M-004

WASI を利用したメッセージブローカ実現可能性の検討
A Feasibility Study of WASI-based Message Broker

笹倉 脩生[1)]　乃村 能成[2)]
Shuki Sasakura　Yoshinari Nomura

1. はじめに

仮想的な命令セットアーキテクチャに WebAssembly (Wasm) [1][2] がある．Wasm バイナリをブラウザを介さず動作させる要求からシステムリソースへのインタフェース WebAssembly System Interface (WASI) が開発され，仕様の開発と実装が進行中である．現在，ランタイムは標準仕様 WASI Preview 1 をサポートし，WASI Preview 2 を実装中である．

本稿では WASI によるサーバプログラムの実現可能性について調査した結果を報告する．ここでは，ネットワーク (ソケット) とスレッドに注目する．具体的には，これらの機能について複数の Wasm ランタイムの特徴を比較，メッセージブローカを実装，性能を評価した．

2. Wasm ランタイムの比較

2.1 概要

Wasm 仕様にもとづくさまざまなランタイムが開発されている．その性能や機能，開発状況は異なるため，作成するプログラムに応じて選択する必要がある．比較対象として Wasmtime[3]，Wasmer[4]，WasmEdge[5] の 3 つを取り上げる．Wasmtime は標準仕様にもとづき開発されている．Wasmer は WASI Preview 1 を拡張した WASIX を実装している．WasmEdge はクラウド向けの拡張機能を実装している．これらのランタイムについてソケットとスレッドの機能を比較し，以降の項で説明する．比較の際，プログラム実装言語は Rust を想定している．

2.2 ソケット対応状況

ソケットは，3 つすべてのランタイムで使用できる．Wasmtime は WASI Preview 1 で定義されている機能 (accept，send，receive，shutdown) に限り使用できる．Wasmer は WASI Preview 1 の拡張機能である WASIX，WasmEdge は wasmedge_wasi_socket ライブラリが実装されている．これを用いることで，Wasmer と WasmEdge は WASI Preview 1 で定義されている機能以外の OS 固有のソケット機能を使用できる．

2.3 スレッド対応状況

スレッドは，WasmEdge のみ使用できない．Wasmtime は WASI Preview 1 の拡張スレッド機能，Wasmer は WASIX によりスレッドを作成できる．

比較した結果，Wasmtime と Wasmer はソケットとスレッドの機能，WasmEdge はソケットの機能を使用できる．しかし，Wasmtime のソケット機能は WASI Preview 1 で定義された機能に限り使用できる．そのため，ソケットとスレッドの機能を共に使用する場合は，Wasmer を第一の候補とできる．

1） 岡山大学大学院環境生命自然科学研究科，Graduate School of Environment, Life, Natural Science and Technology, Okayama University
2） 岡山大学環境生命自然科学学域，Faculty of Environment, Life, Natural Science and Technology, Okayama University

表 1　測定環境

通番	項目	内容
1	OS	Ubuntu 22.04.3 LTS
2	Kernel	6.5.0-14-generic
3	CPU	AMD Ryzen 9 5950X (32) @3.4GHz
4	memory	64GB
5	compiler	rustc 1.74.0 (79e9716c9 2023-11-13)
6	wasm runtime	wasmedge version 0.13.5 および wasmer 4.2.5

3. WASI を利用したメッセージブローカ

3.1 実装の方針

メッセージブローカは複数クライアントからメッセージを受信し，それを保存するキューを持つ．マルチスレッドではキューをスレッド間で共有，ロックで管理する．そのため，ソケットとスレッドの機能を用いる．両方の機能を用いるため，2 章の比較にもとづきランタイムとして Wasmer を用いてマルチスレッドメッセージブローカを実装する．実装言語には Rust を用いる．

4. Wasm プログラムの性能測定

4.1 概要

WASI によるサーバプログラムの性能を確認する．まず，単純な TCP 通信のソケット性能を測定し，次にマルチスレッドメッセージブローカの性能を測定する．ネイティブバイナリ，WASI を用いる Wasm 実装を Rust で実装する．比較対象として，具体的に以下の 2 つの測定を行う．

1. 単純なメッセージ転送プログラムのネイティブ実装 (NativeTCP)， Wasm 実装 (WasmTCP) の性能比較
2. マルチスレッドメッセージブローカのネイティブ実装 (Native+MT)， Wasm 実装 (Wasm+MT) の性能比較

4 つの実装を測定対象とし，スループットを比較する．また，物理的なネットワーク回線を介して測定すると，ネットワーク通信がメッセージ転送のオーバヘッドの支配的要因となるため，単一計算機内での通信により比較する．測定環境を**表 1** に示す．

4.1.1 単純な TCP 通信のソケット性能測定

単純なメッセージ転送プログラムのネイティブ実装 (NativeTCP)， Wasm 実装 (WasmTCP) の測定を行う．このプログラムは次の動作を行う．

1. クライアントの接続を待ち，接続後メッセージを受信，受信した回数メッセージを送信する
2. メッセージの受信・送信時刻を全て出力する

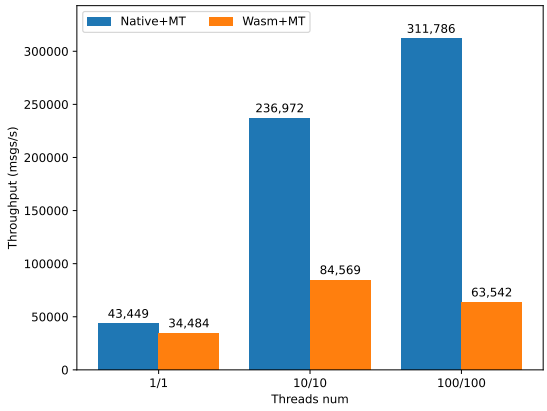

図 1　マルチスレッドメッセージブローカのスループット

ランタイムは WasmEdge を用いる．メッセージサイズを 1，2，4KB に変え，100,000 個のメッセージを転送する．

4.1.2 マルチスレッドを用いた場合のスレッド性能

マルチスレッドメッセージブローカのネイティブ実装 (Native+MT) と Wasm 実装 (Wasm+MT) の測定を 2 種類行う．ランタイムは 3.1 節に示したように，Wasmer を用いる．まず，メッセージブローカがメッセージを転送する動作を測定する．この際，メッセージサイズを 1KB，メッセージ数を 100,000 個として，Receiver 数 と Sender 数の組を (1/1)，(10/10)，(100/100) と変化させる．

2 つ目は，スレッド性能の差をより詳しく見るため，メッセージブローカがメッセージを受信する動作のみ行う．つまり，実際には Receiver を実行しないで Sender からのメッセージをキューイングする．メッセージサイズを 1KB，キューの数を 10 として，Sender 数を 1 から 40 まで変化させる．各 Sender は，各 Receiver 全てを宛先として同数のメッセージを送信する．

4.2 測定結果

4.2.1 単純な TCP 通信のソケット性能測定

NativeTCP と WasmTCP のスループット測定結果について以下で説明する．メッセージサイズ 1KB の場合，スループットは NativeTCP で 583,415 (msgs/s)，WasmTCP は 197,919 (msgs/s) となり，WasmTCP は NativeTCP の約 35%となった．また，メッセージサイズを 4KB に増加させると，スループットは 1KB の場合に比べ NativeTCP で約 65%，WasmTCP で約 85%になった．そのため，WasmTCP の方がメッセージサイズの増加による変化が小さく，メッセージサイズ 4KB では WasmTCP のスループットは NativeTCP の約 45% と差が小さくなった．

4.2.2 マルチスレッドを用いた場合のスレッド性能

マルチスレッドメッセージブローカのスループットを**図 1** に示し，以下で説明する．Wasm+MT のスループットは Native+MT に比べ，最も差が小さい $(1, 1)$ で約 80%，最も差が大きい $(100, 100)$ で約 20%であった．また，Wasm+MT では $(10, 10)$ の場合に比べて $(100, 100)$ の場合の性能が低下しており，スレッド数増加がかえってオーバヘッドになっている．

そこで次に，スレッドによる台数効果の差を見るためにメッセージ受信処理のスループットを**図 2** に示し，以下で説明する．Native+MT の Sender 数 1 に対する比率は，Sender 数 16 まで上昇した後一度減少し，Sender 数

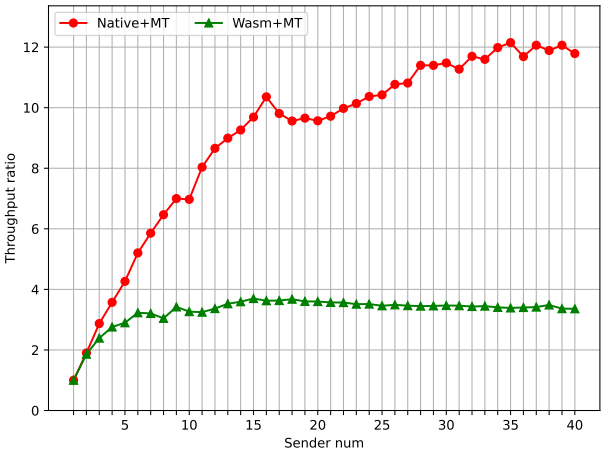

図 2　Sender 数を変化させたメッセージブローカのスループット

35 まで上昇した．上昇傾向が変化する Sender 数はおおよそ実行環境のコア数 (16 コア/32 スレッド) と一致している．一方，Wasm+MT の比率は，Sender 数 9 まで上昇し，比率 4 程度で停滞した．Wasm+MT でも Sender 数の増加に伴いスループットが増加しているため，スレッド数の増加によりスループットが上昇していることがわかるものの，Native+MT に比べ，Sender 数が 1 で約 50%，6 で約 30%，16 で約 20%，35 で約 15%になった．Sender 数 1 の Wasm+MT は Native+MT に比べてスループットが低く，少ないスレッド数で上昇が停止する．つまり，Wasm ランタイムのスレッド生成・管理のオーバヘッドが大きくなることが考えられる．原因の調査と性能改善手法の検討が必要である．

5. おわりに

本稿では，WASI によるメッセージブローカの実現可能性を検討した．ソケットとスレッドについてランタイムを調査し，WASI によるメッセージブローカの実現方式を検討，実装した．また，単純なメッセージ転送プログラムとマルチスレッドメッセージブローカについてネイティブ実装と Wasm 実装を比較した．その結果，メッセージブローカでは 1〜2 スレッドではネイティブに近い性能を示したものの，スレッド数増加による性能向上は小さいことがわかった．

参考文献

[1] WebAssembly: WebAssembly, WebAssembly (online), available from ⟨https://webassembly.org/⟩ (accessed 2023-06-07).

[2] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A. and Bastien, J.: Bringing the web up to speed with WebAssembly, *SIGPLAN Not.*, Vol. 52, No. 6, p. 185–200 (online), available from ⟨https://doi.org/10.1145/3140587.3062363⟩ (2017).

[3] Bytecode Alliance: Wasmtime A fast and secure runtime for WebAssembly, Bytecode Alliance (online), available from ⟨https://wasmtime.dev/⟩ (accessed 2023-11-15).

[4] Wasmer: Wasmer: Run, Publish & Deploy any code - anywhere, Wasmer (online), available from ⟨https://wasmer.io/⟩ (accessed 2023-11-15).

[5] Cloud Native Computing Foundation: WasmEdge, Cloud Native Computing Foundation (online), available from ⟨https://wasmedge.org/⟩ (accessed 2023-11-15).