環境情報学部論文

TF ライブラリの高性能化とデータ一貫性の確保

2022 年 1 月

71970013 / t19501yo
荻原　湧志

TF ライブラリの高性能化とデータ一貫性の確保

慶應義塾大学
環境情報学部
2022 年 1 月
荻原　湧志

TF ライブラリの高性能化とデータ一貫性の確保
Consistent and scalable TF library

学籍番号：71970013 / t19501yo
氏名：荻原　湧志
Yushi Ogiwara

Robot Operating System(ROS) はロボットソフトウェア用のミドルウェアソフトプラットフォームであり、近年多くの研究用ロボットで用いられている。TF ライブラリは ROS で頻繁に使用されるパッケージであり、ロボットシステム内の座標変換を追跡し、データを変換する標準的な方法を提供するために設計されたものである。ROS の開発初期には複数の座標変換の管理が開発者共通の悩みの種であると認識されていた。このタスクは複雑なために、開発者がデータに不適切な変換を適用した場合にバグが発生しやすい場所となっていた。また、この問題は異なる座標系同士の変換に関する情報が分散していることが多いことが課題となっていた。そこで、TF ライブラリは各座標系間の変換を有向森構造として管理し、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。しかしながら、この有向森構造にはデータの暗黙的な線形補間による一貫性の欠落、及び非効率な並行性制御により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という問題があることがわかった。そこで、我々はデータベースのトランザクション技術における再粒度ロッキング法、及び並行性制御のアルゴリズムの一種である 2PL を応用することにより、この問題を解決した。提案手法では、スレッド数が 12 までスケールアップすることを示した。また、多くのアクセスパターンにおいて提案手法は既存手法より高いスループットを出すことを示した。

研究指導教員：川島 英之

目 次

図 目 次

第1章　はじめに

1.1　研究背景

ロボットを使って作業を行う場合、ロボット自身がどこにいるのか、ロボットにはどこにどんなセンサーがついており、また周りの環境のどこにどんなものがあるかをシステムが把握することが重要である。例えば、図 1.1 のように部屋の中にロボットと、ロボットから観測できる二つの物体があるケースを考える。図中にてロボットは円形、物体は星形で表現される。ロボットが向いている方向は円の中心から円の弧へつながる直線の方向で表している。途中で交わる二つの矢印は各座標系の位置と原点、姿勢を表す。ここでは、地図座標系、ロボットの座標系、二つの物体それぞれの座標系が示されている。

システムはロボットに搭載されたセンサーからのデータを元に各座標系間の位置関係を随時更新する。座標系間の位置関係は並行移動成分と回転成分で表現できる。例えば、自己位置推定プログラムは LiDAR から点群データが送られてくるたびにそれを地図データと比較して自己位置を計算し、ロボットが地図座標系にてどの座標に位置するか、ロボットがどの方向を向いているかといった、地図座標系からロボット座標系への位置関係を更新する。物体認識プログラムはカメラからの画像データが送られてくるたびに画像中の物体の位置を計算し、ロボット座標系から物体座標系への位置関係を更新する。

このように、各座標系間の位置関係の更新にはそれぞれ異なるセンサー、プログラムが使われる。各センサーの計測周期、及び各プログラムの制御周期は異なるため、各座標系間の位置関係の更新頻度も異なるものとなる。図 1.2では、地図座標系からロボット座標系への位置関係データと、ロボット座標系から物体座標系への位置関係データがそれぞれ異なるタイミングで登録されていることを示している。

ここで地図中での物体の位置を把握するために、地図座標系から物体座標系への位置関係を取得する方法について考える。地図座標系から物体座標系への位置関係は地図座標系からロボット座標系への変換とロボット座標系から物体座標系への変換を掛け合わせれば計算ができるが、図 1.2のように各変換データは異なるタイミングで来るため、最新の変換データを取得するプログラムは複雑なものとなる。A の時刻で地図座標系から物体座標系への変換データを計算しようとするとロボット座標系から物体座標系への最新の変換データを取得できるが、地図座標系からロボット座標系への変換データはまだ取得できない。このため、最新の変換データ θ を取得する、もしくは過去のデータを元にデータの補外をすると必要がある。B の時刻で地図座標系から物体座標系への変換データを計算しようとすると地図座標系からロボット座標系への最新の変換データを取得できるが、ロボット座標系から物体座標系への変換データはその時間には提供されていない。このため、α と β のデータから線形補間を行う、もしくは最新の変換データ β を取得する必要がある。また、地図座標系からロボット座標系への変換とロボット座標系から物体座標系への変換は別のプログラムで管理されており、座標系同士の変換に関する情報が分散している。

このように、ROS の開発初期には複数の座標変換の管理が開発者共通の悩みの種であると認識されていた。このタスクは複雑なために、開発者がデータに不適切な変換を適用した場合にバグが発生しやすい場所となっていた。また、この問題は異なる座標系同士の変

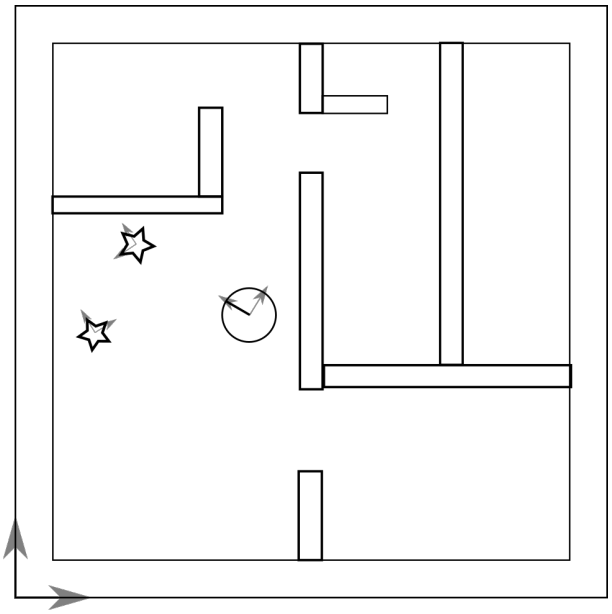

図 1.1: 部屋の中のロボット

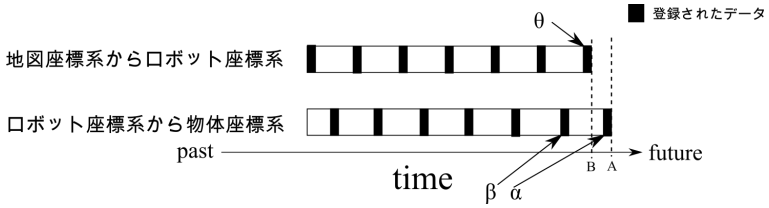

図 1.2: 位置関係の登録のタイムライン

換に関する情報が分散していることが多いことが課題となっていた。

そこで、TF ライブラリは各座標系間の変換を有向森構造として一元管理し、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。まず、図 1.1を表す木構造は図 1.3で表現できる。木構造のノードが各座標系を表し、木構造のエッジは子ノードから親ノードへの変換データが存在することを表す。

各ノードは TF ではフレームと呼ばれ、ノード中の文字列は各座標系に対応するフレーム名が書かれている。図 1.3では地図座標系のフレーム名は map、ロボット座標系のフレーム名は robot、物体 1 の座標系のフレーム名は object1 となる。木構造は子ノードから親ノードへポインタが貼られ、子ノードから親ノードを辿ることができる。このため、map から object1 への座標変換を計算するには object1 から map への座標変換の計算し、その逆数を取る必要がある。

子ノードから親ノードへの位置関係は子ノードが保持する。図中では示されていないが、木のルートノードのエッジは NULL ポインタを指している。

先程説明したように、各フレーム間の座標変換情報はそれぞれ異なるタイミングで登録される。これに対処するため、TF では各フレーム間の座標変換情報を過去一定期間保存する。図 1.3において各フレーム間の座標変換情報が登録されたタイミングを表すのが図 1.4である。横軸は時間軸を表し、左側が過去、右側が最新の時刻を表す。黒色のセルはデータがその時刻にデータが登録されたことを表す。時刻 A では robot から map への座標変換の情報が得られるが、object1 から map への座標変換の情報は時刻 A には存在しない。そこで、

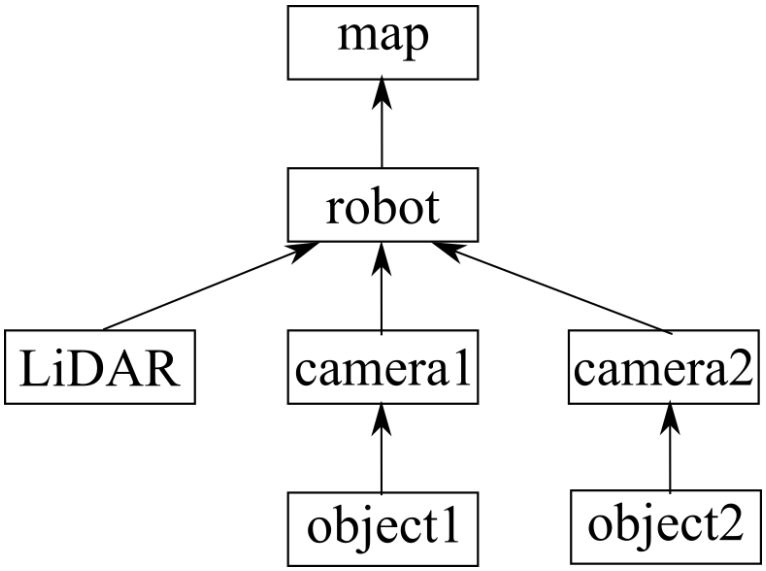

図 1.3: 図 1.1に対応する木構造

TF では前後のデータから線形補間を行うことにより該当する時刻の座標変換データを計算する。つまり、TF は該当する時刻の座標変換データが保存されている、もしくは前後の値を元に線形補間ができる場合にはその時刻の座標変換データを提供できる、とみなす。灰色の領域は線形補間により座標変換データが提供可能な時間領域を表す。

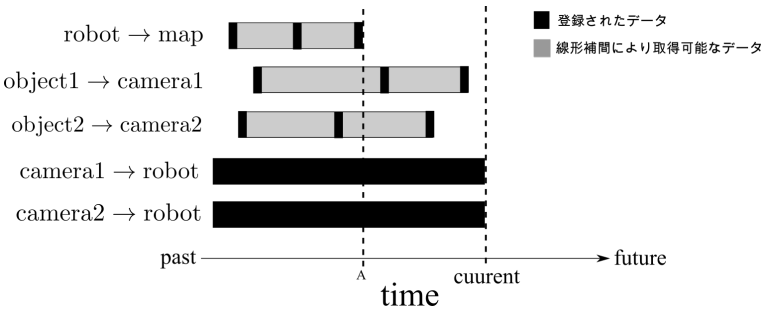

図 1.4: 図 1.3における位置関係登録のタイムライン

図 1.3における位置関係登録のタイムラインが図 1.4のようになっているとき、TF では object1 から map への最新の位置関係は次のように計算する。

まず、object1 から map へのパスを確認する。ここでは object1 から map へのパスは $object1 \to camera1$, $camera1 \to robot$, $robot \to map$ であることがわかる。

次に、どのパスにおいてもなるべく最新の座標変換を提供できる時刻を確認する。図 1.4を確認すると、$object1 \to camera1$, $camera1 \to robot$, $robot \to map$ において最新の座標変換情報が登録された時刻が最も古いのは $robot \to map$ である。このため、時刻 A がここでは要件を満たす。

最後に、時刻 A での各パスのデータを取得し、それらを掛け合わせる。$robot \to map$ については登録されたデータを使い、$object1 \to camera1$, $camera1 \to robot$ については線形補間によってデータを取得する。

1.2 研究課題

前述したようにTFはロボットシステム内部の座標系間の位置関係を一元管理する機構を提供する。しかしながら、これには以下のような問題点が挙げられる。

問題1：ジャイアント・ロック

TFの森構造には複数のスレッドがアクセスするため並行性制御が必要となるが、既存のTFでは一つのスレッドが森構造にアクセスしている際は他のスレッドは森構造にアクセスできないアルゴリズムとなっている。これは、マルチコアが常識となっている現代では大きな問題となる。

問題2：データの一貫性

上記の説明のように、TFのフレーム間の座標変換計算インターフェースは最新のデータを使わない可能性がある。そこで、最新のデータを取得できるインターフェースを追加する。

1.3 研究方針

前述した問題1については、データベースの並行性制御技術における細粒度ロッキング法を適用して解決する。細粒度ロッキング法は、並行性制御においてロックするデータの単位をなるべく小さくし、並行性を向上させる手法である。問題2については、データベースのトランザクション技術における2PLを適用した新たなインターフェースを提供することにより解決する。2PLとは、複数のデータに対するロック・アンロックのタイミングを二つのフェーズに分けることにより並行性を向上させつつデータ操作の一貫性を確保する手法である。

1.4 貢献

本研究ではデータベースのトランザクション技術における再粒度ロッキング法、及び並行性制アルゴリズムの一種である2PLを応用することにより、問題1および問題2を解決した。

1.5 構成

本論文の構成は次の通りである。第二章では関連研究について述べる。第三章では既存のTFの森構造とその問題点について述べる。第四章では提案手法である森構造への再粒度ロックの導入とデータ一貫性のためのインターフェイスの提供について述べる。第五章では提案手法の評価結果を述べる。第六章では本研究の結論を述べる。第七章では今後の課題について述べる。

第2章　関連研究

データベース分野におけるロボットの研究の例として GAIA platform[5] が挙げられる。

TF のようにデータを時系列的に管理するものとして SSM が挙げられる。

データベースの技術をロボットに適用するという内容では GAIA[5] が挙げられる。ロボットというより自律システム

C++に宣言的にデータ変更時のルールを記述できる。これによって簡単にイベントベース trigger 付きの UDF みたいな？

これは RDB ベースで reactive な挙動を提供する。

プロダクトレベルのものを目指す ROS2[6] や Autoware[7] でも、liveliness などの指標 DDS が導入されたが、データベースの並行性制御の導入はない。

本研究のようなアプローチは存在しない。

2PL 以外の並行性制御アルゴリズムとしては Silo[8] が挙げられる。

第3章　既存のTF森の構造とその問題点

3.1　構造

TF ライブラリでは図 3.1 のように各座標系間の位置関係を森構造で管理し、複数の木の登録が登録できる。ノードが各座標系を表し、エッジは子ノードから親ノードへの座標変換データが存在することを表す。各ノードはフレームと呼ばれ、ノード内の文字列は各座標系に対応するフレーム名が書かれている。

各フレーム間の座標変換情報は過去 10 秒間保存される。このため、各フレーム間の座標変換情報が登録された時刻を図 3.2のようなタイムラインで表現できる。黒のセルは登録されたデータを表し、灰色のセルは線形補間により座標変換データが取得可能な時刻を表す。横軸が時間軸を表し、左側が過去、右側が最新の時刻を表す。

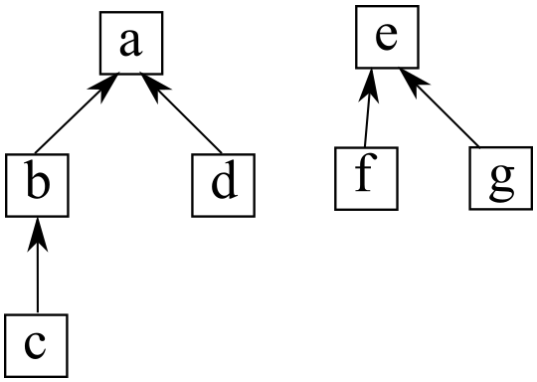

図 3.1: 複数の木構造

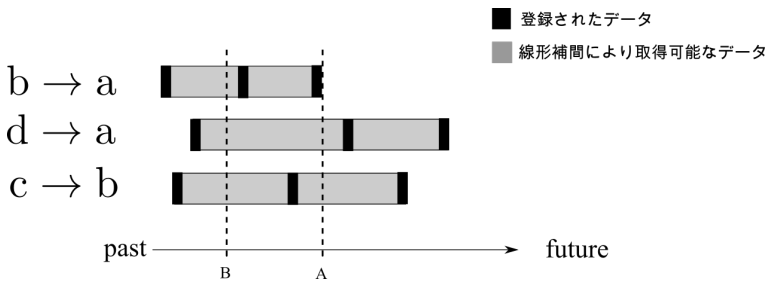

図 3.2: タイムライン

3.2　lookupTransform

二つのフレーム間の座標変換情報を取得するには lookupTransform メソッドを使う。

登録されたフレームが図 3.1、登録された座標変換情報のタイムラインが図 3.2の状況において、lookupTransform メソッドを用いてフレーム c からフレーム d への座標変換を計算するアルゴリズムを説明する。

1. フレーム c から木構造のルートノードへのパスを取得する。ここではルートノードは a となり、フレーム c からフレーム a へのパスは c→b と b→a となる。
2. フレーム d から木構造のルートノードへのパスを取得する。同じようにルートノードは a となり、フレーム d からフレーム a へのパスは d→a となる。
3. 得られた三つのどのパスにおいてもなるべく最新の座標変換を提供できる時刻を確認する。ここでは時刻 A が要件を満たす。
4. 時刻 A における各パスの座標変換データを計算する。b→a については登録されたデータを利用でき、c→b と d→a については線形補間されたデータを利用できる。
5. 最後に、フレーム c からフレーム a への座標変換と、フレーム a から d への座標変換を掛け合わせる。フレーム c からフレーム a への座標変換は c→b と b→a の

最後に、フレーム c からフレーム a への座標変換と、フレーム a から d への座標変換を掛け合わせる。フレーム c からフレーム a への座標変換は c→b と b→a の
例外パターンの説明
また、lookupTransform は指定した時刻のデータを取得することもできる。
擬似コード

Algorithm 1 lookupTransform

```
e = walkToTopParent
if n < 0 then
  X ← 1/x
  N ← −n
end if
```

まずは running example を示す。なるべく C++レベルでの説明は避けるべきか。とりあえず、途中で親が変わらないと仮定する。じゃないと説明が面倒
1. source から root への座標変換を計算 2. target から root への座標変換を計算 3. source から root への座標変換 * root から target への座標変換を計算
まずは普通のケースから。同じ親を持ち、まずはこの時間で見ていくまずは d から r への座標変換を計算次に e から r への座標変換を計算 r から e への座標変換は、単に e から r への逆変換をとれば良い。あとは d から r への変換 * r から e への変換で計算できる
このように target が直接の
ここら辺は説明省いても良い ????
この例のように、source と target が同じ木の中にない場合、つまり同じ root を共有しない場合にはエラーとなる。
running example が十分であれば、擬似アルゴリズムでの説明を省けるかも。

Algorithm 2 walkToTopParent(time, source_id, target_id)

```
frame_id = source_id
top_parent_id = frame_id
// source frame から root へのパスをたどる
while frame_id ≠ 0 do
  cache = getFrame(frame_id)
  if cache = NULL then
    // 木構造の root に到達
    top_parent_id = frame_id
    break
  end if
  paret_id = cache から座標変換と親の id を取得
  if frame_id == target_id then
    // target frame は source frame の祖先なので早期リターン
    累積したデータから座標変換を計算
    return 0
  end if
  座標変換を蓄積
  top_parent_id = frame_id
  frame_id = parent_id
end while
// target_id から root へのパスをたどる
frame_id = target_id
while frame_id ≠ top_parent_id do
  cache = getFrame(frame_id)
  if cache = NULL then
    // 木構造の root に到達
    break
  end if
  parent_id = cache から座標変換と親の id を取得
  if frame_id == source_id then
    // source frame は target frame の祖先なので早期リターン
    累積したデータから座標変換を計算
    return 0
  end if
  座標変換を蓄積
  frame_id = parent_id
end while
if frame_id ≠ top_parent_id then
  // source frame と target frame は同じ木構造に属していない
  return エラーコード
end if
// source frame と target frame は祖先関係にはないが、同じ木に属している
source frame から target frame への座標変換の計算
return 0
```

3.3 setTransform

座標変換を登録する。特に説明の必要はない

3.4 問題点

ここより上の部分では、どの程度エラーケースを、どの程度正確な説明が必要か不明瞭。とりあえず以下を書き下す。

giant lock

最新のデータを返さない example を示そうこの場合には、b に引っ張られて a の最新のデータが取れない

線形補間により、一貫性がないケースがある。例えば関節間の制約やジンバルロック、特異点など制御プログラムが意図しない状態を見てしまうかもしれない

第4章　提案手法

まず、ginat lock をしない方法を導入読み書きを行う際には必要なノードのみ lock する
setTransform とは異なり、lookupTransform は各ノードの情報の読み込みのみ行う。

読み込み処理であれば複数のスレッドからの同時アクセスを行っても data race は発生しない。このため、

S lock は data race が発生しない read の時に、X lock は二つ以上のスレッドから write 操作されると data race が発生するのを防ぐために使われる。

(S-X はもう古い？read と write の方が説明が適切？)

さらに、shared lock と exclusive lock を導入するこれは次のような表で表現できる表は列が既にかかっているロック (N は何もかかっていない)、行はかけようとしているロックを表し、o であれば新たにかけようとしているロックの確保に成功し、x であればロックがかけられないことを表す例えば、S lock が既にかけられていても新たに S lock を他のスレッドが書けることが可能になる。対し、X lock が既にかかっている場合には S lock はかけられず、S lock がかかっている場合にも X lock はかけられない。X lock をかけることができるスレッドは常に一つだけである。

続いて、最新のデータを取らない問題と線形補間によりデータの一貫性がなくなる問題について

既存の lookupTransform では、最新のデータを取得しようとしても過去のデータを参照してしまう、また線形補間されてしまう

最新のデータをそれぞれ取ってくるという方法もサポートする。(しかしこれだけでは十分ではない。 <- この導入はいまいち。) 冒頭で説明したように、ROS は分散アーキテクチャを採用する。TFMessage には各フレーム間の座標変換情報を複数登録できるが、TF は複数の座標変換を setTransform を複数回呼び出すことにより実現している。これにより、中間の状態を見てしまう。これは図のように説明できる。この図のように、giant lock を毎回の setTransform で取ってはいるが、その操作が終わり次の setTransforms を呼ぶまでの間は lock が外される。これにより、一貫性のない状態を見てしまう恐れがある。

そこで、我々は最新のデータを atomic に取得する lookupLatestTransform、及び複数の座標変換を一度に atomic に登録できる seTransforms を追加した。複数のデータに対する読み込み、書き込みを atomic に行うために、我々は 2PL[?] を実装した。

しかしながら 2PL には dead lock の問題がある。例えば次の例、これは木を登る方向と下る方向の両方があるからこうなる。

データを reorder できれば 2PL では deadlock は発生しない (DAG が構築できる)、が TF 木ではできそうにない。

そこで我々は deadlock prevent の方法として No-wait[Bern 1981] を採用した。これは write lock をかけようとして失敗したら最初からやり直す。

Wound wait, nonpreemptive (Concurrency Control in Distributed Database Systems PHLIP.A BERNSTEIN AND NATHAN GOODMAN P196)

transaction に priority を足すとかは？

第5章　評価

実験を行う

スレッド数を上げるとスループット伸びる joint を増やすと緩やかにスループットが落ちる iter を増やすと 1000 以降で急激にスループットが下がる。no wait による弊害か？lock の確保に失敗しているのかも read ratio は高くなるほどブロックされる可能性が減る read len は安定しているように見える。なぜ trn で read len が小さいとこうなるのか。。。2PL でロックする箇所が変わるから？

write len を上げるとやはりブロックされる部分が増える

第6章　結論

　再粒度ロックの導入により、パフォーマンスを上げることができた特に、スレッド数の増加とともにスループットが下がる問題を解決できた
　データの一貫性は必須、というデータも示したい。

第7章　今後の課題

ここでは取り上げなかった TF 木の問題点として、部分的に座標情報の登録に失敗するケース。rollback などもいれ transactional に行う必要

また、insert/delete が大量に発生するケースも考える。

謝辞

本研究を進めるにあたり、慶應義塾大学准教授川島英之先生とサイボウズラボ株式会社星野喬様に頂きました優れた御指導により、私の研究はとても有意義で満ち足りたものとなりました。また、慶應義塾大学川島研究会秘書藤川綾様には幾多の手続きを丁寧にサポートして頂き，円滑な出張や書類作成，研究環境整備を行うことができました。そして、CCBench開発者である株式会社ノーチラス・テクノロジーズ田辺敬之様。この研究に関わっていただいたすべての方に深く感謝を申し上げます。

参考文献

[1] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, “Ros: an open-source robot operating system,” in ICRA Workshop on Open Source Software, 2009.

[2] T. Foote, "tf: The transform library," 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA), 2013, pp. 1-6, doi: 10.1109/TePRA.2013.6556373.

[3] Philip A. Bernstein and Nathan Goodman, "Concurrency Control in Distributed Database Systems" in ACM Computing Surveys, 1981, pp. 185-221

[4] "BufferCore.h", `https://github.com/ros/geometry2/blob/noetic-devel/tf2/include/tf2/buffer_core.h`

[5] "GAIA platform", `https://www.gaiaplatform.io`

[6] "ROS2", `https://docs.ros.org/en/rolling/`

[7] "Autoware", `https://tier4.jp/en/autoware/`

[8] Stephen Tu, Wenting Zheng, Eddie Kohler †, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In SOSP, pages 18–32. ACM, 2013