Vol.2023-OS-158 No.3
2023/2/21

ROS TFのトランザクショナル設計とタイムライン破損

荻原 湧志[1]　萬 礼応[2]　大矢 晃久[2]　川島 英之[3]

概要：ロボット用ミドルウェアの代表格である ROS(Robot Operating System) において，Transform Library(TF) ライブラリは各座標系間の変換情報を有向森構造で管理し，効率的な座標変換情報の登録，座標変換の計算を可能にした．ROS の中核的なパッケージであるにも関わらず，TF ライブラリには次の三つの問題があることがわかった．まず，非効率な並行性制御により，TF の有向森構造へのアクセスが完全に逐次化され，アクセスするスレッドが増えるに従ってパフォーマンスが低下する問題がある (P1)．次に，データの鮮度よりも時間的整合性を優先するため，座標変換を計算する際に最新のデータを参照しないという問題がある (P2)．最後に，有向森構造の変更によって座標変換の計算時に意図しないエラーが生じるという問題がある (P3)．本論文では，データベースの並行性制御法における 2PL 及び Silo を応用することにより，P1 と P2 を解決した．これらの手法は，読み取りのみのワークロードにおいて従来手法の最大 429 倍のスループットを示し，読み書きを組み合わせたワークロードにおいて従来手法の最大 1276 倍のデータ鮮度となることを示した．

キーワード：ROS，TF，トランザクション，並行性制御，Silo

1. はじめに

1.1 動機

Robot Operating System(ROS) はロボット分野で広く利用されており [1], [2]，その内部機構は，低オーバーヘッドのプロセス間通信 [3]，リアルタイム性 [4]，メニーコアアーキテクチャのための NoC 通信方式 [5] などを実現させるため，活発に改良されてきている．Transform Library(TF)[6] は，各座標系間の変換を有向森データ構造で管理し，効率的な座標系間の変換情報の登録，座標系間の変換の計算を可能にした．

図 1 の左側は部屋の中のロボットと，ロボットから観測される二つの物体がある様子を表しており，右側の木構造は部屋，ロボット，そして各物体の座標系の位置関係を表している．各座標系はノードで表現され，TF ではフレームと呼ばれる．フレーム間のエッジは，子ノードから親ノードへ座標変換情報 (CTI: Coordinate Transform Information) が有る事を示している．CTI を TF に登録するセンサデバイスはそれぞれ異なるセンシング周期を持つため，CTI は異なるタイミングで TF 上に登録される．時間的に整合性のあるデータを提供するため，TF は線形補間によって各フレーム間の座標変換情報を計算する．

[1] 慶應義塾大学政策・メディア研究科
[2] 筑波大学システム情報系
[3] 慶應義塾大学環境情報学部

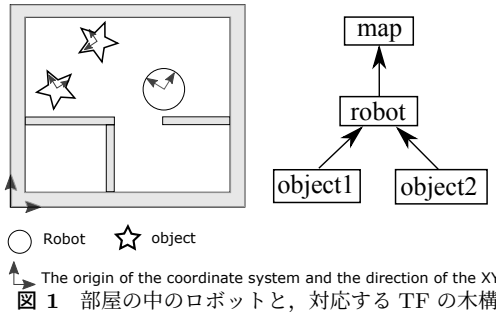

図 1　部屋の中のロボットと，対応する TF の木構造．

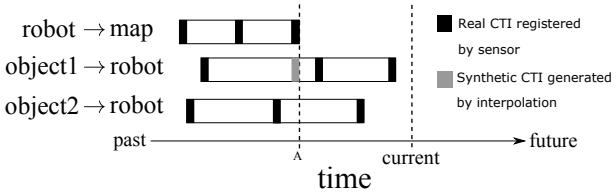

図 2　図 1 における位置情報の登録のタイムライン．

図 2 のタイムラインは，図 1 における各フレーム間の CTI が登録されるタイミングを表したものである．フレーム `object1` から `map` までの最新の CTI は 2 段階に分けて計算される．第一段階では，TF は `object1` から `map` までの全てのエッジにおいてそれぞれの最新の CTI の時刻を集め，それらのうち最も古い時刻を算出する．図 2 の場合，`object1`→`robot` と `robot`→`map` の最新の CTI のうち，時刻が古いのは時間 A で提供される `robot`→`map` の CTI なので，時刻 A が選択される．第二段階では，TF は

時刻 A における各フレーム間の CTI を取得，もしくは線形補間により生成し，これらを掛け合わせることによって `object1` から `robot` への最新の CTI を得る．変換と登録の手続きはそれぞれ lookupTransform(lookupTF) と setTransform(setTF) メソッドによって行われる．

1.2 研究課題

TF は複数の座標系間の位置関係を管理する，ROS の中核的なソフトウェアモジュールである．しかしながら，現在の TF の設計と実装には次の 3 つの問題がある．

(1) 問題 1：低いスケーラビリティ． 既存の TF は CTI を森構造で管理しているが，これはジャイアントロックを用いて排他制御を行っているため，多くのコアを搭載しているマシンにおいてもスループットやレイテンシーの面で性能劣化を引き起こす．

(2) 問題 2：データ鮮度の低さ． lookupTF メソッドはデータの鮮度よりも時間整合性を優先する仕様となっている．これにより，制御や自己位置推定の品質が劣化する可能性がある．また，この仕様によって，座標系間の位置関係が変化しない場合でも定期的に CTI を登録する必要があり，計算資源を不要に使うことになる．

(3) 問題 3：タイムラインの破損． 登録する CTI の親フレームの情報を変えることにより，有向森構造を変更することができるが，ユーザー側が正しく CTI を登録しないと，lookupTF メソッドでの CTI の計算時に意図しないエラーや無限ループが発生してしまう．

1.3 貢献

問題 1 を解決するために，自明な解決策として細粒度ロック [7] を使用することは不適切である．この手法は，ジャイアントロックとは異なり，複数のスレッドが同時に森構造にアクセスする際に各フレームをロックすることによって並行処理を可能にする．しかし，この方法ではフレームの挿入や削除が同時に実行された場合，逐次実行とは異なる結果になってしまう．これはアノマリーと呼ばれるが，これについては 2.2 章で説明する．

このようなアノマリーを防ぎ，且つ，Serializability を提供するために，我々は二相ロッキング (2PL) と Silo[8] という二つの並行性制御プロトコルに基づく *Transactional TF* を提案し，これらの方式をそれぞれ **TF-2PL** と **TF-Silo** と呼ぶ．Transactional TF は lookupTX と setTX という 2 つのインタフェースを提供し，これらはそれぞれ lookupTF と setTF を並行制御プロトコルで改良したものである．

本研究では，並行性制御プロトコルとして 2PL と Silo を採用し，それらの結果を比較した．その結果，224CPU コアでの読み取り専用ワークロードにおいて，TF-Silo は既存手法に比べ最大で 429 倍のスループット，445 倍小さいレイテンシを実現することを明らかにした．

問題 2 を解決するために，lookupTX に最新のデータからフレーム間の座標変換を計算する機能を追加した．本方式は細粒度ロックと異なり，並行性制御プロトコルを用いているためにアノマリーは発生しない．既存方式と比較し，TF-Silo を用いた場合にはデータの鮮度は最大 1276 倍となった．

問題 3 の解決策については，6 節で述べるように今後の課題とする．

また，現実的なワークロードとして，複数台の自動運転車を管理するシナリオをエミュレートしたところ，従来方式が線形劣化するのに対し，TF-2PL と TF-Silo は安定して低いレイテンシを示すことがわかった．

従来の TF ライブラリのソースコードに 1964 行の変更を加え，この新しいコードを [9] に公開した．

本論文はこれまでの研究を拡張したものであり，今回新たにフレームの挿入・削除によるタイムラインの破損，及びアノマリーについての内容を追加した．我々の知る限り，スケーラビリティ，データ鮮度，および一貫性のために ROS にトランザクションの概念を組み込んだものは本研究をおいて他にない．

1.4 構成

本論文の構成は次の通りである．第二節では既存の TF の森構造とその問題点について述べる．第三節ではデータの鮮度・一貫性を確保する提案手法について述べる．第四節では提案手法の評価結果を述べる．第五節では関連研究について述べ，第六節では本研究の結論と今後の課題について述べる．

2. TF: Transform Library

TF は複数の座標系間の位置関係を管理し，座標系間の CTI(Coordinate Transformation Information) を有向森構造で管理する．木構造内のノードはフレームと呼ばれ座標系を表し，フレーム間のエッジは CTI がそのフレーム間に存在することを示す．各フレームはその親フレームに対する CTI を管理する．つまり，子フレームにアクセスすることで子フレームと親フレーム間の CTI にアクセスできる．

2.1 インターフェース

TF には 20 個ほどのインターフェースがあるが [10]，その中で最もよく使われるのが lookupTF と setTF である．

2.1.1 lookupTF インターフェイス

このインターフェイスはソースフレームからターゲットフレームまでの最新の CTI を計算するもので，二つのフレーム間のパスを以下の二段階で走査することにより計算される．

(1) 第一段階では，ソースフレームからターゲットフレー

ムまでのパス上の各フレームの CTI の最新時刻を算出する．TF はフレーム間の CTI を両端キューを使って記憶しており [11]，最新の CTI はこのキューの頭要素に在るため，この段階ではパス上のすべての頭要素のタイムスタンプを収集する．そして，TF はそれらの中から最小のタイムスタンプ t_{min} を選択する．

(2) 第二段階では，第一段階で得られた t_{min} を基に，ソースフレームからターゲットフレームまでの座標変換を計算する．ソースフレームからターゲットフレームまでのパスにおける各フレームの t_min における CTI は，t_min 以前の最も新しい CTI と t_min 以降の最も古い CTI の 2 つによる線形補間を用いて計算される．次に，パス上の t_min における CTI を掛合わせることにより，TF は時間的一貫性のある座標変換を取得する．フレームの挿入や削除により，木構造に変化が生じてあるフレームの親フレームが変化することがある．このため，線形補間時の t_min 前後の 2 つの CTI の親が異なる場合には，古い方の CTI を使用する [12].

2.1.2 setTF インターフェイス

setTF インターフェイスは，新しい CTI をタイムラインに追加する．上述したように，タイムラインは両端キューとして実装されるので，時間 t の新しい CTI は t 以前の最も新しい CTI の後にキューに挿入され，古い CTI はキューの端から追い出される．

2.2 問題点

問題 1:低いスケーラビリティ

lookupTF と setTF は複数のスレッドから同時に呼び出されることがある．メニーコアアーキテクチャにおいては，複数のスレッドによる並列アクセスをサポートするために，基盤となるデータ構造が並行処理をサポートしている必要がある．しかしながら，TF ではジャイアントロック によって森構造全体が保護されるため，このデータ構造には 2 つ以上のスレッドが同時にアクセスすることはできない．この設計は，メニーコアアーキテクチャが提供する並列性を完全に排除し，スループットとレイテンシーの面で性能劣化を引き起こす．

問題 2:データ鮮度の低さ

2 フレーム間の座標変換を計算する際に，lookupTF では 2 フレーム間のパス上全てのフレームが CTI を提供できる時刻において算出されるため，各フレームの最新の CTI が使われないという問題がある．これにより時間整合性は取れるが，データの鮮度が落ちるため，制御や自己位置推定における品質を劣化させる可能性がある．現在，TF は最新の CTI に基づいてフレーム間の座標変換を計算するインタフェースを提供していない．

図 3 は，座標系間の位置関係があまり変化しないため

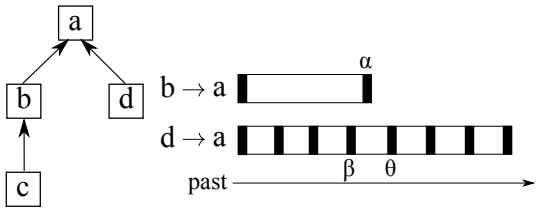

図 3 不必要な更新が必要な例.

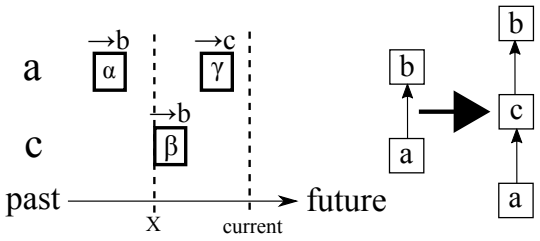

図 4 TF の仕様により，過去の木構造を参照する例.

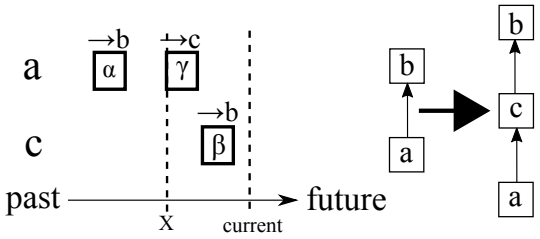

図 5 タイムラインが破損する例 1.

b→a の CTI は頻繁に更新されないが，d→a の CTI は頻繁に更新される場合を示している．b から d への座標変換を計算するために b→a と d→a を使う場合，TF は古い β と θ を使用しなければならない．これを防ぐため，座標系間の位置関係が全く変わらなくても，TF においては定期的に CTI を登録し直す必要がある．この仕様はプロセッサ、メモリ、ネットワークを不必要に消費する．

問題 3:タイムラインの破損

TF では，登録する CTI の親フレームの情報を変えることによって有向森構造を変更することができる．しかしながら，ユーザーが誤った方法で CTI を登録することによって lookupTF でのフレーム間座標変換の計算時に意図しないエラーが発生してしまう．ここでは，このような状態をタイムラインの破損と呼ぶ．

2.1.1 節で説明したように，木構造の変化によって線形補間時の 2 つの CTI の親フレームが異なる場合には，古いほうの CTI が使用される．例えば，図 4 で時刻 **current** にて a から b への座標変換を lookupTF で計算すると，β と γ を参照することにより，第一段階で時刻 X で線形補間を行うことが決定される．これにより，第二段階では α を参照し，結果的に最新の木構造ではなく過去の木構造を参照する，という現象が発生する．これは，a と b の間に c が挿入されることにより発生している．この TF の仕様は，これから説明するように意図しない実行時エラーや無限ループの発生につながる．

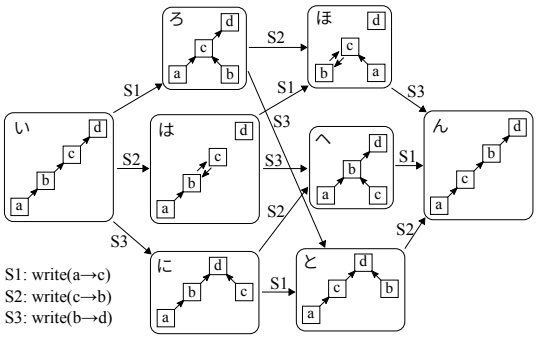

図 6 逆転時の状態遷移図.

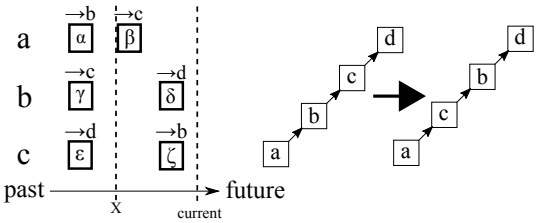

図 7 タイムラインが破損する例 2.

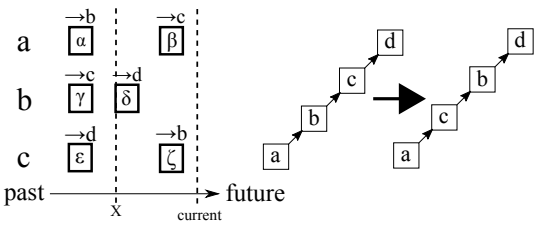

図 8 タイムラインが破損する例 3.

図 5 は図 4 のタイムラインにおいて β と γ の時刻関係を逆転したものである．時刻 X で線形補間が行われるが，フレーム c は時刻 X での線形補間ができず，実行時エラーとなってしまう．

図 6 は，a→b→c→d という構造において，b と c の位置関係を逆にする，つまりフレーム間の位置関係が逆転する場合の状態遷移図を表している．b と c の位置を逆転させるには，

- a の親を c に変える．
- c の親を b に変える．
- b の親を d に変える．

という三つの操作が必要となり，それぞれ S1，S2，S3 と名付けた．各操作を適用する順番が変わることにより，**ろ**，**は**，**に**，**ほ**，**へ**，**と**，という六つの構造変化中の状態が存在する．

図 7 は，b と c の逆転において正しく CTI が登録されないことによりタイムラインが破損する例を示している．a から d への lookupTF を行うと，β, δ, ζ を読むことにより，第一段階では時刻 X で線形補間を行うことが決定される．第二段階では β, ϵ を読むが，これは中間状態である，状態**ろ**，を参照することになってしまう．

同じようにして，図 8 で a から d への lookupTF を行うと、中間状態である状態**に**を参照してしまう。図 9 で a から d への lookupTF を行うと、中間状態である状態**は**を参照し、無限ループに陥ってしまう。

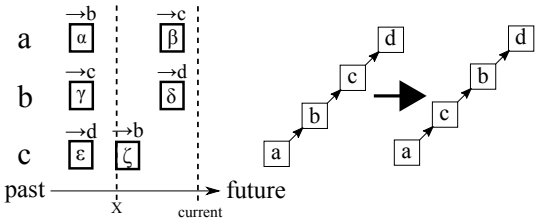

図 9 タイムラインが破損する例 4.

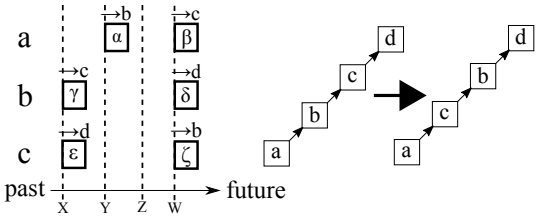

図 10 逆転によってアノマリーが発生する例.

表 1 アノマリーを発生させるスケジュールの例.

$t1$: lookupTF(a→d)	Time	$t2$
read(a(α))	Z	
	W	setTF(a→c(β))
	W+1	setTF(b→d(δ))
	W+2	setTF(c→b(ζ))
read(b(δ))	W+3	
read(a(α))	W+4	
read(b(γ))	W+5	
read(c(ϵ))	W+6	

自明な解決策とその罠

問題 1 を解決するために，自明な解決策として細粒度ロックを用いることは不適切である．この方法は，複数のスレッドが同時に森構造にアクセスする際に各フレームをロックすることによって並行処理を可能にする．しかし，この方法ではフレームの挿入や削除が同時に実行された場合，逐次実行とは異なる結果になってしまう．

図 10 と表 1 は，木構造の変化中と lookupTF が並行に実行された時に アノマリーが発生する例を表している．図 10 はフレーム b と c が逆転するタイムラインを表している．ここで，タイムラインの破損が発生しないように β, δ, ζ を同じ時刻に登録している．表 1 はアノマリーが発生するスケジュールを表している．まず，スレッド $t1$ が lookupTF(a→ d) の処理を時刻 Z で開始する．α を読むことにより，このスレッドは次に b を読む．ここで，スレッド $t2$ が始まって β, δ, ζ を登録する．$t1$ の処理に戻り，δ を読むことで b の親が d であることを確認する．これにより，lookupTF の第一段階が完了し，第二段階では時刻 Y で線形補間を行うことを決定する．第二段階が始まって α, γ, ϵ を読み，それらを掛け合わせて処理を完了

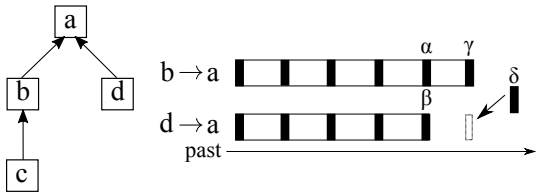

図 11 CTI が同時に登録されるケース.

する．この実行結果は，$t1$ と $t2$ が直列に実行された場合とは異なる結果になる．$t2$ が始まるより前に $t1$ が始まった場合，lookupTF は第一段階で時刻 X を選択し，$t1$ が始まるより前に $t2$ が始まった場合には lookupTF は第一段階で時刻 W を選択する．これは，表 1 のスケジュールにおいて，lookupTF は第一段階において図 6 の状態**へ**を参照していることに起因する．したがって，このスケジュールでは Serializability が失われている．

表 2 アノマリーを発生させるスケジュールの例 2.

$t1$: lookupTF-latest(b→d)	Time	$t2$: γ と δ を同時刻に登録
	X	setTF(b→a(γ))
read(b(γ))	X+1	
read(d(β))	X+2	
	X+3	setTF(d→a(δ))

問題 2 を解決するために，最新の CTI に基づいてフレーム間の座標変換の計算を行うインターフェイス (ここでは lookupTF-latest と呼ぶことにする) を提供するだけでは不十分である．図 11 と表 2 は，最新の CTI をもとに b から d への座標変換を計算する lookupTF-latest と，b と d の各フレームの最新の CTI を同時刻に登録するスレッドが同時に実行される時の様子を表している．このタイムラインでは b と d の CTI が同時刻で登録されているため，ユーザーは lookupTF-latest においても同時刻の CTI が使われることを期待する．しかしながら，setTF が複数の CTI をアトミックに登録できないことに起因して表 2 のスケジュールのように異なる時刻の γ と β が使われてしまうことがある．

3. Transactional TF

我々は，二相ロッキング (2PL) と Silo [8] という二つの並行性制御プロトコルを基にした *Transactional TF* を提案し，これらをそれぞれ **TF-2PL**，**TF-Silo** と名付ける．2PL と Silo は Serializable なので，我々の手法も同様に Serializable である．2PL は悲観的アルゴリズムによって競合が多いワークロードに適しており，対して Silo は楽観的アルゴリズムによって競合が少ないワークロードに適している [13].

3.1 デザイン

Transactional TF は，lookupTX と setTX という 2 つのインタフェースを提供する．これらは，それぞれ並行性制御プロトコルによって lookupTF と setTF が Serializable であるように改善した版である．lookupTX はソースフレームからターゲットフレームへの座標変換情報を時間的整合性を保ちながらアトミックに計算し，setTX は複数の CTI をアトミックに書き込む．これらのインタフェースを用いることで，複数のスレッドから CTI を Serializable に効率的に読み書きすることができる。ここで，lookupTX は時間的整合性を保つために過去の CTI から計算するために Serializability のみ提供する (つまり，古いデータを読み込むために linearizablity がない [7]) ことを注記しておく．

問題 2 を解決するために，lookupTX に各フレームの最新 CTI のからフレーム間の座標変換を計算するオプションを追加した．このオプションを使うと時間的整合性は失われるが，各フレームの最新の CTI を使用することで，フレーム間のパスにおいて最新の座標変換を提供することができる．さらに，この機能を使えば 2.2 節で説明した不要な CTI の登録を防ぐこともできる．2PL と Silo は strict serializable(つまり，Serializable かつ linearizable[7], [14]) であり，このオプションは最新の CTI を使うので，このオプションも strict serializablity を保証する．

アルゴリズム 1 に TF-2PL の処理を処理を記述した．木構造を読むため， lookupTX-2PL はソースフレームからターゲットフレームまでのパスを二回辿る．第一段階では，パス内の最新の CTI のタイムスタンプのうち最も古い時刻を *getCommonTime* プロシージャによって計算する．この遷移中に，lookupTX-2PL は各フレームを読み込みロックする．これは 2PL における成長フェーズに該当する．第二段階では，第一段階で取得した時間でのパス上の CTI を算出し，それらを掛け合わせる．このパスは 2.1.1 節で説明したように第一段階のものとは異なる場合があるため，lookupTX-2PL は第二段階においてもアクセスする各フレームを読み込みロックし，これも成長フェーズとなる．フレーム間の座標変換の計算の完了後，全てのロックは収縮フェーズにて解放される．3 行目にて説明されているように，getCommonTime プロシージャをスキップすることで，各フレームの最新の CTI を使用してフレーム間の座標変換を計算する．

木構造を更新するために setTX-2PL は lookupTX-2PL と同じように各フレームをロックするが，競合の解決方法が異なる．デッドロックを防ぐため，setTX-2PL は競合を検知した場合は即座に全てのロックを解放し，処理を初めからやり直す．この技法は *2PL-NoWait* [13] と呼ばれている．lookupTX-2PL で成長フェーズにて競合を検知した場合には，全てのロックを解放して処理をやり直す代わりに，ロックが解放されるまで待機することを注記しておく．

Algorithm 1 TF-2PL

```
 1: function lookupTX-2PL(bool use_latest)
 2:     time := TIME_LATEST;
 3:     if use_latest then
 4:         time := getCommonTime-2PL();
 5:     for f :=source; f ≠target; f := f.next do
 6:         f.readLock();                              ▷ 成長フェーズ
 7:         src_trans×= f.trans_at(time);
 8:     end for
 9:     ∀f.unlock(); return src_trans;                 ▷ 縮小フェーズ
10: end function
11: procedure getCommonTime-2PL
12:     common_time := TIME_MAX;
13:     for f :=source; f ≠target; f := f.next do
14:         f.readLock();                              ▷ 成長フェーズ
15:         common_time := min(f.latest_time, common_time);
16:     end for
17:     return common_time;
18: end procedure
19: function setTX-2PL
20:     for c ∈ CTI set do                             ▷ 成長フェーズ
21:         f := c.getFrame();
22:         if f.tryWriteLock() = FAIL then
23:             ∀f.unlock() and retry;
24:     end for
25:     for c ∈ CTI set do
26:         f := c.getFrame(); f.update(c);
27:     end for
28:     ∀f.unlock();                                   ▷ 縮小フェーズ
29: end function
```

Algorithm 2 TF-Silo

```
 1: function lookupTX-Silo(bool use_latest)
 2:     time := TIME_LATEST;
 3:     if use_latest then
 4:         time := getCommonTime-Silo();
 5:     for f :=source; f ≠target; f := f.next do ▷ Read phase
 6:         copy f to local buffer;
 7:         src_trans ×= f.trans_at(time);
 8:     end for
 9:     if ∃f is changed or latched                ▷ Validation phase
10:         retry;
11:     return src_trans;
12: end function
13: procedure getCommonTime-Silo
14:     common_time = TIME_MAX;
15:     for f :=source; f ≠target; f := f.next do ▷ Read phase
16:         copy f to local buffer;
17:         common_time := min(f.latest_time, common_time);
18:     end for
19:     return common_time;
20: end procedure
21: function setTX-Silo
22:     Sort CTI;
23:     for c ∈ CTI set do                         ▷ Lock phase
24:         f := c.getFrame(); f.latch();
25:     end for
26:     for c ∈ CTI set do                         ▷ Write phase
27:         f := c.getFrame(); f.update(c);
28:     end for
29:     ∀f.unlatch();                              ▷ Unlock phase
30: end function
```

TF-Silo の処理の流れをアルゴリズム 2 に示した．木構造を更新するには，setTX-Silo はまず各フレームをラッチまたはアンラッチの状態を 1 ビットで管理し，アクセスする全てのフレームをラッチする．競合が発生した場合には，setTX-2PL のように処理を始めからやり直すのではなく，ラッチが解放されるまで待機する．全てのラッチを取得した後，各フレームの CTI を更新し，ラッチを解放する (これは Silo コミットプロトコルの *lock phase* と *write phase* に対応する)．

木構造を読むため，lookupTX-Silo はまずソースフレームからターゲットフレームまでのパス上のフレームをスレッドローカルなバッファにコピーする (これは Silo の *read phase* に対応する)．その後，すべてのフレームが更新されていないか，またはラッチされていないかをチェックし競合の検知を行う．競合が検知されなかった場合には，算出結果を返し (これは Silo のコミットプロトコルの *validation phase* に対応する)，競合が検知された場合には処理を始めからやり直す．

3.2 実装

TF ライブラリの実装は Github のリポジトリ [15] で公開されている．このリポジトリのデフォルトブランチは neotic-devel であるが，ROS2 のバージョンを含め TF の木構造の並行性制御アルゴリズムはどのブランチでも変わらない．このため，本研究では Ubuntu18.04 を搭載したマシンに ROS Melodic Morenia をインストールし，TF ライブラリの Github のリポジトリ [15] の melodic-devel ブランチの実装を変更して実験を行った．C++言語で実装をし，1964 行分の変更を行った．この実装は [9] で公開されている．

4. 評価

提案手法と既存手法を比較するため，以下二つのワークロードにおいてそれぞれを評価した：

(1) 1,000,000 万のフレームが連結している TF の木構造に，最大 224CPU コアから並行にアクセスするワークロード．

(2) 一台のエッジサーバで複数の自動車の位置情報を管理するワークロード．

その結果，ワークロード (1) において提案手法は既存手法に比べ最大で 429 倍のスループット，445 倍小さいレイテンシ，1276 倍のデータ鮮度となった．また，ワークロード (2) において既存手法がスレッド数の増加に対して線形劣化するのに対し，提案手法は安定して低いレイテンシを示すことがわかった．詳細は，[16] にて記載した．

5. 関連研究

ROS の内部構造は積極的に改善されている．LOTROS[3] は `nodelet` パッケージを置き換える，オーバーヘッドの小さい IPC を提案した．CompROS は ROS2 向けのリアルタイム機能を提案し，ROS-lite[5] はメニーコアの組込みプラットフォーム向けの NoC 通信方式を提案した．

ROS の他にも，ロボット用ミドルウェアの研究が盛んに行われている．GAIA [1] はデータベースと C++を組み合わせたイベント駆動型フレームワークであるが，その技術的な詳細は明らかにされていない．

Sensor sharing manager (SSM) [17] は TF のような時系列データを管理するライブラリである．各種センサデータを共有メモリ上のリングバッファで管理することで，複数のアプリケーションで時間的に同期したデータを高速に取得することができる．

インメモリデータベースの並行性制御を高速化するために，Siloo [8], MOCC [18], TicToc [19], latch-free SSN [20], [21], [22], Cicada [23] などの様々な手法が提案されている．しかしながら，これらのデータベース向けのプロトコルは YCSB や TPC-C などの Web や在庫管理を想定したベンチマークで評価されており，ロボットへの応用は考慮されていない．

我々の知る限り，我々の既存研究 [16], [24] を除いて，本研究はトランザクションのコンセプトを ROS のコアモジュールに組み込むものとして初めての試みである．本研究は我々の既存研究の内容に加え，タイムライン破損，フレームの挿入・削除によるアノマリーの説明を新たに追加した．

6. 結論と今後の課題

TF は座標系間の変換情報を管理する ROS の中核的なモジュールであり，ジャイアントロックによって並行処理が行えない (P1)，最新の CTI から座標変換を計算するインターフェイスが無い (P2)，タイムラインの破損という現象がある (P3)，という三つの問題がある．

我々は，並行性制御プロトコルに基づく Transactional TF を提案することにより，P1 及び P2 を解決した．本手法は，並行性制御プロトコルの Serializability に基づいてアノマリーを回避した上でスケーラビリティを保証する．また，最新の CTI から座標変換を計算するインターフェイスも追加した．

提案手法である TF-2PL and TF-Silo はスループット，レイテンシ，データ鮮度の面において既存手法を凌駕することを示し，また現実的なシナリオにおいても効果的であることを示した．P3 の解決については，今後の課題とする．

本研究は，トランザクショナル設計によって ROS の高精度化，高品質化を実現した．これは筆者らの知る限り，これまでにないアプローチである．これはデータベースとロボットを融合した研究領域の第一歩であり，この研究領域の開拓を進める．

謝辞 この成果は，科研費 JP19H04117 ならびに国立研究開発法人新エネルギー・産業技術総合開発機構 (NEDO) の委託業務（JPNP16007）の結果得られたものです．

参考文献

[1] : GAIA platform, www.gaiaplatform.io.
[2] Kato, S., Tokunaga, S., Maruyama, Y., Maeda, S., Hirabayashi, M., Kitsukawa, Y., Monrroy, A., Ando, T., Fujii, Y. and Azumi, T.: Autoware on board: enabling autonomous vehicles with embedded systems, *ICCPS* (2018).
[3] Iordache, C., Fendyke, S. M., Jones, M. J. and Buckley, R. A.: Smart Pointers and Shared Memory Synchronisation for Efficient Inter-process Communication in ROS on an Autonomous Vehicle, *IROS* (2021).
[4] Dehnavi, S., Koedam, M., Nelson, A., Goswami, D. and Goossens, K.: CompROS: A composable ROS2 based architecture for real-time embedded robotic development, *IROS* (2021).
[5] Azumi, T., Maruyama, Y. and Kato, S.: ROS-lite: ROS Framework for NoC-Based Embedded Many-Core Platform, *IROS* (2020).
[6] Foote, T.: tf: The transform library, *TePRA* (2013).
[7] Weikum, G. and Vossen, G.: *Transactional Information Systems*, Morgan Kaufmann Publishers Inc. (2002).
[8] Tu, S., Zheng, W., Kohler, E., Liskov, B. and Madden, S.: Speedy Transactions in Multicore In-Memory Databases, *SOSP* (2013).
[9] Ogiwara, Y.: Transactional TF, github.com/Ogiwara-CostlierRain464/geometry2.
[10] Foote, T.: buffer_core.cpp, github.com/ros/geometry2/blob/melodic-devel/tf2/src/buffer_core.cpp.
[11] Foote, T.: time_cache.h, github.com/ros/geometry2/blob/noetic-devel/tf2/include/tf2/time_cache.h.
[12] Foote, T.: time_cache.cpp, github.com/ros/geometry2/blob/noetic-devel/tf2/src/cache.cpp.
[13] Tanabe, T., Hoshino, T., Kawashima, H. and Tatebe, O.: An Analysis of Concurrency Control Protocols for In-Memory Databases with CCBench, *PVLDB*, Vol. 13, No. 13 (2020).
[14] Herlihy, M. and Shavit, N.: *The Art of Multiprocessor Programming, Revised Reprint*, Morgan Kaufmann Publishers Inc. (2012).
[15] Foote, T.: TF, github.com/ros/geometry2.
[16] Ogiwara, Y., Yorozu, A., Ohya, A. and Kawashima, H.: Transactional Transform Library for ROS, *IROS* (2022).
[17] Takeuchi, E. and Tsubouchi, T.: Sensory data processing middlewares for service mobile robot applications, *SICE-ICASE* (2006).
[18] Wang, T. and Kimura, H.: Mostly-Optimistic Concurrency Control for Highly Contended Dynamic Workloads on a Thousand Cores, *PVLDB*, Vol. 10, No. 2 (2016).
[19] Yu, X., P. A. S. D. and Devadas, S.: TicToc: Time Traveling Optimistic Concurrency Control, *SIGMOD*, pp. 1629–1642 (2016).
[20] Wang, T., J. R. F. A. and Pandis, I.: The Serial Safety Net: Efficient Concurrency Control on Modern Hard-

ware, *DaMoN* (2015).
[21] Kim, K., W. T. J. R. and Pandis, I.: ERMIA: Fast Memory- Optimized Database System for Heterogeneous Workloads, *SIGMOD*, pp. 1675–1687 (2016).
[22] Wang, T., J. R. F. A. and Pandis, I.: Efficiently Making (Almost) Any Concurrency Control Mechanism Serializable, *VLDB*, Vol. 26, No. 4, pp. 537–562 (2017).
[23] Lim, H., K. M. and Andersen, D.: Cicada: Dependably Fast Multi-Core In-Memory Transactions, *SIGMOD*, pp. 21–35 (2017).
[24] Ogiwara, Y., Yorozu, A., Ohya, A. and Kawashima, H.: Making ROS TF Transactional, *ICCPS Poster* (2022).