卒業論文

軽量 ROS ランタイム mROS 2 の評価に向けたロボット制御アプリケーションの実装

公立はこだて未来大学
システム情報科学部　複雑系知能学科
知能システムコース　1020259

中村 碧

指導教員　　松原 克弥

提出日　　2024 年 1 月 25 日

BA Thesis

Implementing Robot Applications Toward Evaluating 'mROS 2' a Lightweight ROS Runtime

by

Aoi Nakamura

Intelligent Systems Course, Department of Complex and Intelligent Systems
School of Systems Information Science, Future University Hakodate

Supervisor:　　Katsuya Matsubara

Submitted on January 25th, 2024

Abstract–

The burgeoning application of the Robot Operating System (ROS) in the development of robot software signifies a trend towards the integration of distributed robot systems with cloud computing. However, the challenge of node placement flexibility, especially given the disparities in CPU architecture between cloud servers and robotic devices, poses a significant hurdle to dynamic node reconfiguration. This obstacle has prompted previous research efforts to explore dynamic placement mechanisms, which unfortunately resulted in increased overhead. The introduction of mROS 2-Wasm, alongside mROS 2-POSIX, represents an innovative approach to this issue, yet their performance has largely been evaluated through network throughput microbenchmarks, leaving a gap in understanding their effectiveness in real-world scenarios. This study aims to bridge this gap by providing a comprehensive comparison and evaluation of mROS 2-POSIX, mROS 2-Wasm, and ROS 2, focusing on their communication performance and memory footprint. The findings from this research indicate that both mROS 2-POSIX and mROS 2-Wasm are capable of achieving the anticipated performance levels in complex application environments, thereby validating their potential for enhancing the efficiency and flexibility of distributed robot systems.

Keywords: cloud robotics, Robot Operating System, WebAssembly

概 要： ロボットソフトウェア開発において，ROS の利用が増加している．ROS を活用したロボットソフトウェアの構築ではクラウドを用いた分散ロボットシステムが注目されているが，ノード配置の柔軟性が課題となっている．特に，クラウドとロボットの CPU アーキテクチャの違いから，動的なノード再配置が難しい．先行研究での動的配置機構はオーバーヘッドの増加が問題とされ，mROS 2-POSIX を用いたアプローチとして mROS 2-Wasm も開発された．しかし，性能評価はネットワークスループットのマイクロベンチマークに留まり，実アプリケーションにおける有用性が十分に評価されていない．本研究では，mROS 2-POSIX と mROS 2-Wasm，ROS 2 の性能を比較評価する．実験結果によって得た通信性能やメモリサイズをもとに，mROS 2-POSIX と mROS 2-Wasm が複雑なアプリケーション上でも期待される性能を発揮できることを示す．

キーワード： クラウドロボティクス,Robot Operating System,WebAssembly

目次

第 1 章

序論

1.1 背景

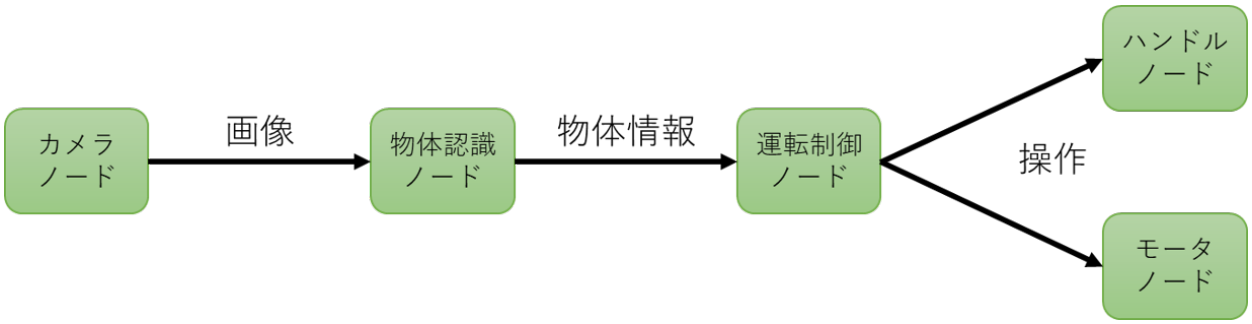

図 1.1: ROS プログラミングモデルの例

さまざまな産業向けやエンターテイメント関連のロボットシステム，自動車の自動運転技術や IoT システムのソフトウェア開発をサポートするフレームワークとして Robot Operating System（以下，ROS）の普及が進んでいる [1]．ROS のプログラミングモデルは，システムの各機能を独立したプログラムモジュール（ノード）として設計することにより，汎用性と再利用性を向上させて，各機能モジュール間のデータ交換を規定することで，効率的かつ柔軟なシステム構築を可能にしている．例として図 1.1 に示す．カメラを操作して周囲の環境を撮影するノード，画像からオブジェクトを識別するノード，オブジェクトのデータをもとに動作制御を実行するノードを連携させることで，自動運転車の基本機能の一部を容易に実装できる．

ROS のプログラミングモデルは，ロボット/IoT とクラウドが協力する分散型システムにおいても有効である．ロボットシステムのソフトウェア処理は，外界の情報を取得する「センサー」，取得した情報を処理する「知能・制御系」，実際に動作するモーターなどの「動力系」の 3 要素に分類できる [2]．クラウドとロボットが連携して高度なロボットシステムを実現するクラウドロボティクス [3] において，主に知能・制御系のノードを高い計算能力を持

つクラウドに優先して配置することで，高度な知能・制御処理の実現を促進できる．さらに，ロボットが取得した情報や状態などをクラウドに集約・保存することで，複数のロボット間での情報共有と利用を容易にする．

1.2 課題

一方で，現行の ROS 実装では，各ノードの配置をシステム起動時に静的に設定する必要があり，クラウドとロボット間の最適なノード配置を事前に設計する必要がある．実際の環境で動作するロボットは，ネットワークの状況やバッテリー残量の変動など，システム運用前に予測することが困難な状況変化に対応する必要があり，設定したクラウドとロボット間のノード配置が最適でなくなる可能性がある．このような状況変化への対応として，ノードを動的に再配置するライブマイグレーション技術があるが，多くの場合でクラウドとロボット間の CPU アーキテクチャが異なり，命令セットがそれぞれ違うため，実行中のノードをシステム運用中にマイグレーションすることは技術的に困難である．

1.3 目的

菅ら [4] は，WebAssembly（以下，Wasm）を用いることで，クラウドとロボット間での実行状態を含む稼働中ノードの動的にマイグレーションする手法を実現した．Wasm とは Web 上で高速にプログラムを実行するために設計された仮想命令セットアーキテクチャのことで，1 つのバイナリが複数のアーキテクチャで動作するため，異なる CPU アーキティクチャ間でのマイグレーションに適しているといえる．課題として，ROS を Wasm 化したことでライブマイグレーション後のファイルサイズのオーバーヘッドが増大し，ノードの実行時間が大幅に増えてしまう問題が残った．柿本ら [5] は，組込みデバイス向けの軽量な ROS ランタイム実装である mROS 2-POSIX を採用し，ROS ランタイムの Wasm 化にともなうオーバヘッド増加に対処した．しかし，mROS 2-POSIX はネットワークスループットのマイクロベンチマークに留まっている [6]．また，柿本らによって実現した mROS 2-POSIX を Wasm 化した mROS 2-Wasm も実際のアプリケーションでは評価されていない [5]．そのため，ライブマイグレーション後のオーバーヘッド増加を解決するロボットソフトウェア基盤として，アプリケーションが複雑化した場合の動作が明らかでない．

本研究では，ノードの動的マイグレーションに向けた mROS 2-POSIX と mROS 2-Wasm の性能評価を行い，ネイティブ ROS と比べて動的配置機構実現後のロボットソフトウェア基盤として優位性を明らかにすることを目指している．アプローチとして，動的配置機構を実現するのに適したアプリケーションを mROS 2-POSIX，mROS 2-Wasm，ネイティブ ROS で実装し，通信時間とメモリ消費量を比較評価する．ネイティブ ROS にはラズパイマウス [22] と呼ばれる 2 つの車輪を回転することで動作するロボットのライントレース実装

があり，本実装として，mROS 2-POSIX と mROS 2-Wasm にそのアプリケーションを移植する．このライントレースノードは，組込みデバイス上で動作し，主な機能は Pub/Sub 通信を使用しているため，本研究の実装するアプリケーションとして適しているうちの一つである．このライントレースノードの実装により，mROS 2-POSIX と mROS 2-Wasm のロボットソフトウェア基盤としての拡張性を示すことができ，評価によって，実アプリケーション上で mROS 2-POSIX と mROS 2-Wasm の軽量性を示す．

1.4 章構成

本評価では，ライントレースノードを各環境で動作させ，いくつかのトピックに対して，Pub/Sub 通信のにかかる時間を通信性能とし，計測を行った．また，各環境で RSS（Resident Set Size 物理消費メモリ）を測定するために実行時のプロセス ID を取得し，そのプロセスに割り当てられている RSS を計測した．この結果を実験結果として本論文に示す．

本論文は，全 7 章から構成されている．第 1 章は，本研究における背景と課題，目的について述べた．第 2 章は，ROS や mROS 2-POSIX，WebAssembly について述べる．第 3 章では，Wasm 仮想マシンを用いた mROS 2-POSIX 環境である mROS 2-Wasm に関する先行研究について述べる．第 4 章では，本研究のライントレースノードを各環境の実装について説明する．第 5 章では，実験結果を示し，結果から得られた知見から考察を述べる．第 6 章では，関連研究について述べる．第 7 章では，本研究のまとめと今後の課題について述べる．

第 2 章

関連技術

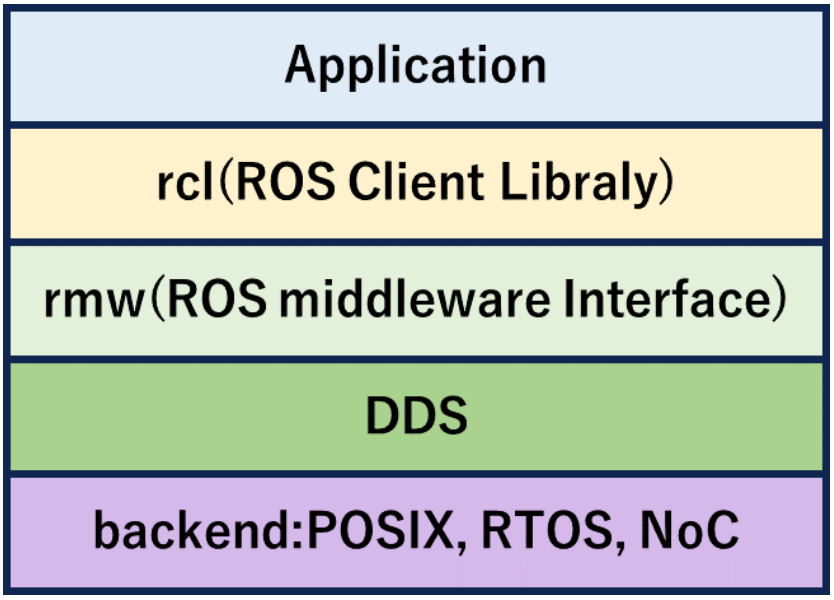

図 2.1: ROS 2 の構成図

表 2.1: QoS の影響

Publisher	Subscription	適合性
Best effort	Best effort	Yes
Best effort	Reliable	No
Reliable	Best effort	Yes
Reliable	Reliable	Yes

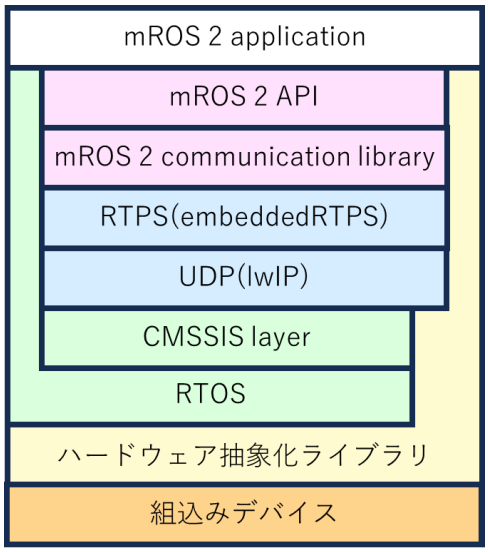

図 2.2: mROS 2 の内部構成

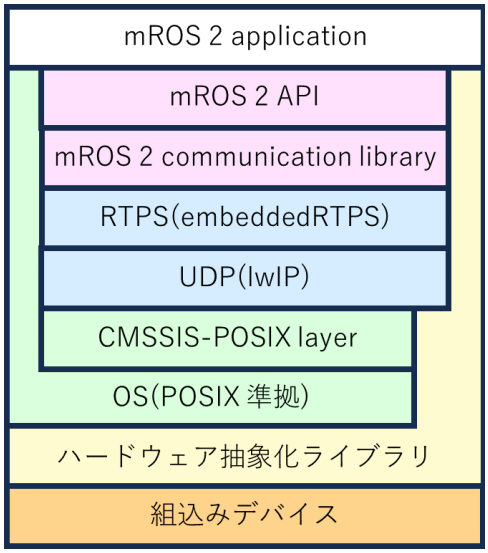

図 2.3: mROS 2-POSIX の内部構成

2.1 ROS

ロボットシステムを開発するにあたって現在は ROS 2 が主流であるが，その前身である ROS は，スタンフォード人工知能研究所の研究プロジェクトとから移管された Willow Garage 社によって開発が始まったロボットソフトウェア開発基盤である．最初の正式なディストリビューション版は, 2010 年 3 月にリリースされた Box Turtle で, その後, Fuerte, Groovy, Hydro, Indigo, Jade, Kinetic, Lunar, Melodic, Noetic, Foxy, Galactic, Humble といったバージョンがリリースされている．ROS の利点は，分散型ロボットシステムの実現に適切である通信ミドルウェアの実装や RTPS（Real Time Publish Subscribe）通信プロトコル，プロジェクト管理やデバックおよびシミュレーションなどのための広範囲なツール群，豊富な OSS（Open Source Software）のパッケージ（ソフトウェアのビルド，実行，共有を容易にするために設計されたフォルダ構造とファイルの集合）やライブラリ（特定の機能やアルゴリズムを実装するプログラムコードの集合)，世界規模の活発なオープンソース開発コミュニティという 4 つの側面にある [8].

まず，通信ミドルウェアの実装や RTPS 通信プロトコルによって，開発者が複雑なネットワーク通信の詳細を把握することなく分散型システムの構築に集中できるようになり，リアルタイム性が要求されるアプリケーションの性能を向上させることができる．

次に，プロジェクト管理やデバックおよびシミュレーションなどのための広範囲なツール群によって，ロボットシステムの開発速度と品質を向上させることができる．

また，豊富な OSS のパッケージやライブラリによって，ロボットシステムの開発速度を加

速させ，コストを削減することができる．

最後に，世界規模の活発なオープンソース開発コミュニティによって，ロボットシステムの開発における技術的な課題を解決するための情報を素早く得ることができる．これら 4 つの利点によって，ROS は様々な場所で利用されており，ロボットシステムの開発においては，ROS を利用することが一般的である．

パッケージやライブラリに関しては公式のものだけでも多種多様であり, ロボット本体の制御, モーターの制御, センサーの制御, 画像処理, 音声認識, 自律走行, SLAM (Simultaneous Localization and Mapping）など，ロボットシステムに必要な機能を網羅している．短期間で機能の大枠を組み立てることができるため，研究開発，教育，産業用途においても，幅広い分野で利用されている．

ROS はすでに 10 年以上の歴史を持っており，ロボット工学の発展に大きく貢献してきた．日本の企業ではソニーの aibo の事例が ROS を使った製品として代表的であり，他の企業でも商用商品に採用されることも多い．ロボット開発を取り巻く環境や ROS が研究用から商用にも活用され始めるという変遷を受け，2014 年より第 2 世代バージョンである ROS 2 の開発が始まった．これは ROS 1 が研究用途において，十分な性能を持っていたが，商用製品においてはリアルタイム性やセキュリティの問題があった．ROS 1 の設計は以下のようにされている．

- 単一のロボットで動作することを想定
- 多くのリソースを抱えているマシン上で動作されることを想定
- リアルタイム性は考慮しない
- 優れたネットワーク上で動作されることを想定（有線接続など）
- 研究，主に学術的なアプリケーションを想定
- 規制や禁止事項がなく，最大限の柔軟性がある（プログラムが main() から始まることを決定していないなど）

そのため，リアルタイム性を要求する産業用やリソースの少ないマシンに利用されることが増えるとこの設計が問題になることが多かった．

また，ROS 1 では利用が広まるにつれて，以下の課題が浮上してきた．

1. メッセージ通信の仲介役として ROS Master が必要である．ROS Master が何らかの原因でダウンすると，全てのプロセスがダウンしてしまうという問題がある
2. リアルタイム性を考慮していない設計になっている
3. メッセージを暗号化せずに送受信を行っているため，セキュリティに問題がある
4. 実行するマシンにリソースが少ないと動作が不安定になるという問題がある
5. Ubuntu（Linux）でないと使いにくい

そのため ROS 2 では，上記の問題を解決するため，リアルタイムおよび組込みシステム向け

の DDS（Data Distribution Service）[7] と呼ばれる通信ミドルウェアを採用し，通信の信頼性を確保するための QoS（Quality of Service）制御の機能を導入した．

ROS のミドルウェア実装である DDS[?] の採用により，(1) の問題点がなくなり，QoS と DDS の組み合わせにより，(2) に対してリアルタイム性の実現を目指した．(3) のセキュリティの問題は ROS 2 でメッセージの暗号化が施されたことにより解決した．また，(4) に対応するためにリソースの少ないマシンで動作できるように開発されている．さらに，(5) の Ubuntu 以外の OS にも対応した．

ROS 2 の構成を図 2.1 に示す．ROS 2 ではユーザーが実装するアプリケーション層があり，その下にクライアントライブラリである rcl の層がある．rcl クライアントライブラリ層には通信概念を公開する API の定義がされており，DDS の上に構築された．rcl は，C++, Python といった各プログラミング言語用のクライアントライブラリに共通する機能を提供している API である．その下部に rmw（ROS Middleware Interface）と DDS がある．rmw は様々な種類の DDS 実装，各 backend の差を吸収，最適化を施している層であり，rcl クライアントライブラリに対して共通のインタフェースを提供している．DDS はリアルタイム通信を行うための API とプロトコルを提供している国際ミドルウェア標準 [24] に基づいて実装された層である．RTPS は DDS から呼び出される transport 層に位置する通信プロトコルである．backend として，RTOS（Real Time Operating System）や POSIX などの ROS 2 は様々な環境で動作するためを表す層がある．

2015 年 8 月には最初のディストリビューションであるアルファ版がリリースされ，2017 年の 12 月に ROS 2 が本リリースされた．2023 年の Metrics Report[23] によると，ROS は 550365601 回ダウンロードされている．そのうち 30% が ROS のディストリビューションである Noetic で，32% が ROS 2 のディストリビューションである humble である．2023 年で ROS は Noetic，ROS 2 は humble が主流のディストリビューションであった．本研究は ubuntu 22.04 でリリースされた ROS 2 の humble ディストリビューションを使用している．

2.1.1 Publisher と Subscriber

ROS では基本的なノード間のデータ通信として Publish/Subscribe（Pub/Sub）通信型の非同期な通信プロトコルを採用している．データの送信側を Publisher（出版者）とよび，受信側を Subscriber（購読者）と呼ぶ．通信経路として，トピック（Topic）を介した通信が行われる．トピックで送受信されるデータはメッセージ（Message）と呼ばれ，車輪の角速度や回転量，現在位置の 3 次元座標など，基本型を組合せた任意の型を定義することができる．同じ名前のトピックに対して，様々な個数や種類のノードが任意のタイミングで登録，変更，削除ができる．Publisher 側は自由なタイミングで Topic に向けて通信を行い，非同期に動作する Subscriber 側は，Publisher からメッセージを受け取った際に対応するコールバック関数が実行される．

ノードは Publisher や Subscriber，またその両方をプログラム次第で変化できるため，ユーザが考えたオリジナルのノードを作成することができる．この仕様のため，ノード同士の依存が少なくなり，ロボットシステム全体の機能の追加や削除が容易である特性は，柔軟なロボットシステムの構築の一助となる．

2.1.2 Service 通信と Action 通信

ROS 2 では, Publisher と Subscriber 以外にも Service 通信と Action 通信がある．Service 通信は，サーバーとクライアントの 2 つのノード間でリクエストとレスポンスをやり取りする通信である．これは既存のクライアント・サーバーモデル [15] によく似ており，Subscribe するのではなく，ノードがメッセージの値を欲しいタイミングでリクエストする仕組みになっている．そのため，ロボットに対する命令やデータの取得，計算結果を受け渡しに適しており，Service 通信はリアルタイム性や連続的なデータには向かないが，確実に応答するというロボットシステムにおいて重要な役割を果たす．さらに，Service 通信は同期的な通信であるため，Service 通信を行うノードは，Service 通信が完了するまで他の処理を行うことができない．そのため，クライアントの処理を間接的にブロックすることができる．Service 通信はパラメータ設定や取得に使用されることが多い．例として，ロボットの速度やセンサー感度の設定に使用される．他にも，ロボットの状態確認や計算や処理のリクエストなどに用いられる．

一方，Action 通信は，Service 通信と同様にサーバーとクライアントの 2 つのノード間でリクエストとレスポンスをやり取りする通信である．しかし，Action 通信は Service 通信と異なり，リクエストに対するレスポンスを即座に返すのではなく，リクエストに対するレスポンスを返すまでの間に，進捗状況を返すことができる．この通信方式によって，長時間のタスクやフィードバックが可能なタスク，中断可能なタスクに適しているため，開発者は柔軟性を保ちながら，ロボットシステムを構築することができる．長時間のタスクの例として，地図の作成や長距離のナビゲーションなどがある．他にも中断可能なタスクの例として，ロボットが掃除をしている最中に，別のエリアを掃除するよう指示を変更するなどがある．さらに，Action 通信は非同期的な通信であるため，Action 通信を行うノードは，Action 通信が完了するまで他の処理を行うことができる．これによって，複雑なタスクを行うノードでも Action 通信を用いることで同時に処理することが可能であり，多くのリソースがあるマシンで複数の Publisher や Subscriber，Action，Service 通信を抱える高度なノードを動作させることができる．

2.1.3 QoS

QoS は，通信の信頼性や遅延時間，帯域幅などの通信品質を制御するための機能である．ROS 2 には QoS の設定として，データの到達を保証する Reliability，多少のデータの欠損

を許容する Best_Effort があり，ROS 2 のノードにはデフォルトで Reliability が設定されている．ノード間の通信において，QoS による影響を表 2.1 に示す．Best_Effort の Publisher と Best_Effort の Subscriber の場合，Subscriber は Publisher からデータを受信できる．同様に Reliable の Publisher と Reliable の Subscriber の場合，Subscriber は Publisher からデータを受信できる．また，Publisher の QoS 設定が Best_Effort，Subscriber の QoS 設定が Reliable の場合，Subscriber は Publisher からデータを受信できない．Reliable の Publisher と Best_Effort の Subscriber の場合，Subscriber は Publisher からデータを受信できる．このように，Subscriber の QoS 設定によって，通信ができるかどうかが変わる．

2.2 mROS 2-POSIX

mROS 2-POSIX とは ROS 2 の軽量実行環境である mROS 2 を POSIX に準拠した OS でも使用できることを目的に高瀬らによって作成された POSIX 対応済みの mROS 2 である [6]．mROS 2-POSIX は mROS 2 同様，ROS 2 を採用するロボットシステムにおいて通信方式とメモリ軽量な実行環境を確立することができる組み込み技術を導入することにより，分散型のロボットシステムにおける応答性やリアルタイム性の向上，消費電力の削減が可能になる．mROS 2-POSIX に実装されている DDS の embeddedRTPS[17] は組込み向けの軽量な DDS であり，ROS 2 のデフォルトの DDS として利用されている FastRTPS[] と通信ができる．embeddedRTPS は，Pub/Sub 通信の相手と経路を自律的に探索できるように設計され，ROS 2 および RTPS の利点が，組み込み技術導入時に損なわれないようになっている．

mROS 2 と mROS 2-POSIX の内部構成の違いを図 2.2 と図 2.3 に示す．ユーザーアプリケーションからの階層順で，mROS 2 通信ライブラリ，通信プロトコルスタック，RTOS，ハードウェア抽象化ライブラリによって構成される．mROS 2-POSIX API 層および通信ライブラリ層は，メッセージを非同期に Publish や Subscribe するためのコミュニケーションチャネルである ROS 2 の Topic に相当する API および通信機能を提供する階層である．本階層は，ROS 2 のネイティブなクライアント通信ライブラリである rclcpp と互換性を保つように設計されている．mROS 2 通信ライブラリは，ユーザアプリケーションに対して，ROS 2 通信のトピックに関する基本的な API を提供している．主な API として void mros2::init()，void mros2::Node::create_node()，void mros2::Publisher::create_publisher(), void mros2::Subscriber::create_subscriber() がある．利用可能な機能は制限されているものの，組込み技術を導入する ROS 2 開発者は，汎用 OS 向けのプログラミングスタイルを踏襲しながら C++ によって mROS 2 のアプリケーションを実装できる．その制限として，mROS 2-POSIX は Service 通信や Action 通信には対応していない．通信プロトコルスタックには，前述の embeddedRTPS を採用している．UDP については組込み向けの C による軽量実装である lwIP（Light Weight IP）[18] が採用され

ている．RTOS には，非常に短い間隔で正確な時間測定を可能にする高分解能タイマ（High Resolution Timer）やシステムが待機状態のときにエネルギー消費を削減できるティックレス（Tickless）の低消費電力な処理遅延機能など，高いリアルタイム性と安全性が求められる軽量な組込みシステムに適した設計がなされている．

2.3 WebAssembly

WebAssembly（Wasm）は，C，C++，C#，Rust などの言語で書かれたプログラムをコンパイルのターゲット Web 上でプログラムを高速に実行するために設計された，スタックベースの仮想マシンで実行される仮想命令アーキテクチャである [24][27]．サンドボックスな環境でアプリケーションは実行されるため，ハードウェアや言語，プラットフォームに依存せず，ネイティブに近い実行速度でコードを実行できるという性質がある [24]．Wasm の性質から Web ブラウザ以外の実行環境として利用する取り組みがあり，Web の外部で Wasm を実行するための API 群として WASI（WebAssembly System Interface）[25] が提案された．WASI を用いることでファイルやディレクトリ，ネットワークソケットなど，様々なリソースに Wasm からアクセスできるようになり，Web ブラウザ以外の実行環境として動作する Wasm ランタイムの開発が進んだ．

2.3.1 WAMR

WebAssembly の実行環境として WAMR（WebAssembly Micro Runtime）[26] がある．WAMR は，組込みや IoT，クラウドなど，様々なプラットフォームで動くようにメモリ消費量が小さくなるよう設計された，オープンソースの Wasm ランタイムである．Wasm プログラムの実行方式としては Wasm バイナリを逐次実行するインタプリタ方式と（Classic），Classic インタプリタを基礎としながらも最適化技術を取り入れ，実行速度を高速にした Fast インタプリタと，事前に Wasm バイナリをネイティブバイナリにコンパイルして実行する AoT（Ahead Of Time），Wasm バイナリを実行時にネイティブバイナリにコンパイルして実行する JIT（Just In Time）があり，そのうち AoT と JIT ではネイティブと同等の実行速度で動作する．また，マルチスレッドやスレッド管理を行う pthread API（POSIX スレッドの標準 API）をサポートする組込みライブラリや Socket API をサポートする組込みライブラリも提供されている．

第 3 章

先行研究

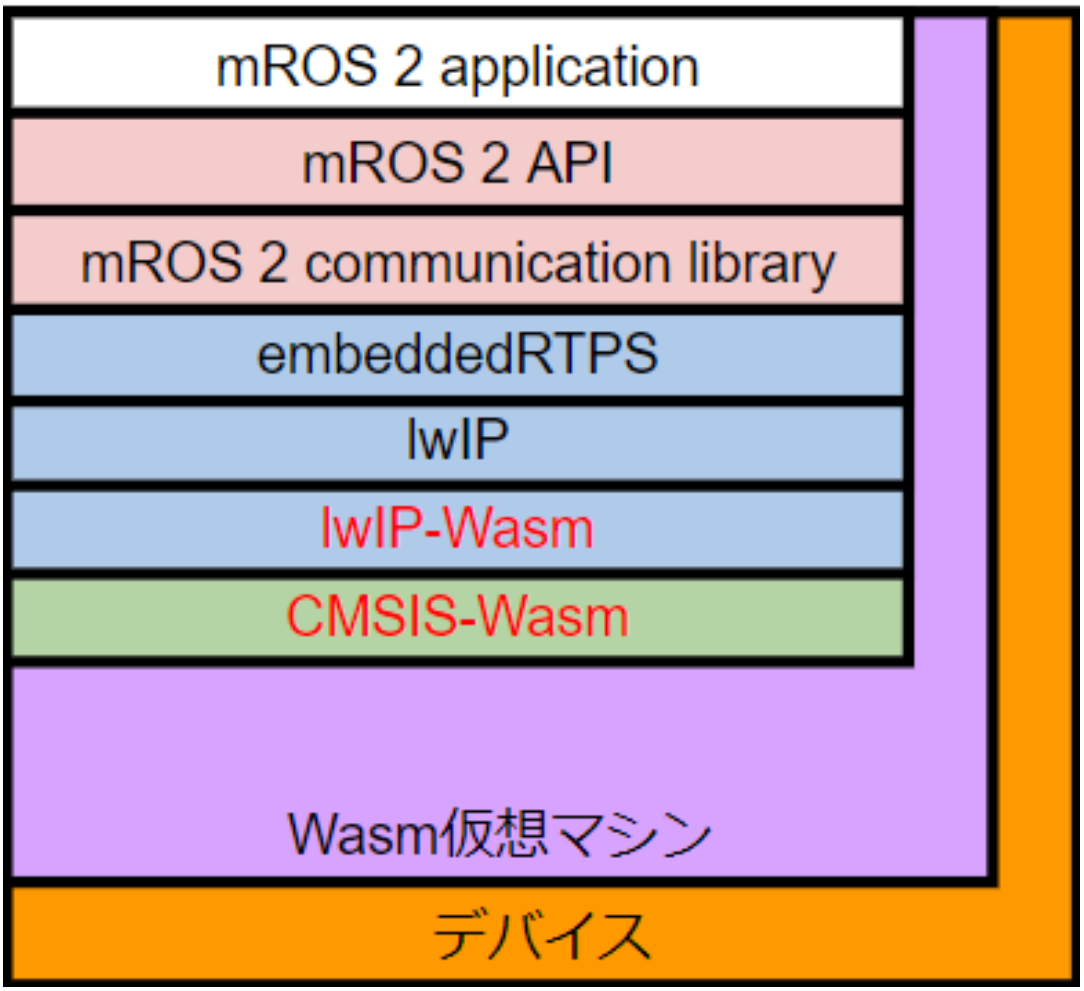

図 3.1: mROS 2-Wasm の構成図

柿本らによって，mros2-POSIX を wasm 環境で実行する mROS 2-Wasm が提案された．

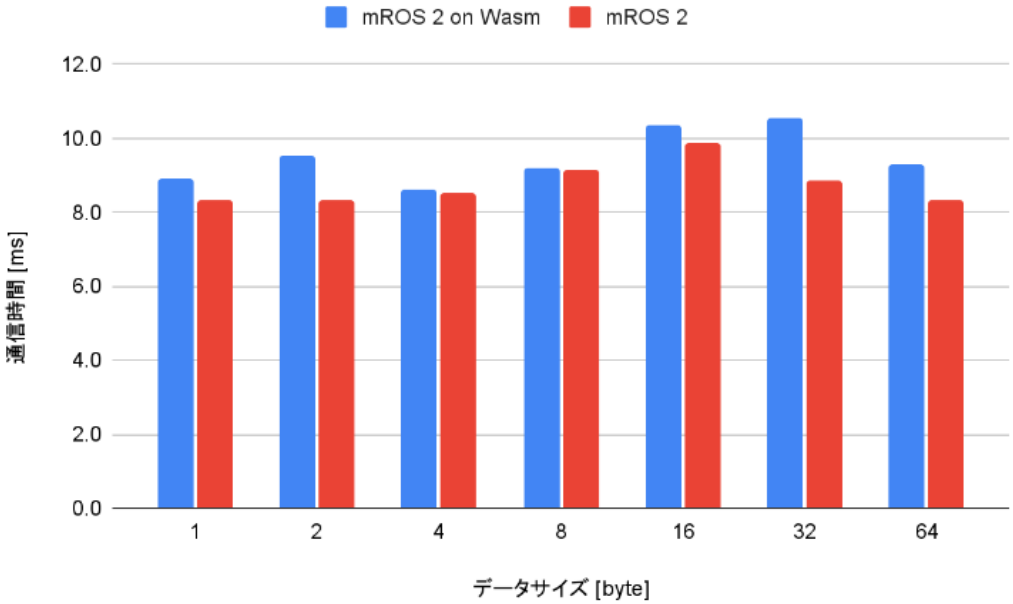

図 3.2: mROS 2-Wasm の通信性能

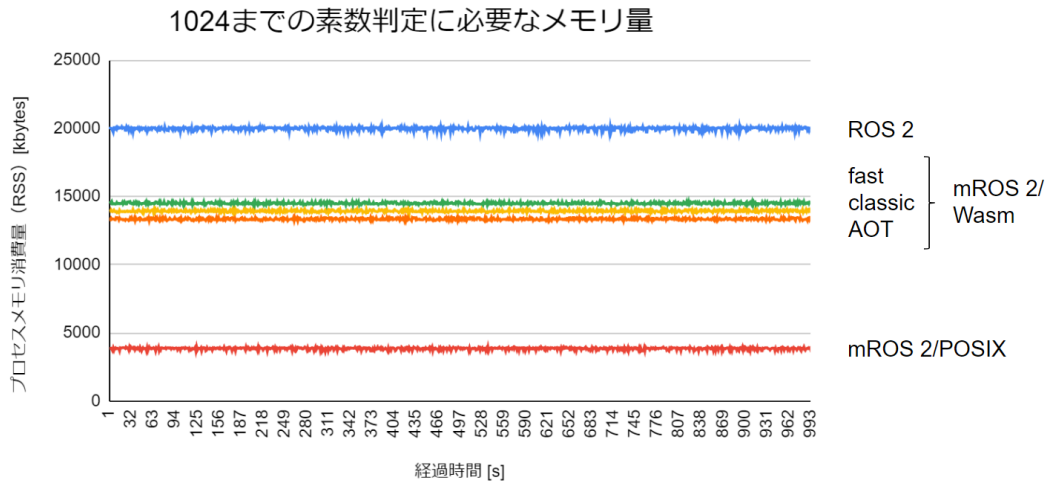

図 3.3: mROS 2-Wasm のメモリ使用量

[5]

mROS 2 を Wasm 化させるにともない，使用する Wasm ランタイムに関して以下の制約がある．

- ROS ランタイムにはスレッド操作やネットワーク通信の処理が必要であるため，Wasm ランタイムにはこれらの機能が必要である

- リソースの限られているエッジデバイスで動作させることを想定する必要があるため，Wasm ランタイムと ROS ランタイムによるリソース消費を最小限に抑える必要がある

Wasm はサンドボックスな環境で実行されるため，OS の機能に依存した層がある mROS 2-POSIX をそのままではコンパイルすることができない．図 2.3 で示したとおり，mROS 2-POSIX で OS に依存している層は CMSIS-POSIX と lwIP-POSIX である．CMSIS-POSIX は mROS 2 内部で RTPS 通信を行うための機能として，スレッド管理機能，排他制御機能，メッセージキュー管理機能，時間管理機能が pthread（プログラム内で複数の実行スレッドを作成・制御するための API を提供）などを用いて実装されている．lwIP では，UDP マルチキャストをおこなための実装が Socket や CMSIS-POSIX が提供する機能を用いて実装されている．

これらの依存を解消するため，CMSIS-POSIX を Wasm 対応した CMSIS-WASM，lwIP-POSIX を Wasm 対応した lwIP-WASM を柿本らは実装した．mROS2-wasm の構成を図 3.1 に示す．なお，mROS 2 はオープンソースソフトウェアであることから上位レイヤに変更が加わっても変更を取り込みやすいよう，既存のビルドシステムに極力手を加えずにシステムを構築されている．

CMSIS-WASM を実装する機能のうち，排他制御機能，メッセージキュー管理機能はスレッド管理機能に依存している．時間管理機能は手を加えることなく Wasm コンパイルが可能であったため，実際に実装されたのはスレッド管理機能である．実装に際して，WAMR の pthread API を用いてスレッド管理機能を実装されているが WAMR pthread ライブラリの既知の問題が障害となった．以下にその問題を述べる．

- timespec 構造体をサポートしていない
- wasi-sysroot の errno と互換性がない
- pthread_attr_t 構造体をサポートしていない

timespec 構造体は，スレッドの同期処理などに使われる pthread_cond_timedwait 関数から使用され，待機時間の長さを指定するために使われる．WAMR では timespec 構造体が使われず，useconds を用いる必要があった．そのため待機死体時間をマイクロ秒単位に変換し引数として WAMR の pthread_cond_timewait に渡すことで timespec 構造体を用いずに同様の動作をさせた．

errno に関してはシステムが正常に動作している際に使われることがないため，仕様が避けられている．

スレッド生成時にそのスレッドの属性を設定するのに使われる pthread_attr_t 構造体は，WAMR ではサポートされていない．しかし，pthread_attr_t はスレッドを生成する関数のみで使用されており，デフォルトの属性から変更されずにスレッド適用されていたため，削

除されている．CMSIS-WASM は上記のように柿本らによって実装された．

lwIP-WASM は，Socket を用いて通信機能の実装が行われているため，WASI を用いる必要がある．WAMR には WASI を用いて実装された Socket API が提供されているため，それを用いて実装が行われた．

WAMR の Socket API では，setsockopt 関数において IPPROTO_IP レベルで設定できるオプションは 5 つに限られている．以下に示すのがそのオプションである．

- IP_MULTICAST_LOOP
- IP_ADD_MEMBERSHIP
- IP_DROP_MEMBERSHIP
- IP_TTL
- IP_MULTICAST_TTL

一方, mROS 2-POSIX の lwIP-POSIX では, IP_MULTICAST_IF, IP_ADD_MEMBERSHIP, IP_MULTICAST_TTL が使用されており，IP_MULTICAST_IF が WAMR の Socket API には存在しないためネットワークインターフェースの指定ができないという問題があった．Socket は IP_MULTICAST_IF によるネットワークインターフェースの指定がない場合はデフォルトのネットワークインターフェースが指定されるため，IP_MULTICAST_IF を設定する setsockopt 関数を削除して実装が行われた．また，マルチキャスト通信の配信範囲を制御する TTL を設定する IP_MULTICAST_TTL の指定時に，WAMR の Socket API ではオプションデータの長さが 4bytes でないときプログラムが終了してしまう．lwIP-POSIX ではこの長さが 1bytes で定義されていたため 4bytes に変更した．このように WAMR の Socket API に合わせた細かい変更を行うことで，lwIP-WASM が実装された．システムのビルドは，外部のプロジェクトのビルド・インストールなどを可能にする ExternalProject を用いて mROS 2 のディレクトリ外から行うことができる．以上の実装により，実行中の mROS 2-POSIX を Wasm 化することができた．

mROS 2-Wasm は通信時間とメモリサイズと計算処理性能の評価が行われた．

計算処理性能の評価では，ノード内で擬似作業として 1 以上の整数を順に素数判定し，1024 までの素数を見つける処理を行う実装をし，その時間を計測，評価した．コンパイル方式は AoT と Classic インタプリタと Fast インタプリタを使って計測した．

通信時間では mROS 2-Wasm と mROS 2 を比較評価した．その結果を図 3.2 に示す．計測方法はクラウド想定のデバイスから文字列データを送信し，ロボット想定のデバイスからデータを受けとると，そのままデータを返し，クラウド想定のデバイスが受け取るまでの時間，RTT（Round Trip Time）を計測している．mROS 2-Wasm のコンパイル方式は AoT と JIT を使わず Classic インタプリタでコンパイルしたバイナリファイルを使って計測した．

メモリサイズの評価では，mROS 2-POSIX と mROS 2-Wasm と mROS 2-POSIX を比

較評価している．その結果を図 3.3 に示す．この計測では，計算処理性能で実装された処理のプロセス ID の RSS（Resident Set Size）と VSS（Virtual Set Size）を取得し評価をした．今後の課題として，mROS 2-Wasm に実行状態の保存，復元機構を実装することが残った．

第 4 章

実装

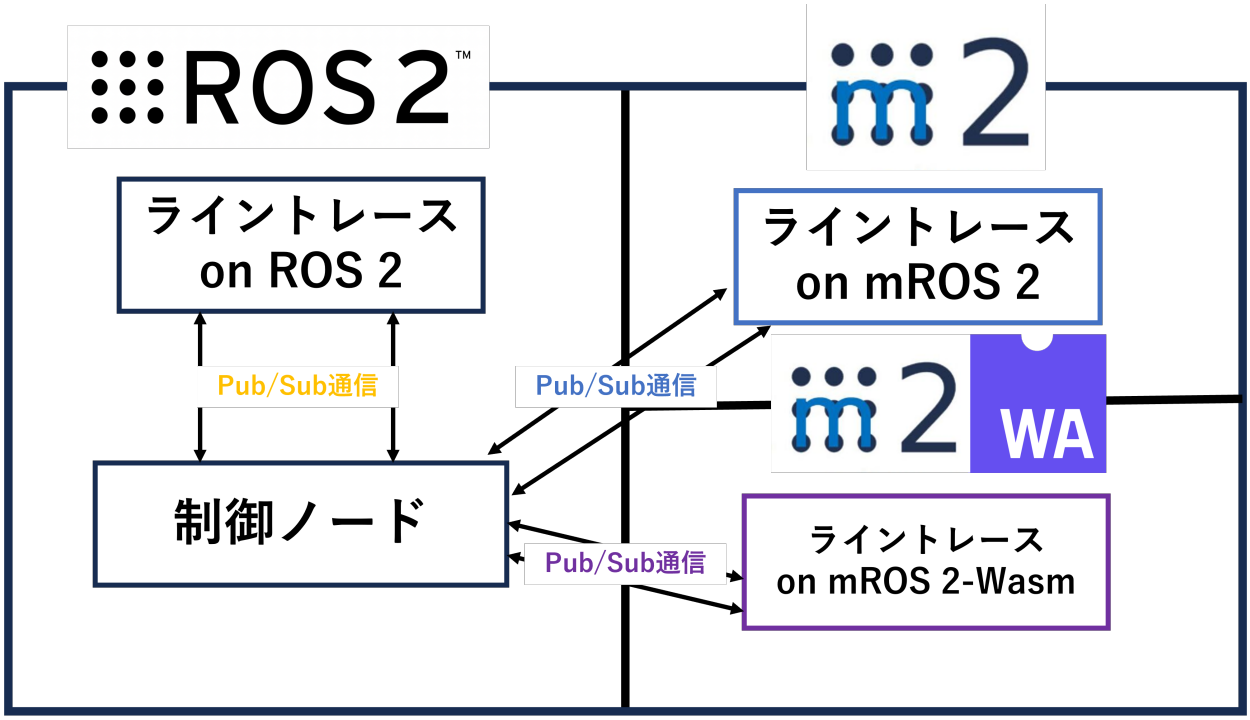

図 4.1: 実装アプリケーション構成

本章では，mROS 2-POSIX と ROS 2 の性能を比較評価するにあたって，実装するアプリケーションの概要について説明する．

4.1 実装要件

mROS 2-POSIX と mROS 2-Wasm と ROS 2 を比較評価するためのアプローチとして ROS 2 で実際に動作するアプリケーションが望ましい．第 1 章で述べた通り，mROS

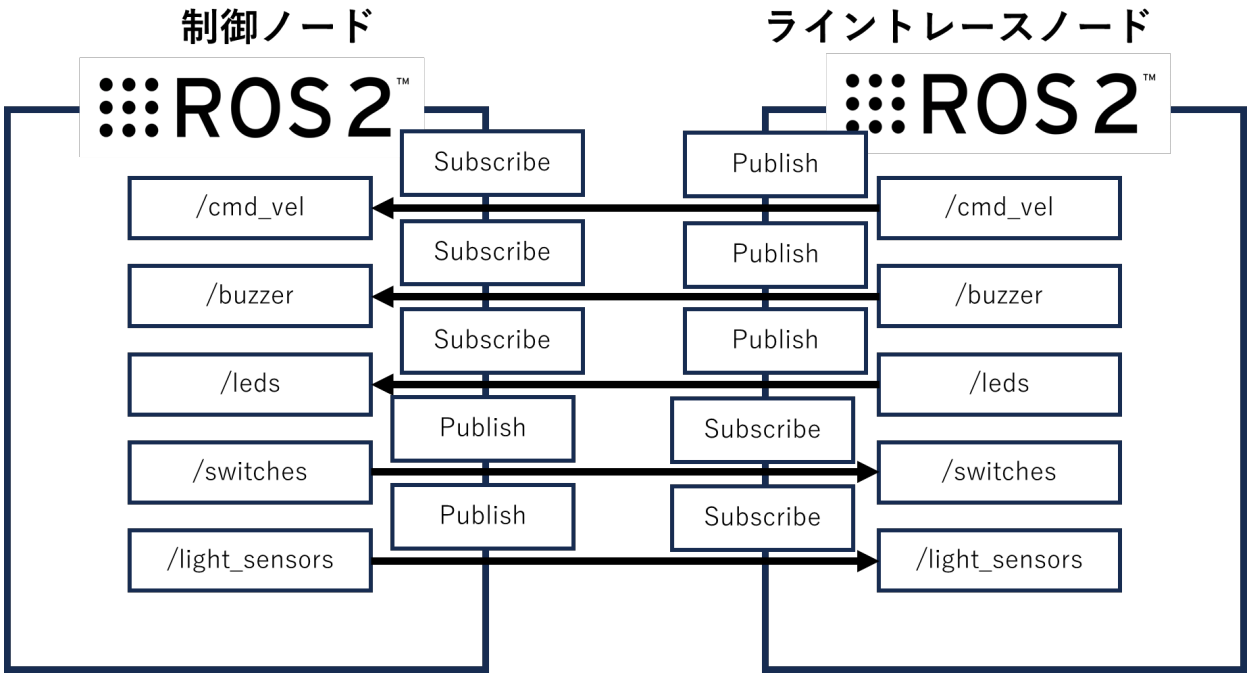

図 4.2: 実装アプリケーションの Pub/Sub 通信構成

2-POSIX の評価はネットワークスループットのマイクロベンチマークに留まり，実アプリケーションにおける有用性が十分に評価されていない．そのため，mROS 2-POSIX と mROS 2-Wasm と ROS 2 に以下の条件を満たしたアプリケーションを実装し，比較評価を行う．実装するアプリケーションの要件として，Pub-Sub 通信を主な機能として使用しているアプリケーションであること，組み込みデバイス上で動作できるアプリケーションであること，OS への依存が少ないアプリケーションであることの 3 つを要件とした．まず，ROS 2 には Pub/Sub 通信や Service 通信，Action 通信があるが，mROS 2-POSIX と mROS 2-POSIX を基盤とした mROS 2-Wasm には，Pub/Sub 通信のみの実装になっている．これは mROS 2-POSIX で使用されている embeddedRTPS が組込み向けの通信プロトコルであり，軽量化を図るため Pub/Sub 通信のみの実装になっているからである．そのため，ROS 2 のアプリケーションを mROS 2-POSIX と mROS 2-Wasm に移植する際には，Pub/Sub 通信を主な通信手段としているアプリケーションが望ましい．

次に，本研究は第 1 章で述べた通り，動的配置機構実現後のロボットソフトウェア基盤としてどのような優位性があるのか明らかにすることを目指している．したがって，実装するアプリケーションは動的配置機構が適用される Rassberry Pi や Jettson nano のような組込みデバイス上で動作できるアプリケーションが望ましい．

そして，第 1 章で述べた動的配置機構は，OS への依存が少ないアプリケーションが望ましいと考える．これは，OS の環境に依存したアプリケーションであればあるほど，マイグ

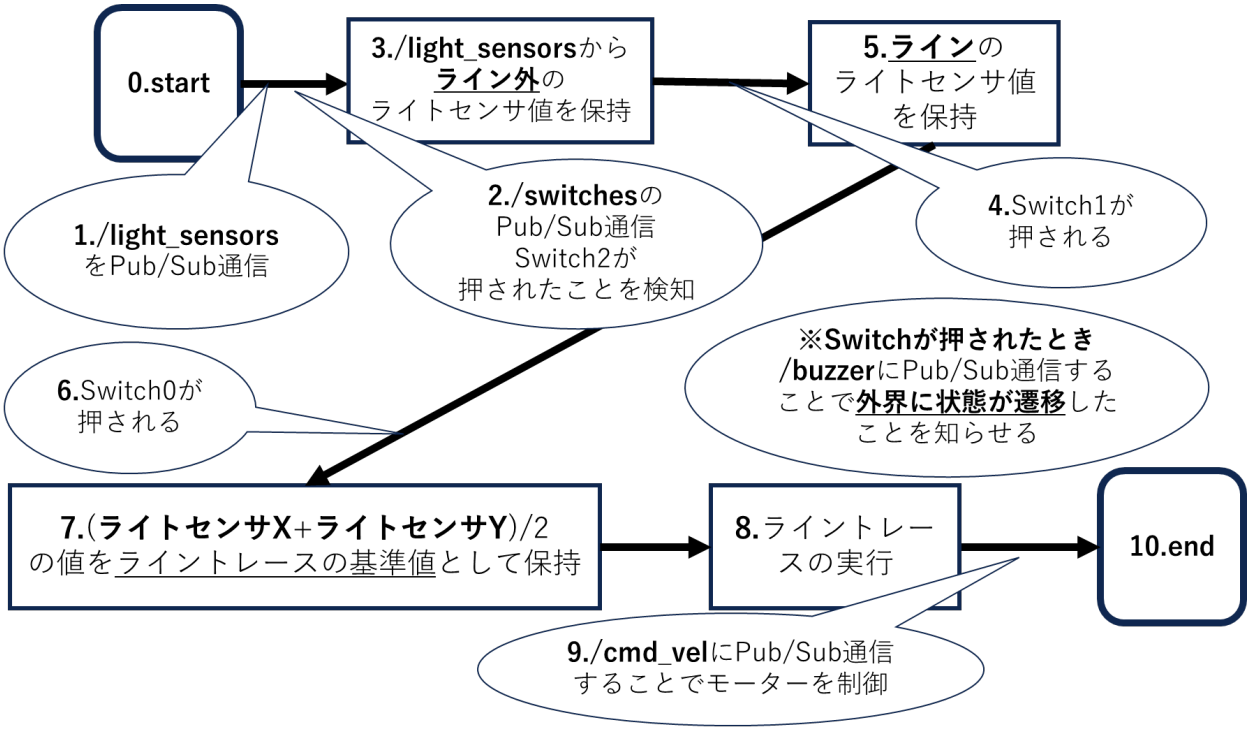

図 4.3: 実装アプリケーションの動作フロー

レーション後の環境で正常に動作しない可能性が大きくなるためである．

以上の条件を満たすアプリケーションとして，本研究ではラズパイマウスで動作する ROS 2 のライントレースノードを mROS 2-POSIX と mROS 2-Wasm に移植し，比較評価を行う．

ライントレースとは，ロボットや自動車などの自動制御システムが，地面に引かれた線に沿って移動する技術のことである．主に，教育や趣味，競技の分野で用いられる技術で，センサーを使って線を認識し，それに従って機体を制御する．この実現には，光センサや赤外線センサを使って線の色や反射特性を検出し，そのその情報に基づいてモーターを制御して進路を調整する．主に線が白色で背景が黒，もしくはその逆が一般的である．

このライントレースはロボティクスや自立走行車において基本的な技術であり，将来的により大きなアプリケーションに発展できる可能性がある．そのため，評価だけでなく mROS 2-POSIX と mROS 2-Wasm のロボットソフトウェア基盤として拡張性示すことができると考え，ラズパイマウスのライントレースノードを各環境に実装することとした．

4.2 実装アプリケーションの構成

本研究では ROS 2 で動作するライントレースノードを mROS 2-POSIX に移植し，mROS 2-POSIX と mROS 2-wasm，ROS 2 の性能を比較評価する．比較評価のためのアプリケー

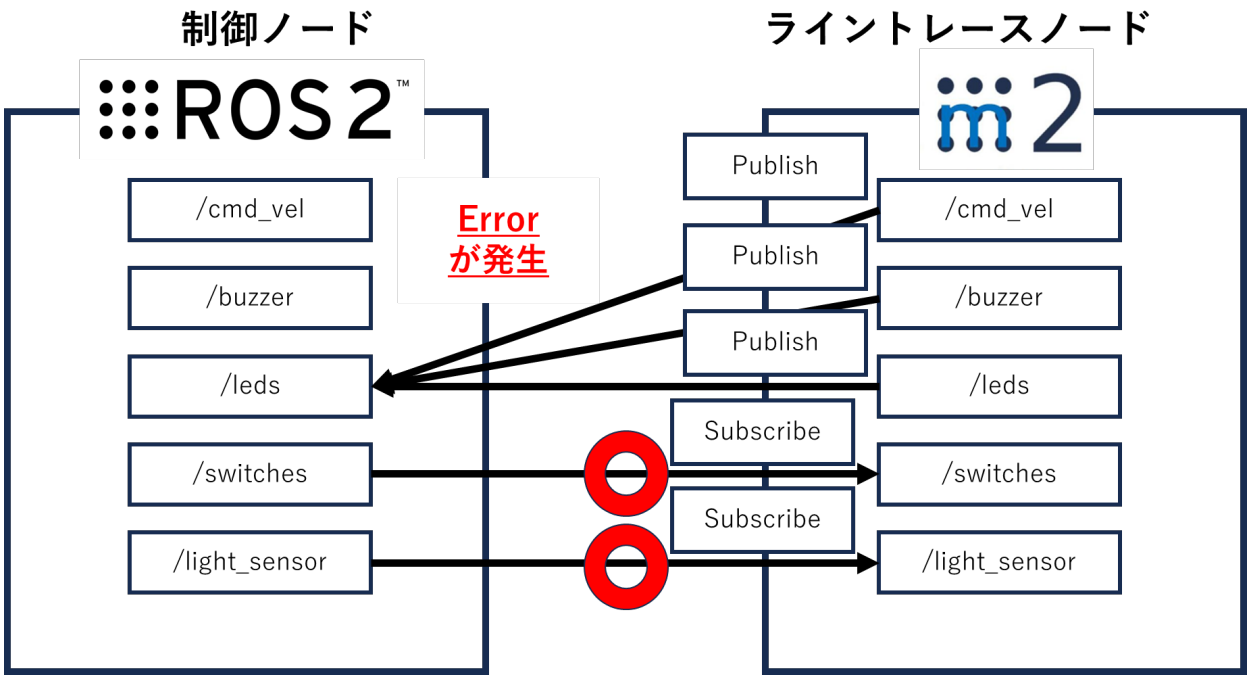

図 4.4: mROS 2-POSIX のバグ

ションの構成図を図 4.1，図 4.2 に示す．

図 4.1 は mROS 2-POSIX と mROS 2-wasm，ROS 2 の実装アプリケーションの構成である．ラズパイマウスのライントレースアプリケーションは，実際にラズパイマウスを制御する制御ノードと，ライントレースの実装ノードの 2 つに分かれている．制御ノードはラズパイマウスのデバイスファイルにアクセスし，モータを回転させたり，LED や Switch，ライトセンサの値を取得する．ライントレースノードは，制御ノードから受け取った LED，Switch，ライトセンサの値を駆使してモータ制御の命令を制御ノードに向けて通信する．そのため，制御ノードは OS の依存が大きく，ライントレースノードは OS の依存が少ない．第 4 章で述べた通り，マイグレーションは OS 依存があるアプリケーションに適用すると，適用後に動作しない可能性がある．したがって，OS に依存しないノードであるライントレースノードを mROS 2-POSIX と mROS 2-wasm それぞれに実装した．

図 4.2 はライントレースノードとその制御ノードの Pub/Sub 通信についての構成図である．ライントレースノードと制御ノードは相互に 5 つのトピックを介して通信を行う．以下に通信を行う 5 つのトピックを示す．

- /cmd_vel
- /buzzer
- /leds
- /switches

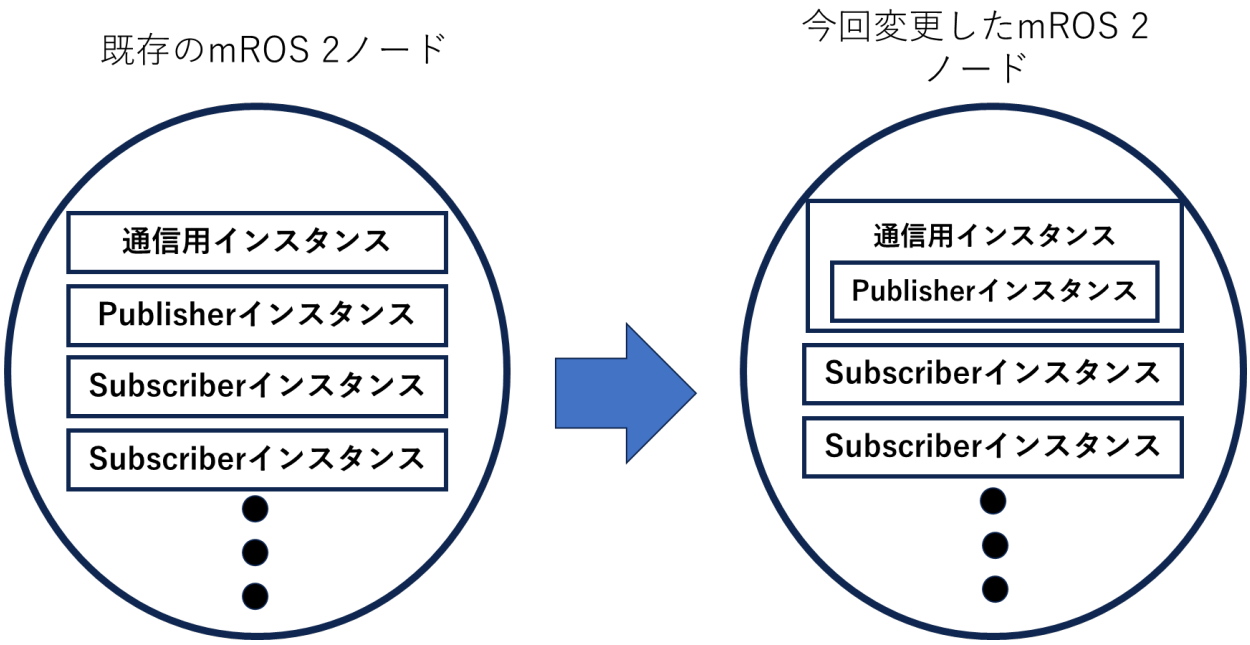

図 4.5: mROS 2-POSIX のバグ fix

- /light_sensors

/light_sensors と/switches はライントレースノードが Subscribe するトピックであり，/cmd_vel，/buzzer，/leds はライントレースノードが Publish するトピックである．/light_sensors は制御ノードから常に Publish されるライトセンサの値で，光が反射しやすい白い部分をライトセンサに近づけると取得する整数値が大きくなり，光を吸収する黒に近い部分をライトセンサに近づけると取得できる整数値が小さくなる．/switches は/light_sensors 同様に制御ノードから常に Publish されるスイッチの状態を表す値である．スイッチが押されると true が Publish され，スイッチが押されていないと false が Publish される．/cmd_vel はライントレースノードが Publish するトピックであり，ラズパイマウスのモーターを制御するためのトピックである．/buzzer はライントレースノードが Publish するトピックであり，ラズパイマウスから音を鳴らすことができるトピックである．/leds はライントレースノードが Publish するトピックであり，ラズパイマウスの LED を制御するためのトピックである．これらのトピックを介して Pub/Sub 通信を行えるよう，Python で記述されたライントレースノードを C++ に書き換え，mROS 2-POSIX と mROS 2-wasm に移植した．また，動作フローなどで通信時間に影響がでないように，ROS 2 と mROS 2-POSIX，mROS 2-wasm の動作フローを統一した．動作フローは図 4.3 に示す．

まず，各ノードが立ち上がると，ライントレースノードと制御ノード間で，/light_sensors トピックで Pub/Sub 通信が行われる．ここでライントレースノードはライトセンサの値を

取得する．また，同時にほかのトピックでの間でも通信が行われる．次に，各 Switch が押されたタイミングでライトセンサからライン外の値とライン内の値を保持し，その差を 2 で割った数を基準値にして保持する．そして，ライントレースが実行され，保持した基準値に基づいて/cmd_vel トピックに対してモーター制御の命令を送る．このように，ライントレースが実行されるノードを各環境に実装した．また，mROS 2-Wasm は第 2 章で述べた通り，AOT，JIT，Classic インタプリタ，Fast インタプリタの 4 つのコンパイル方式があるが，本研究では Classic インタプリタでコンパイルしたバイナリを用いて評価を行う．

4.3 実装アプリケーションの問題と解決策

ROS 2 で実装されているノードを mROS 2-POSIX に実装する際に，mROS 2-POSIX 固有の様々な問題や実装に関する問題に遭遇した．以下にその問題を述べる．

- mROS 2-POSIX の Publisher のバグ fix
- mROS 2-POSIX の QoS 設定の変更
- ライントレースノードの Service 通信部分の削除
- mROS 2-POSIX が同じ型のメッセージを複数 Pub/Sub 通信できない

既存の mROS 2-POSIX 実装では，Publisher にバグが存在し，適切に動作しない．mROS 2-POSIX と ROS 2 で通信を行う場合には payload error が発生する．これは，実際のアプリケーションのような複数の Publisher と Subscriber を用いるアプリケーションで，mROS 2-POSIX 実行し ROS 2 と通信を試みようとすると，発生する．この error は ROS 2 の Subscriber に対して，想定以上のデータが送信された場合に表示される．ros2 topic echo を用いてトピックに送信されているデータを確認したところ図 4.4 に示すように，ROS 2 の Subscriber に対して mROS 2-POSIX で Publisher として定義されているすべてのメッセージが送信されていることが分かった．これは mROS 2-POSIX の Publisher のインスタンスが 1 つのみの生成になっていることが原因である．mROS 2-POSIX は通信のために DDS 実装である embeddedRTPS の機能のインスタンスを生成している．主に通信用のインスタンスと Publisher のインスタンス，Subscriber のインスタンスが生成される．バグ修正前のインスタンスの状態とバグ修正後のインスタンス状態をを図 4.5 に示す．Publisher インスタンスにその情報が上書きされ，最後に作成された Publisher のトピック宛にすべてのメッセージが送信されている．そのため，本実装では通信用インスタンスに対して，Publisher のインスタンスを生成するように修正した．

第 2 章で述べた通り，ROS 2 には通信データをどのように扱うか設定できる QoS 設定が存在する．この QoS 設定は ROS 2 で Reliable が設定されており，mROS 2-POSIX はデフォルトで Best_effort が設定されている．第 2 章の表 2.1 に示すように Subscriber の QoS が Reliable，Publisher の QoS が Best_effort の場合通信が成立しない．embeddedRTPS には

QoS 設定として Best_effort と Reliable が ROS 2 同様に用意されており，mROS 2-POSIX か ROS 2 のノード間で通信するために，mROS 2-POSIX の QoS 設定の変更を行った．

mROS 2-POSIX では Service 通信が使えないため，Service 通信を使わないノードの実装を実現する必要がある．OS 2 のライントレースノードの実装では，モーターの on/off に Service 通信を使っている．2 章で説明した通り，Service 通信は ROS 2 の通信方法の 1 つであり，クライアント・サーバーモデルに近い通信方法である．ROS 2 のライントレースノードの実装では，この Service 通信を用いてラズパイマウスのモーターの on/off を操作しているが，mROS 2-posix には Service 通信の実装がされておらず，そのまま移植することはできない．本実装では，mROS 2-POSIX の実装部分でモーターの on/off の処理を削除し，ノードを立ち上げた後，ROS 2 の制御ノードに対してターミナルから ros2 service call を使用することで，モーター電源の on/off を操作した．

本実装では同じメッセージ型を使用することはなかったが，既存の mROS 2-POSIX で同じメッセージ型を使用する Publisher もしくは Subscriber を立ち上げると，ノードのビルドが成功しない．これは，mROS 2-POSIX が任意のメッセージ型を扱うための Template を用いた実装部分で，Pub/Sub のメッセージ型を格納するリストに重複を許していたからである．そのため，本実装では，重複を許さないようにリストの格納部分を修正し，同じメッセージ型を使用してもコンパイルエラーが発生しないようにした．これらの問題は mROS 2-Wasm にも同様に存在したが，mROS 2-POSIX を実装する際に解決した方法と同じであったため，解決した方法と同様で行った．

第 5 章

実験

表 5.1: 実験環境

ハードウェア	Raspberry Pi 3B+
OS	Ubuntu 22.04 LTS
CPU	1.4GHz 64-bit quad-core ARM Cortex-A53
メモリ	1GB LPDDR2 SDRAM
ROS 2	ROS 2 humble

/light_sensors	ROS2	mROS2	mROS2-Wasm
平均値	733.40 μ s	656.59 μ s	2101.89 μ s
最大値	830 μ s	780 μ s	2870 μ s
最小値	650 μ s	580 μ s	1740 μ s
標準偏差	58.14 μ s	51.16 μ s	282.89 μ s

表 5.2: /light_sensors の通信時間の統計

/cmd_vel	ROS2	mROS2-POSIX	mROS2-Wasm
平均値	42757.34 μ s	36954.83 μ s	46760.56 μ s
最大値	74220 μ s	72490 μ s	83190 μ s
最小値	16550 μ s	2380 μ s	14170 μ s
標準偏差	23451.16 μ s	20486.34 μ s	20424.32 μ s

表 5.3: /cmd_vel の通信時間の統計

	ROS2	mROS2-POSIX	mROS2-Wasm
RES(Resident Size)	32.51MB	1.66MB	17.37MB
SHR(Shared Memory)	25.16MB	1.48MB	1,73MB
RSS(Resident Set Size)	57.67MB	3.14MB	19.10MB

表 5.4: 各環境の物理消費メモリ（RSS）

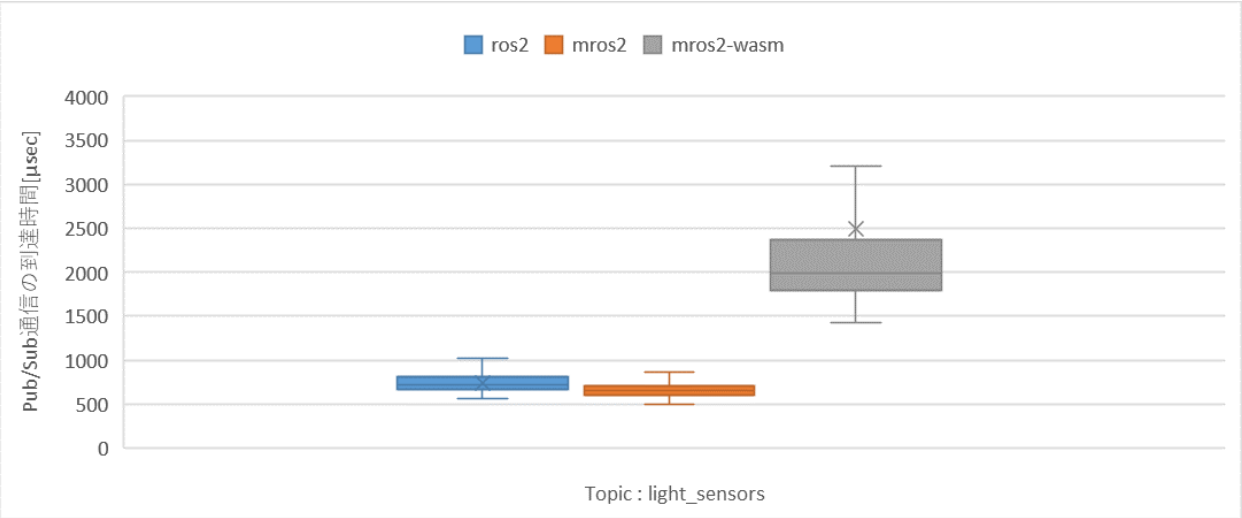

図 5.1: トピック: /light_sensors の通信時間の箱ひげ図

本章では，実装したアプリケーションを用いて，mROS 2-POSIX と mROS 2-Wasm と ROS 2 の性能を比較評価した実験について述べる．

5.1 概要

本研究で mROS 2-POSIX，mROS 2-Wasm，ROS 2 の性能を比較評価するために，それぞれの環境に実装したライントレースノードをラズパイマウスにジョイントしている Rassberry Pi 3B + 上で動作させ，制御ノードとライントレースノードの通信時間とメモリ消費量を比較する実験を実施した．この評価の目的は，ROS 2 と mROS 2-POSIX，mROS 2-Wasm の通信性能とメモリ消費量を実アプリケーション上の値をもとに比較し，実アプリケーション上でも mROS 2-POSIX が動的配置機構のソフトウェア基盤として適しているか，そして，mROS 2-Wasm を実アプリケーション上で動作させた場合でも，mROS 2-POSIX の軽量という利点を損なわないかを示すことである．

実験の実行環境を表 5.1 に示す．なお，第 2 章で述べた通り，ネイティブ ROS 2 のランタイムディストリビューションは最新版である humble を採用した．

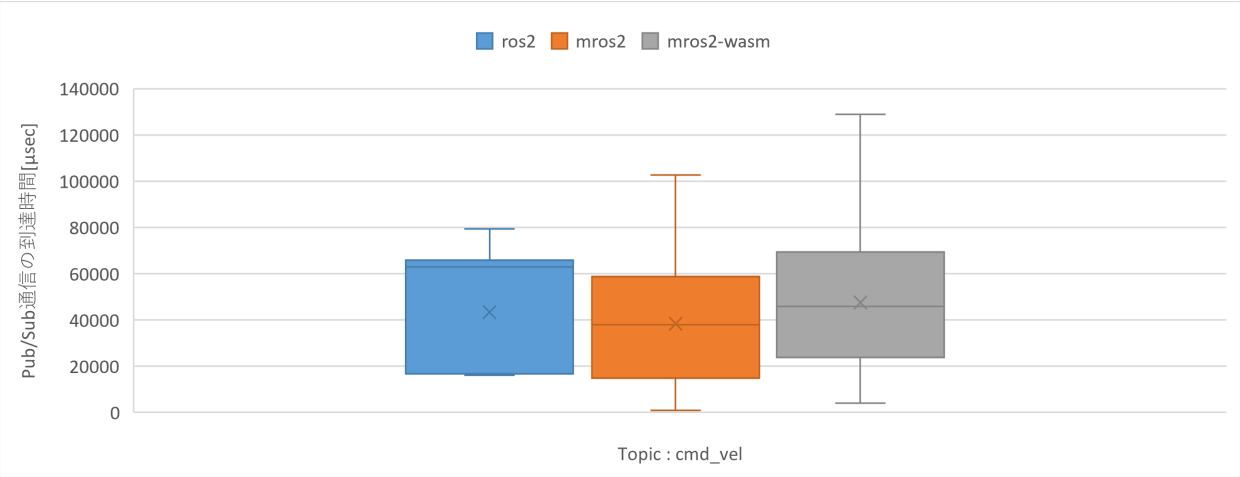

図 5.2: トピック: /cmd_vel の通信時間の箱ひげ図

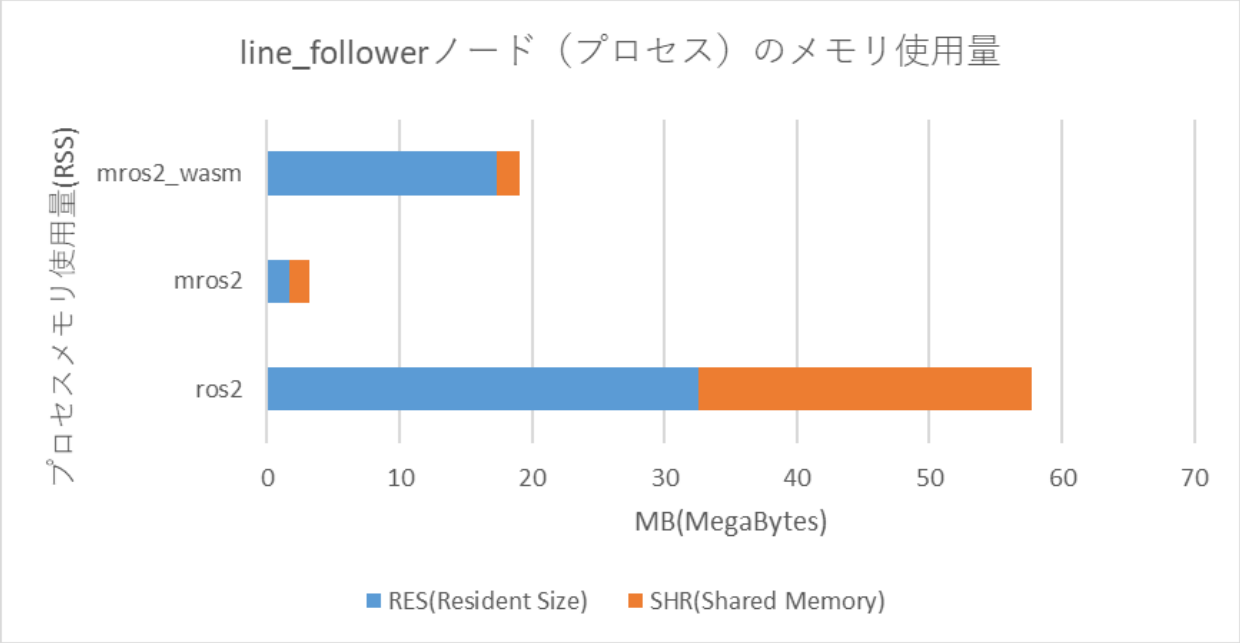

図 5.3: 各環境で RES，SHR を合わせた RSS の比較

通信時間の計測は，2 つのノード間で Pub/Sub 通信にかかる時間を Linux 標準の clock_gettime() を用いた．clock_gettime() といっても，様々な実装があり，実時間を計測するシステム全体で一意な時間を取得する CLOCK_REALTIME や，ある開始地点からの単調増加の時間で表現される CLOCK_MONOTONIC などがある．今回計測に用いたのは CLOCK_MONOTONIC である．これは，時間を計測するトピックは，/light_sensors と/cmd_vel である．このトピックはライントレースする際に使用する主なトピックであり，

第 4 章で述べた通り，/light_sensors はライトセンサの値を，/cmd_vel はラズパイマウスのモーターを制御するためのトピックである．/light_sensors は Int16 型が 4 つ格納されている配列であり，/cmd_vel は geometry_msgs/Twist 型で，float64 型が 6 つ格納されている配列である．この 2 つのトピックの Publish した現在時刻，Subscribe した現在時刻を取得し，その差分を通信時間とした．評価にあたって，試行回数を 1000 回とし，平均値，最大値，最小値，標準偏差を計算し，通信時間の箱ひげ図を作成した．/light_sensors と/cmd_vel の Pub/Sub の関係は第 5 章で述べた通り，/light_sensors は制御ノードから Publish されるトピックであり，mROS 2-POSIX，mROS 2-Wasm は Subscribe するトピックである．/cmd_vel は mROS 2-POSIX，mROS 2-Wasm が Publish するトピックであり，制御ノードが Subscribe するトピックである．

メモリ消費量の計測は，mROS 2-POSIX，mROS 2-Wasm，ROS 2 の各環境でのラインレースノードに対して，実行しているランタイムのプロセス ID を取得し，そのプロセスに割り当てられている RSS（物理消費メモリ）を計測した．各環境で同様の計測を行い，RSS は時間によって変化しなかったため，この RSS を用いて，グラフを作成した．

5.2 結果

実験結果を図 5.1 から図 5.3 および表 5.2 から表 5.4 に示す．

図 5.1 では各環境ごとにトピック/light_sensors の Publish した現在時刻と Subscribe した現在時間の差を計算し，箱ひげ図として通信時間のばらつきを示している．ROS 2 と mROS 2-POSIX の通信時間のばらつきは比較的同等であるが，mROS 2-Wasm を見ると，非常に大きなばらつきがあった．

表 5.2 は，トピック/light_sensors に対して各環境ごとに通信時間の平均値，最大値，最小値，標準偏差を示している．表 5.2 を見ると，一番小さい平均値を記録したのが mROS 2-POSIX であり，次いで ROS 2，mROS 2-Wasm の順である．mROS 2-POSIX と ROS 2 の平均時間の差は約 77 μ sec であり，mROS 2-POSIX と ROS 2 の通信時間の平均がおおよそ同程度，または mROS 2-POSIX の Pub/Sub 通信の方が ROS 2 よりも若干高速であることだった．標準偏差は測定値の分布が平均値のまわりにどの程度集まっているか示す指標で小さい順に，mROS 2-POSIX，ROS 2，mROS 2-Wasm の順である．mROS 2-POSIX と ROS 2 の差はほとんどなく，若干 ROS 2 のばらつきが大きいが，ほとんど同程度であった．mROS 2-Wasm の通信時間の平均が mROS 2-POSIX と ROS 2 はおおよそ同等であることのに比べて，およそ 3 倍になっており，標準偏差は，mROS 2-POSIX と ROS 2 がおおよそ同等であるのに対し，mROS 2-Wasm はおよそ 5 倍の値になっている．

図 5.2 は，図 5.1 と同様に各環境ごとの/cmd_vel の通信時間のばらつきを示す箱ひげ図である．図 5.2 を見ると，mROS 2-POSIX と ROS 2 と mROS 2-Wasm の通信時間のばらつきはほぼ同等であった．

表 5.3 は，表 5.2 同様トピックの/cmd_vel の通信時間の平均値，最大値，最小値，標準偏差を示している．表 5.3 の平均値は，小さい順に mROS 2-POSIX，ROS 2，mROS 2-Wasm の順である．mROS 2-POSIX と ROS 2 の差は 5082 µ sec であり，mROS 2-POSIX の方が遅延が少なかった．また，ROS 2 と mROS 2-Wasm の差は 4003 µ sec であり，ROS 2 の方が遅延が少なかった．標準偏差を見ると，小さい順に mROS 2-Wasm，mROS 2-POSIX，ROS 2 の順である．これは平均値の順と逆であり，mROS 2-Wasm と mROS 2-POSIX の差はほとんどなく，ROS 2 の標準偏差は mROS 2-Wasm と mROS 2-POSIX で大きくことなっている．

図 5.3 は，ライントレースノードを各環境で動作させたときの RES（Resident Size），SHR（Shared Memory）を合わせた RSS の棒グラフである．表 5.4 は，各環境の RES，SHR，RSS をそれぞれ数字で示している．図 5.3，表 5.4 を見ると，mROS 2-POSIX のメモリ消費量がもっとも小さく，次いで mROS 2-Wasm，ROS 2 の順である．mROS 2-POSIX は ROS 2 や mROS 2-Wasm と比べて非常に小さなメモリ消費量であった．ROS 2 との差は 54.53MB である．mROS 2-Wasm は ROS 2 よりもメモリ消費量が 38.57MB と少なく，mROS 2-POSIX のほうが 15.96MB 少ない．

5.3 考察

図 5.1，表**??**，図 5.2，表 5.3 から，mROS 2-POSIX は ROS 2 と比べて Pub/Sub 通信の遅延が少なく，安定かつ高速に通信できていることが分かった．これは，mROS 2-POSIX が組込み用デバイス向けである DDS の embeddedRTPS が軽量な TCP/IP スタックである lwIP を採用しているためだと考える．そして，ROS 2 と mROS 2-Wasm の通信時間を比べると，ROS 2 側が Subscribe を行い，mROS 2-Wasm 側が Publish を行うトピック/cmd_vel では平均の遅延の差は約 4000 µ sec，mROS 2-Wasm の遅延が大きかったものの，標準偏差は ROS 2 よりも小さかった．mROS 2-Wasm が ROS 2 から Publish されたメッセージを Subscribe しているトピック/light_sensors では，mROS 2-Wasm の遅延が ROS 2 の時よりも平均で 3 倍ほど大きく，標準偏差はおよそ 5 倍になっていた．これは，mROS 2-Wasm が Classic インタプリタでコンパイルされ実行されているのが原因であると考える．しかし，Classic インタプリタで実行されたノードでも Publish の通信時間は ROS 2 とそこまで大きな差がないということが示された．この原因として，Publisher の実装がコールバック関数を呼び出す必要のある Subscriber と比べて，メッセージ処理に時間がかかってしまい，遅延が大きくなると考える．

通信性能の評価では，実アプリケーション上で mROS 2-POSIX は ROS 2 よりも高速で動作することが示された．また，mROS 2-Wasm は Classic インタプリタでコンパイルされた場合，遅延がほかの環境よりも大きくなることが示された．

図 5.3 と表 5.4 から，mROS 2-POSIX は ROS 2 と比べてメモリ消費量が少ないというこ

とがわかった．これは mROS 2-POSIX が ROS 2 で実装されているアプリケーションでも軽量なランタイムとして機能しているということを示している．mROS 2-Wasm は ROS 2 と比べてメモリ消費量が少なく，mROS 2-POSIX と比べてメモリ消費量が大きいということがわかった．これは，mROS 2-Wasm は mROS 2-POSIX を Wasm 化したものであり，Wasm 分のメモリ消費が mROS 2-POSIX に計上されているため，mROS 2-POSIX よりもメモリ消費量が大きいからである．ネイティブの ROS 2 よりもメモリ消費量が少ないことで，ROS 2 を動作させるよりも，mROS 2-Wasm を動作させたほうがメモリ消費量の削減が可能であることが示された．

今回の実験によって，実アプリケーションを動作させても，mROS 2-POSIX は ROS 2 よりメモリ消費量と通信性能で優れており，リソースの限られたデバイスでノードを実行する場合，mROS 2-POSIX のほうが ROS 2 よりも適していることを示すことができた．mROS 2-POSIX を Wasm 化した mROS 2-Wasm は実アプリケーションを Classic インタプリタでコンパイルした場合，通信性能で ROS 2 よりも劣るものの，メモリ消費量で ROS 2 より優れていることから，ROS 2 を Wasm 化するより mROS 2-POSIX を Wasm 化することでメモリ使用量の削減ができることを示せた．ただ，通信性能の劣化はコンパイル方式を JIT もしくは AOT にすることで改善することができる可能性がある．先行研究でソフトウェア基盤として採用された mROS 2-POSIX は実アプリケーション上でも軽量ソフトウェア基盤として最適であり，mROS 2-Wasm を実アプリケーション上で動作させた場合でも，mROS 2-POSIX の軽量という利点を損なわず Wasm 化することができていることを示した．

第 6 章

関連研究

この章では本研究の関連研究について述べる．

6.1 mROS 2: 組込みデバイス向けの ROS 2 ノード軽量実行環境

組込みデバイス向けの高効率な ROS 2 通信方式およびメモリ軽量な実行環境を確立するために，高瀬らは提案として軽量ランタイムである mROS 2 を設計，実装，評価した．mROS 2 を評価するにあたって，ROS と micro-ROS を用いている．micro-ROS は ROS 2 の組込みデバイス向けの軽量実行環境である．mROS 2 と micro-ROS の違いは，ROS 2 と通信する際の Agent ノードの有無である．Agent ノードを立ち上げなければならない，micro-ROS は通信に遅延が発生する恐れがある．通信性能の評価実験として std_msgs::msg::Int32 のメッセージをエコーバックするアプリケーションを用いている．アプリケーションでの RTT（Round Trip Time）を計測することで通信性能を評価した．結果は，mROS 2 の RTT が一番小さくなり次いで汎用デバイス同士をつないだ ROS 2 環境が速かった．メモリサイズの比較では，mROS 2 が ROS 2 よりもメモリ消費量が少なかった．この評価手法のメモリサイズ消費量について本研究で参考にした．通信性能の評価手法は本研究では適用しない．ライントレースノードと制御ノードの関係はラウンドトリップの関係にないためである．この実験結果から組込みデバイス上で動作した mROS 2 の RTT は汎用デバイス上で動作した ROS よりも高速であることが分かった．

6.2 ROS 2 ノード軽量実行環境 mROS 2 における任意型メッセージの通信処理方式

ROS 2 の軽量ノード実行環境である mROS 2 において，任意のメッセージ型を扱うための通信処理方式を高瀬らは提案した．具体的には，mROS 2 における通信処理機構を，メッ

セージ型に関して共通の処理および固有の処理に分離し，固有の処理を行うファイルはメッセージ型ごと生成し通信フローに組み込むことで，任意のメッセージ型による通信を可能にした．通信性能において，型変換の処理を含む通信遅延時間をμｓで計測し，提案手法と過去のバージョンと比較することで，提案手法の有用性を示した．この通信性能においての評価手法を本研究では参考にした．課題として，任意型の配列を含む型による通信は未対応である．また，250Bytes 以上のメッセージサイズが不可能であるという課題がある．

第 7 章

結論

本章では，本研究のまとめと今後の課題について述べる．

7.1 まとめ

本研究では，クラウドロボティクスにおける実行状態を持つ ROS ノードの動的配置機構の実現後に，mROS 2-POSIX と mROS 2-Wasm と ROS 2 上で実アプリケーションを動作させ，その性能を比較評価することで，ロボットソフトウェア基盤としてどのような優位性があるのか明らかにすることを目指した．

先行研究で実現した mROS 2-Wasm や mROS 2-POSIX にはネットワークスループットのマイクロベンチマークに留まっており，実アプリケーション上での有用性が十分に評価されていなかった．本研究で，動的配置機構実現後のアプリケーションとして，組込み用デバイスで動作することができるアプリケーションかつ，Pub/Sub 通信を主に使用しているアプリケーションでり，OS に依存していないアプリケーションという 3 つの条件を満たしているノード実装として，ROS 2 にすでに実装されているラズパイマウスで，ライントレースを行うノードを mROS 2-POSIX と mROS 2-Wasm に移植し，通信性能とメモリサイズの評価を実施した．

結果として，通信性能は mROS 2-POSIX，mROS 2-Wasm が ROS 2 からのメッセージを Subscribe する/light_sensors トピックにおいて，mROS 2-POSIX が最も高速で，次に ROS 2，最後に mROS 2-Wasm であった．このトピックでは，mROS 2-Wasm が他の環境と比べて非常に遅延が大きくなった．mROS 2-POSIX と mROS 2-Wasm が ROS 2 にメッセージを Publish する/cmd_vel トピックにおいては，mROS 2-POSIX が最も高速で，次に ROS2，最後に mROS 2-Wasm であった．しかし，トピック/light_sensors の時と比べて mROS 2-Wasm の遅延はほかの環境と大きく差がなかった．

メモリサイズの比較では，RSS は ROS 2 がもっとも大きく，次に mROS 2-Wasm，最後に mROS 2-POSIX であった．

考察として，通信性能は，mROS 2-POSIX は実アプリケーション上でも高速に動作することができることが示され，mROS 2-Wasm が Classic インタプリタでコンパイルされた場合，遅延が大きくなることが示された．

メモリサイズの比較では，mROS 2-POSIX が実アプリケーションを用いても ROS 2 よりメモリ消費量が圧倒的に少なく，mROS 2-Wasm も Wasm 分オーバーヘッドが増加した場合でも，ROS 2 の RSS より小さくなることが示された．

今回の実験によって，実アプリケーションを動作させても，mROS 2-POSIX は ROS 2 よりメモリ消費量と通信性能で優れており，リソースの限られたデバイスでノードを実行する場合，mROS 2-POSIX のほうが ROS 2 よりも適していることを示すことができた．また，mROS 2-Wasm を実アプリケーション上で動作させた場合でも，mROS 2-POSIX の軽量という利点を損なわず Wasm 化することができており，ROS 2 を Wasm 化するより mROS 2-POSIX を Wasm 化することでメモリ使用量の削減ができることを示せた．

7.2 今後の課題

今後の課題は，柿本ら [5] の mROS 2-Wasm で JIT，AOT の両方のコンパイルを実施し，そのバイナリファイルを用いて性能を評価することである．

先行研究での今後の課題として，mROS 2-Wasm が JIT，AOT コンパイル後，ノードを立ち上げることができないという問題があった．そのため，Classic インタプリタでコンパイルしたノードを実行することで通信性能の評価を実施した．しかし，Classic インタプリタでコンパイルしたノードは，AOT コンパイルしたノードに比べて通信の遅延が非常に大きくなったため，JIT，AOT コンパイルしたノードをもとに通信性能を評価することが必要である．

現在の mROS 2-Wasm には保存・復元機能が実装されていない．管ら [4] の研究で ROS 2 の動的配置機構が実現されたが，mROS 2-POSIX にスレッド操作等の処理が含まれているため，WAMR の実行状態を保存，復元する機構をそのまま適用することができる．大学院進学後は，スレッド操作等の処理を含む場合の実行状態の実態や，その扱いについて調査し，動的配置機構を実現することでアーキテクチャ中立な ROS ランタイム実行状態を異種デイバス間でマイグレーションすることを目指す．

謝辞

この度の研究を通じて、多大なるご指導とご支援を賜りました松原　克弥先生に心からの感謝の意を表します．また、日々の研究生活において、絶えず励ましと支えを提供してくださった研究室の仲間たち、先輩方にも深く感謝申し上げます．

参考文献

[1] ROSWiki：ROS/Introduction, urlhttp://wiki.ros.org/ROS/Introduction..（Accessed on 01/23/2024）

[2] ロボット政策研究会: ロボット政策研究会 報告書 RT 革命が日本を飛躍させる ,https://warp.da.ndl.go.jp/info:ndljp/pid/286890/www.meti.go.jp/press/20060516002/robot-houkokusho-set.pdf(2006).

[3] Kehoe et al. explored cloud-based robot grasping utilizing the Google object recognition engine, presenting their findings in the 2013 IEEE International Conference on Robotics and Automation, pages 4263-4270.

[4] 菅文人, 松原克弥: クラウドロボティクスにおける異種デバイス間タスクマイグレーション機構の検討，研究報告組込みシステム (EMB)，Vol. 2022， No. 36， pp. 1-7(2022).

[5] 柿本翔大，松原克弥: クラウド連携を対象としたアーキテクチャ中立な ROS ランタイムの実現，情報処理学会研究報告， Vol. 2023-EMB-62，No. 51 ，pp. 1-7(2023).

[6] 高瀬英希，田中晴亮，細合晋太郎: ロボットソフトウェア軽量実行環境 mROS 2 の POSIX 対応に向けた実装および評価，日本ロボット学会誌，Vol. 2023-EMB-41，No. 8，pp. 724-727(2023).

[7] Object Management Group: About the DDS Interoper-ability Wire Protocol Version 2.5 (online), https://www.omg.org/spec/DDSI-RTPS/2.5 (2024.01.25).

[8] 高瀬英希:ROS (Robot Operating System) の紹介と IoT/IOT 分野への展開,RICC-PIoT workshop 2022.

[9] Fastdds Simple Discovery Settings5.3.2, https://fast-dds.docs.eprosima.com/en/latest/fastdds/discovery/simple.html.（Accessed on 01/24/2024）

[10] Fastrtps ros index https://index.ros.org/r/fastrtps/.（Accessed on 01/24/2024）

[11] RTI Connext DDS, https://www.rti.com/en/.（Accessed on 01/24/2024）

[12] Micro XRCE-DDS, https://micro-xrce-dds.docs.eprosima.com/en/latest/index.html.（Accessed on 01/25/2024）

[13] Eclipse Cyclone DDS, https://cyclonedds.io/.（Accessed on 01/25/2024）

[14] Gurum DDS, https://gurum.cc/index_eng.（Accessed on 01/25/2024）

[15] クライアント・サーバモデル, https://www.ibm.com/docs/ja/txseries/8.2?topic=computing-clientserver-model.（Accessed on 01/25/2024）

[16] “micro-ROS — ROS 2 for microcontrollers,” https://micro.ros.org/.（Accessed on 01/25/2024）

[17] ”embeddedRTPS”, https://github.com/embedded-software-laboratory/embeddedRTPS.（Accessed on 01/25/2024）

[18] ”lwIP”, https://savannah.nongnu.org/projects/lwip/.（Accessed on 01/25/2024）

[19] ”TOPPERS/ASP3 kernel”, https://www.toppers.jp/asp3-kernel.html.（Accessed on 01/25/2024）

[20] ”POSIX”, https://ibm.com/docs/ja/zos/2.5.0?topic=ulero-posix.（Accessed on 01/25/2024）

[21] ”CMSIS-POSIX”, https://www.keil.com/pack/doc/CMSIS/RTOS2/html/index.html.（Accessed on 01/25/2024）

[22] ”Raspberry Pi Mouse”, https://rt-net.jp/products/raspberrypimousev3/.（Accessed on 01/25/2024）

[23] ”2023 ROS Metrics Report”, https://discourse.ros.org/t/2023-ros-metrics-report/35837.（Accessed on 01/25/2024）

[24] WebAssembly, https://webassembly.org/.（Accessed on 01/25/2024）

[25] WASI, https://wasi.dev/.（Accessed on 01/25/2024）

[26] bytecodealiance/wasm-micro-runtime（wamr）, https://github.com/bytecodealliance/wasm-micro-runtime#.（Accessed on 01/25/2024）

[27] WebAssembly - mdn, https://developer.mozilla.org/en-US/docs/Web（Accessed on 01/25/2024）