Vol.2022-ARC-248 No.36
Vol.2022-SLDM-198 No.36
Vol.2022-EMB-59 No.36
2022/3/10

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

菅 文人[1,a)]　松原 克弥[1,b)]

概要：高性能なクラウドサーバ（以降，クラウド）とロボットが連携することで，より高度な制御・知能処理を実現するクラウドロボティクスでは，計算負荷，ネットワーク状況やハードウェア制約などを考慮して，クラウドとロボットへタスクを分散配置することがシステム実現の要となる．しかし，実世界で稼働するロボットは，周辺環境変化の影響を受けて変化するバッテリー残量やネットワーク状況などを高精度に予測することが難しいため，あらかじめ最適なタスク配置を静的に決めることが難しいという課題がある．このような状況変化に合わせて，クラウドとロボット間でタスクを動的に再配置することが望まれるが，クラウドとロボットの CPU アーキテクチャは異なる場合が多く，CPU のレジスタ構成や命令セットの違いにより，実行中のタスクを異種デバイス間で移行することは容易ではない．本研究では，クラウドロボティクスで動作する ROS ノードを対象とした異種デバイス間マイグレーション機構の実現手法を検討する．既存の ROS ノードプログラムを WebAssembly にコンパイルするための WebAssembly 用 ROS フレームワークを実装することで，クラウドとロボットの両方で動作する ROS ノードを実現する．さらに，WebAssembly 仮想マシンに仮想マシン実行状態の保存復元機構を実装することで，実行状態をともなった ROS ノードの動的マイグレーションを可能にする．

キーワード：マイグレーション，CPU アーキテクチャ，クラウドロボティクス，Robot Operating System，WebAssembly，仮想マシン

1. はじめに

ロボット導入の分野拡大や通信インフラの充実化にともなって，クラウドとロボットが連携することで高度な制御・知能処理を実現するクラウドロボティクスが注目されている．ロボットシステムのソフトウェア処理は「センサ」,「知能・制御系」,「駆動系」の三要素に大きく分類することができる [1] が，クラウドロボティクスを採用したシステム（以降，クラウドロボット・システム）では，ハードウェアへのアクセスが必要なセンサ系や駆動系のタスクをロボット側で実行し，計算処理能力の高いクラウド側で知能・制御系のタスクを実行するように分担することが多い．特に，高負荷な知能・制御系の処理をクラウド側に配置することで，限られた計算性能しか持たないロボットでも，高度な機能が実現できるようになる．Robot Operating System（以降，ROS）は，各処理をノードという単位で独立実装して，ネットワークで連携させることで実装を行うプログラミングモデルを採用している．前述のクラウドロボット・システムでは，各 ROS ノードをクラウドとロボットに分散配置することで柔軟にシステムを構成できる．

現在の ROS は，各ノードの配置をシステム起動時に指定する必要があるため，クラウドロボット・システムにおけるクラウドとロボットの間の最適な処理分担を静的に決める必要がある．特に，CPU アーキテクチャが異なることが多いクラウドとロボット間では，実行状態を持つ ROS ノードの処理をシステム稼働中に移行させることは難しい．しかし，実空間で動作するロボットは，周辺環境変化や通信回線，バッテリー残量などのシステム起動時に想定しきれない影響を受けることがあり，起動時に設定したクラウド-ロボット間の処理分担が最適ではなくなる可能性がある．

本研究は，クラウドとロボットの各プラットフォームで動作する ROS ノードの配置をシステム稼働中に動的に変更できるようにすることで，様々な状況変化に柔軟に対応できるクラウドロボット・システムを実現することを目的とする．本稿では，CPU アーキテクチャが異なるクラウドとロボット間で，ROS ノードを対象とした動的マイグレー

[1] 公立はこだて未来大学
Future University Hakodate, Hakodate, Hokkaido, Japan
[a)] b1018262@fun.ac.jp
[b)] matsu@fun.ac.jp

ション機構の実現手法を検討する．CPU アーキテクチャ中立なノード実行状態の表現形式として，WebAssembly（以降，Wasm）を ROS へ導入する．C++で実装された ROS ライブラリ API を用いて実装される ROS ノードを Wasm バイナリとしてコンパイルし，Wasm 仮想マシン上で動作させるために解決すべき技術的課題を抽出して，それらへの対処方法を考案する．さらに，Wasm 仮想マシン上で動作する ROS ノードの実行状態をファイル等へ保存し，さらに，保存された実行状態を別プラットフォームで復元する，異種デバイス間 ROS ノードマイグレーションの実現手法を検討する．また，基本的な ROS API に対応したプロトタイプ実装を使って，ROS ノードの Wasm 化によるオーバヘッドと実行状態の保存・復元にかかるオーバヘッドを計測する実験によって，本提案機構の有効性を確認する．

実行中の各 ROS ノードがクラウドとロボットの間を柔軟に移行できることにより，例えば，自動運転制御などのタイムクリティカルな知能・制御系処理をクラウドとロボットのどちらで実行すべきかを，通信回線の状況変化やバッテリー状況変化に対応する省電力制御に対応して最適な選択が可能となる．また，本機構を活用することで，異種ロボットが連携するマルチロボット・システムにおける各ロボットの連携処理や，マルチアクセス・エッジコンピューティングにおけるエッジ-データセンタ間の処理分担においても，柔軟な意思決定が可能となる．

以降，第 2 章では，クラウドロボット・システムにおける ROS ノード配置の課題を提示する．第 3 章では，クラウド-ロボット間 ROS ノードマイグレーションの実現手法を提案する．第 4 章でプロトタイプ実装について述べ，第 5 章では，実装を用いたオーバヘッド計測に関する実験結果を示す．第 6 章で，関連する先行研究を紹介し，本研究の位置づけを明らかにする．最後に，第 7 章で，まとめと今後の課題について述べる．

2. クラウドロボット・システムにおける ROS ノード配置

クラウドロボット・システムではクラウドとロボット間のネットワーク通信が必要である．ロボットの移動によりネットワーク通信状況が変化した場合，リアルタイム性が損なわれたり，クラウドとの連携が出来なくなる可能性がある．また，ロボットにはバッテリー制約や計算リソースの制約がある．バッテリー残量の減少や計算負荷の増加により，ロボットの停止や処理の遅延が発生する可能性がある．このようなネットワーク環境やロボットの負荷状況の変化により，ロボットシステムの継続が困難になる場合がある．ネットワーク通信状況が変化した場合，クラウド上のノードをロボットに移行することでネットワーク通信が不要になり，ロボット単体でシステムを継続することができる．ロボット側の負荷が変化した場合，ロボット側の高負荷なノードをクラウド側に移行することで，バッテリー消費や計算負荷が低減され処理の遅延を抑制することが出来る．このように，システムの状況に応じてデバイス間でノードを移行し，適切な配置にすることで，ロボットシステムを継続することが可能になる．

ノードの移行方法は，ノードの種類によって異なる．入力内容によってのみ出力が決まる（以降，ステートレス）ようなノードは，移行元でノードの停止を行い，移行先でノードの起動を行えば移行が完了する．ノードの処理が前周期の実行状態を参照している（以降，ステートフル）場合においては，そのプロセスの実行状態も移行しなければならない．しかし，CPU アーキテクチャが異なるデバイス同士では，プロセスの実行状態に互換性が無い．プロセスの実行状態としてはコード，メモリ（ヒープやスタック），レジスタ値が存在する．コードは CPU 固有の命令セットにより構成される．メモリはヒープやスタック上に存在するデータであり，これらのメモリ上でのデータ表現は ABI によって異なる．レジスタ値は，CPU のレジスタ構成によって異なる．そのため，プロセスの実行状態に互換性が無い異種デバイス間では実行中のノードを移行することは容易ではない．クラウドとロボットは，それぞれの動作特性から異なる CPU アーキテクチャが搭載されることが多い．TREND FORCE によると，2017 年時点のサーバ市場で x86 プロセッサのシェア率が 96%を占めている [2]．ロボットの CPU には，低コストや低電力といった特性が要求されるため，ARM が採用されることが多い．そのため，現在広く利用されているクラウドとロボットは異種デバイスである可能性が高く，実行中のノードを動的に移行できないという課題がある．

3. クラウド-ロボット間 ROS ノードマイグレーション

本研究ではクラウドロボット・システム上のタスクとして，ROS ノードを対象とする．異種デバイス間でステートフルなタスクのマイグレーションを行うには，前章で述べたように，プロセスの実行状態を CPU アーキテクチャに依存しない形式で表現する必要がある．本研究の提案機構では，図 1 に示すように，仮想マシン上でノードを実行させることでプロセスの実行状態を異種デバイス間で共通な形式にする．

仮想マシンは CPU アーキテクチャに依存しない仮想的な命令セットを持ち，レジスタに依存しないスタックベースのものとして Wasm を導入した．他の仮想マシンと比較して，多数のプログラミング言語を対象としており，ネイティブコードと同等の処理速度で動作することが設計目標として挙げられている．ROS ノードを Wasm 仮想マシン上で動作させ，ノードの実行状態を保存・復元する機構を提案する．

Vol.2022-ARC-248 No.36
Vol.2022-SLDM-198 No.36
Vol.2022-EMB-59 No.36
2022/3/10

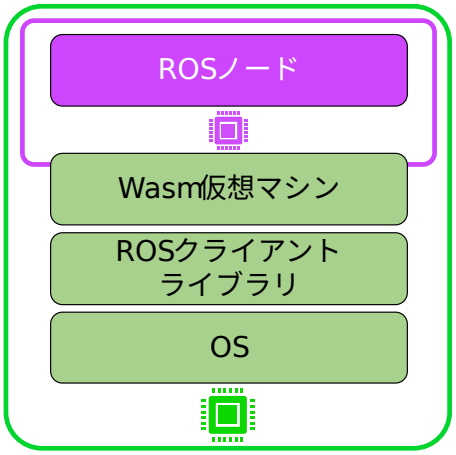

図 1 CPU アーキテクチャ中立な ROS ノード

3.1 ROS ノードの CPU アーキテクチャ中立化

ROS ノードの実行状態を異種デバイス間で共通の形式にするため，ROS ノードのプログラムを Wasm へコンパイルする必要がある．通常の ROS ライブラリは CPU アーキテクチャ依存のネイティブバイナリとして存在しているが，Wasm のプログラムとネイティブバイナリを直接リンクする事は出来ない．Wasm のプログラムからネイティブライブラリを使用する場合，ライブラリのプログラムも Wasm へコンパイルする必要がある．

ROS ノードのプログラムは，ユーザが実装する計算処理と，ROS ライブラリによって提供される ROS API によって構成されるため，計算処理と ROS API のプログラムをソースコードの状態からコンパイルする必要がある．現在の Wasm は，スレッド操作やネットワーク機能をサポートしておらず，これらの機能を持つプログラムは Wasm へコンパイルできないという制約がある．ROS API のプログラムはスレッド操作やネットワーク機能を含むため，Wasm の制約上コンパイルを行うことはできない．そのため，本研究では計算処理のプログラムのみ Wasm へコンパイルし，ROS API はネイティブの ROS ライブラリの関数を利用する．Wasm 仮想マシン上で動作する ROS ノードから，ネイティブ ROS ライブラリの ROS API を呼び出すための機構を実装する．

3.2 ROS ノード実行状態の保存と復元

Wasm 仮想マシン上で実行されたノードをマイグレーションするには，Wasm 仮想マシンの内部構造を保存し，復元する必要がある．内部構造には，モジュールインスタンス，関数インスタンス，グローバルインスタンス，メモリインスタンス，スタックフレームがある [3]．モジュールインスタンスは他の内部構造への参照の情報を持つ．関数インスタンスは関数の型，仮想命令セットで構成されたコード，関数インデックスの情報を持つ．Wasm 仮想マシン外部の関数インスタンスは関数の型と参照の情報を持つ．グローバルインスタンスは，グローバル変数の型と値の情報を持つ．メモリインスタンスは，64KB を 1 ページとしたメモリ領域を持つ．スタックフレームは，実行中の関数インデックス，スタック，スタックポインタを持つ．これらの内部構造の内，モジュールインスタンスと関数インスタンスは実行中に状態が変化せず，Wasm プログラムをロードすることで復元できる．それ以外の実行状態はノードの実行中に状態が変化するため，移行元でノード停止後に実行状態を保存し，移行先で復元できる必要がある．本研究では，Wasm 仮想マシンにノードの実行を停止させて実行状態を保存する機構，及び，保存した実行状態を復元し，ノードを再実行させる機構を実装する．

4. プロトタイプ実装

本研究ではオープンソースの Wasm 仮想マシンである WebAssembly Micro Runtime（以降，WAMR）[4] に対し，仮想マシン実行状態の保存・復元機構を実装する．WAMR は実行時のバイナリサイズが小さく，メモリ使用量が少ないという特徴があり，ロボット上での動作に適していると言える．

従来の ROS ノードプログラムは，各 CPU アーキテクチャに向けて用意されたライブラリとリンクすることで実行ファイルを生成する．本研究では ROS ノードを Wasm へコンパイルするため，特定のアーキテクチャに依存した ROS ライブラリとリンクすることは出来ない．Wasm のプログラムから，ネイティブの ROS ライブラリを利用する場合，Wasm へコンパイルするプログラムと，Wasm 仮想マシン外部の両方にグルーコードを実装する必要がある．仮想マシン上でノードを実行することで，ノードの実行状態は異種デバイス間で共通になる．本章では，Wasm 仮想マシン上の ROS ノードプログラムからネイティブの ROS ライブラリの機能を呼び出すためのグルーコードの実装と，ROS ノードの実行状態を保存・復元する機構の実装について述べる．

4.1 Wasm 向け ROS API グルーコード

本研究で実装するグルーコードは C++言語を対象としており，既存の ROS ノードプログラムにも適用可能になるよう実装する．ROS ノードプログラムで使われる ROS API は以下の 4 つに分類することが出来る．

- ROS オブジェクト API
- コールバック実行制御 API
- ロギング API
- ユーティリティ API

ROS オブジェクト API は，ROS ノード，publisher/subscriber，Service-server/Service-client 等のオブジェクトを生成するための関数や，それらのオブジェクトのメンバ関数等が含まれる．コールバック実行制御

Vol.2022-ARC-248 No.36
Vol.2022-SLDM-198 No.36
Vol.2022-EMB-59 No.36
2022/3/10

API は，コールバック関数を持つオブジェクトを制御する関数が含まれる．ロギング API は，ログ生成のためのマクロや関数が含まれる．ユーティリティ API は，例外処理，メモリ管理，時間データ取得等の関数が含まれる．本研究では，ステートフルなタスクとして，特定の回数分メッセージを配信し続ける ROS ノードを対象とする．この ROS ノードプログラムで使用されている ROS API の内，ROS オブジェクト関数に分類される ROS ノード生成 API，Publisher 生成 API，publish API の実装を行った．

4.1.1 ROS ノード生成 API

この関数の引数はノード名を表す一つの文字列型オブジェクトで，戻り値は生成された ROS ノードオブジェクトのアドレスである．Wasm の関数の引数では整数と浮動小数点数しか扱うことが出来ないため，文字列型のノード名を引数をで渡すことは出来ない．Wasm プログラムのグルーコードとして，受け取ったノード名を表す文字列型オブジェクトを配列にシリアライズし，配列の先頭アドレスと長さをそれぞれ整数型の値として Wasm 仮想マシン外部のグルーコードに渡す関数を，ネイティブの ROS ノード生成 API と同じ関数のシグネチャで定義する．Wasm 仮想マシン外部のグルーコードでは，受け取った文字列の先頭アドレスと長さから文字列型オブジェクトをデシリアライズし，ネイティブの ROS API の引数に指定して ROS ノードを生成する．ネイティブの ROS API で生成した ROS ノードオブジェクトのアドレスは，ROS ノードを一意に識別するための整数値を Key とした連想配列の Value として保持する．Wasm プログラムでは Wasm 仮想マシン外部のメモリ空間にアクセスできないため，識別子を指定することで任意の ROS ノードに対して操作を行う．

4.1.2 publisher 生成 API

この API の引数はトピックを表す一つの文字列型オブジェクトと QoS の値で，戻り値は生成されたパブリッシャオブジェクトのアドレスである．テンプレート関数であるためパブリッシュするデータ型を指定する必要がある．パブリッシャは一つの ROS ノードに関連付けて生成されるため，生成済みの ROS ノードの識別子，トピック名，パブリッシュするデータ型を Wasm 仮想マシン部のグルーコードに渡す関数を ROS ノード生成関数と同様に実装する．Wasm 仮想マシン外部のグルーコードでは，ROS ノード生成関数のグルーコードと同様に文字列データの変換処理を行い，ROS ノード識別子を Key にして連想配列から生成済みの ROS ノードのアドレスを取得する．トピック名，ROS ノードのアドレス，パブリッシュするデータ型を引数としてネイティブの ROS API に渡すことでパブリッシャを生成する．パブリッシャ生成関数では QoS の設定が必要となるが，本研究では QoS を全てデフォルト値に設定するため，QoS の設定は省略する．ROS ノードと同様，ネイティブの ROS API で生成したパブリッシャオブジェクトのアドレスは，連想配列を用いて保持する．

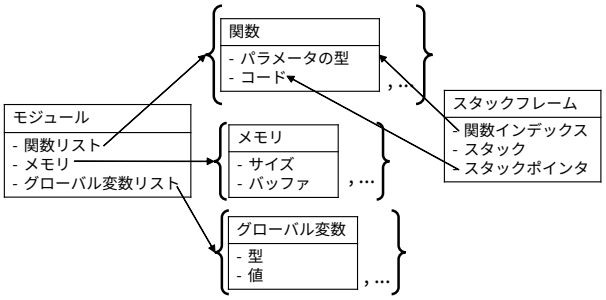

図 2 WAMR における仮想マシンの内部構造

4.1.3 publish API

この API の引数はパブリッシュする ROS メッセージのみであり，テンプレートクラスのパブリッシャのメンバ関数である．ROS メッセージは任意のデータ型の組み合わせによって表現されることがあるため，ROS メッセージの型の数分グルーコードが必要になる．本研究で対象とする ROS ノードプログラムでは，std_msgs::msg::String 型の実装のみ行った．

4.2 ROS ノード実行状態の保存・復元機構

提案機構で実行したノードの実行状態を移行するには，Wasm 仮想マシンで動作する計算処理の実行状態と，ネイティブの ROS API によって生成された ROS オブジェクトを CPU アーキテクチャに依存しない形式で保存し，復元する必要がある．

4.2.1 Wasm 仮想マシン

Wasm 仮想マシンの内部構造には，モジュールインスタンス，関数インスタンス，グローバルインスタンス，メモリインスタンス，スタックフレームがある [3]．これらの内部構造は図 2 のように構造体として定義されているため，各構造体のメンバ変数を保存復元する必要がある．モジュールインスタンスと関数インスタンスは実行中状態が変化しない．これらの状態は Wasm のプログラムファイルをロードすることで復元可能であるため，実行中のノードから保存する必要はない．グローバルインスタンス，メモリインスタンス，スタックフレームは実行中に状態が変化するため，構造体のメンバ変数を保存復元処理を実装する．

スタックフレームのメンバ変数であるスタックポインタは，関数インスタンスのメンバ変数であるコードのバッファを指すポインタ型変数である．コードのバッファは，Wasm 仮想マシンを実行するたびにランダムなアドレス空間に割り当てられる．移行先のデバイスで Wasm 仮想マシンを実行する際，異なるアドレス空間に割り当てられる可能性が高いため，移行元のスタックポインタの値は使用できない値となる．移行元で復元できる値へ変換するため，スタックポインタが指すバッファの先頭アドレスを引いたオフセット値を計算し，保存する．復元時には，移行先で

Vol.2022-ARC-248 No.36
Vol.2022-SLDM-198 No.36
Vol.2022-EMB-59 No.36
2022/3/10

表 1 実験環境

	ロボット	クラウド
CPU	Cortex-A57 1.43GHz	Intel（R） Core（TM） i5-8350U 1.70GHz
メモリ	4GB	8GB
OS	Ubuntu 18.04	Ubuntu 20.04
ROS	ROS2 Foxy	ROS2 Foxy
ストレージ	microSDHC UHS-I Class10 32GB	NVMe SSD 128GB

割り当てられたバッファの先頭アドレスにオフセット値を足した値をスタックポインタに代入することで復元する．

4.2.2 ROS

ROS API によって生成されたオブジェクトの実行状態は，Wasm 仮想マシンの外部に存在する．本研究では，ROS オブジェクトの内部状態は移行せず，移行先で再度 ROS API を呼び出すことで ROS オブジェクトの移行を行う．ROS API 呼び出し時に Wasm 仮想マシン外部のグルーコードで引数を保存する．移行先で保存した引数を元に再度 ROS API を呼び出し，新たに ROS オブジェクトを生成することで Wasm 仮想マシン外部の実行状態を復元する．

5. 実験

本研究で実装した提案機構を用いた ROS ノードと，従来の ROS ノードをクラウドとロボットをそれぞれ想定したプラットフォームで実行させ，実行時間及び，Wasm 仮想マシン実行状態の保存・復元にかかる時間と実行状態のデータサイズを評価する実験を実施した．実行時間の計測には特定の回数分メッセージを配信し続ける ROS ノードを対象とした．本実験の実施中は，実験対象の ROS ノードを 1 つだけ実行し，さらに実験対象の ROS ノードが配信するトピックから購読するサブスクライバーを 1 つだけ実行した．1 回の実行で配信する回数は 100,000 回とし，メッセージのデータサイズは 256bytes から 4Kbytes までとした．Wasm 仮想マシン実行状態の保存・復元にかかる時間及び実行状態のデータサイズの計測には，フィボナッチ数を再帰関数によって求めるプログラムを対象とした．Wasm 仮想マシン内で再帰呼び出し中の関数の数は 100 から 1,000 までとした．試行回数は 50 回とし，その平均を算出した．今回実験を行った環境について表 1 に示す．

5.1 計測結果

実験結果を図 3 から図 7 に示す．図 3 では，実装した提案機構を用いた ROS ノードと，従来の ROS ノードをクラウド上で実行した時の実行時間を比較しており，図 4 ではロボット上で実行した時の実行時間を比較している．提案機構で実装した ROS ノードでは配信するメッセージのデータサイズが増加するにつれ，実行時間も増加している．従来の ROS ノードでは，配信するメッセージのデータサイズに関わらず，実行時間は一定である．

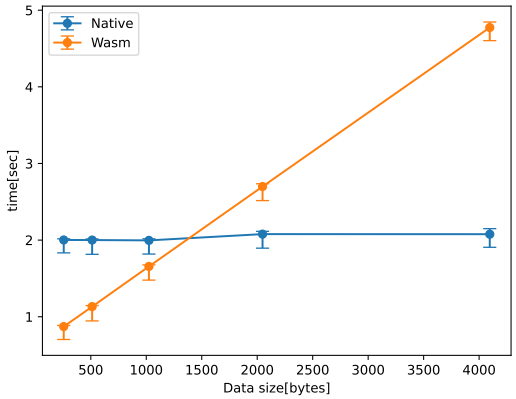

図 3 データ送信処理ノードの実行時間（クラウド）

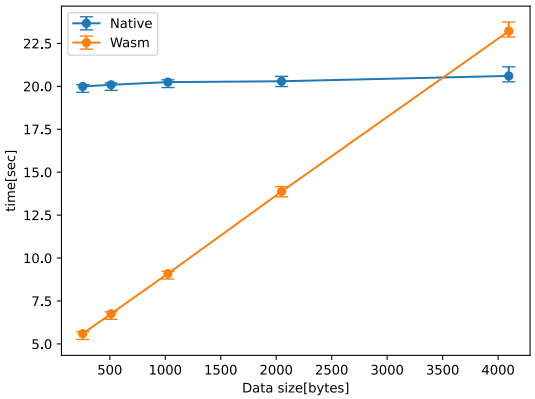

図 4 データ送信処理ノードの実行時間（ロボット）

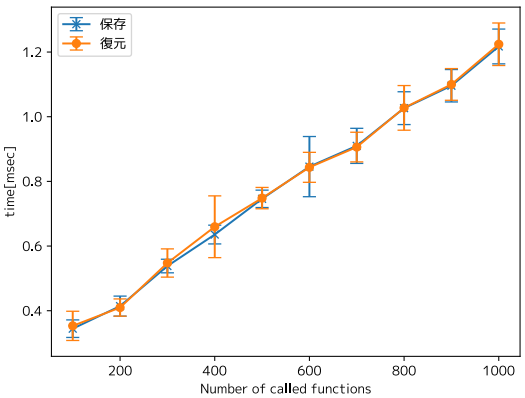

図 5 実行中 ROS ノードのマイグレーション時間（クラウド）

図 5，図 6 から，再帰関数の呼び出し数が増加することで実行状態の保存・復元にかかる時間が増加している．

図 7 から，関数呼び出し数が増加することで実行状態のデータサイズが増加している．メモリのサイズは 131Kbytes，グローバル変数のサイズは 4bytes，Wasm の命令を実行するインタプリタ関数のローカル変数のサイズ

Vol.2022-ARC-248 No.36
Vol.2022-SLDM-198 No.36
Vol.2022-EMB-59 No.36
2022/3/10

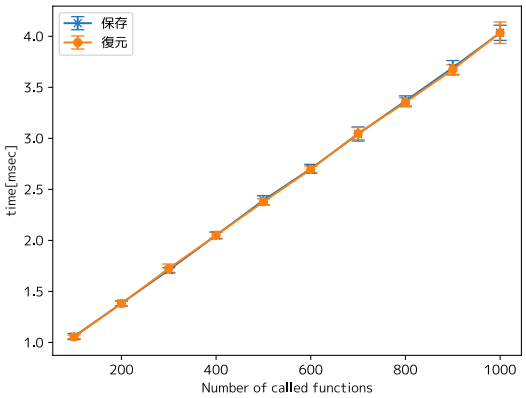

図 6 実行中 ROS ノードのマイグレーション時間（ロボット）

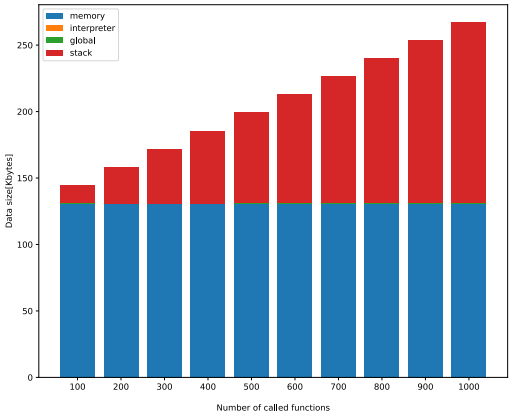

図 7 保存した ROS ノード実行状態のサイズ

は 28bytes で一定である．

5.2 考察

図 3，図 4 から，提案機構ではデータサイズが増加するにつれ，実行時間も増加していることが示されている．これは，Wasm 上の文字列型オブジェクトを外部のグルーコードに渡す際，一度文字列型オブジェクトを文字列の先頭アドレスと長さに分解してから引数として渡し，それらの値から文字列型オブジェクトを再生成しているためであると考えられる．Wasm において，整数と浮動小数点数以外を外部のグルーコードと交換する場合，型変換の処理が必要になるため，実行時間においてデータサイズが課題となる．データサイズが小さい時に従来よりも Wasm 上での実行時間が短いのは，本研究で実装したグルーコードでは QoS 制御を行っていないためである．

図 5，図 6 から，関数呼び出し数が増加することで実行状態の保存・復元にかかる時間が増加し，実行状態を保存したファイルサイズが増加していることが示されている．Wasm 仮想マシンは，関数を実行する際にスタックを 1 つ割り当て，関数の実行が終了する際にスタックを解放する．そのため，関数を再帰的に呼び出す場合，呼び出しの深さが深くなるほどスタックの数が増え，それに伴い実行状態の保存復元にかかる時間とファイルサイズが増加する．

6. 関連研究

Fukutomi らは，ROS システムを構成するデバイスのリソースをマスタースレーブ方式によって管理し，各デバイスのリソースの変化に応じてでノードを分散させる手法を提案した [5]．この研究で提案されている資源管理機構は ROS のミドルウェアにリソースマネージャを実装することで実現されている．リソースマネージャは ROS マスターの機能として動作し，各マシンのリソース情報（CPU，メモリ，ディスク）の取得管理，各ノードの割り当てとマイグレーションを行う．同一の ROS システム内では，同じ名前のノードを複数動作させる事はできない．同じ名前を持つ他のノードが実行されたとき，前に実行されたノードは終了させられる．そのため，ステートレスなノードのマイグレーションは他のデバイス上で同じ名前のノードを実行することにより行われる．ステートフルなノードは，プログラムの実行状態をマイグレーションする必要がある．この手法では，実行中のプログラムの変数の状態を公開させ，マイグレーション時に変数のデータを初期化させるという二つの制約を課し，ステートフルなノードのマイグレーションを実現している．この制約はユーザにマイグレーションのためのプログラム開発を要求するため，既存のオープンソースのノードには適用できない．

Suezawa は JVM（Java Virtual Machine）の実行状態の永続化システムを実装した [6]．実行中の Java アプリケーションを中断させ，Java スタック，ヒープ，メソッド領域，スレッドの状態を保存し，別のマシン上で復元することで，中断したポイントからアプリケーションを再開させることが出来る．しかし，このシステムは Java アプリケーションにしか対応しておらず，一般的な C++言語で記述された ROS ノードプログラムを対象とすることはできない．

7. おわりに

本研究では，クラウドロボット・システムを対象として，CPU アーキテクチャが異なるクラウドとロボットの間でステートフルな ROS ノードをマイグレーションするための機構を提案した．Wasm 仮想マシン上で ROS ノードを実行できるようにすることで，CPU アーキテクチャ中立なノード実行状態を実現した．Wasm 化においては，ノードとして実装された処理のみを Wasm 化し，ネイティブバイナリである ROS ライブラリの ROS API を Wasm 化したノードから呼び出せるようにした．Wasm が対応するデータ型が限られている制約に対して，ROS API に指定するデータやオブジェクトを Wasm 仮想マシンと ROS ライブラリ間で授受できるようにするシリアライズ処理を実

装した．プロトタイプ実装では，基本的な ROS API を対象にして，シリアライズ処理が実現できることを確かめた．また，組み込み向けの Wasm 仮想マシンである WAMR を対象として，仮想マシン実行状態の保存・復元機構の実装手法を確立した．ネイティブの ROS ライブラリ内にあるノード実行状態を保存・復元するために，ROS API 呼び出し時の引数を記録しておくことで，ノード実行状態の復元時に ROS ライブラリを同じ引数で再度呼び出すことで実行状態を復元できるようにした．さらに，本マイグレーション機構の有用性を確認するために，プロトタイプ実装を用いて，仮想マシン上での実行によるオーバヘッドとノード実行状態の保存・復元にかかる時間，保存した実行状態のデータサイズを計測する実験を行った．本実験結果より，ノードが ROS API 呼び出し時に指定されたデータのシリアライズにかかるオーバヘッドが，データサイズに比例して大きくなることがわかった．

今後の課題として，マルチスレッド処理を行う実アプリケーションへの対応がある．ROS フレームワークの内部実装にはスレッドを用いている処理があり，現在の Wasm 仕様がマルチスレッドに対応していないために Wasm 化することができない．本実装に用いた WAMR は，独自にマルチスレッドをサポートしているが，将来的に Wasm 仕様にマルチスレッド対応が追加された際に無効化される可能性がある．実際，現在，Wasm の仕様にマルチスレッド機能を追加することが検討されている [7]．Wasm 仕様がマルチスレッドに正式に対応することにより，ROS フレームワーク全体を Wasm 仮想マシンで動作させることを検討できる．ROS フレームワーク全体を Wasm 化することで，前述の ROS API 呼び出しにおけるシリアライズ処理が必要なくなり，ネイティブと比べて遜色のない性能で ROS ノードが実行できるようになることを目指す．

謝辞 本研究は，JSPS 科研費 JP21K11832 の助成を受けたものです．

参考文献

[1] Nedo: ロボット白書 2014 (2014). https://www.nedo.go.jp/content/100567345.pdf. (参照 2022-1-25).
[2] Trend force:trendforce finds x86 processors continues to corner server market this year with global shipment share estimated at 96 https://www.trendforce.com/presscenter/news/20171005-9880.html. (参照 2022-1-25).
[3] WebAssembly Core Specification. https://webassembly.github.io/spec/core/. (参照 2021-11-3).
[4] bytecodealiance: Webassembly micro runtime. https://github.com/bytecodealliance/wasm-micro-runtime. (参照 2022-1-25).
[5] Daisuke Fukutomi, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Resource manager for scalable performance in ros distributed environments. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1088–1093, 2019.
[6] Sara Bouchenak. Making java applications mobile or persistent. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 01)*, San Antonio, TX, January 2001. USENIX Association.
[7] Webassembly (2019) threads. https://github.com/webassembly/threads. (参照 2022-1-25).