

# ROS ノード軽量実行環境 mROS の ユーザ定義メッセージ型の対応のための機能拡張

祐源 英俊<sup>1,a)</sup> 高瀬 英希<sup>1,2,b)</sup>

受付日 2020年6月30日, 採録日 2020年12月1日

**概要:** mROS は, ロボットソフトウェアの開発支援プラットフォームである ROS を, 組み込み機器上で動作可能とする軽量実行環境である. 本研究では, ROS におけるユーザ定義のメッセージ型のヘッダファイルを mROS に向けて自動生成する手法を提案する. ROS で規定されたメッセージの型ならびにユーザが独自定義する型を, mROS 上で扱えるようにする. あわせて, ユーザ定義メッセージ型を mROS において取り扱えるようにするための, mROS 通信ライブラリの新たな動作フローの設計を示す. これらの提案手法により, 従来研究の成果において mROS が有していたメッセージの型に関する制約を解消することができる. さらに本研究では, 提案手法に基づく機能拡張を mROS に施したうえで, その性能を mROS の従来実装および他手法 *rosserial* と比較評価する. これにより, 本研究成果は組み込み機器を活用する ROS ベースのロボットシステムの開発を容易化することに貢献することを示す.

**キーワード:** ROS, 組み込みシステム, リアルタイム OS, 通信

## A Functionality Expansion of the Lightweight Runtime Environment mROS for the User-defined Message Types

HIDETOSHI YUGEN<sup>1,a)</sup> HIDEKI TAKASE<sup>1,2,b)</sup>

Received: June 30, 2020, Accepted: December 1, 2020

**Abstract:** mROS is a lightweight execution environment that enables node programs of ROS to be executed on embedded devices. In this research, we aim to remove the constraint on message types that mROS can handle. We propose an approach that generates header files of message types automatically for mROS. We also propose an operation flow of mROS communication library. Proposed approach and flow make it possible for the mROS environment to handle various message types, including primitive types and types defined by users. Furthermore, we implement the proposed flow to mROS and conduct the comparative evaluation against mROS with the previous implementation and the *rosserial*, another lightweight ROS node execution environment. By these, we show that this research contributes to the easier development of ROS based robot systems with embedded devices.

**Keywords:** ROS, embedded devices, real-time operating systems, communication

### 1. はじめに

近年, ロボットシステムの開発を支援するソフトウェアプラットフォームである ROS [1] が注目を集めている.

ROS の提供する機能の 1 つとして, システムを構成するソフトウェアコンポーネントどうしの通信を可能にする, 通信ミドルウェアの機能がある.

この機能を活用することにより, 所望のシステムを, ノードと呼ばれる機能部品プログラムの組合せによって表現することができる. これによって, 本来は複雑かつ広範な知識が要求されるロボットシステムを, 容易にかつ効率的に構築することができる.

この ROS の通信機能では, トピックを介したメッセー

<sup>1</sup> 京都大学大学院情報学研究科  
Graduate School of Informatics, Kyoto University, Kyoto  
606-8501, Japan

<sup>2</sup> JST さきがけ  
PRESTO Program, Japan Science and Technology Agency,  
Kawaguchi, Saitama 332-0012, Japan

a) emb@lab3.kuis.kyoto-u.ac.jp

b) takase@i.kyoto-u.ac.jp

ジのやりとりにより、出版購読型のデータ通信を実現する。この出版購読型通信モデルとは、情報を送信する出版者および情報を受信する購読者が、通信チャンネルであるトピックを介して通信を行うものである。ROSでは、このトピックを通過するメッセージの型を、自由に指定することができる。メッセージの型としては、数値や文字列などの基本型に加え、ユーザがそれらを自由に組み合わせて定義するユーザ定義型も利用可能である。

我々はこのROSの活用の幅を広げ、ROSを用いるシステムの性能を向上させることを目的に、mROS [2]を開発している。mROSは、TOPPERS/ASP [12]カーネルの上で実行されるROSノードの軽量実行環境である。mROSを利用することにより、本来はLinux環境が必要なROSノードを、ASPカーネルが搭載可能な組み込み機器上で実行することができる。これにより、ROSを用いるシステムのリアルタイム性能確保およびシステム内の計算資源の最適化が行える。さらに、ROSを活かした効率の良い組み込みシステム開発もできるようになると期待される。

mROSでは、mROSアプリケーションが出版および購読することのできるメッセージの型が、文字列型に限定されていた。これは、mROS通信ライブラリに実装する機能を抑え、プログラムサイズおよびプログラムの規模を抑えようとしたためである。しかし、このようにmROSが通信に利用できる型を限定することにより、mROSアプリケーション開発および利用において、次のような問題が生じていた。具体的には、まず、文字列型以外のメッセージを扱う際のエンコード処理が、通信動作時のオーバーヘッドになる。このエンコード処理は、通信のために送信時にはメッセージを文字列型に変換し、受信時には文字列型から元の型に変換するものである。このため、mROSアプリケーションだけでなく、mROSアプリケーションと通信を行うROSノードに対しても、同様の処理を追加する必要がある。このようなエンコード処理をメッセージ型ごとにユーザが用意すると、mROSのプログラムサイズが肥大化する可能性がある。またROSノードアプリケーションとの互換性が低く、mROSアプリケーションの開発が煩雑になっているという問題もある。

そこで本研究では、mROSについてのこの課題を解決し、mROSの性能、汎用性および移植性の向上を目指す。このために、mROSにおいて、ユーザ定義メッセージ型を含む任意のメッセージ型による通信をできるようにする。ROSのシステムに組み込み機器を導入する利点および課題点から、本研究の提案により満たされるべき要件について検討する。検討した要件に基づき、mROSにおいて任意のメッセージ型を処理するための開発手法、および、mROS通信ライブラリの動作フローを提案する。メモリ量に制限がある組み込み機器での運用においては、動的メモリ割当ての抑止およびメモリ量の最小化は必須の要件であ

る。このため、提案手法の設計および実装にあたっては、ROSの通信ライブラリ内で用いられているboostライブラリは用いないようにする。提案に基づく機能拡張をmROSに施し、本研究で定義する要件が満たされているかを確認する性能評価を行う。性能評価では、シリアル通信によってROSノードおよび組み込みアプリケーション間の通信を実現する、rosserialとの比較評価も行う。他手法との性能評価は、文献 [2] など、既存の研究では行われていなかった。本研究においてはこの評価も行うことにより、他手法に対するmROSの優位性を明確にする。

本研究の貢献は次にあげるとおりである。まず、mROSに対し、文字列型以外のメッセージ型を扱うための機能拡張が行えるようになる。これは、mROSの通信ライブラリのプログラムサイズを肥大化させることなく行うことができる。この機能拡張によって、通信の際に必要な文字列型へのエンコード処理および文字列型からのデコード処理が不要となり、通信時のオーバーヘッドが削減される。さらに、既存のROSノードアプリケーションとのプログラムの互換性も向上する。

本文の構成は以下のとおりである。2章で、背景としてROSおよびROSを用いるシステムに組み込み技術を導入する利点を説明する。また、ROSと組み込み技術の結び付けに関連する研究を紹介する。3章で、本研究で扱うmROSについて説明したうえで、その課題点を指摘する。さらに、課題をもとに本研究で満たすべき要件を定義する。4章で、本研究で提案する開発手法およびmROS通信ライブラリの動作フローについて説明する。5章で、本研究で行う評価について説明する。また、その結果を掲載し、結果に基づく議論を行う。6章で、本研究のまとめおよび今後の展望を述べる。

## 2. ROSへの組み込み機器の導入

本章では、まずROSについて説明する。次に、ROSへの組み込み機器技術の導入について、その利点および問題点を述べる。

### 2.1 ROS

ROS [1] は、ロボットシステムの開発を支援するソフトウェアプラットフォームである。元々は研究用のプロトタイプ機を効率良く開発するため、スタンフォード大学およびWillow Garage社によって開発された。現在は、Open Roboticsによって、維持管理、リリース、産業界への導入促進が行われている。

ROSは、ロボットシステム開発のための通信ミドルウェア、ビルドツール、デバッグツール、および、シミュレーションツールなどを提供する。なかでも代表的な機能として、出版購読型通信モデルに基づいたシステムコンポーネント間のデータ通信を実現する、通信機能があげられる。

出版購読型通信モデルとは、情報を送信する出版者および情報を受信する購読者が、通信チャネルであるトピックを介して通信を行うという通信方式である。ROSにおいては、購読者であるサブスクリバノードが、出版者であるパブリッシャノードから、通信情報であるメッセージを受け取ることによってデータ通信が実現されている。

ROSでは、トピックを通過するメッセージの型を、トピックごとに指定することができる。トピックに指定する型は、数値および文字列型など基本的なデータ型であるプリミティブ型 [14] に加え、それらを組み合わせることにより定義されるユーザ定義型も利用可能である。ユーザ定義型は、いくつかの内部変数を持つ構造をしている。内部変数にはプリミティブ型、既存のメッセージ型およびそれらの配列が利用可能である。

この出版購読モデルに基づいた通信は、すべてトピックを介して行われる。このため、システム内のノードどうしの結合が弱くなる。したがって、システムに対するノードの追加および削除を非常に容易に行うことができ、システムの構成を柔軟に変更できるようになる。一方、このROSの通信機能は、Linuxの通信ミドルウェアとして提供されている。このため、ROSのノードを実行し、ノード間のデータ通信を行わせるためには、Linux環境が必要である。

なお、現在はROSのほか、次世代版ROSであるROS 2 [9]もリリースされている。ROS 2は通信動作の実装を一新し、組み込み機器など比較的小さいデバイスでも実行することを想定したものとなっている。しかし、利用者の数および利用可能なプログラム資産は、依然としてROS 2よりROSの方が多いため、本研究では、ROSのみについて扱う。

## 2.2 組み込み技術導入の利点

前述のとおり、ROSのノードを実行するためには、原則としてLinux環境が必要である。Linux OSは、汎用OSに分類される。このため、実行されるプロセスが多く、カーネルが大規模となっている。したがって、リアルタイム性能を確保することが難しい。ここでのリアルタイム性能とは、決められた時間内にタスクを終わらせることができる能力を指す、このことはノードプログラムの応答性低下へとつながり、意図しない機器の動作を引き起こす可能性がある。

この問題は、システムの一部ノードを、組み込みシステム向けのリアルタイムOS上で実行することにより解決できると考えられる。リアルタイムOSは、リアルタイム性能を確保することを重視して設計された軽量のOSである。備える機能が限定的である反面、実行されるプロセスが少なく、大規模な汎用OSに比べてタスクスケジューリングが容易である。さらにリアルタイムOSは、その実行に必要な計算資源が、汎用OSに比べ少ない。このため、消費電

力およびコストが抑えられた組み込み機器に搭載することが容易である。以上のことから、リアルタイムOSを搭載する組み込み機器上でROSのシステムの一部のノードを実行することにより、そのノードアプリケーションの応答性を向上させることが期待される。

また、Linuxは大規模なソフトウェアプログラムであるため、搭載にあたり要求される計算資源の水準が高い。このため、消費電力、メモリ、および計算能力などの計算資源が豊富であるデバイスを採用する必要がある。ROSを用いて分散システムを構築する場合、比較的単純な機能を持つノードを実行し、ホストとの通信を行わせるというシステム構成も想定される。そのようなノードを高機能デバイスで実行すれば、高水準の計算資源が無駄になる。

この問題は、ノードの機能およびシステムの要求に合わせて、計算資源が限定的な組み込み機器を採用することにより解決できると考えられる。リアルタイムOS上のROSノード実行環境を活用することにより、より計算資源が限定された機器によってROSノードが実行できる。これにより、システムが利用する計算資源を、より適したものにすることができる。

## 2.3 関連研究

文献 [5] は、自動制御で動作しエリアの探索および目的地への移動を行う車椅子についての研究である。ROSのシステムにArduinoを活用することで、低コストかつ柔軟なシステムを実現している。文献 [6] では、ROSを用いてモバイルロボットおよびクラウドを連携させた、VSLAMのボトルネックの緩和を目指したシステムを提案している。ROSを活用し、SLAMに必要な計算をクラウド上のノードに負荷分散させている。いずれも、情報の収集、送受信、および、受信情報に基づく動作のみを行う、いわゆるエッジ機器がシステム構成中に存在している。両文献中においてはノートPCを採用しているが、ここをmROSを実行する組み込み機器に置き換えられると考えられる。これにより、エッジ機器の小型化および省電力化ができる。

続いて、ROSと組み込み機器の通信および協調を行う研究を紹介する。文献 [4] は、組み込み機器上においてROSノードを実行するプロトコルおよびライブラリである、`rosserial` についての公式ドキュメントである。

`rosserial` を利用するシステムの例を、図 1 に示す。`rosserial` を活用すれば、ほかROSノードとの通信を行うプログラムを組み込み機器上で実行できる。ただし、利用できる通信方式はシリアル通信のみである。このため、ネットワークを用いる場合に比べ、伝送速度および伝送範囲に劣る。また、図 1 に示すとおり、ブリッジである `rosserial_server` を実行する必要がある。`rosserial_server` は、組み込み機器上のクライアントアプリケーションとはシリアル通信を、ホストPC上などの他のROSノードとはネットワーク通信

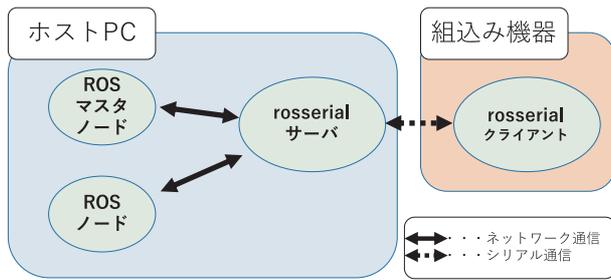


図 1 rosserial を利用するシステムの例  
Fig. 1 Example of the system using rosserial.

を行う。次世代版 ROS である ROS 2 [9] は、組み込み機器への対応を念頭に置いて開発されている。さらに、組み込み向け ROS 2 実行環境環境である micro-ROS [10] も開発されている。この 2 つの手法は、ともに通信ミドルウェアに Micro-XRCE-DDS [11] を使用している。Micro-XRCE-DDS は、組み込み機器で実行する Micro-XRCE-DDS-Client およびホスト PC 上で実行する Micro-XRCE-DDS-Agent の間で通信を行う。このため、Micro-XRCE-DDS-Agent がブリッジとなる。この場合、ブリッジが単一障害点となり、またブリッジを挟むことにより通信遅延時間が増大する。また、ROS 2 は ROS の主要な概念を継承しているものの実装が大きく変更されており、API レベルで互換性がない。したがって、すでに ROS を用いて構成されたシステムを ROS 2 および組み込み環境の micro-ROS を利用する場合は、その開発者は ROS 2 の API や利用方法について学ぶ学習コストがかかる。そのうえで、既存資産を ROS 2 に向けて移植することが必要になる。文献 [7] は、ROS と、リアルタイムで動作するソフトウェアフレームワークである LUNA との通信を実現するブリッジについて提案および実装を行っている。LUNA と直接の通信を行う LUNA ブリッジが ROS トピックに対しメッセージの送信および受信を行うことにより、間接的に ROS ノードと LUNA アプリケーションの通信を実現している。しかし、2 者間の通信にはブリッジを挟まなければならない。また、ROS および LUNA という 2 つのフレームワークを用いているため、学習コストが高く、コードの再利用性が低い。文献 [8] では、IEC61131-3 に準拠した、ソフトウェア PLC の主要な製品である CODESYS と ROS の通信インタフェースを提案している。両者間のインタフェースとして、ROS メッセージの送受信および CODESYS の共有メモリへのアクセスを行う ROS ノードを実装している。これも、ブリッジを含むシステム構造が必要である。また、ユーザ定義メッセージ型や配列については、対応可能性が示唆されているのみである。

### 3. mROS

#### 3.1 mROS

mROS [2] は、2018 年から著者らが開発している、ROS



図 2 mROS のソフトウェア構成  
Fig. 2 Software structure of mROS.

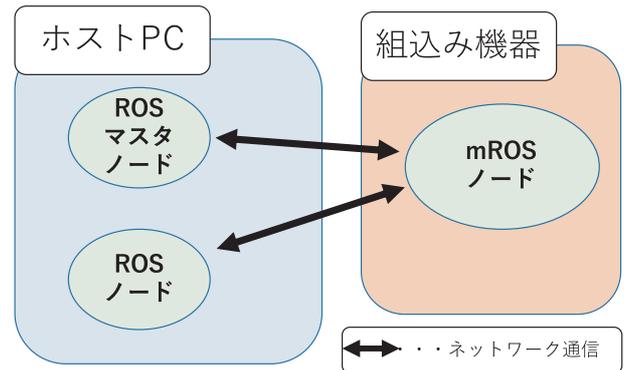


図 3 mROS を利用するシステムの例  
Fig. 3 Example of the system using mROS.

ノードの軽量実行環境である。mROS は、リアルタイム OS である TOPPERS/ASP カーネル [12]、および、TCP/IP プロトコルスタックを搭載可能な、ミッドレンジの組み込み機器を対象としている。ここで想定するミッドレンジの組み込み機器とは、次のような仕様を満たすものである。

- 数百 MHz 程度の動作周波数のプロセッサを持つ。
- 数 MB 程度の内臓メモリ（記憶領域）を持つ。
- ネットワーク通信のインタフェースを持つ。

現在は、ARM Cortex-A9 のプロセッサを搭載した、GR-PEACH をターゲットボードとして開発が進められている。

mROS のソフトウェア構成図を、図 2 に示す。mROS は、ROS ノードとの通信を実現する mROS 通信ライブラリを、TOPPERS/ASP カーネルの上で提供している。これを利用することにより、ROS のノードと通信を行うプログラムを、mROS アプリケーションとしてリアルタイム OS 上で実行することができる。また、ASP カーネルがサポートする mbed ライブラリのドライバ類も利用することができる。mROS では、mbed ライブラリが提供する軽量 TCP/IP プロトコルスタックである lwIP を使用している。

mROS を利用するシステムの例を図 3 に示す。図に示されるとおり mROS は、ブリッジを介することなく、ネットワークで接続された PC 上で実行される ROS ノードと直接通信を行うことができるという特徴を持つ。通信の際にブリッジを挟む必要がないため、ブリッジを挟む場合に比べて通信路の遅延が小さくなることが期待される。開発

言語は C++ が利用可能である。

### 3.2 課題点

本節では、本研究で考慮すべき mROS の課題点を議論する。

まず、組み込み機器および PC 間の通信速度がある。システムのリアルタイム性能を極力確保するためには、通信にかかる遅延がなるべく短く、その遅延時間の揺らぎが小さくなければならない。従来の mROS の実装では、通信において、文字列型のメッセージのみを扱うことができた。このため、文字列型以外のメッセージ型を利用する場合は、ユーザアプリケーション内で文字列型へのエンコード処理、および文字列型からのデコード処理を行う必要がある。このエンコード処理が、通信時の遅延時間およびその揺らぎの増大につながっている。

続いて、実行環境のプログラムサイズがあげられる。mROS の従来の実装では、通信時には文字列型メッセージのみを受付けるようにしている。これは、複数のメッセージ型に関する処理を実装に含めないことにより、mROS 通信ライブラリの高機能化およびプログラムサイズの肥大化を回避するためである。本研究で行う機能拡張により、通信ライブラリの実装が複雑となり、プログラムサイズが大きくなってしまふことが懸念される。

ROS ノードプログラムとの互換性および移植性も重要な要素である。mROS は、オープンソースで開発されている ROS のノードをそのまま実行できることを目標としている。このため、既存の ROS ノードプログラムとの互換性および移植性を意識した設計がなされている<sup>\*1</sup>。しかし、従来の mROS の実装では、通信に利用できるメッセージの型が文字列型に限定されていた。このため、文字列型以外のメッセージを利用するノードを用いる場合、利用するメッセージ型を文字列型に変換する処理を、ユーザによってアプリケーション上に実装する必要がある。したがって、開発作業が複雑かつ煩雑なものとなり、ROS の開発効率の良さを生かしたシステム開発ができなくなっている。

## 4. 提案手法

### 4.1 設計要件

前節で指摘した、通信時間、プログラムサイズ、および ROS ノードプログラムとの互換性の 3 つの課題を考慮しつつ、本研究で満たすべき設計要件を定める。

まず、ROS において扱うことのできるユーザ定義メッセージ型を含む、文字列型以外のメッセージ型による通信を可能とする。このために、以下の型を内部の変数として

保持するメッセージ型を、mROS において利用できるようにする。

- ROS のプリミティブ型 [14]
- プリミティブ型の配列
- 上記の型の変数を内部に持つ、ユーザ定義メッセージ型

文献 [2] では、送信可能なデータは QVGA フォーマットの画像データを扱えることを考慮し、512KB 程度まで扱えることを目標としている。また受信可能なデータは、制御命令データであることを想定し、1KB 程度までの大きさまでを目標としている。本研究においても、この目標を達成できることを目指す。

通信遅延およびその揺らぎについては、ユーザ定義型を利用する場合に関しては、mROS 従来実装のパフォーマンスを上回ることを目標とする。従来実装における型の変換および文字列型としての通信処理にかかる時間と比較し、本研究の成果物がより短い遅延時間で通信できることを目指す。また、文献 [2] では、リアルタイム性能確保のための議論もなされている。通信の遅延時間について、大きな外れ値が発生することなく一定の幅に収まることを目標とし、それを示すための検証も行っている。本研究においては、通信遅延時間の揺らぎが既存実装を上回らないことを目指す。さらに、メッセージのサイズが同等であれば、この揺らぎがメッセージの型によって大きく変化しないようにもする。

プログラムのサイズが大きくなりすぎると、mROS を搭載できる組み込み機器が、十分なメモリを持つ機器のみに限定される。前節で述べたとおり、任意のメッセージ型に対応できる仕様を採用する場合、プログラムのサイズが肥大化する可能性がある。このため、設計ではこの点を考慮し、プログラムのサイズが mROS の従来実装のものに比べて大きくならないようにする。特に ROS の通信ライブラリ内で用いられている boost ライブラリは、プログラムの過度な肥大化を招くため、これを回避するために用いないこととする。

効率の良いシステム開発のためには、mROS アプリケーションおよび ROS ノードの、移植性および互換性が高いことが重要である。移植性が高いと、mROS アプリケーションおよび ROS ノードプログラムの相互の移植が容易に行える。また、互換性が高いと、ROS を用いたシステム開発の経験者が、少ないコストで開発できる。移植性に関して、ROS ノードプログラムのコードを mROS アプリケーションの作成に使いまわすことのできるようにする。具体的には、ノードプログラムの実装のうち、メッセージの送受信動作のみを行う記述に関しては、コピー&ペーストにより移植できるほどの移植性を確保することを目指す。また、mROS アプリケーションと通信を行う ROS ノードプログラムの変更が不要であるようにもする。この移植性を

<sup>\*1</sup> ただし、mROS が固有にかかえる移植性の課題は存在する。特に、mROS で採用している TOPPERS カーネルは POSIX 非準拠であるため、この API を用いている ROS ノードプログラムを mROS 上で実行することは難しい。

```
float32 x
float32 y
float32 z
Time time
```

図 4 ユーザ定義メッセージ型の例

Fig. 4 Example of the user-defined message type.

確保することにより、両者の互換性も確保できる。

#### 4.2 実現方針

本章では、本研究の目的を実現するための方針について検討する。

まず、ROS におけるユーザ定義メッセージ型について説明する。ユーザ定義型は、プリミティブ型の変数、すでに定義されたメッセージ型の変数、および、それらの配列を内部変数として持つ構造をしている。

これを定義して利用するためには、まずメッセージの構造を決定する定義ファイルを作成する必要がある。メッセージ型を定義する定義ファイルの例を図 4 に示す。このメッセージ型は、位置情報およびその時刻を他のノードに送信するための、位置座標メッセージを想定して作成したものである。図では、x, y, および z の 3 種類の名前が割り当てられた浮動小数点数、および、time という名前が割り当てられた Time 型の変数を内部変数に持つメッセージ型の定義が示されている。Time 型は、ROS が提供する基本的なメッセージ型の 1 つである。内部で浮動小数点数型の変数により時刻情報が保持される。この定義ファイルをもとに、ROS のビルドツールからプログラムが生成される。これをノードプログラムで利用することにより、ユーザ定義メッセージ型を利用できるようになる。本研究の目標は、このような形式で定義されるユーザ定義型を、mROS アプリケーションで扱えるようにすることである。

ユーザ定義型を用いた ROS ノードとの通信を実現するために mROS 通信ライブラリが行うべき処理は、メッセージ型に固有の処理およびすべてのメッセージ型に共通な処理の 2 種類に大別できる。提案する mROS 通信ライブラリでは、メッセージ型に共通の処理からメッセージ型固有の処理を呼び出す実行フローを採用する。これにより、呼び出される処理をメッセージ型に応じて変更することで、通信で扱うメッセージ型を柔軟に変更できるようにする。

この処理仕様をより具体的に示す。まず、メッセージ型に固有の処理を、型ごとに定めるようにする。その処理を、メッセージ型 1 つに対しそれぞれ作成されるファイル内に含めるようにする。これにより、ファイル生成時に、利用しないメッセージ型のファイルに対し不要な変更を加えないようにする。

また、上述の処理仕様に基づくように mROS 通信ライブラリの具体的な動作フローを設計する。ROS ノードと

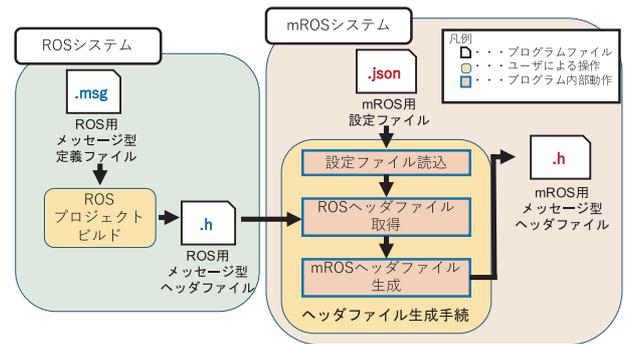


図 5 ヘッダファイル生成手法

Fig. 5 Operation flow of header file generation.

の通信のために mROS 通信ライブラリが行うべき動作は、出版および購読トピックの登録、メッセージの送信、メッセージの受信の 3 つに大別される。このそれぞれについて、実現のための方針を検討する。

出版および購読ノードの登録時は、ノード名、トピック名、および、トピックで通信するメッセージの型名など必要な情報をマスタへ送信する。このため、この必要な情報を返すメッセージ型固有の処理を定め、この処理を呼び出すことによりこれらの値を取得できるようにする。

メッセージ送信時は、メッセージはビット列に変換され、そのサイズを付加されたうえで共有メモリに格納される。その後、パブリッシュタスクにより共有メモリからメッセージのデータが取り出され、送信される。このうち、メッセージ型に応じたシリアライズ処理、および、メッセージのサイズ計算処理を、メッセージ型固有の処理として行えるようにする。

メッセージ受信時は逆に、まずメッセージをビット列として共有メモリへ受信する。その後、メッセージの型に応じたデシリアライズを行ったうえで、コールバック関数に受け渡される。このうち、メッセージのデシリアライズ処理をメッセージ型固有の処理として実行できるようにする。

#### 4.3 メッセージヘッダファイル生成手法

前章で述べた処理仕様を満たすよう、メッセージ型固有の処理をヘッダファイルとして生成する手法を提案する。

図 5 に、提案する手法のフローを示す。

図の左半分を示す緑色部分が ROS のシステムでのフローを、右半分を示す橙色部分が本研究で提案する mROS システム内でのフローを表す。mROS システム内において、ユーザが行う手順を黄色、プログラムなどで自動で実行させる手続き動作を赤色で表している。ROS のシステム内において、メッセージ型ヘッダファイル (.h 形式) は ROS プロジェクトのビルドを実行することにより、定義ファイル (.msg 形式) より生成される [14]。

ROS システム内で生成されたメッセージ型ヘッダファイルをもとに、mROS アプリケーション向けのメッセージ

型ヘッダファイルを生成する手順について説明する。まずユーザは、設定ファイルに、ROS ワークスペースのディレクトリ、ROS ライブラリのディレクトリ、および利用するメッセージ型の情報を記述する。続いて、設定ファイルに記述する ROS のワークスペースのディレクトリを基準に、利用するメッセージ型に対応する ROS システム内のメッセージ型ヘッダファイルを参照する。そしてこれをもとに、組み込みデバイス上での動作を想定した、mROS 専用のメッセージ型のヘッダファイルを生成する。まず ROS システム内のヘッダファイルから、型名、型の定義、および通信の整合性検証のためのハッシュ合計値を取得する。これらの値は、生成するヘッダファイル内にそのままコピーして利用する。さらに、メッセージ型の定義から型の内部変数の情報を取得し、それをもとにメッセージサイズの計算を行う処理、シリアライズ処理、および、デシリアライズ処理を生成する。シリアライズおよびデシリアライズのための関数は、メッセージ型の各内部変数に対し、変数の型に応じた処理を順番に行うように生成する。

メッセージ型の内部変数は、プリミティブ型、他のメッセージ型、Header 型およびその配列のいずれかである。Header 型はタイムスタンプおよびフレーム ID などを内部に含む、基本型の 1 つである。Header 型も、プリミティブ型の組合せにより構成されている。ゆえに、任意のメッセージ型は、プリミティブ型、Header 型、および、それらの配列の組合せで表現することができる。これは、メッセージ型が入れ子構造であっても、その内部変数であるメッセージ型を順にたどれば、いずれは上述した基本的な変数の組合せで表現できるからである。したがって、プリミティブ型、Header 型、およびそれらの配列に対する処理を生成できるならば、任意のメッセージ型に対し、そのシリアライズ処理およびデシリアライズ処理を行う関数を生成することができる。

#### 4.4 mROS 通信ライブラリ動作フロー

本節では、提案する mROS 通信ライブラリの具体的な動作フローについてそれぞれ説明する。

##### 4.4.1 出版および購読ノード登録処理

提案する、出版および購読ノード登録時の動作フローを以下に示す。

- (1) 扱うメッセージ型の種類を、関数呼び出し時の引数から判別する。
- (2) 通信に際して必要な情報を、その情報を返すメッセージ型固有の処理を呼び出すことにより取得する。
- (3) 取得した情報を XMLRPC 通信でマスタに送信する。

ノードの登録動作は、ノードおよびトピックに関する情報をマスタに送信する。この情報は、前章の手法により ROS システム内のファイルから抽出された値を利用し、この値を返すため専用の処理を呼び出すことにより取得す

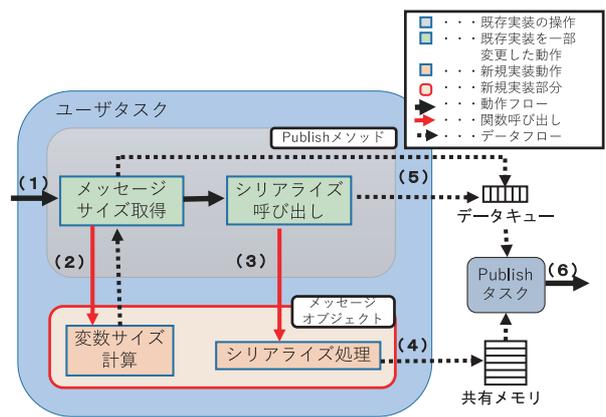


図 6 メッセージ送信時の動作フロー  
Fig. 6 Operation flow of message sending.

る。これは、値の取得方法をメッセージ型の種類によらない共通のものとするためである。

##### 4.4.2 メッセージ送信処理

提案するメッセージ送信時の動作フローを図 6 に示す。

図中において表される、動作の説明を以下に示す。

- (1) 送信するメッセージをオブジェクトとして受け取る。
- (2) 引数として受け取ったオブジェクトに対し、メッセージサイズ計算の関数を呼び出す。
- (3) 同オブジェクトに対し、共有メモリの所定箇所のポインタを引数として、シリアライズの関数を呼び出す。
- (4) 呼び出されたシリアライズの関数が、引数として渡されたポインタの指すメモリ領域に、メンバ変数の値をコピーする。その後、コピーした値のサイズに応じ、ポインタを動かす。
- (5) 前手順でメッセージの値をコピーした領域の先頭ポインタ、および、手順(2)で返されたメッセージ全体のサイズを Publish タスクのデータキューに格納する。
- (6) データキューにデータが格納されると、Publish タスクがただちに実行される

Publish タスクはデータキューから受け取る情報をもとに送信データを構成し、送信する。図中の灰色領域はメッセージ型に共通の処理（以下、共通処理と呼ぶ）、図中の橙色領域はメッセージ型に固有な処理（以下、個別処理と呼ぶ）を示す。またこの橙色領域に相当するプログラムは、前節に述べた手法により生成されたものを利用する。図中の赤矢印は、処理の呼び出しを表している。この赤矢印が示すとおり、共通処理が個別処理を呼び出す動作フローとなっている。メッセージ内容の共有メモリへの書き込みも個別処理が行う。個別処理を呼び出した共通処理へは、メッセージのサイズのみが返される。これは、扱われるメッセージ型にかかわらず整数型の値である。これにより、共通処理と個別処理との結合度を下げている。これにより、共通処理には変更を加えず、個別処理を所望のメッセージ型に対応するものに差し替えるだけで、その型

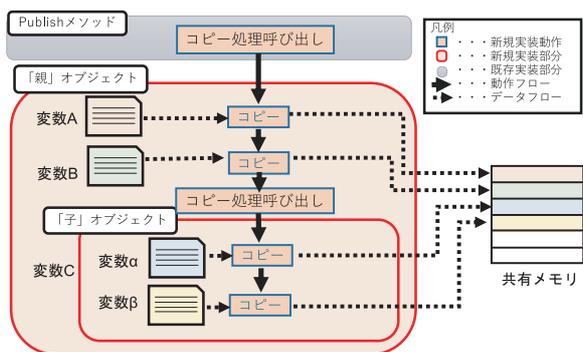


図 7 入れ子メッセージの処理動作フロー

Fig. 7 Operation flow of processing of nested messages.

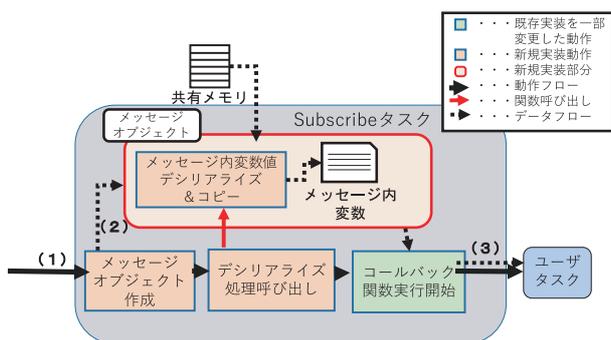


図 8 メッセージ受信時の動作フロー

Fig. 8 Operation flow of message receiving.

に関する処理が行えるようにする。

またこのフローにより、入れ子構造を持つメッセージ型の処理も実現することができる。ここでメッセージ型オブジェクトを「親」、その型内部の1変数型として利用されているメッセージ型オブジェクトを「子」とする。この「親」をシリアライズする際の動作フローを例として説明する。この動作を表す図を、図 7 に示す。

図が示すとおり、Publish メソッドが親オブジェクトに対して行うシリアライズ処理呼び出しと同様の形式で、親オブジェクトが子オブジェクトのシリアライズ処理を再帰的に呼び出す。また、親オブジェクトと子オブジェクトで、共有メモリ領域を指すポインタを共有する。これにより、入れ子構造を持つ型のシリアライズ処理を実現する。

#### 4.4.3 メッセージ受信処理

提案するコールバック処理に関連する一連の動作を表す図を図 8 に示す。また、行われる動作の詳細を以下に示す。

- (1) 受信したメッセージの型に対応するオブジェクトを生成する。
- (2) 作成したオブジェクトに対し、受信したデータが格納される共有メモリへのポインタを引数として、デシリアライズ関数を呼び出す。デシリアライズ関数内では、ポインタが示す値をメッセージオブジェクトに順に格納する。
- (3) メッセージオブジェクトへのポインタを引数とし、関

数ポインタの指す関数を実行する。

図中の「コールバック関数実行開始」と書かれた矩形に相当する処理は、デシリアライズにより変数値が適切に設定されたメッセージオブジェクトを、所定のコールバック関数に渡す動作である。この動作に向けて伸びる破線矢印は、この動作の前々手順で作成されたメッセージオブジェクトを利用することを意味している。デシリアライズ関数からの返り値を利用するというのではない。送信時同様、デシリアライズ処理および他の処理との結合度が低く、メッセージ型に対し柔軟な処理が行える。また、Publish メソッドにおけるシリアライズ処理同様、再帰的な手続き呼び出しによる、入れ子構造を持つメッセージ型のデシリアライズも可能である。

## 5. 評価

本章ではまず、本研究において行う性能評価における、評価項目および評価環境について説明する。続いて、評価結果を掲載し、結果をもとに議論をする。

### 5.1 評価項目

評価項目は、有効性確認、通信速度、プログラムサイズ、および移植性の4項目とした。

有効性確認では、ユーザ定義型を含む任意のメッセージ型による通信が行えるようになったことを、実際にメッセージの送受信を行うことにより検証した。通信速度評価では、メッセージの通信の際に発生する遅延時間を計測した。プログラムサイズの評価では、組込み機器に搭載されるプログラムファイルのサイズを計測した。移植性評価では、既存の ROS ノードプログラムを組込み機器上へ移植する際に、プログラムにどれだけ変更を加える必要があるかを計測した。

### 5.2 評価環境

評価対象の環境は、mROS および roserial の2つとした。両環境とも、ホストとしてノート PC を用意し、そのうえで ROS マスタノードおよび評価用の ROS ノードを実行した。ノート PC の環境は以下のとおりである。

- Core i7 4500U (1.8 GHz)
- Ubuntu 16.04 LTS
- ROS Kinetic

また、組込み機器には GR-PEACH [13] を使用した。環境は、以下のとおりである。

- Cortex-A9 (400 MHz)
- TOPPERS/ASP Kernel 1.9.2
- mROS v1.1
- roserial (jade-dev ブランチ)\*2

\*2 roserial の公式ドキュメントには、Kinetic では jade-dev ブランチのバージョンを使用するよう記載されている。

実験時は、ノート PC および GR-PEACH を、イーサネットケーブルによりルータを介して同一ネットワークに接続した。当該ネットワークには、ノート PC および GR-PEACH の 2 機器のみが接続された状態とした。また、rosserial 使用時のシリアル通信のボーレートは 115,200 bps とした。

### 5.3 有効性確認

ROS のプリミティブ型を扱う場合について有効性の確認を行ったところ、内部にメソッドを持つ Time 型および Duration 型を除くすべての型で動作することを確認した\*3。また、プリミティブ型を組み合わせたユーザ定義メッセージ型についても、すべての組合せについて確認してはいないものの、確認した限りのものではすべて正常に動作した。ここでは、本文献中で例にあげていたユーザ定義メッセージ型である Coordinate 型、および、入れ子構造および配列構造を内部に持つメッセージ型である Image 型についての、有効性確認の結果を示す。

#### 5.3.1 ユーザ定義メッセージ型

ユーザ定義メッセージ型の有効性確認では、図 4 に示した 3 次元座標情報を格納するユーザ定義メッセージ型を用いた。この型を Coordinate 型と名付け、以降はこの名称を使用する。評価は、mROS から ROS への送信、および、ROS から mROS への送信の 2 方向に対して行った。組込み機器およびホスト PC のうち、一方で送信ノードを、他方で受信ノードを実行し、両者の間で 1 方向の通信を行わせた。受信ノードで受信したメッセージを表示し、送信ノードから送信したものと一致していることを確認した。評価に用いた、Coordinate 型のメッセージの送信を行うプログラムコードを図 9 に、受信を行うコードを図 10 に示す。ROS ノードおよび mROS アプリケーションのプログラム実装は、その大部分が共通している。このため、両者のものをまとめて掲載する。

まず、mROS から ROS への送信評価を行った。結果の出力を、図 11 に示す。

図 11 に示すとおり、ホスト PC 上の ROS ノードにおいて、Coordinate 型のメッセージが正しく受信できていたことが分かる。図 9 に示すとおり、送信ノードは送信のたびに各座標を表す変数の値を更新している。図 11 におけるこの値の出力を見ると、この値の更新処理が反映されていることが読み取れる。このことから、mROS アプリケーションにおいて格納した値が正しく送信できていたことも確認できる。

次に、ROS から mROS への送信評価を行った。結果の出

```
void pub_task(){
  ros::init(argc,argv,"mros_node");
  ros::NodeHandle nh;
  ros::Publisher chatter = nh.advertise
    <mros_comm_measure::Coordinate>("ping",1);
  mros_comm_measure::Coordinate msg;
  while(true){
    msg.x += 0.1;
    msg.y += 10.03;
    msg.z += 3.003;
    get_utm(&start_time);
    msg.time.sec = (int)(start_time/1000000);
    chatter.publish(msg);
    wait_ms(1000);
  }
}
```

図 9 Coordinate 型の送信を行うコード

Fig. 9 Code to send messages of Coordinate type.

```
void sub_task(){
  ros::init(argc,argv,"mros_pub_node");
  ros::NodeHandle nh;
  ros::Subscriber sub = nh.subscribe("pong",1,callback);
  ros::spin();
}
void callback(mros_comm_measure::Coordinate *msg){
  ROS_INFO("I heard [x:%d,...", (int)(msg->x * 1000),...);
}—
```

図 10 Coordinate 型の受信を行うコード

Fig. 10 Code to receive messages of Coordinate type.

```
I heard[x: 7.899995, y: 792.370789, z: 237.237167] at 80 sec
I heard[x: 7.999995, y: 802.400818, z: 240.240173] at 81 sec
I heard[x: 8.099995, y: 812.430847, z: 243.243179] at 82 sec
I heard[x: 8.199995, y: 822.460876, z: 246.246185] at 83 sec
I heard[x: 8.299995, y: 832.490906, z: 249.249191] at 84 sec
I heard[x: 8.399996, y: 842.520935, z: 252.252197] at 85 sec
I heard[x: 8.499996, y: 852.550964, z: 255.255203] at 86 sec
I heard[x: 8.599997, y: 862.580994, z: 258.258209] at 87 sec
I heard[x: 8.699997, y: 872.611023, z: 261.261200] at 88 sec
I heard[x: 8.799997, y: 882.641052, z: 264.264191] at 89 sec
```

図 11 ホスト PC での Coordinate 型の受信結果

Fig. 11 Result of message receiving on the host PC.

```
I heard [x:3899, y:292870, z:90086] at 10 sec
I heard [x:3999, y:302900, z:93089] at 10 sec
I heard [x:4099, y:312930, z:96092] at 10 sec
I heard [x:4199, y:322960, z:99095] at 10 sec
I heard [x:4299, y:332990, z:102098] at 10 sec
```

図 12 mROS 上での Coordinate 型の受信結果

Fig. 12 Result of message receiving on the mROS.

力を、図 12 に示す。図 12 より、GR-PEACH 上の mROS アプリケーションにおいて、Coordinate 型のメッセージが正しく受信できていたことが分かる。ASP カーネルのログ出力では、浮動小数点数を表示することができない。このため、受け取った浮動小数点数の値を 1,000 倍し、整数として出力した。座標を表す値を確認すると、メッセー

\*3 たとえば、ROS の Time 型には Time::now() というメソッドが存在し、これを用いることによりシステム時刻を取得できるようになっている。本研究では、このようなメソッドは対象外としている。

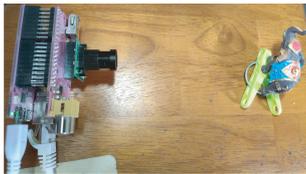


図 13 GR-PEACH を用いて画像を取得した様子  
Fig. 13 Taking images with GR-PEACH.



図 14 mROS により取得した画像  
Fig. 14 Image sent from mROS.

ジごとに更新されていたことが見て取れる。これより、送信ノードが行うメッセージ値の更新処理が正しく出力結果に反映されているといえる。このことから、ROS ノードから送信した値が mROS アプリケーションで正しく受信できていたことが確認できる。

### 5.3.2 Image 型

続いて、ROS の `sensor_msgs` パッケージにより提供される `Image` 型を用い、カメラ画像の送信テストを行った。`Image` 型は、画像データの画素を配列として保持するほか、`width` および `height` などプリミティブ型のパラメータ、および `Header` 型の変数を含んでいる。したがって、`Image` 型の送信をもって、入れ子構造および配列を持つメッセージ型が扱えるようになったことを確認できる。このノードを実行する様子を図 13 に、受信した画像を ROS が提供する `image_view` ノードで表示した結果を図 14 に示す。図 13 に示すとおり、GR-PEACH のカメラに映るように、象の模型を配置した。図 14 を見ると、GR-PEACH のカメラから取得した画像がホスト PC 上の `image_view` ノードで確かに受信できていたことが読み取れる。

## 5.4 通信遅延

それぞれの実行環境を用いてメッセージの送受信を行った場合に発生する、通信遅延時間を測定した。

### 5.4.1 評価方法

測定に用いたシステム構成を表す図を図 15 に示す。図に示すとおり、システムは送信ノード、受信ノード、およびエコーバックノードの 3 ノードから構成した。これらのノードは、`ping` および `pong` の 2 トピックを用いて通信を行った。このシステムでは、まず送信ノードが `ping` トピックに出版し、エコーバックノードに向けてメッセージを送信する。メッセージを受信したエコーバックノード

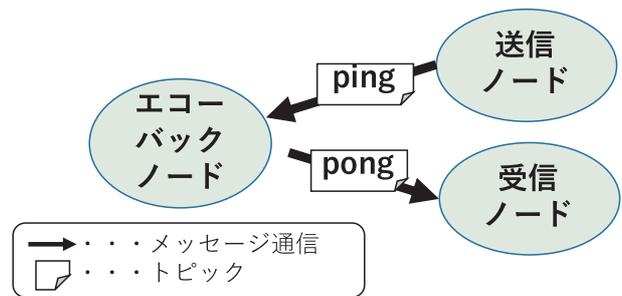


図 15 通信速度の評価に用いるシステムの構成  
Fig. 15 System structure for the evaluation of communication speed.

は即座にメッセージを `pong` トピックに出版し、受信ノードに向けてメッセージを送信する。評価する通信遅延時間は、送信ノードからメッセージの送信を開始する際の時刻、および、受信ノードがメッセージを受信した直後の時刻の差分とした。すなわち、送信ノードからホスト PC にメッセージを送信して、それが返送されるまでにかかるラウンドトリップ時間 (RTT) を測定対象の時間とした。評価では、エコーバックノードを組込み機器に、送信ノードおよび受信ノードをホスト PC に搭載した。また時刻の計測および評価値の計算は、ホスト PC 上で行った。

評価は、文字列型に関するもの、および、ユーザ定義メッセージ型に関するものの 2 通りを実施した。

文字列型を用いた評価では、サイズを様々なに変化させた文字列を利用した。評価では、まず送信ノードからエコーバックノードに向けて 1B の文字列が送信された。文字列を受信したエコーバックノードは、即座に受信ノードに向け、文字列を返送した。この返送される文字列は、1B-32,768B まで、試行ごとに 2 の冪乗で変化するようにした。これにより、様々なサイズのメッセージに対する通信遅延時間の変化を評価した。

ユーザ定義メッセージ型を用いた評価では、前節の有効性確認評価でも用いた `Coordinate` 型を利用した。評価では、まず送信ノードからエコーバックノードに向けて、`Coordinate` 型のメッセージが送信された。メッセージを受信したエコーバックノードは、即座に同内容のメッセージを受信ノードに向けて返送した。このとき、mROS 従来実装の評価においては、`Coordinate` 型および文字列型間の変換時間も評価対象の遅延時間に含めるものとした。すなわち、送信ノードで送信前に文字列へと変換する時間、エコーバックノードで受け取った文字列を `Coordinate` 型に変換し、さらにもう 1 度文字列型に変換し直す時間、および、受信ノードで受け取った文字列を `Coordinate` 型に変換する時間も評価値に含まれるようにした。これにより、型のエンコード通信処理時間に与える影響を評価した。

実際の評価においては、図 15 に示すシステムを実現するプログラムを実行環境ごとに実装し、それを用いて行っ

表 1 1 B-512 B 文字列利用時の通信遅延時間の平均値 [ $\mu\text{sec}$ ]

Table 1 Average communication delay time with 1 B-512 B sized string messages.

	1 B	2 B	4 B	8 B	16 B	32 B	64 B	128 B	256 B	512 B
rosssserial	9,133.36	9,121.31	9,387.14	9,575.56	10,031.84	11,615.23	14,237.36	19,861.96	30,803.89	30,878.46
従来実装	1,149.32	1,059.83	1,181.51	1,196.54	1,184.94	1,123.38	1,173.78	1,159.18	1,178.44	1,245.17
新実装	1,029.19	1,043.86	1,075.62	1,044.64	1,023.44	1,051.23	1,076.88	1,087.85	1,114.79	1,170.59

表 2 1,024 B-32 KB 文字列利用時の通信遅延時間の平均値 [ $\mu\text{sec}$ ]

Table 2 Average communication delay time with 1,024 B-32 KB sized string messages.

	1,024 B	2,048 B	4,096 B	8,192 B	16,384 B	32,768 B
従来実装	1,395.18	2,085.42	2,547.91	3,797.71	5,962.55	13,409.56
新実装	1,318.13	1,913.39	2,379.44	3,539.96	5,599.99	15,983.26

表 3 Coordinate 型使用時の通信遅延時間の平均値 [ $\mu\text{sec}$ ]

Table 3 Average communication delay time with Coordinate type messages.

rosserial	11,572.43
mROS 従来実装	1,597.27
mROS 新実装	995.46

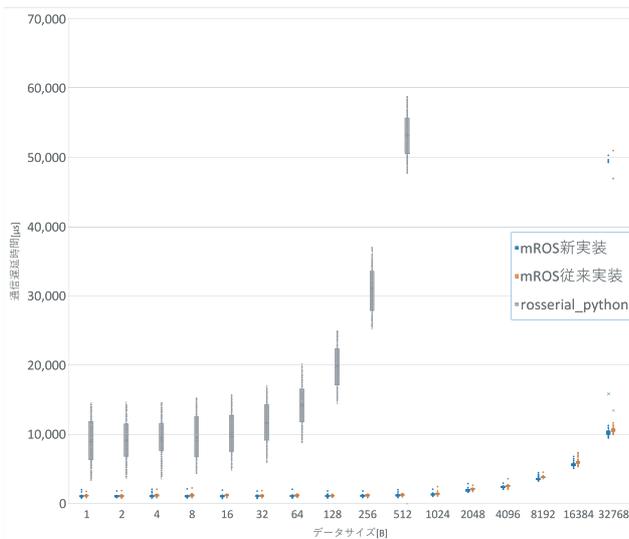


図 16 文字列型について行った通信時間評価結果

Fig. 16 Evaluation result of communication speed with string type messages.

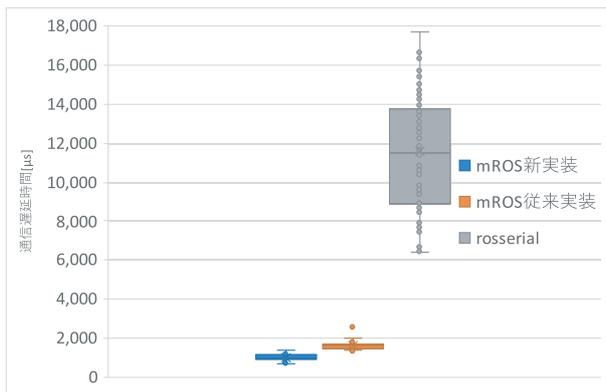


図 17 Coordinate 型について行った通信時間評価結果

Fig. 17 Evaluation result of communication speed with Coordinate type messages.

た. mROS では, エコーバックノードに相当するタスクが, ROS ノードと直接ネットワーク通信をするようにした. 一方 rosserial では, rosserial クライアントアプリケーションにエコーバックノードの機能を実装したうえで, rosserial

サーバを介して通信を行うようにした.

時刻の取得には, 両環境ともに, C++の `std::chrono::high_resolution_clock::now()` 関数を用いた.

なお, 計測結果に影響を及ぼさないようにするため, 評価中はデバッグ出力など評価に関係のない処理および出力は行わせないようにした.

#### 5.4.2 評価結果

文字列および Coordinate 型について評価を行い, 結果を箱ひげ図としてプロットしたものを図 16 および図 17 に示す. いずれの図も, 縦軸に計測した通信時間 (RTT) を  $\mu$  秒単位で, 横軸に送信した文字列のサイズをバイト単位で示している. また, それぞれの試行における遅延時間の平均値を, 表 1, 表 2 および表 3 に示す.

rosserial においては, 1,024 B 以上のメッセージを扱うことができなかつたため, その結果は掲載していない. それ以外のサイズのメッセージ送受信に関しては, すべての環境でのすべての試行において, 送信した 200 メッセージすべてを失うことなく受信した.

#### 5.4.3 通信遅延評価についての議論

文字列型の送信に関しては, 遅延時間およびその揺らぎに関して, 1 KB 未満の文字列であればおよそ 1 ms 前後で送信できていたことが読み取れる. 一方, 16 KB を超えるデータの送信では遅延時間が非常に大きくなっていた. 特に 32 KB のデータの送信においては, 平均値から大きく外れた値も存在していた. ROS において, 16 KB を超えるデータの通信時にはパケットの分割が発生することが文献 [3] で指摘されている. 本評価においても同様の事態が発生し, パケット分割の影響で遅延が大きくなっていたと

考えられる。このため、リアルタイム性能をより確保するためには、メッセージのサイズが16KBを上回らないことが好ましいといえる。mROSの新実装および従来実装を比較すると、文字列の送信に関して、新実装の方が通信性能が若干向上していたことが分かる。この原因の1つとして、本研究に基づく機能拡張を行う際に、通信ライブラリの一部機能が意図せずに最適化されたことが考えられる。また、`rosserial`による送信結果と比較すると、非常に短い時間かつ小さい揺らぎで送信できていたことが分かる。これは、シリアル通信およびネットワーク通信という通信路の違いが結果に大きく反映されているためと考えられる。

`Coordinate`型を用いた場合に関しては、本研究による提案をもとにユーザ定義メッセージ型による通信を行った場合、遅延時間の小ささの観点では最も良い性能となった。遅延時間の揺らぎも小さくなっているが、この点はmROS従来実装のものとはあまり大きな差はない。ここから、文字列型で送信した場合およびユーザ定義型で送信する場合、通信遅延時間の揺らぎが変化するわけではないといえる。

組込み機器に搭載するアプリケーションにおいて、数値から文字列への変換処理は、`ostream`を、文字列から数値への変換は`sscanf`命令を用いて行った。ホストPC上のROSノードでは、数値から文字列への変換処理は`to_string`命令を、文字列から数値への変換処理は組込みアプリケーション同様`sscanf`命令を用いて行った。`ostream`より`to_string`を用いた方が高速に処理できると考えられる。しかし、組込みアプリケーション向けにコンパイルすることができなかったため、今回は採用しなかった。この部分が、型エンコード処理の中でも特に大きなオーバーヘッドとなっていた可能性が考えられる。`rosserial`では、`Coordinate`型を用いた通信を行うことができた。しかし、遅延時間およびその揺らぎの両方とも、mROSを用いた場合に比べて大きくなった。

これらの結果より、mROSが搭載可能なミッドレンジの機器を採用する場合は、本研究の提案に基づいたmROSの新実装が、より良い通信性能を発揮できるといえる。

## 5.5 プログラムサイズ

### 5.5.1 評価方法

組込み機器に搭載する実行可能ファイルのプログラムサイズを比較した。評価対象のファイルは、ビルドにより生成され、`objcopy`コマンドでバイナリファイルに変換されるファイルを利用した。mROSであれば`asp.bin`が、`rosserial`であれば`asp`がこれに該当する。評価ではLinuxの`size`コマンドを用いて、各ファイルの各セクションごとのプログラムサイズを取得した。

評価は、通信性能の評価においてGR-PEACHに搭載したものと同一アプリケーションのビルド生成物を使用した。

表4 プログラム全体のサイズ評価結果

Table 4 Evaluation result of the program file.

	text [Byte]	data [Byte]	bss [Byte]	dec [Byte]
rosserial	125,860	2,660	63,012	191,532
従来実装	427,684	2,964	4,693,728	5,124,376
新実装	418,500	2,932	3,645,156	4,066,588

すなわち、ホストPCと通信を行う、エコーバックノードのプログラムを評価対象とした。

### 5.5.2 評価結果

ビルド生成物のプログラムファイルについて評価した結果を表4に示す。

### 5.5.3 議論

表4に示す結果のうち、始めに従来実装および本研究に基づく新実装の結果を比較する。

まず、新実装の結果は従来実装のものに比べ、`bss`セクションの値が大きく縮小された。これは、通信に用いられる共有メモリ領域が一部メッセージサイズに合わせて最適化されたためである。従来実装では、可変長の文字列および画像などの、サイズが大きいデータを扱うことを想定していた。そのため、それらのデータも扱えるよう、通信時に用いられる共有メモリ領域を大きく確保していた。一方、新実装では、この領域が縮小された。本研究に基づく機能拡張をmROSに施す際、扱うメッセージ型が固定長であれば、そのサイズに合わせてコンパイル時に共有メモリ領域を自動で調整するフローを実装した。可変長配列など、メッセージ型の大きさが特定できない場合は、ユーザにより確保するメモリ領域の大きさを指定できるようにした。本評価で扱った`Coordinate`型は固定長のメッセージ型であったため、このフローによって共有メモリ領域の一部が、型のサイズに合わせて縮小されたと考えられる。また、`text`セクションの値も若干減少していたことが分かる。これは、要件定義の節で述べた懸念に反する結果となった。考えられる原因としては、型のエンコード処理の挿入により、従来実装のプログラムサイズが若干増大しているということがあげられる。また機能拡張を行う際に、コードの一部が意図せず最適化された可能性も考えられる。

新実装のプログラムでも、依然として3.6MBほどの大きな`bss`領域を確保していた。この問題は、さらなるプログラムの最適化により解消が可能であると考えられる。しかし、新実装のプログラム全体のサイズは、現段階ですでに5MBを下回っていた。このため新実装のmROSは、ターゲットとするミッドレンジの組込み機器に搭載することができるサイズに収まっているといえる。

次に、新実装および`rosserial`を比較する。`rosserial`のプログラムサイズは全体でおよそ200KBほどであった。これは、mROSのサイズを大きく下回るものであった。`rosserial`の評価対象のファイルには、ASPカーネルのプ

表 5 移植性コストの評価結果

Table 5 Evaluation result of porting cost.

	エッジ			ホスト			合計
	追加	変更	削除	追加	変更	削除	合計
rosserial pub	1	6	1	0	0	0	8
rosserial sub	3	4	1	0	0	0	8
従来実装 pub	19	3	0	2	1	0	25
従来実装 sub	13	1	0	2	1	0	17
新実装 pub	2	2	0	0	0	0	4
新実装 sub	2	2	0	0	0	0	4

ログラムも含まれる。これより、rosserial をベアメタルで実行する場合、さらに小さいプログラムサイズで実現されると考えられる。一方 mROS は、RTOS および TCP/IP プロトコルスタックを必要とする。このため、実行環境は rosserial のものより大きくなった。このことは、両環境の text セクションの評価結果に 300 KB ほどの差があったことから確認できる。

## 5.6 移植コスト

### 5.6.1 評価方法

既存の ROS ノードプログラムを組込みソフトウェアに移植する際にかかる、移植コストを評価した。評価では、図 4 に示したユーザ定義型である、Coordinate 型の送受信を行うプログラムを評価対象とした。このプログラムを、まず ROS のノードとして実装した。その後、そのプログラムを mROS の従来実装、提案に基づく mROS の新実装、および rosserial に向けて移植した。その際に、プログラムをコピー&ペーストしてから、どれほどの変更を加える必要があるかを評価した。具体的には、コピー&ペーストしたコードに対し、追加、削除、および変更をしたプログラムの行数をカウントし、それを移植コストの評価値とした。また、組込み機器に移植したアプリケーションと通信を行う ROS ノードプログラムにも変更を加える必要がある場合もある。この場合、その移植コストも評価値に含めるものとした。評価は、mROS の従来実装、本研究の成果に基づく mROS の新実装、および、rosserial を対象として行った。

### 5.6.2 評価結果

評価結果を表 5 に示す。表のうち、見出しに pub という単語がついている行が送信ノードの移植結果を、sub という単語がついている行が受信ノードの移植結果を表している。

### 5.6.3 議論

mROS の従来実装および新実装を比較すると、新実装では従来実装に加え、移植コストが抑えられていたことが分かる。従来実装では、メッセージ型をプログラムコード内で利用するためには、メッセージ型のクラスの定義、および、文字列型とのエンコード処理を追加する必要があっ

た。さらに、mROS 通信ライブラリの API および ROS 通信ライブラリの API の相違点を埋める変更も必要であった。加えて、従来実装の場合、組込み機器上のプログラムと通信を行う ROS ノードに対しても変更を加えなければならなかった。これらの移植コストの高さは、移植作業を煩雑にし、また移植時のエラー発生率を高めていると考えられる。一方、本研究の成果に基づく mROS の新実装では、上述したような変更が不要となった。これにより、表に示す結果のとおり、低い移植コストでアプリケーションの移植が行えるようになっている。

mROS の新実装および rosserial の両者を比較すると、rosserial では mROS より多い移植コストが必要であったことが分かる。mROS は、通信ライブラリの実装がやや複雑である代わりに、通信ライブラリのインタフェースは ROS ノードプログラムのもと同じになるように設計されている。一方 rosserial の提供する通信ライブラリのインタフェースは、ROS ノードプログラムのもとは異なる点が多い。これは、ROS ノードプログラムのインタフェースと同一のものにすることよりも、ライブラリの内部実装を簡潔にすることを優先したためと考えられる。この結果、ROS ノードのプログラムを rosserial のクライアントアプリケーションとして移植する際には、通信の設定および呼び出しを行う箇所はすべて変更を加える必要がある。一方 mROS のアプリケーションとして移植する場合は、この箇所の修正は必要ない。

## 6. 結論

本研究は、ROS ノードの軽量実行環境である mROS の汎用性向上を目的とし、通信メッセージ型に関する制約の解消を目指した。このための、メッセージ型ヘッダファイル生成手法および mROS 通信ライブラリ動作フローを提案した。提案手法は、メッセージ型に対応するヘッダファイルを、ROS システム内から取得した情報によって生成するものである。また、動作フローは任意のメッセージ型の処理に対応するため、ヘッダファイル内に定義されたメッセージ型固有の処理を呼び出す構造とした。提案内容に基づく機能拡張を mROS に実装し、性能評価を実施した。有効性の確認により、ユーザ定義のメッセージ型を含む任意のメッセージ型が、mROS において利用可能となったことを確認した。さらに通信時間、プログラムサイズ、および移植性の 3 項目に関して定量的評価を行った。評価では、mROS の既存実装および他手法 rosserial との比較も行った。本研究で行った提案により、mROS を活用したシステム開発がより容易となった。これにより、ROS をベースとするシステムへの、組込み技術の導入がより促進される。また、mROS の汎用性も向上し、多様なシステム開発において活用することができるようになった。

今後の課題としては、mROS の実装を改善することによ

る性能向上, および他手法とのさらなる詳細な比較評価などがあげられる. 特に実装に関しては, 現時点で対応できていないユーザ定義メッセージ型を要素に持つ配列型への対応, および, カーネルが提供する共有メモリの活用によるメッセージ変換処理の改善があげられる.

謝辞 本研究は, JST さきがけ JPMJPR18M8 および JSPS 科研費 JP18K18024 の支援を受けたものである.

#### 参考文献

- [1] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R. and Ng, A.Y.: ROS: An opensource Robot Operating System, *ICRA Workshop on Open Source Software*, Vol.3, p.5 (2009).
- [2] Takase, H., Mori, T., Takagi, K. and Takagi, N.: mROS: A Lightweight Runtime Environment of ROS 1 nodes for Embedded Devices, *Journal of Information Processing*, Vol.28, pp.150–160 (Feb. 2020).
- [3] Maruyama, Y., Kato, S. and Azumi, T.: Exploring the Performance of ROS2, *Proc. 13th International Conference on Embedded Software*, p.5, ACM (2016).
- [4] Ferguson, M.: roserial, available from (<http://wiki.ros.org/roserial>.)
- [5] Zhengang, L., Yong, X. and Lei, Z.: ROS-Based Indoor Autonomous Exploration and Navigation Wheelchair, *2017 10th International Symposium on Computational Intelligence and Design (ISCID)*, Vol.2, pp.132–135 (2017).
- [6] Patrick, B., Mohan, M., Paul, R., John, J.P., Mo, J. and Lutcher, B.: Cloud-Based Realtime Robotic Visual SLAM, *Proc. 2015 Annual IEEE Systems Conference (SysCon)*, pp.773–777 (2015).
- [7] Bezemer, M.M. and Broenink, J.F.: Connecting ROS to a real-time control framework for embedded computing, *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*, pp.1–6, IEEE (2015).
- [8] Tiago, P., Rafael, A. and Germano, V.: Bridging Automation and Robotics: An Interprocess Communication between IEC 61131-3 and ROS, *2018 IEEE 16th International Conference on Industrial Informatics (INDIN)*, pp.1085–1091 (2018).
- [9] ROS Index: ROS 2 Overview, available from (<https://index.ros.org/doc/ros2/>)
- [10] micro-ROS, available from (<https://micro-ros.github.io/>)
- [11] eProsima: Micro-XRCE-DDS, available from (<https://github.com/eProsima/Micro-XRCE-DDS>)
- [12] TOPPERSproject: TOPPERS/ASP kernel, available from (<https://www.toppers.jp/asp-kernel.html>.)
- [13] ルネサスエレクトロニクス: GR-PEACH, 入手先 (<http://gadget.renesas.com/ja/product/peach.html>.)
- [14] msg: ROS wiki, available from (<http://wiki.ros.org/msg>)



祐源 英俊 (学生会員)

2019年京都大学工学部情報学科卒業. 同年同大学院情報学研究科通信情報システム専攻修士課程入学. 組込みリアルタイムシステムの実行環境に関する研究に従事.



高瀬 英希 (正会員)

2012年名古屋大学大学院情報科学研究科博士課程後期課程修了. 博士(情報科学)取得. 2009~2012年日本学術振興会特別研究員 DC1. 2012年京都大学大学院情報学研究科助教, 2019年より准教授. 2018年より科学技術振興機構さきがけ研究者, 現在に至る. 組込みリアルタイムシステムの実行環境およびシステムレベル設計技術に関する研究に従事. 最近の興味は関数型言語によるIoTシステムの設計手法. 本会2007年度コンピュータサイエンス領域奨励賞, 船井情報科学振興財団船井研究奨励賞(2015年4月)等を受賞. 電子情報通信学会, 日本ロボット学会各会員.