Vol.2021-ARC-244 No.22
Vol.2021-SLDM-194 No.22
Vol.2021-EMB-56 No.22
2021/3/25

時刻同期機構を備えた分散型ロボット開発ミドルウェアの特徴と有効性

郡司 凌太[1,a)]　福田 浩章[1,b)]　長谷川 忠大[1,c)]

概要：ロボットは人間の生活を支えるために, 様々な場面で利用されている. 近年ではロボットに求められる要求が一層複雑になっており, ロボットの制御ソフトウェアの開発には広範な知識と経験が必要となる. そのため, ロボットの開発コストは増加している. このコストを削減するために, ROS (Robot Operating System) などのミドルウェアが開発され, 広く利用されている. SSM (Streaming data Sharing Manager) もこのようなミドルウェアの 1 つであり, PC 上でタイムスタンプ付きのセンサデータの読み出しと書き込みを行う機能を提供する. これは, センサデータの計測時刻を考慮したロボットの制御に有効であり, データの計測間隔の異なる複数のセンサを用いて制御する自律移動ロボットなどの制御ソフトウェアでは特に有効である. 一方で, SSM ではロボットの制御を 1 台の PC で行うことを前提としているため, CPU 負荷の大きいプロセスが複数存在すると, 他プロセスに影響を与えることがある. その結果, センサデータの取りこぼしや計算の遅延などの予期せぬ挙動を引き起こしてしまうという問題があった. そのため SSM を拡張して, SSM で動作するプロセスを異なる PC に分散することを可能にする Distributed Streaming data Sharing Manager (DSSM) というミドルウェアが提案されている. しかし, DSSM を用いてロボットを構築し走行させた事例は少なく, DSSM の実装がロボットに与える影響や特徴は明らかになっていない. そこで本論文では, SSM で構築された自律移動ロボットを DSSM を用いて 3 台の PC に分散し, 様々な条件下で走行実験を行う. そして, その際の PC への負荷やログデータを解析することによって, DSSM の有用性と特徴を検証する.

1. 研究背景と目的

ロボットは現在, 作業の効率化や人間には危険な作業の代替など, 様々な場面で利用されている. ロボットはセンサやアクチュエータなどのハードウェアとそれを制御するためのソフトウェアシステム (以下, 制御システム) で構成される. しかし近年, ロボットの用途は多様化しており, それに伴って制御システム開発には広範な知識と経験が求められるようになっている [4][5]. そのため, 過去の制御システムの再利用や複数人で制御システムを容易に開発することができるようなミドルウェアシステムが求められている. このような背景から, Robot Operating System (ROS) [7][9] や RT-Middleware (RTM) [1][2][3] などのミドルウェアシステムが開発され, 使用されている. 多くのミドルウェアシステムでは, 制御システムで行われるセンサデータの計測やハードウェアの制御はそれぞれ単一のプロセスで実行される. その中でも, 複数のセンサを用いて構築されるロボットでは, センサデータをまとめて処理するプロセス (以下, メインプロセス) が制御システムに存在する. メインプロセスでは, プロセス間通信 (以下, IPC) を用いて他プロセスからセンサデータを取得するが, この場合, センサデータの計測時刻 (以下, タイムスタンプ) が重要になる. もし, タイムスタンプが異なるセンサデータを用いて制御を行うと, 周囲の状況が正しく把握できず, 適切な制御ができなくなる. 従って, メインプロセスではセンサデータのタイムスタンプを同期することが必要となる.

Streaming data Sharing Manager (SSM) [11][12] はタイムスタンプとセンサデータの管理に必要な煩雑な処理を隠蔽し, センサデータの時刻同期を行うことを容易にする API を提供するミドルウェアシステムである. SSM では, ロボットの機能要素は個々のプロセスとして実行され, IPC に共有メモリを用いてセンサデータをやり取りすることで制御システムが構築される. すなわち, SSM で実行される全てのプロセスは同一の PC に存在する必要がある. そのため SSM で構築した制御システムでは, 負荷の大きいプロセスが複数存在すると, CPU の処理が間に合わず, 他のプロセスに影響を与えて, 処理の遅延やセンサデータのと

[1] 芝浦工業大学
[a)] ma19030@shibaura-it.ac.jp
[b)] hiroaki@shibaura-it.ac.jp
[c)] thase@shibaura-it.ac.jp

Vol.2021-ARC-244 No.22
Vol.2021-SLDM-194 No.22
Vol.2021-EMB-56 No.22
2021/3/25

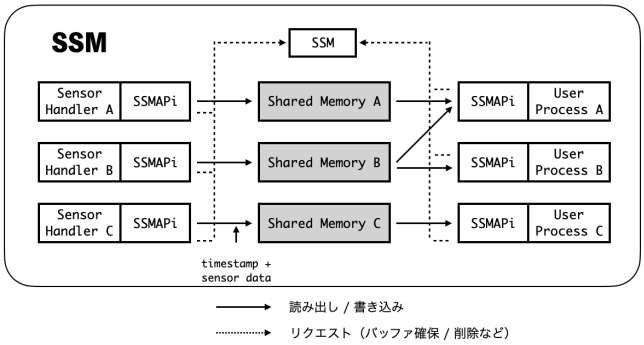

図 1 SSM を用いて構築した制御システムの例

りこぼしなどの予期せぬ不具合が発生するという問題があった.

この問題に対処するために, SSM と同様の API を持ち, ネットワークを介して SSM が管理する共有メモリにアクセスする Distributed Streaming data Sharing Manager (DSSM) が提案されている [13]. DSSM では TCP/IP 通信を用いて異なる PC に存在するプロセス同士でセンサデータをやりとりする手段を提供する. これにより PC の負荷に応じて, プロセスを複数の PC に分散できるようになる. また複数の PC を使用する場合, それぞれの PC が参照する時計が異なるため, タイムスタンプの一貫性は失われる. そのため, DSSM では Network Time Protocol (NTP) を用いて PC 間のタイムスタンプを同期することでこの問題に対処している. このようにして, DSSM は複数の PC を用いて制御システムを構築することを可能にしている. その一方で, DSSM を用いてロボットを構築し, 実際に走行させた事例は少なく, 通信で使用する TCP/IP やタイムスタンプの一貫性を保証するために使用する NTP がロボットの制御に与える影響は明らかになっていない.

そこで本研究では, DSSM と 3 台のラズベリーパイを使用して, 複数のセンサとハードウェアを用いて制御される自律移動ロボットを構築し, 走行実験を行う. そして, 実験中に計測した CPU 使用率と共有メモリに保存されたセンサデータのログを分析することで DSSM の有用性と特性を検証する.

本論文は, 2 節で SSM と DSSM について述べる. 3 節では走行実験とその結果について述べ, 4 節でその考察を述べる. 最後に 5 節で本論文の結論と今後の課題を述べる.

2. SSM / DSSM

本節では, SSM と DSSM のアーキテクチャについて述べる.

2.1 SSM

図 1 に SSM を用いて構築した制御システムの例を示す. SSM は周期的に計測されるセンサデータをタイムスタンプと紐づけてリングバッファ型の共有メモリに管理する. そして, センサデータの読み書きなどの共有メモリを操作するための API (以下, SSMAPi) を開発者に提供する. SSM では, SSMAPi を使用してデータを共有メモリに書き込むプロセスをセンサハンドラと呼ぶ. センサハンドラが SSM にセンサデータを書き込むには, 最初に, SSMAPi を使って SSM に共有メモリの生成を依頼する. この際, SSM が生成した共有メモリを区別するために, 開発者は一意の識別子を設定し送信する. SSM が共有メモリを確保すると, センサハンドラにそのメモリのポインタを返信する. この一連の操作を行うことで, センサハンドラは SSMAPi を介してセンサデータを共有メモリに書き込むことができるようになる. 一方で図 1 の *UserProcessA, B, C* のように, 共有メモリからセンサデータを読み出すプロセスは, 最初に SSMAPi を使って対象のセンサデータが存在する共有メモリの識別子を SSM に送信する. SSM は識別子を使って共有メモリを検索し, そのメモリのポインタをユーザプロセスに返す. その後, ユーザプロセスは SSMAPi を介してセンサデータを読み出すことが可能になる. また, SSMAPi を介してセンサデータが書き込まれた場合, SSM が自動的に現在時刻をタイムスタンプとしてセンサデータに紐づけて保存される. そのため, SSM で管理されるセンサデータは全て時刻を指定して読み出すことが可能になる.

このように, SSM は共有メモリにセンサデータとタイムスタンプを紐づけて管理することによって, 時刻を遡ったセンサデータの参照を可能にしている. そして, 書き込みや時刻を指定したセンサデータの読み出しなどの共有メモリの操作を行う, SSMAPi を提供する. そのため, 開発者は SSMAPi を利用することでセンサデータの時刻同期に必要な複雑な処理を実装せずに, 時刻同期が必要な制御システムを構築することが可能になる. 一方で, SSM では IPC に共有メモリを用いるため, 全てのプロセスは同一の PC に存在する必要がある. しかし, これまでの SSM を使用した自律移動ロボットの運用から, 制御システム内に負荷の大きいプロセスが複数存在すると, CPU の処理が間に合わず, 処理の遅延やセンサデータのとりこぼしなどの予期せぬ不具合が発生することわかっていた. 従って, SSM を用いて制御システムを構築する場合, システム内のそれぞれの処理が CPU に与える負荷を考慮して, 制御に使用するアルゴリズムなどを選択する必要があった.

2.2 DSSM

2.1 節で述べたように, SSM では使用する PC の CPU 負荷を考慮して制御システムを構築する必要があった. DSSM はこの問題を解決するミドルウェアであり, SSM を拡張し, TCP/IP を用いたソケット通信を利用する共有メモリの操作を行う API を開発者に提供する. これを利用することで制御システムを複数台の PC から構築できるようになり,

Vol.2021-ARC-244 No.22
Vol.2021-SLDM-194 No.22
Vol.2021-EMB-56 No.22
2021/3/25

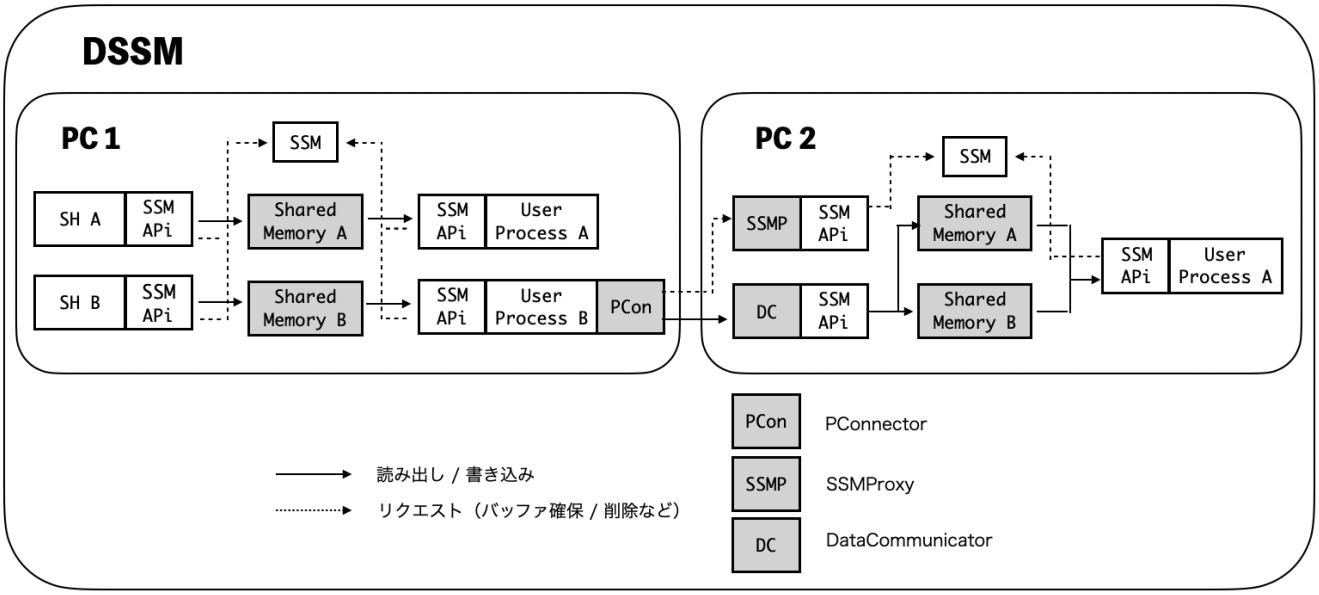

図 2　DSSM を用いて構築した制御システムの例

CPU 負荷を分散することが可能になる.

図 2 に DSSM を用いて構築した制御システムの例を示す. DSSM ではネットワークを介して異なる PC に存在する共有メモリにアクセスしたいセンサハンドラは, PC1 の *User ProcessB* のように SSMAPi の代わりに PConnector と呼ばれる API を使用する. PConnector は SSMAPi とほぼ同様のインターフェースを持ち, TCP/IP 通信を使用して異なる PC に存在する共有メモリを操作する手段を提供する. PConnector がバッファの確保などのリクエストを受けると, アクセスする PC に独立したプロセスとして実行されている SSMProxy というモジュールにそのリクエストを送信する. SSMProxy はネットワークを介して共有メモリにアクセスするセンサハンドラごとに, 共有メモリが存在する PC にインスタンス化されるモジュールである. そして, SSMProxy は PConnector からリクエストを受け取り, SSMAPi を用いて実際に共有メモリを操作する. DSSM は SSM の拡張ではあるが, SSM が数年かけて開発され, 様々な研究室で運用されている実績を考慮して, 既存のソースコードに変更を加えずにネットワークを介した共有メモリの操作を可能にするために, SSMProxy を介して共有メモリを操作するようにしている. また, PConnector を使用するセンサハンドラがセンサデータを書き込む場合, SSMProxy は別スレッドで DataCommunicator というモジュールを実行する. DataCommunicator は PConnector からのセンサデータの書き込みや読み出しのリクエストを処理するモジュールである. これは PConnector と SSMProxy 間の単一の TCP/IP 通信では, SSM のセンサハンドラが複数の共有メモリにセンサデータを書き込めるという動作を再現できないためである. このように DSSM では 3 つのモジュールを新たに用意することで, ネットワークを介した共有メモリの操作を可能とする.

表 1　ntpd を用いて PC と NTP サーバを同期した際のオフセット

	平均	最大値	最小値	標準偏差
offset(ms)	***1.41***	***5.93***	0.015	1.20

一方で, 複数の PC で 1 つの制御システムを構築する DSSM では, 全てのセンサハンドラが同じ時計を参照するとは限らないため, タイムスタンプの一貫性を保証できないという問題がある. そこで DSSM では, Network Time Protocol (NTP)[6] を用いて, この問題に対処する. 具体的には, 1 台の PC を NTP サーバとして実行し, 他の PC はこの PC に自身のシステムクロックを同期するように NTP デーモン (ntpd) を設定する. このようにして, DSSM ではタイムスタンプの一貫性を保証する. また, NTP による時刻同期の正確性を評価するために, 2 台の PC を用いて, 1 時間に 1 回, ntpd のオフセットを計 140 回記録した. 表 1 に記録したオフセットの平均を示す. これによると, オフセットの最大値は約 6ms で平均は 1.41ms となっており, これは 3 節で DSSM の有用性の検証で用いる, 自律移動ロボットの制御システムの用件を満たす.

3. 実験

本節では, DSSM と 3 台のラズベリーパイを用いて自律移動ロボットを構築して走行実験を行うことにより, DSSM の有用性と制御システムに与える影響を検証する. 最初に, 実験に使用する自律移動ロボットのセンサとハードウェアの仕様について述べる. その後, 自律移動ロボットの制御システムについて述べ, 最後に本節で行う実験の詳細とその結果について述べる.

Vol.2021-ARC-244 No.22
Vol.2021-SLDM-194 No.22
Vol.2021-EMB-56 No.22
2021/3/25

表 2　ラズベリーパイ 3 Model B+の性能

名前	Raspberry Pi 3 model B+
CPU	Broadcom BCM2837B0, Cortex-A53 64-bit SoC @ 1.4GHz
メモリ	1GB LPDDR2 SDRAM
ネットワーク	Gigabit Ethernet over USB 2.0 (maximum throughput 300Mbps)
容量	8GB mini SD

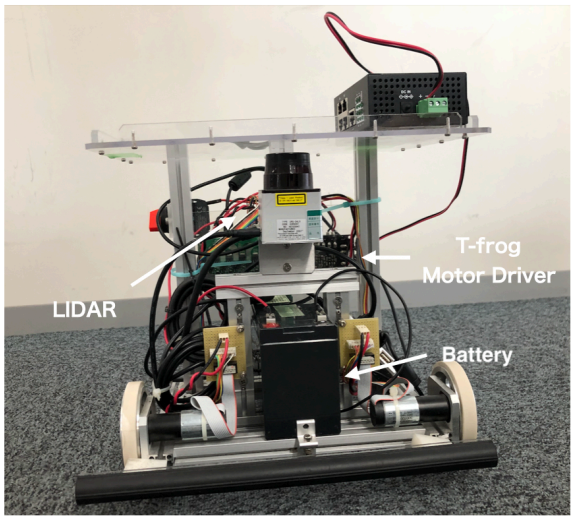

図 3　本実験で用いる自律移動ロボット

表 3　実験に使用するセンサ

Sensor	Name
Motor Driver	Tsuji Electronics TF-2MD3-R6
LiDAR	Hokuyo URG-04LX

3.1　センサとハードウェア

表 2 に実験に使用するラズベリーパイ 3 の性能を示す. また, 図 3 に本実験で用いる自律移動ロボットを示す. この自律移動ロボットには表 3 に示した 2 つのセンサが取り付けられている. 以下にセンサの機能の詳細を述べる.

Motor Driver　このセンサは自律移動ロボットの両車輪に取り付けられているロータリエンコーダから角速度を取得する.

Light Detection and Ranging (LiDAR)　このセンサは自律移動ロボットの正面に取り付けられ, ロボットから周囲の障害物までの距離をデータとして取得する.

3.2　自律移動ロボットの制御システム

実験を行う前に制御システムで使用する, 自律移動ロボットが走行する環境の情報を持つ地図 (以下, 環境地図) を作成する. 環境地図は, 自律移動ロボットを経路に沿って手動で走行させることによって, LiDAR センサから得られる自律移動ロボットと障害物との距離データを元に作成する. 図 4 に本実験で用いる自律移動ロボットの制御システムを

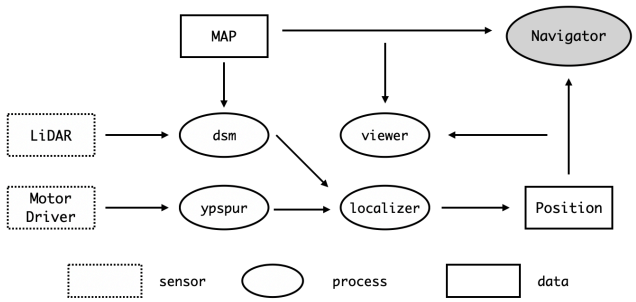

図 4　制御システム

示す. 制御システムでは自律移動のために主に 5 つのプロセスを実行する. 各プロセスは SSMAPi や PConnector を用いてデータをやり取りしながら動作する. 以下にこのプロセスの詳細について述べる.

(1) dsm は, LiDAR センサから取得した自律移動ロボットと周囲の障害物との距離データを用いて, 自律移動ロボットの自己位置を推定する処理である. この処理では, 環境地図を用いたスキャンマッチング [8] により自己位置を推定する. dsm で行うスキャンマッチングは 100ms 周期で最大 10,000 回のマッチングを試みるので, CPU 負荷が比較的大きい処理である. また, この自己位置推定ではパーティクルフィルター [10] を適用しているので, 推定位置は確率に依存することになる.

(2) ypspur は dsm と同じく, モータードライバで計測した角速度を用いてオドメトリを計算し, 自己位置を推定する処理である. この処理では, 最後に更新した自律移動ロボットの位置とオドメトリを加算して, 新たな自己位置を推定する. このプロセスでは 5ms の周期で処理が行われるが, スキャンマッチングと比較して十分に負荷の小さいプロセスになっている.

(3) localizer は dsm と ypspur で推定した自己位置推定の結果を融合し, 自律移動ロボットの自己位置を決定する処理である. 自己位置の融合には加重平均を用いており, より正確に自己位置を推定できるような重み値を採用している. また, このプロセスで推定した自己位置は ypspur と navigator でも用いられる.

(4) navigator は, 設定した経路に沿って自律移動ロボットを走行させる処理である. このプロセスでは, 環境地図と localizer で計算した自己位置を元に, 自律移動ロボットの次の進行方向を決定する.

(5) viewer は自律移動ロボットが現在の位置に移動するまでの軌跡を環境地図に表示する処理である. このプロセスでは localizer で計算した自己位置を環境地図を使って実際にディスプレイに表示するため, CPU 負荷は大きい.

Vol.2021-ARC-244 No.22
Vol.2021-SLDM-194 No.22
Vol.2021-EMB-56 No.22
2021/3/25

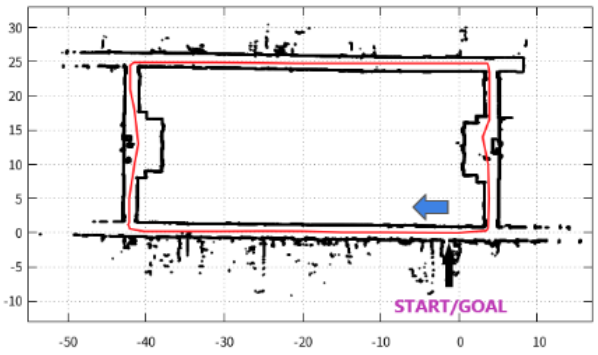

図 5　環境地図と自律移動ロボットの経路

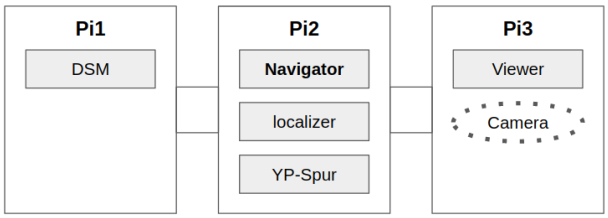

図 6　プロセスの配置

表 4　ラズベリーパイ 4 Model B の性能

名前	Raspberry Pi 4 model B
CPU	Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
メモリ	4GB LPDDR4-3200 SDRAM
ネットワーク	Gigabit Ethernet
容量	32GB mini SD

3.3　実験概要

図 5 に本実験で使用する環境地図と事前に設定した経路を示す．自律移動ロボットの経路はスタートとゴール地点が同位置で，矢印の方向，すなわち時計回りに走行し，環境地図を 1 周するように設定した．この経路は直線と 4 つの右折で構成されている．最初にこの経路の妥当性を検証するために予備実験として，SSM と DSSM を用いて 3 台のラズベリーパイ 3 (Pi1, Pi2, Pi3) に図 6 のように制御プロセスを分散させて走行実験を行った．3.2 節で述べたように，dsm と viewer は CPU 負荷の大きいプロセスであるため，それぞれ異なるラズベリーパイ (Pi1, Pi3) で動作させる．そのため，dsm は Pi1，navigator や ypspur, localizer などの処理を Pi2, viewer を Pi3 で実行する．予備実験は SSM, DSSM ともに同回数行ったが，実験の成否は安定せず，特に 3 つ目の右折地点を通り過ぎて壁に衝突してしまう現象が多発した．この原因を解明するために，実験中に共有メモリに書き込まれたデータを記録し，実験結果を分析した．その結果，この経路には目印となるような障害物が少なく，dsm で行うスキャンマッチングによる自己位置推定が難しく，3 つ目の右折地点において現実の位置と推定した位置に大きなずれが生じていたことが判明した．また DSSM を用いた場合，dsm の計算時間が SSM と比較してわずかに増加していることも判明した．これは今回使用する自律移動ロボットの制御には大きな影響を及ぼさなかったが，共有メモリのバッファの大きさを変更する必要が生じる可能性がある．そのため，この予備実験の結果を踏まえて本実験では，DSM の設定を一部見直した上で，DSSM の有用性とプロセスに与える影響を検証するために，DSSM と SSM を用いて自律移動ロボットを 10 回周回させる実験を異なる 3 つの環境で行った．

表 5　各実験の成功率

実験	SSM 成功率 (%)	DSSM 成功率 (%)
Ex 1	90	90
Ex 2	70	80
Ex 3	80	80

Ex1　予備実験と同様に 3 台のラズベリーパイ 3 と DSSM を用いて，図 6 のように制御プロセスを分散させて走行実験を行う．Ex1 では，SSM と DSSM を用いて同様の実験を同回数行うことで，TCP/IP 通信によるデータの送受信における遅延や NTP による時刻同期による影響が，自律移動ロボットを動作させるために十分に小さいことを確認する．実験中は *sar* コマンドを用いて CPU 使用率の計測を行い，SSM を使用した場合と DSSM を比較して，CPU 使用率がどの程度減少するかを検証する．また予備実験で判明した，DSSM がプロセスに与える影響を検証するために，実験中に共有メモリに書き込まれたデータも外部ファイルに記録する．

Ex2　Ex2 では，既存の制御システムに CPU 負荷の大きいプロセスを追加した場合に DSSM が PC に与える影響を検証する．そのため Ex2 では，カメラセンサとその処理を行うプロセスを追加する Pi3 に追加する．このプロセスは 100ms ごとにカメラセンサからデータを計測し，そのデータを処理した結果を Pi2 に書き込む．更に，カメラデータを目視で確認するためのビューワーも Pi3 で実行する．

Ex3　Ex3 では，制御システムで使用する PC の性能を向上させた場合に DSSM が PC に与える影響を検証するために，ラズベリーパイ 3 をラズベリーパイ 4 に置き換えて，Ex2 と同様の環境で実験を行う．表 4 にラズベリーパイ 4 の性能を示す．

3.4　実験結果

実験結果を表 5, 6, 7, 8 に示す．以下にそれぞれの実験の結果を述べる．

Ex1　表 5 にあるように，Ex1 では DSSM を用いてロボットを制御した場合にも SSM と同回数，経路を完走す

Vol.2021-ARC-244 No.22
Vol.2021-SLDM-194 No.22
Vol.2021-EMB-56 No.22
2021/3/25

表 6　Ex1 の CPU 使用率

PC	user	nice	system	iowait	idle	total
Pi_2(SSM)	42.89	0	7.43	***0.0083***	49.6	***50.40***
Pi_1(DSSM)	9.62	0	1.54	0.061	88.90	11.10
Pi_2(DSSM)	31.22	0	4.43	***0.106***	64.25	***35.75***
Pi_3(DSSM)	23.15	0	5.95	0.058	70.84	***29.16***

表 7　Ex2 の CPU 使用率

PC	user	nice	system	iowait	idle	total
Pi_2(SSM)	50.18	0	9.73	0.0082	39.7	***60.03***
Pi_1(DSSM)	9.49	0	1.56	0.055	88.88	11.12
Pi_2(DSSM)	28.66	0	4.24	0.15	66.98	33.01
Pi_3(DSSM)	27.39	0	7.40	0.071	65.13	***34.87***

表 8　Ex3 の CPU 使用率

PC	user	nice	system	iowait	idle	total
Pi_2(SSM)	46.83	0	11.31	0.026	41.83	***58.17***
Pi_1(DSSM)	6.98	0	3.81	0.031	89.18	10.82
Pi_2(DSSM)	26.26	0	3.64	0.027	70.07	***29.93***
Pi_3(DSSM)	22.18	0	8.86	0.014	68.99	31.01

表 9　dsm の計算時間の平均

実験	SSM(ms)	標準偏差	DSSM	標準偏差
Ex 1	***28.74***	8.49	***37.55***	12.23
Ex 2	***33.68***	15.58	36.58	12.11
Ex 3	14.35	5.39	21.1	5.16

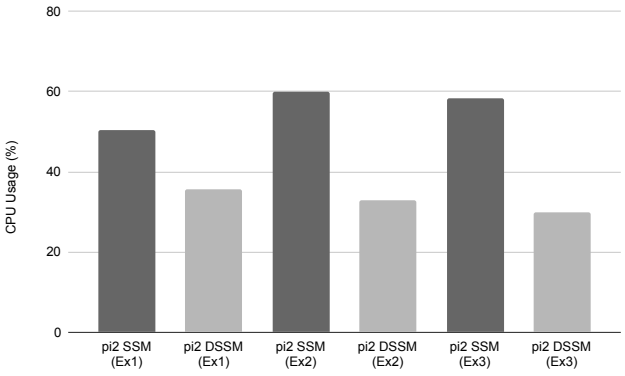

図 7　Pi2 の CPU 使用率

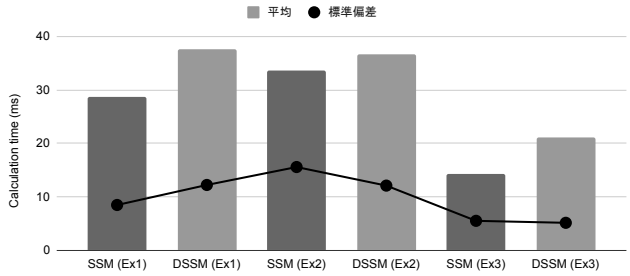

図 8　dsm の計算時間の平均と標準偏差

ることができた. これは TCP/IP 通信によるデータの通信遅延や NTP の時刻同期による影響が, 制御システムにとって十分に小さいことを示している. 表 6 を見ると, DSSM の Pi2 における CPU 使用率の合計は SSM と比較して, 50.40%から 35.75%に減少している. 一方で, CPU 使用率を詳しく見ると, *iowait* は SSM と比較して DSSM ではわずかに上昇している. これは DSSM はソケットを用いてデータを送受信しているため, SSM と比較して IO への負荷が増加したためだと思われる. また, 共有メモリに書き込まれたデータのログを分析したところ, dsm の計算時間の平均は SSM が約 28ms であるのに対し, DSSM では約 38ms と増加していることが分かった. 図 6 のあるように, DSSM において dsm の処理は 1 台のラズベリーパイで実行しているため, dsm の計算時間の平均は減少することはあっても増加することはないと我々は想定していた. そのため, この結果は想定外であった.

Ex2　Ex2 では経路を完走できた回数は SSM と DSSM のどちらも殆ど変わらず, SSM で 7 回, DSSM で 8 回であった. しかし表 7, 9 を見ると, カメラプロセスを追加した Ex2 の SSM は Ex1 と比較して 50.40%から 60.03%と約 10%増加し, dsm の計算時間の平均は 28.74ms から 33.68ms と約 5ms 増加している. 一方, DSSM では, カメラプロセスを追加した Pi3 の CPU 使用率は, 29.16%から 34.87%と約 5%増加しているが, 他のラズベリーパイの CPU 使用率と dsm の計算時間には殆ど変化がない. 更に, Ex1 と Ex2 の dsm の計算時間平均の標準偏差は SSM では約 2 倍になっているが, DSSM では殆ど変化がない. これらの結果から, DSSM では他 PC に CPU 負荷の大きいプロセスを追加しても, 制御システム全体に大きな影響を与えないことがわかる.

Ex3　Ex1, Ex2 と同様に実験の成功回数は SSM と DSSM のどちらも同じである. 表 8 を見ると, DSSM の CPU 使用率は SSM と比較して, Pi2 の CPU 使用率は 58.17%から 29.93%に減少している. 一方, 表 9 にあるように, ラズベリーパイ 4 を用いた場合にも, DSSM における dsm の計算時間の平均は SSM と比較して増加している.

4. 考察

本節では, 3 節で行なった実験とその結果を踏まえて DSSM の有用性について考察を行う.

最初に, 今回行った実験では, 自律移動ロボットは SSM と DSSM のどちらを用いた場合にもほぼ同回数, 経路を完走することができた. そのため, TCP/IP 通信によるデータの通信遅延や NTP の時刻同期による影響は十分に小さかったと考えられる. 一方で, SSM と DSSM のどちらを用いて自律移動ロボットを制御した場合も, 経路の完走に失敗した場合があった. これは dsm における自己位置推定の結果がパーティクルフィルターに依存しており, 確率に左右されるからである. この設定を今回の実験で用いる環境

Vol.2021-ARC-244 No.22
Vol.2021-SLDM-194 No.22
Vol.2021-EMB-56 No.22
2021/3/25

地図と経路に合わせて更に調整することで実験の成功確率を限りなく 100%に近付けること可能であったが, 本実験の目的は DSSM の有用性を示すことであったため行わなかった.

図 7 に各実験における Pi2 の CPU 使用率を示す. これを見ると, 全ての条件で DSSM の CPU 使用率は SSM と比較して約 15%以上減少している. このことから DSSM は, 複数の PC に CPU 負荷を分散させながら, SSM を用いた場合と同等の自律移動ロボットの制御を可能にしていることが分かる. また, Ex1 と Ex2 の SSM では CPU 使用率が約 10%増加しているのに対して, DSSM における Ex1 と Ex2 の CPU 使用率は殆ど同値である. これは, DSSM を使用することで CPU 負荷が大きいプロセスを限られたオーバーヘッドで異なる PC に分散することが可能であることを示している. これらの理由から, CPU リソースを大きく必要とするプロセスは DSSM を用いた方が有効に機能すると考えられる. さらに Ex2 と Ex3 の DSSM における Pi2 の CPU 使用率は, SSM の CPU 使用率の約半分ほどに削減できており, CPU リソースの多い PC を用いた場合にも DSSM は有効に動作すると考えられる.

その一方で, 今回の実験ではスキャンマッチングを用いて自律移動ロボットの自己位置推定を行う dsm の計算時間の平均が, 1 台のラズベリーパイで全てのプロセスを実行する SSM よりも, DSSM の方が大きいという結果が得られた. 図 8 に dsm の計算時間の平均と標準偏差を示す. この結果はログデータの分析によって, 今回行った全ての実験で当てはまることが分かった. そのため, DSSM を使用する開発者はこの遅延が発生することを考慮した上で, 指定した時刻のセンサデータを読み取ることができるように, 共有メモリのサイズを決定する必要がある. しかし, 図 8 の Ex1 と Ex2 の dsm の計算時間を比較するとわかるように, DSSM を用いて制御システムを構築した場合, 他 PC に CPU 負荷の大きいプロセスを追加したとしても, 計算時間に大きな変化はない. 更に Ex3 を見るとわかるように, より性能の高い PC を使用することでこの遅延は小さくすることができる.

以上の実験結果から, DSSM を用いることで複数台の PC から SSM と同様の制御システムを構築できることが分かった. また, DSSM を用いた場合, ネットワークを直接使用しない処理でもその影響を受けて, 計算時間が増加することがわかった. SSM を使用して構築された既存の制御システムを DSSM に移植する場合, この影響によって指定した時刻にセンサデータが共有メモリに存在しない可能性がある. ただし, この影響は使用する PC の性能によって変化し, また, SSM と異なり他 PC のプロセスの影響を受けないため, この遅延は予め計測した上で適切な PC を選択することで回避可能である.

5. 結論と今後の課題

近年, ロボットを制御するソフトウェアシステムを開発するコストを低減させるために, 様々なミドルウェアシステムが開発され, 使用されている. SSM もそのようなミドルウェアシステムの 1 つであり, タイムスタンプを用いてセンサデータの書き込みと読み出しを行う API を提供している. SSM 上では, センサデータの計測やハードウェアの操作などのロボットの機能要素は, それぞれ独立したプロセスとして動作する. そして, プロセス間のデータのやりとりには共有メモリを用いることで, 現在時刻から遡ってデータを参照することが可能にしている. しかし, SSM は共有メモリを使用するため, 1 台の PC で全ての制御プロセスを動作させる必要がある. そのため, CPU 負荷の大きいプロセスが他の影響をプロセスに影響を与えて, ロボットの制御プロセスに予期せぬ不具合が発生するという問題があった. この問題を解決するミドルウェアとして SSM を拡張して, 異なる PC にプロセスを分散できる仕組みを提供する DSSM がある. DSSM では, TCP/IP 通信を用いて SSM が管理する共有メモリを操作する API を開発者に提供する. この API を使用することで開発者はプロセスを複数の PC に分散することが可能になる. 本論文では, DSSM の有用性と制御システムに与える影響や DSSM のミドルウェアとしての特性を検証するために, ラズベリーパイを用いて以下の 3 つの実験を行った.

Ex1 DSSM と 3 台のラズベリーパイ 3 を用いて自律移動ロボットの制御システムを構築し, あらかじめ設定した経路を, SSM と DSSM で構築した自律移動ロボットそれぞれ 10 回ずつ走行させる.

Ex2 Ex1 の制御システムに新しいセンサとそれを処理する CPU 負荷の大きいプロセスを追加し, 同様の実験を行う.

Ex3 ラズベリーパイ 3 と比較して性能が高いラズベリーパイ 4 を用いて, Ex2 と同様の実験を行う.

その結果, DSSM を用いることで CPU 負荷を複数の PC に分散しながら, SSM と同様に自律移動ロボットを制御することが可能であることが分かった. また経路の妥当性を検証するための予備実験と本実験の結果から, DSSM ではネットワークを直接利用しないプロセスであっても, その影響によりプロセスの処理時間が SSM と比較して増加することがわかった. これは SSM で構築された既存の制御システムを DSSM に移植した場合, 指定した時刻のセンサデータが共有メモリに存在しないという問題が発生する可能性がある. しかしその影響は測定可能であり, また, PC の性能によってこの遅延は変化するため, 開発者はこの影響を考慮して制御システムを変更したり, ハードウェアを選択したりできる. また DSSM では, 小さいオーバーヘッ

ドでプロセスを異なる PC に分散することができるため, 負荷の大きいプロセスの追加も容易になる.

今後の課題としては, 実際の道路や不安定な路上での走行実験が必要であると考えられる. 本研究で使用した環境地図と経路は予備実験の結果にもあるように, 特徴が少なく, 自己位置推定が難しいという問題があった. そのため, SSM や DSSM に関係のない部分での実験の失敗が多発した. これを踏まえて, より特徴の捉えやすい環境での実験が必要だと考えられる. また, DSSM では TCP/IP 通信を使用したデータのやり取りを提供している一方で ROS/ROS2 や RTM ではリアルタイム性を重視するために UDP をベースとした通信プロトコルも選択肢として提供している. これを踏まえて, DSSM でもその実装を検討することが必要である.

参考文献

[1] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. Composite component framework for rt-middleware (robot technology middleware). In *IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pp. 1330–1335, Monterey, USA, July 2005.

[2] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and Woo-Keun Yoon. Rt-middleware: distributed component middleware for rt (robot technology). In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 3933–3938, 2005.

[3] Noriaki Ando, S. Kurihara, Geoffrey Biggs, T. Sakamoto, H. Nakamoto, and Tetsuo Kotoku. Software deployment infrastructure for component based rt-systems. *Journal of Robotics and Mechatronics*, Vol. 23, pp. 350–359, 06 2011.

[4] Michael Goodrich and Alan Schultz. Human-robot interaction: A survey. *Foundations and Trends in Human-Computer Interaction*, Vol. 1, No. 3, pp. 203–275, January 2007.

[5] James Kramer and Matthias Scheutz. Development environments for autonomous mobile robots: A survey. *Auton. Robots*, Vol. 22, No. 2, p. 101132, February 2007.

[6] David L.Mills. *Computer Network Time Synchronization: The Network Time Protocol*. CRC Press, 2006.

[7] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *Proceedings of the 13th International Conference on Embedded Software*, pp. 1–10, Pittsburgh, USA, November 2016.

[8] Kazuya Okawa. Self-localization estimation for mobile robot based on map-matching using downhill simplex method. *Journal of Robotics and Mechatronics*, Vol. 31, No. 2, pp. 212–220, 2019.

[9] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. Vol. 3, 01 2009.

[10] Sebastian Thrun. Particle filters in robotics. UAI'02, p. 511518. Morgan Kaufmann Publishers Inc., 2002.

[11] ROBOT PLATFORM PROJECT WIKI. Ssm, 2021/01/12 閲覧. `https://www.roboken.iit.tsukuba.ac.jp/platform/wiki/ssm/index`.

[12] 竹内栄二朗. ロボット用ミドルウェアを利用した自律移動ロボット開発. 計測と制御, Vol. 57, No. 10, pp. 741–744, 2018.

[13] 郡司凌太, 福田浩章. Dssm：時刻同期を考慮した分散型データストリーム管理ミドルウェア. ソフトウェア工学の基礎 26 日本ソフトウェア科学会 FOSE 2019 （レクチャーノート／ソフトウェア学）, pp. 157–162, 11 2019.