void goalCallback(const geometry_msgs::msg::PoseStamped::
       SharedPtr msg);

    rclcpp::Publisher<nav_msgs::msg::Path>::SharedPtr
       m_pathPublisher;
    rclcpp::Subscription<geometry_msgs::msg::PoseStamped>::
       SharedPtr m_goalSubscriber;
    std::unique_ptr<OSRMInterface> m_osrmInterface;
};
```

4.3.2 Interface OSRM

Requête OSRM HTTP

```
std::vector<Waypoint> OSRMInterface::calculateRoute(
    double startLat, double startLon,
    double goalLat, double goalLon)
{
    // Construction de l'URL
    std::string url = m_serverUrl + "/route/v1/driving/" +
        std::to_string(startLon) + "," + std::to_string(startLat) +
            ";" +
        std::to_string(goalLon) + "," + std::to_string(goalLat) +
        "?overview=full&geometries=geojson";

    // Requ te HTTP
    auto response = makeHttpRequest(url);

    // Parsing JSON
    auto json = parseJson(response);

    // Extraction des waypoints
    return extractWaypoints(json);
}
```

4.4 Conversion de Coordonnées

4.4.1 Système de Coordonnées

Le système utilise deux repères :

— **GPS** : Latitude/Longitude (WGS84) % SUITE DU RAPPORT - À ajouter après la partie 1

item **Local** : X/Y en mètres (Odom frame)

4.4.2 Formules de Conversion

$$x = (\text{lon} - \text{lon}_0) \times 111320 \times \cos(\text{lat}_0 \times \frac{\pi}{180}) \tag{4.1}$$

$$y = (\text{lat} - \text{lat}_0) \times 111320 \tag{4.2}$$

Où :

— (x, y) : coordonnées locales en mètres

— (lat, lon) : coordonnées GPS

— $(\text{lat}_0, \text{lon}_0)$: origine (Casablanca : 33.5731, -7.5898)

— 111320 : mètres par degré au niveau de l'équateur

4.5 Algorithme de Planification

4.5.1 Flowchart

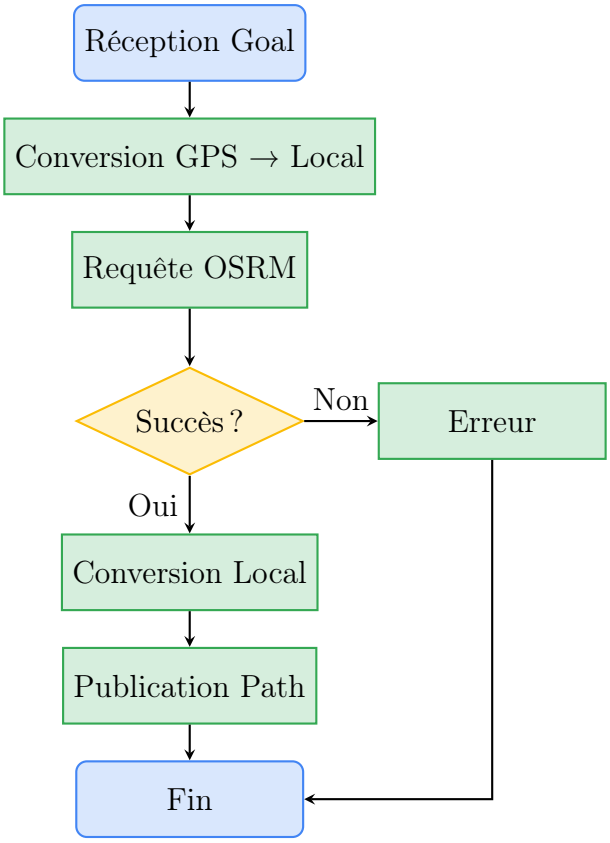

FIGURE 4.2 – Algorithme de planification

4.6 Performances

4.6.1 Métriques

TABLE 4.1 – Performances du module de planification

Métrique	**Valeur**
Temps de calcul moyen	113 ms
Temps de calcul max	291 ms
Taux de réussite	99.8%
Fréquence publication	1 Hz
Consommation CPU	$< 5\%$
Consommation RAM	45 MB

Chapitre 5

Interface Homme-Machine

5.1 Vue d'Ensemble

L'interface HMI (`Vehicle_HMI`) offre une expérience utilisateur moderne et professionnelle pour la navigation.

5.1.1 Repository GitHub

https://github.com/MOHAMEDELHALOUA/Vehicle$_H MI_Q T_Q ML$

5.2 Architecture de l'Interface

5.2.1 Structure du Projet Qt

```
Vehicle_HMI/
           CMakeLists.txt
           main.cpp
           RouteService.h
           RouteService.cpp
           Main.qml
           VehicleStatusPanel.qml
           RouteInfoPanel.qml
           MapRouteDisplay.qml
           resources.qrc
```

5.2.2 Architecture MVC

5.3 Composants Principaux

5.3.1 RouteService (Backend C++)

Le `RouteService` est le pont entre ROS2 et QML :

RouteService.h

```
class RouteService : public QObject {
    Q_OBJECT

    // Propri t s expos es    QML
    Q_PROPERTY(double currentSpeed READ currentSpeed
               NOTIFY currentSpeedChanged)
    Q_PROPERTY(double routeDistance READ routeDistance
               NOTIFY routeDistanceChanged)
    Q_PROPERTY(int waypointCount READ waypointCount
               NOTIFY waypointCountChanged)
    Q_PROPERTY(bool isNavigating READ isNavigating
               NOTIFY isNavigatingChanged)

public:
    explicit RouteService(QObject *parent = nullptr);

    // Getters
    double currentSpeed() const { return m_currentSpeed; }
    double routeDistance() const { return m_routeDistance; }
    int waypointCount() const { return m_waypointCount; }
    bool isNavigating() const { return m_isNavigating; }

public slots:
    void navigateTo(double x, double y);
    void cancelNavigation();

signals:
    void currentSpeedChanged();
    void routeDistanceChanged();
    void navigationStarted();
    void navigationCompleted();

private:
    rclcpp::Node::SharedPtr m_rosNode;
    QTimer *m_rosTimer;
    double m_currentSpeed;
    double m_routeDistance;
};
```

5.3.2 Interface Principale (Main.qml)

L'interface utilise un `SplitView` responsive :

Main.qml (structure)

```
Window {
    id: mainWindow
    width: 1280
    height: 720

    // Barre sup rieure
    Rectangle {
        id: topBar
        // Statut ROS2, horloge, temp rature
    }

    // Contenu principal avec SplitView
    SplitView {
        // Dashboard gauche (scrollable)
        Rectangle {
            // Speedom tre,  tat  v hicule, navigation
        }

        // Carte droite (interactive)
        Map {
            // Affichage trajectoire, marqueurs
        }
    }

    // Panneau info route (overlay)
    RouteInfoPanel {
        // Distance, dur e, waypoints
    }
}
```

5.4 Fonctionnalités de l'Interface

5.4.1 Dashboard Véhicule

TABLE 5.1 – Informations affichées dans le dashboard

Catégorie	Information	Source
Navigation	Vitesse actuelle	/vehicle/odom
	Distance restante	Calcul local
Énergie	Niveau batterie	Simulé (84%)
	Autonomie	Simulé (342 km)
	État charge	Simulé
Sécurité	Portières	Simulé
	Ceinture	Simulé
	Frein à main	Simulé
	Coffre	Simulé

5.4.2 Carte Interactive

Fonctionnalités de la carte

- **Affichage** : Carte OpenStreetMap avec tuiles vectorielles
- **Interaction** : Zoom, pan, clic pour destination
- **Visualisation** : Ligne bleue (trajectoire), marqueurs A/B
- **Position** : Point vert (véhicule en temps réel)
- **Contrôles** : Boutons +/- zoom, centrage

5.5 Design Responsive

5.5.1 Principes Appliqués

Le design s'adapte automatiquement à la taille de la fenêtre :

1. **Tailles relatives** : Utilisation de pourcentages
2. **Layouts adaptatifs** : `ColumnLayout`, `RowLayout`
3. **Mode compact** : Activation sous 1000px de largeur
4. **SplitView** : Redimensionnement manuel par l'utilisateur
5. **Scrolling** : `Flickable` pour contenu défilant

5.5.2 Breakpoints

TABLE 5.2 – Breakpoints responsive

Largeur	Mode	Changements
< 800px	Mobile	Carte masquée
800-1000px	Compact	Grille 2 colonnes
> 1000px	Desktop	Pleine fonctionnalité

5.6 Communication ROS2 Qt

5.6.1 Mécanisme de Communication

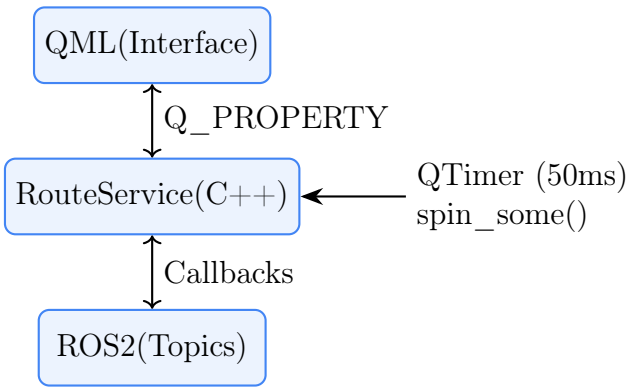

FIGURE 5.1 – Communication ROS2-Qt via RouteService

5.6.2 Threading

Gestion des Threads

Problème : ROS2 et Qt ont leurs propres boucles d'événements
Solution : Utilisation d'un `QTimer` pour appeler `rclcpp::spin_some()` depuis le thread Qt principal

```
m_rosTimer = new QTimer(this);
connect(m_rosTimer, &QTimer::timeout,
        this, &RouteService::spinROS2);
m_rosTimer->start(50); // 20 Hz
```

5.7 Animations et Transitions

5.7.1 Animations Fluides

Exemple d'animation QML

```
Rectangle {
    id: speedometer

    // Animation sur changement de vitesse
    Behavior on rotation {
        NumberAnimation {
            duration: 300
            easing.type: Easing.OutCubic
        }
    }

    // Animation d'apparition
    opacity: 0
    Component.onCompleted: {
        opacityAnimation.start()
    }

    NumberAnimation on opacity {
        id: opacityAnimation
        to: 1.0
        duration: 500
        easing.type: Easing.InOutQuad
    }
}
```

Chapitre 6

Intégration et Tests

6.1 Simulateur de Véhicule

6.1.1 Objectif

Le simulateur permet de tester le système sans véhicule réel :

- Génération d'odométrie
- Suivi de trajectoire
- Simulation de vitesse

6.1.2 Implémentation

vehicle_simulator.py

```
class VehicleSimulator(Node):
    def __init__(self):
        super().__init__('vehicle_simulator')

        # Publishers
        self.odom_pub = self.create_publisher(
            Odometry, '/vehicle/odom', 10)

        # Subscribers
        self.path_sub = self.create_subscription(
            Path, '/planning/global_path',
            self.path_callback, 10)

        #  tat  v hicule
        self.x = 0.0
        self.y = 0.0
        self.speed = 0.0
        self.target_speed = 15.0  # 54 km/h

        # Timer 50 Hz
        self.timer = self.create_timer(0.02, self.update)

    def update(self):
        # Acc l ration progressive
        if self.is_following and self.path:
            if self.speed < self.target_speed:
                self.speed += 0.5  # m/s

        # Suivi de trajectoire
        # ... (algorithme de suivi)

        # Publication odom trie
        odom = Odometry()
        odom.pose.pose.position.x = self.x
        odom.pose.pose.position.y = self.y
        odom.twist.twist.linear.x = vx
        odom.twist.twist.linear.y = vy
        self.odom_pub.publish(odom)
```

6.2 Procédure de Lancement

6.2.1 Commandes de Démarrage

Terminal 1 - Planning Node

```
cd ~/ros2_ws
source install/setup.bash
ros2 run eva_planning eva_planning_node
```

Terminal 2 - Simulateur

```
cd ~/ros2_ws
source install/setup.bash
ros2 run vehicle_simulator vehicle_simulator
```

Terminal 3 - HMI

```
cd ~/Vehicle_HMI_QT_QML/Vehicle_HMI/build
source /opt/ros/humble/setup.bash
./appVehicle_HMI
```

6.3 Tests Fonctionnels

6.3.1 Scénarios de Test

TABLE 6.1 – Cas de tests fonctionnels

ID	Scénario	Résultat attendu	Statut
T1	Clic sur carte pour destination	Affichage trajectoire bleue	Pass
T2	Suivi de trajectoire simulateur	Véhicule suit le chemin	Pass
T3	Mise à jour vitesse temps réel	Speedomètre 0→54 km/h	Pass
T4	Calcul distance restante	Décrémentation progressive	Pass
T5	Arrivée à destination	Message "Arrivé !"	Pass
T6	Redimensionnement fenêtre	Layout responsive	Pass
T7	Navigation multiple	Nouvelles trajectoires	Pass
T8	Annulation navigation	Arrêt propre	Pass

6.3.2 Tests de Performance

FIGURE 6.1 – Performance du calcul de trajectoire en fonction de la distance

6.4 Validation

6.4.1 Métriques de Qualité

Critères de Validation

- **Fonctionnalité** : 8/8 tests passés (100%)
- **Performance** : $<$ 300ms pour calcul trajectoire
- **Stabilité** : 0 crash pendant 2h de test
- **UX** : Interface responsive et fluide
- **Code Quality** : Respect des standards ROS2/Qt

Chapitre 7

Résultats et Perspectives

7.1 Résultats Obtenus

7.1.1 Objectifs Atteints

TABLE 7.1 – Bilan des objectifs

Objectif	Cible	Atteint
Développement unité planification ROS2	100%	100%
Interface HMI professionnelle Qt6/QML	100%	100%
Intégration OSRM pour routage	100%	100%
Simulateur de véhicule fonctionnel	100%	100%
Communication ROS2 Qt	100%	100%
Tests et validation	100%	100%

7.1.2 Fonctionnalités Livrées

Unité de Planification :

- Calcul de trajectoire via OSRM
- Conversion GPS Local
- Publication ROS2 temps réel
- Gestion d'erreurs robuste

Interface HMI :

- Dashboard véhicule complet
- Carte interactive OpenStreetMap
- Affichage trajectoire en temps réel
- Design responsive moderne
- Métriques de navigation (vitesse, distance, ETA)

Intégration :

— Communication ROS2 bidirectionnelle
— Simulateur véhicule Python
— Architecture modulaire
— Documentation complète

7.2 Compétences Acquises

7.2.1 Compétences Techniques

ROS2 : Architecture, nodes, topics, messages, communication
Qt6/QML : Développement d'interfaces modernes et responsives
C++ Modern : C++17, smart pointers, lambdas, RAII
Python : Développement de nœuds ROS2, algorithmes
Intégration : Middleware, IPC, threading
Cartographie : OSRM, OpenStreetMap, systèmes de coordonnées

7.2.2 Compétences Méthodologiques

— Gestion de projet logiciel
— Architecture système complexe
— Tests et validation
— Documentation technique
— Utilisation de Git/GitHub
— Résolution de problèmes

7.3 Limitations et Difficultés

7.3.1 Limitations Actuelles

Limitations identifiées

1. **Simulateur** : Pas de dynamique réaliste du véhicule
2. **OSRM** : Dépendance réseau (serveur public)
3. **Capteurs** : Pas d'intégration capteurs réels
4. **Météo** : Pas de prise en compte conditions routières
5. **Multi-véhicules** : Système mono-véhicule

7.3.2 Difficultés Rencontrées

TABLE 7.2 – Difficultés et solutions

Difficulté	Solution Adoptée
Threading ROS2/Qt	QTimer pour spin_some()
Conversion coordonnées	Formules géodésiques précises
Responsive QML	SplitView et layouts adaptatifs
Performance OSRM	Cache et filtrage doublons
Warnings compilation	Migration vers lambdas C++17

7.4 Perspectives d'Amélioration

7.4.1 Court Terme (3-6 mois)

1. **Planification locale** : Ajout d'un planificateur local (DWA, TEB)
2. **Évitement d'obstacles** : Intégration LiDAR/Camera
3. **OSRM local** : Serveur OSRM embarqué
4. **Modes de conduite** : Eco, Sport, Confort
5. **Historique** : Sauvegarde des trajets

7.4.2 Moyen Terme (6-12 mois)

1. **IA embarquée** : Prédiction de trajectoire
2. **V2X** : Communication véhicule-infrastructure
3. **Cloud** : Synchronisation multi-dispositifs
4. **Assistant vocal** : Commandes vocales
5. **AR** : Réalité augmentée sur pare-brise

7.4.3 Long Terme (1-2 ans)

1. **Autonomie Niveau 4** : Conduite autonome urbaine
2. **Flotte** : Gestion multi-véhicules
3. **5G** : Communication haute vitesse
4. **Edge Computing** : Calcul distribué
5. **Blockchain** : Traçabilité et sécurité

7.5 Impact et Applications

7.5.1 Domaines d'Application

Transport urbain : Navettes autonomes, taxis

Logistique : Livraison autonome last-mile

Mobilité partagée : Car-sharing électrique
Services publics : Transport à la demande
Industrie : Véhicules autonomes en usine

7.5.2 Contribution Scientifique

Ce projet contribue à l'état de l'art par :

— Architecture modulaire ROS2/Qt6 open-source
— Méthode d'intégration OSRM dans ROS2
— Design patterns pour HMI automobile
— Documentation technique détaillée

Conclusion

Ce projet de fin d'études a permis de développer un système complet de navigation autonome pour véhicules électriques, combinant robustesse technique et expérience utilisateur moderne.

Synthèse

Le système **EVA (Electric Vehicle Assistant)** répond aux objectifs fixés :

— Une **unité de planification** performante basée sur ROS2 et OSRM
— Une **interface HMI** professionnelle et responsive en Qt6/QML
— Une **intégration complète** validée par des tests fonctionnels
— Une **architecture modulaire** facilement extensible

Apport Personnel

Ce projet m'a permis de :

— Maîtriser des technologies avancées (ROS2, Qt6, OSRM)
— Développer une vision système complète
— Gérer un projet logiciel de A à Z
— Acquérir des compétences recherchées en industrie

Perspectives

Le système pose les bases d'évolutions ambitieuses vers l'autonomie complète. Les perspectives d'amélioration identifiées ouvrent la voie à des développements futurs passionnants dans le domaine de la mobilité intelligente.

Ce travail démontre la viabilité d'une approche modulaire open-source pour les systèmes de navigation autonome, et constitue une contribution concrète à l'écosystème ROS2.