

# TF の高性能化とデータ一貫性の確保

萩原湧志, 川島英之

December 20, 2021

**概要:** 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。Robot Operating System (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クング法、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。

## 1 序論

### 1.1 背景

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使われている

グローバル座標系の中でのロボットの位置、ロボットのローカル座標系の中でのセンサーの位置、センサーのローカル座標系の中での物体の位置、という風に別々に管理している

この時、グローバル座標系での物体の位置を計算するにはグローバル座標系からロボットの位置の計算 \* センサーの位置 \* 物体の位置を計算すれば計算できる

TF ではこのような座標系同士の変換を次の図のように有向森構造で管理できる。座標系の原点を frame と呼び、このフレームを node、座標系同士の位置関係を並行移動成分・回転成分で edge で表した木構造で管理する木構造で管理することにより、先程のようなグローバル座標系での物体 A の位置は、二つの frame 間のパスを辿ることで計算できるロボットの座標変換の情報は全てこの TF モジュールで管理される

TF モジュールは ROS の多くのパッケージで使われているが、これには以下のような問題点がある

Giant lock により、木構造が大きくなり多くのスレッドがアクセスするケースでは遅くなる

(setTransform の non atomicity は実はまだ問題にはならない)

提供されているインターフェイスでは「最新」のデータを取るというメソッドでも実際には該当するパスのデータが完全に準備された状態の時間のデータを返している。

また暗黙的に線形補完が行われるため、データの一貫性がなくなる可能性がある

### 1.2 貢献

そこで、データベースのトランザクション機構における並行性制御のプロトコルの一種である 2PL を導入した。

その結果、224 コアのマシンではこの程度の性能差が出た

また、データ一貫性を保証できるような setTransforms, lookupLatestTransform というインターフェースを導入した。

### 1.3 論文構成

本論文の構成は次の通りである。2 章では既存の TF 森の構造とその問題点について述べる。3 章では提案手法である TF 森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。4 章では提案手法の評価結果を述べる。5 章では本研究の結論を述べる。6 章では今後の課題について述べる。

## 2 関連研究

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。

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、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。

## 3 既存の TF 森の構造とその問題点

TF 森の実態は tf2 パッケージ中にある BufferCore クラス [4] である。

### 3.1 構造

まずは TF 森とタイムテーブルから？

各座標系同士の回転移動、並行移動で表現できる位置関係は TF 森で表現される

例えば図のようなマップ座標系、ロボット座標系、カメラ座標系、物体の座標系は図のような木構造に対応する。

また、図のフレーム A, B のように他の木とは分離された木が存在してもよい。このため、TF 森構造となる。

さて、座標変換の情報は変わらない場合と時刻とともに変わる場合がある

さて、座標変換の情報は主にセンサーからデータが送られてくるたびに計算され、TF 森に登録される。このため各フレーム間の座標変換の情報はセンサーの更新頻度に依存し、それぞれ異なるタイミングで更新される。例えば、マップ座標系からロボット座標系の座標変換を計算するプログラムは LiDAR のデータ更新頻度に合わせて自己位置を計算し、マップ座標系からロボット座標系の座標変換を登録する。カメラ座標系から物体の座標系の座標変換を計算するプログラムはビデオカメラのデータ更新頻度に合わせて物体の位置を計算し、カメラ座標系から物体の座標系の座標変換を登録する。カメラの更新頻度と LiDAR の更新頻度が異なる場合、マップ座標系からロボット座標系の座標変換が登録されるタイミング、カメラ座標系から物体の座標系の座標変換が登録されるタイミングにズレが生じる。マップ座標系から物体の座標系を計算する際にデータ同期の問題が生じてしまう。これに対処するため、まず TF 森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。これは図のように表現できる

図は各フレーム間の座標変換情報が提供される時刻を表す。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。

灰色線は対応する時刻の座標変換情報が登録されたことを表す。図の点線の位置での map から apple への座標変換を計算する際、各フレーム間の座標変換情報が必要になる。robot から camera への座標変換情報は常にある物として扱われるが、map から robot、camera から apple の座標変換データは該当する時刻では登録されていない。そのため、TF は前後のデータから線形補間を行うことにより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。

まずは CompactFrameID 等の typedef は説明が面倒なので、展開してしまうか。

座標変換は TransformStorage で表現され、これは以下で構成される回転成分を表す rotation\_。これは Quaternion で表現される並行移動成分を表す translation\_。これは Vector3 で表現される座標変換の時刻を表す stamp。これは ros::Time 型で表現される座標変換の親フレームの frame id を表す frame\_id。これは整数型で表現される座標変換の子フレームの frame id を表す child\_frame\_id。これは整数型で表現される

TimeCacheInterface は特定のフレームから親フレームへの座標変換を登録・管理する抽象クラスである。主に以下のようなメソッドがある bool getData(ros::Time time, TransformStorage & data\_out, std::string \* error\_str = 0)

特定の時刻 time における TransformStorage を取得する。取得できた場合には true を返し、取得できなかった場

合には false を返しエラーの内容を error\_str に代入する。

```
bool insertData(const TransformStorage& new_data)
```

座標変換の情報を登録する。情報の登録に成功した場合には true を、そうでない場合には false を返す。

```
std::pair<ros::Time, CompactFrameID> getLatestTimeAndParent()
```

最新の座標変換情報のうちその時刻と、親フレームの id を返す。

TimeCache は TimeCacheInterface を継承し、あるフレームにおける親フレームへの座標変換を時系列的に管理する。これは TransformStorage の双方向キューで構成され、ユーザーが座標変換情報を登録する際には先頭に push し、保存していた座標変換の時刻が 10 秒以上過去のものとなれば deque から追い出す。getData のアルゴリズムは以下ようになる

StaticCache は TimeCacheInterface を継承し、先程の例の robot-*i*sensor 間の座標変換のように不変な座標変換情報を管理する。

BufferCore は TF 森を管理する。

文字列である frame は TF 森内部では整数型の id で管理している

TimeCacheInterface を継承する TimeCache と StaticCache はそれぞれあるフレームからみた親フレームの id、親フレームの座標変換を保持するため、これらは TF 森における子フレームから親フレームへのエッジ及び子フレームのノードを表す。このため TF 森は子フレームから親フレームへエッジが伸びる有効森構造となっている。

以下のようなインターフェイスがある。bool setTransform(const geometry\_msgs::TransformStamped& transform, const std::string & authority, bool is\_static = false)

座標変換情報を登録する。authority には座標変換情報を登録するプログラムの名前等を指定できる。is\_static を true にすると、StaticCache にデータを登録できる。下記の更新、挿入において詳細を説明する

```
geometry_msgs::TransformStamped lookupTransform(const std::string& target_frame, const std::string& source_frame, const ros::Time& time) const;
```

時刻 time における target\_frame から source\_frame への座標変換を返す。下記の検索の節において詳細を説明する

```
template<typename F> int walkToTopParent(F& f, ros::Time time, CompactFrameID target_id, CompactFrameID source_id, std::string* error_string) const;
```

指定の時刻 time について target\_id から source\_id への座標変換を計算する。返り値は 0 以外の場合にはエラーコードとなり、エラーの原因が error\_string に代入される。

テンプレートパラメータの F にはよく TransformAccum クラスが与えられ、これは通過したパスの座標変換情報を蓄積するオブジェクトである。

主に lookupTransform の中で用いられる。

```
int getLatestCommonTime(CompactFrameID target_frame, CompactFrameID source_frame, ros::Time& time, std::string* error_string) const;
```

target\_frame から source\_frame までのパスにおいて、座標変換のデータが提供できる時刻で最新のものを取得する。そのような時刻が存在する場合には time に代入され 0 を返す。もしそのような時刻がない場合にはエラーコードを返し、エラーの内容を error\_string に代入する。

これは図のように説明できる。

図において map から robot へのパスにおいて必要なのは map から robot への座標変換、robot から sensor への座標変換、sensor から apple への座標変換である。この中で最新のデータが最も古い時刻に来ているのは map から robot への座標変換である。この時刻よりも前ならば map から apple への座標変換データは提供できるが、この時刻より後ではパス中の座標変換が提供できなくなる。このため、getLatestCommonTime ではこの時刻 A が選択される。

ここで、図中では lidar から object への座標変換が最も古い時刻に来ているが、map から robot への座標変換を計算する時にこの座標変換は使わないために時刻 D が選ばれないことに注意する。

(robot -i lidar -i とか別のもっと遅いものも追加！)

```
TimeCacheInterfacePtr BufferCore::getFrame(CompactFrameID
frame_id)
```

与えられた frame\_id に対応する TimeCahceInterface へのポインタを取得する。該当するものがない場合には null を返す。

また、BufferCore には以下のようなフィールドがある  
frames\_: 各フレーム id に対応する TimeCacheInterface のポインタを管理する配列。TF 森はこの配列で表現される。

frame から frame id を検索するテーブル、及び frame id から frame を検索するテーブルが存在する frameIDs\_, frameIDs.reverse

frame\_mutex\_: frames\_, frameIDs\_, frameIDs.reverse は複数のスレッドから操作される。データ競合を防ぐため、各スレッドは frame\_mutex\_ を用いて排他処理を行う。

特定のフレームに対応する TimeCacheInterface を取得するには、まず frameIDs\_ から対応する frame id を取得し、続いて frames\_ から対応する TimeCacheInterface へのポインタを取得する。この二回の検索処理の後、TF 森のフレームに該当する TimeCacheInterface にアクセスできる。

TransformAccum クラスは、lookupTransform にて木構造を辿って座標変換を計算する際に座標変換の計算を蓄積していくクラスである。

```
int gather(TimeCacheInterface* cache, Time time,
string * error_string)
```

cache から time の時の座標変換を取得し、それを TransformAccum インスタンス内の一時変数に記録する。cache から親のフレームの id を取得しそれを返す。

## 3.2 lookupTransform

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えていくまずは 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 が同じ木の中にある場合、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
  parent_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

## 4 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 がかけられている場合にも 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ー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。i- この s つ名の導入はいまいち。) 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 setTransforms 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 PHILIP.A BERNSTEIN AND NATHAN GOODMAN P196)  
transaction に priority を足すとかは？

## 5 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

## 6 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。

## 7 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。

## References

- [1] 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.
- [2] 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.
- [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](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