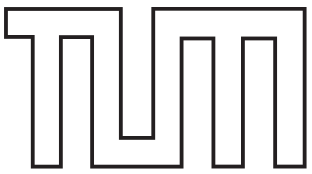

Technische Universität München
Chair of Media Technology
Prof. Dr.-Ing. Eckehard Steinbach

Bachelor Thesis

Decentralized Tree Structured Transformation Graphs

Author:	Leon Nepomuk Dorscht
Matriculation Number:	03724790
Address:	Am Rudolf-Widmann-Bogen 10 82319 Starnberg
Advisor TUM:	Eckehard Steinbach, Prof. Dr.-Ing.
Advisor DLR:	Marco Sewtz, M. Sc.
Begin:	01.Sep.2022
End:	13.Nov.2022

With my signature below, I assert that the work in this thesis has been composed by myself independently and no source materials or aids other than those mentioned in the thesis have been used.

München, November 13, 2022
Place, Date


Signature

München, November 13, 2022
Place, Date


Signature

Abstract

The world of robots consists of numerous coordinate systems representing the robots joint state, sensor positions, object poses and other properties. Those reference frames may be transformed from one to another. Mostly known is the hand-eye calibration, a transformation from the visual sensor to the tool center point (TCP) of the manipulation actuator. On complex systems, this involves a chain of smaller transformations to obtain the final one. Modern solutions use the *Robot Operating System (ROS)* module *tf* to construct a tree containing all transformations. In general, the world is modeled in parent-child relations where each frame may depend on zero or one frame and may have an arbitrary number of child frames. This is used in a broad variety of applications and platforms. However, transforms are expected to be exact and support for uncertainty in *tf* is limited. Therefore, an alternative is under development to jointly estimate poses and the corresponding confidence matrix. This calculation is based on the Lie Algebra which can transform all frames into a tangent-space and concatenate uncertain transforms. Until now this system depends on a central processing node and all clients communicate with it. This generates a communication overhead and induces delays in the communication. This work evaluates a decentralized architecture to overcome the aforementioned drawbacks. Furthermore, an experimental analysis shall show the change in performance utilizing the proposed implementation.

Contents

Chapter 1

Introduction

1.1 Motivation

Cambridge dictionary defines a robot as "a machine controlled by a computer that is used to perform jobs automatically" [Pre22]. These tasks involve modifying the objects around the robot. The robot needs manipulators or actuators to perform these tasks. These elements connect in a complex system, the robot, via various joints. It is necessary to describe the 3D position and 3D rotation, combined in a 6D pose vector, of elements in the world and the robot in different coordinate frames. The camera, for example, has its own frame, which detects an object in the distance, which is its z-axis. The actuator has a different coordinate system, with another fixed z-axis describing the height above the floor. Using multiple frames makes it more efficient to calculate the suitable modifications to the actuator. However, it also needs the coordinates of the detected object in the actuator's frame rather than the camera's. To calculate the coordinates in the actuator frame, we need to apply a transformation to the coordinates calculated in the camera's frame. This transformation of coordinates from one frame to another creates a higher computational task since the joints can change their rotation or translation and a transformation between the poses has to be calculated new every time the robot changes those joint parameters. The *tf*-dude, which is short for transformation-frame-dude, solves this task. This program uses a tree-structured graph as a data structure to save the transformations between coordinate frames. The program can calculate the transformation from one frame to another by combining all the transformations over connecting joints between the two coordinate frames.
Since all real-world applications, mainly mechanical ones, have uncertainty, it is necessary to include an error. Figure 1.1 shows an example of why this is necessary. The robot's actuator position changes are already noticeable when an error of one degree is applied. The x- and y- coordinates of the new position vary by an amount of 4.3696cm in x-direction as well as -10.5711cm in y-direction.

This functionality of including an uncertainty is not included in the so far implemented

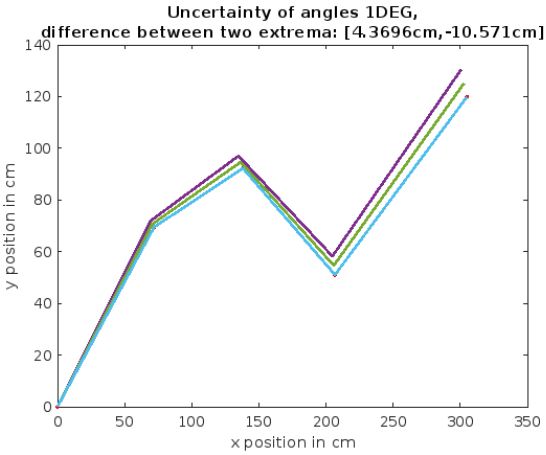

Figure 1.1: Example of a robotic arm with three hinge joints. The green arm is without any uncertainties; the other two are with an applied error of ± 1 degree

tf-dude. To get a transformation with uncertainties, the new implementation uses Lie algebra and saves data about the uncertainty of the transformation between coordinate frames.
Another new feature that this program should implement is the use of decentralized structures. Since the robot constantly needs transformations for computational tasks, a high rate of updates and computation of transformations is necessary. So far, we could use the *ROS tf module* or the *tf*-dude, but we expect a speed improvement when using the decentralization of the processes. This decentralization of the data structure could boost the performance since this might reduce the communication between processes, which takes time, and higher amounts of data due to the relatively large data packages sent over the inter-process communication (IPC). Take, for instance, three times a 6D Pose; each dimension is encoded as a float meaning 4 Bytes. In total, that makes 72 Bytes. Transformations with uncertainties will take up even more space, which the thesis will come to later. To reduce the amount of sent data, the decentralized *tf*-dude has subsystems owning subtrees, which, for instance, only manage the arm of the robot. As long as the robot only needs to calculate the transformation inside this sub-tree or update an element inside it, the process does not have to send data to other sub-trees.

1.2 State of the Art

In 1955, Jacques Denavit and Richard S. Hartenberg formulated the Denavit-Hartenberg convention to standardize how coordinate frames are defined. These conventions made it easier to solve robotic chain kinematics in joint systems. This convention defined rules to determine the axes of a joint, where every joint in the system has one coordinate frame for each movable direction or rotation. The z-axis is here along the rotation or translation

axis of the joint, the x-axis is the cross product of the last z-axis, and the y-axis is chosen in a way, that the frame is a right-handed coordinate system [Web19]. This convention is still in use today in many robotic applications. Later, with more advanced processing systems on robotic hardware, *ROS* introduces the *tf* module, which tracks each joint state individually and constructs a path formula on-the-fly when requested. One of the most used communication and operation frameworks for robots is the open-source *Robot Operating System*, also known as *ROS* [Sta18]. It provides an implementation of most algorithms required to run a robot. These libraries also include the *tf* library, which can perform coordinate frame transformations for robot coordinate frames. This *ROS tf* library already includes transformations between frames, but in contrast, my project does not run decentralized and is, in its standard version, incapable of handling uncertainties. This principle is explained in Chapter 2. *ROS* also is capable of establishing a communication infrastructure between systems. The *German Aerospace Center* (*DLR*) already developed an inter-process communication, the *links and nodes*-library [Pre22]. This project uses this communication framework as a dependency. The *tf*-dude is by default IPC-agnostic. That means the program does not depend on the IPC implementation and might use *ROS* in the future.

1.3 Structure of the Thesis

This thesis focuses on implementing and evaluating the *tf*-dude in a decentralized tree structure. In order to do so, the first part explains the necessary technical background, structured into the three main concepts of the new program. These are coordinate transformation, Lie algebra to handle uncertainties, the underlying tree structure, and the decentralized approach. The methodology by which the program was implemented follows this chapter to introduce how this implementation of the *tf*-dude works. Afterward, it is necessary to describe the methodology of the tests implemented to test the program and explain why these specific analyses are conducted and the expected outcome. The subsequent part captures the factual outcome of the tests, and the metrics by which the results are measured get introduced. These results are then analyzed and discussed to explain how reliable and representative they are. In the end, the thesis will give a summary and an outlook for the future.

Chapter 2

Theoretical Background

As the motivation and state of the art are now explained, the next chapter is concerned with the explanation of the main concepts used in the *tf*-dude. The four main aspects, coordinate transformation, uncertainty, tree structure, and decentralized approach, are all of high importance for my thesis, and a basic understanding is therefore mandatory.

2.1 Coordinate Transformations

To understand coordinate transformations, we first have to define a coordinate frame. A coordinate frame in 3-dimensional space is defined by three vectors of unit length called $x, y, z \in R^3$. Those three vectors are perpendicular to each other [Mey06b]. This definition of the angle to each other ensures that these vectors only have three degrees of freedom. Otherwise, the whole degree of freedom of the system of vectors would be nine. Often, it is helpful to have more than one coordinate frame to describe the position of various points in a system. Figure 2.1 shows an example of those coordinate frames. The three standard vectors have a 90-degree angle to each other. It is also visible that the blue frame has a rotation and translation to the origin of the green frame.

In robotics, it is often the case that every joint, sensor, actuator, object, and robot has its own coordinate system, which is positioned so that the movement of the element can be described more easily. Figure 2.2 shows an example of the improvement these diverse coordinate frames make. The point P is movable in the direction of the x'-axis and can be rotated around the y'-axis. The prime coordinate system is fixed to the rotational axis and turns with the plate. This enables us to describe the position of P in the reference frame $\mathcal{A}$ by only changing the x'-coordinate of point P since it can only move in this direction. To describe the point P in world coordinates, the frame $\mathcal{W}$, we need to transform it. Coordinate transformations can solve this problem.

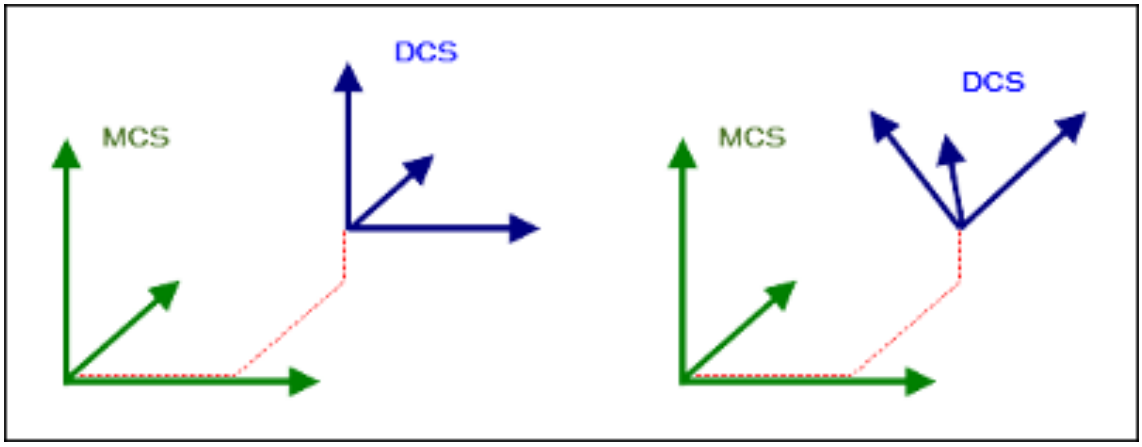

Figure 2.1: Coordinate frame in a fixed coordinate frame [Ele21]

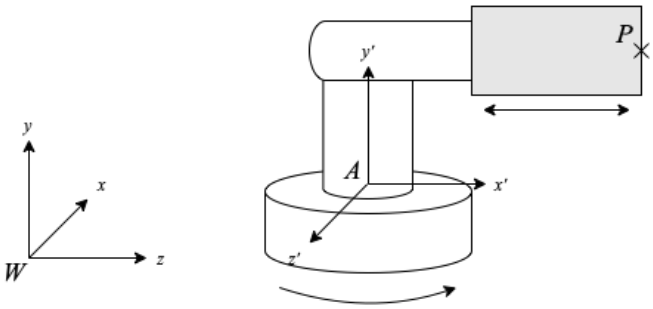

Figure 2.2: Use of coordinate frames. The left side shows the world coordinate frame, and the right is a frame bound to the joint that can rotate. The grey block can move in the direction of the x'-axis.

Rotations

The rotation of a coordinate frame can be described by using a rotation matrix. This rotation is always given by an angle around one of the axis, with their positive direction given by the right-hand rule. The definition of rotation matrices are

$$R_x(\gamma) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix} \tag{2.1}$$

$$R_y(\beta) = \begin{bmatrix} \cos(\beta) & 0 & \sin(\beta) \\ 0 & 1 & 0 \\ -\sin(\beta) & 0 & \cos(\beta) \end{bmatrix} \tag{2.2}$$

$$R_z(\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{2.3}$$

where the index stands for the axis around which coordinate systems rotate. We call the three different rotation angles yaw, pitch, and roll. Yaw stands for α, the rotation around the z-axis, pitch for β, which describes the rotation around the y- and roll for γ, around the x-axis. The rotations can be performed by using the matrices. For example, we have the position of a point $Q_{\mathcal{B}}$ in a system rotated by Angle γ around axis x, $\mathcal{B}$. Now want to transform it into world coordinates $\mathcal{W}$, we need the rotation matrix $R_x(\gamma)$. If we would now multiply the matrix to the left side of the vector, $Q_{\mathcal{W}} = R_x(\gamma)Q_{\mathcal{B}}$.

Translation

However, what if we do not need a coordinate frame that is not a result of rotating it around the world frame but also shifted in space? To describe this, we can use a translation vector. An interpretation of the vector is the following. Imagine the two points P and **P'**, given in Figure 2.3. The position of those points can be described by the vector $\boldsymbol{p}$ and $\boldsymbol{p'}$. The result of formula $\boldsymbol{g} = \boldsymbol{p'} - \boldsymbol{p}$, vector $\boldsymbol{g}$ is called translation vector. It can also be used for Q, in the way, that $\boldsymbol{q'} = \boldsymbol{q} + \boldsymbol{g}$, since PP' is parallel to the segment QQ'[Mey06b].

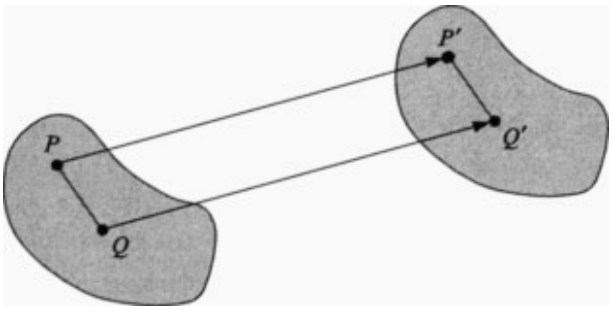

Figure 2.3: Translation of an object [Mey06a].

Homogeneous Coordinates and Transformations

In the last part, we need to look at homogeneous coordinates to understand the 4x4 transformation matrix that the *tf*-dude uses. The use of those matrices is to make it easier to transform coordinate frames. To do rotations as well as translations, we need a way to incorporate the translation into the rotation matrix. This combination of the two is achieved by introducing a "1" into the vectors and expanding the matrices from 3x3 to 4x4.

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{2.4}$$

$$\boldsymbol{p'} = \boldsymbol{T}\boldsymbol{p} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ 0 & 1 \end{bmatrix} \boldsymbol{p} \tag{2.5}$$

Here the translation is included by the translation vector $\boldsymbol{t}$, and the rotation is the rotation matrix $\boldsymbol{R}$. If we do the matrix multiplication, we can see that x' includes the addition of t_1. Equally, we do this addition for the other components. This way, we found an efficient way to transform vectors using matrices. The *tf*-dude saves those matrices to perform matrix multiplication.

2.2 Uncertainty

The matrix described in Section 2.1 can describe the coordinate frame transformation in an ideal world, but this is not the case. Most real systems have imperfections, leading to problems if we idealize them. The *tf*-dude is implemented using Lie algebra to account for real-world imperfections. For this, a frame saves a covariance matrix in addition to the 4x4 homogeneous transformation matrix. This covariance matrix stores the covariance and variance between the various translation and rotation elements, necessary metrics to handle uncertainty.

$$\begin{bmatrix} Var(x) & Cov(x,y) & Cov(x,z) & Cov(x,\alpha) & Cov(x,\beta) & Cov(x,\gamma) \\ Cov(y,x) & Var(y) & Cov(y,z) & Cov(y,\alpha) & Cov(y,\beta) & Cov(y,\gamma) \\ Cov(z,x) & Cov(z,y) & Var(z) & Cov(z,\alpha) & Cov(z,\beta) & Cov(z,\gamma) \\ Cov(\alpha,x) & Cov(\alpha,y) & Cov(\alpha,z) & Var(\alpha) & Cov(\alpha,\beta) & Cov(\alpha,\gamma) \\ Cov(\beta,x) & Cov(\beta,y) & Cov(\beta,z) & Cov(\beta,\alpha) & Var(\beta) & Cov(\beta,\gamma) \\ Cov(\gamma,x) & Cov(\gamma,y) & Cov(\gamma,z) & Cov(\gamma,\alpha) & Cov(\gamma,\beta) & Var(\gamma) \end{bmatrix} \tag{2.6}$$

This matrix is used together with Lie Algebra to compute the transformation between nodes. The in-depth description of the calculation process is out of the scope of this thesis. For more information, refer to [Ead13].
The *frame* data structure stores the described matrices, each as an Eigen [GJ+10] matrix with double precision. Eigen is a C++ library for linear algebra, a dependency for this project's code.

2.3 Tree Structure

The data is encapsulated in a *frame* object as described in Sections 2.1 and 2.2. Nevertheless, this data is unstructured, and if we need more than one transformation, we need a means of data structure to organize and store multiple *Frame* objects, which makes the relation between consecutively executed transformations clear. This organization and clarification are done by using a tree structure in the *tf*-dude. Figure 2.4 shows an example of

this structure. The elements are called nodes, and each one of the nodes has an individual identifier. Every node in the tree except the highest one, called the root, has a predecessor. A node can have none or multiple child nodes. The nodes without children are the leaves of the tree; connections between the nodes are called weighted edges. The weight of those connections stores the transformation data from the parent to the child node.

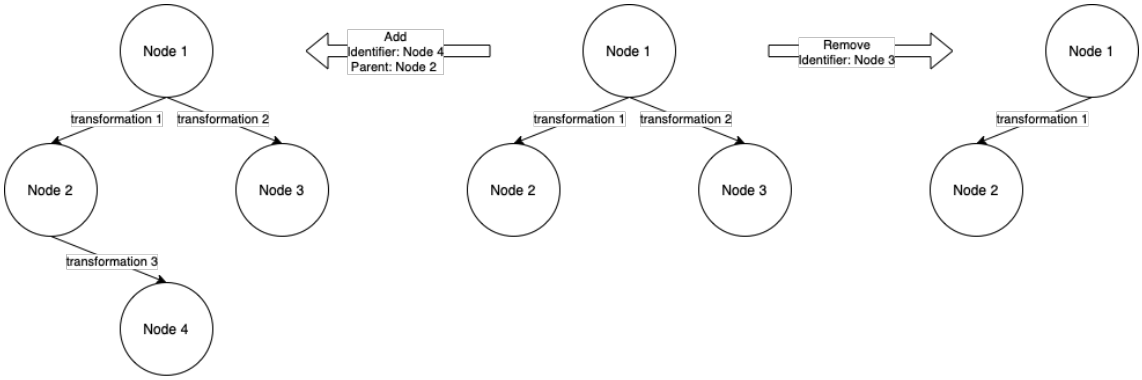

Figure 2.4: Tree structure with the two major algorithms, add and remove. Add applied on the tree results in the left; remove applied on it results in the right tree.

This structure has an implementation of the CRUD operations, which stands for create, read, update and delete. These basic operations first create the tree by adding new nodes with transformations to the tree. Then it is possible to either read the data by giving back the transformation from one element to another or update a path with a new transformation. In the end, the tree can be deleted entirely, or only a single element can be removed. The read operation can give back the transformation along a path that not only includes one connection but can also have an element in between two nodes, for example, from *Node 4* to *Node 1* in Figure 2.4. Those basic operations and the tree structure implementation are already part of the *boost graph library* [SLL00], used in the manager implemented before.

2.4 Decentralized Approach

Since it is now established what data is stored and in which way, we want to focus on decentralization. The so far used and implemented code uses a server to save and alter data. The clients can only send requests to the server and receive a response from the server to do the in Section 2.3 mentioned functions to modify or read the tree. The decentralization is useful if the program needs to be optimized on the execution speed since it takes time to communicate with the server, and the server has to read or modify a relatively big tree. The implemented version uses decentralization, where the server is not needed but instead uses clients communicating with each other. Figure 2.5 shows the difference. On the left-hand side, the server manages the whole tree, and the clients send

requests. The right-hand side shows the scenario using clients and decentralization. Every client owns a part of the divided tree from the before-used server. In the graphic, the upper process owns the grey part, and the lower the white. This decentralization should increase the speed at which the process works since it is possible to modify the smaller trees at a higher speed.

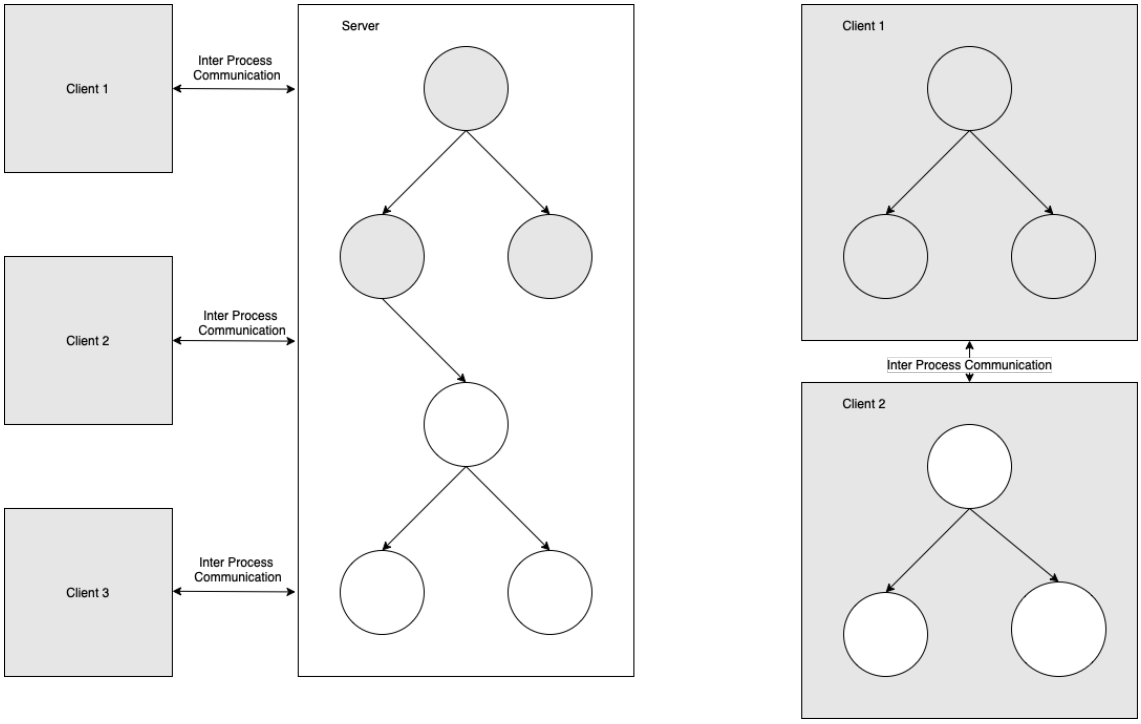

Figure 2.5: The left side shows the centralized system with ownership of the tree structure completely by the server. The right side shows the decentralized approach.

Chapter 3

Methodology

Now that we have gained the first insight into the technical necessities for this project, it is time to understand the function of the code and the way it was implemented. This chapter describes the function, also in corner cases.

This project is an implementation to manage the trees inside a process and the data necessary to manage the tree structure in a decentralized way. The tree structure was implemented and is used in the project without altering it. The project also includes the *links and nodes* [Sch22], an inter-process communication interface that can send data over the network.

The first function I programmed is the add function. It works in the way that the node the tree has to add is given to the tree with the highest possible root. This primary process is the white process seen in Figure 3.1. This graph also shows that an element that should be added as a child of a node in possession of a graph other than the main graph (grey process) has to be given from the first to the latter process. Inter-process communication handles this exchange of data between processes. This means that if a process cannot add the element to the tree it owns, it sends it to the following process, and then this other process tries to add it to its sub-tree. To later test whether an element is in the right place, the get parent function is implemented as an extra debug function. With this, I can check if the node is added and in the correct position inside the tree. An additional test of the particular case that the parent node does not exist is implemented to check the code's behavior. The expected result is that the program does not change the tree.

To add a different process as a child to a process, the add_IPC function is needed. This function adds inter-process communication into a list of child processes and the name of the following root. The last information stored in this list is the transformation frame between the processes.

The remove function has a more complex algorithm than the add function since removing the children of the removed element is necessary. Those nodes are not required anymore

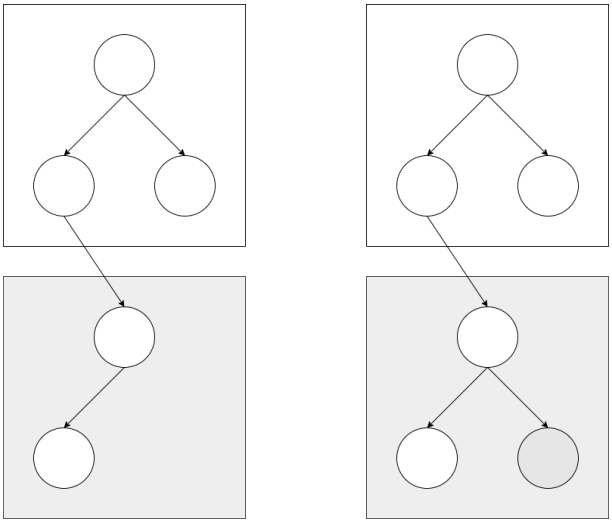

Figure 3.1: Add function on a tree in two different processes. Left: initial state. Right: after calling add function, the grey element got added.

since they do not have a connection to the main tree anymore. Figure 3.2 shows the first case, where the program deletes a normal node. An already implemented function in the *tf*-library handles this remove functionality. This part of the library handles only the tree structure inside a process. Since this only uses before-implemented code, it should work as expected.

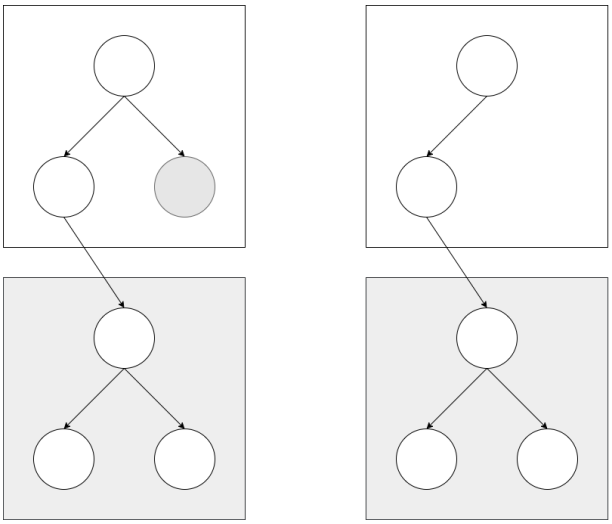

Figure 3.2: Remove function on a tree in two different processes. Left: initial state, the grey element should be removed. Right: remove got executed.

The more complicated case, where the grey process has to be deleted, is shown in Figure 3.3. We check if the parent of the next tree's root still exists in the tree, a standard

operation implemented in the dependency. By that, we can delete the whole tree of the next process if the parent does not exist in the white process anymore. If this parent does not exist, the white process tells the grey process to delete the whole tree and deletes the link in the white process.

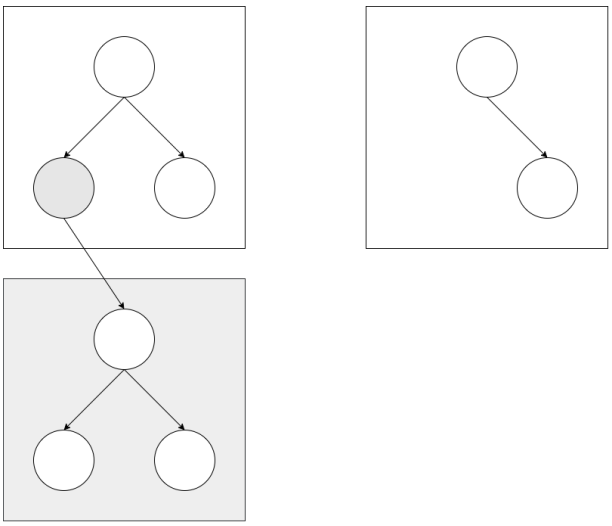

Figure 3.3: Special case of remove function on a tree in two different processes. Left: initial state, the grey element should be removed. Right: the grey element and all its children got removed.

The third algorithm's function is to change the frames between two nodes. This algorithm works similarly to add; it checks if the element is in the tree of the current process, and if not, it calls the update_frame function in the subsequent linked processes. Since this update should be done as fast as possible while the robot is moving, the update function checks first in the process in which it was called if it can perform the update. If this is not the case, it sends a call to the main tree, which starts to check from the top process to the last ones.

The get_path algorithm is necessary to get the path from one node to another. This function is depicted in Figure 3.4. The first picture on the left side shows the path the program should evaluate between which nodes. Node 1 is the start, and node 4 is the end node. The algorithm works in more than one step if the path is over more than one process. In this case, two processes are involved. In the first step, which can be seen in the middle, the algorithm determines the path from the start node (node 1) to the root of the own process (node 2). This result is transferred to the parent process, white, and then inserted into the connection between the connecting node (node 3) and the dummy element that stands for the root of the grey process (dummy node 2). The frame of the dummy node 2 gets updated with the path from node 1 to node 3. Now the white process can evaluate the path from node 1 to node 4 by calculating the path with the standard get_path function implemented in the tree. This calculation is done by calling the before

implemented function of the tree manager with start node dummy node 2 and end node, node 4.

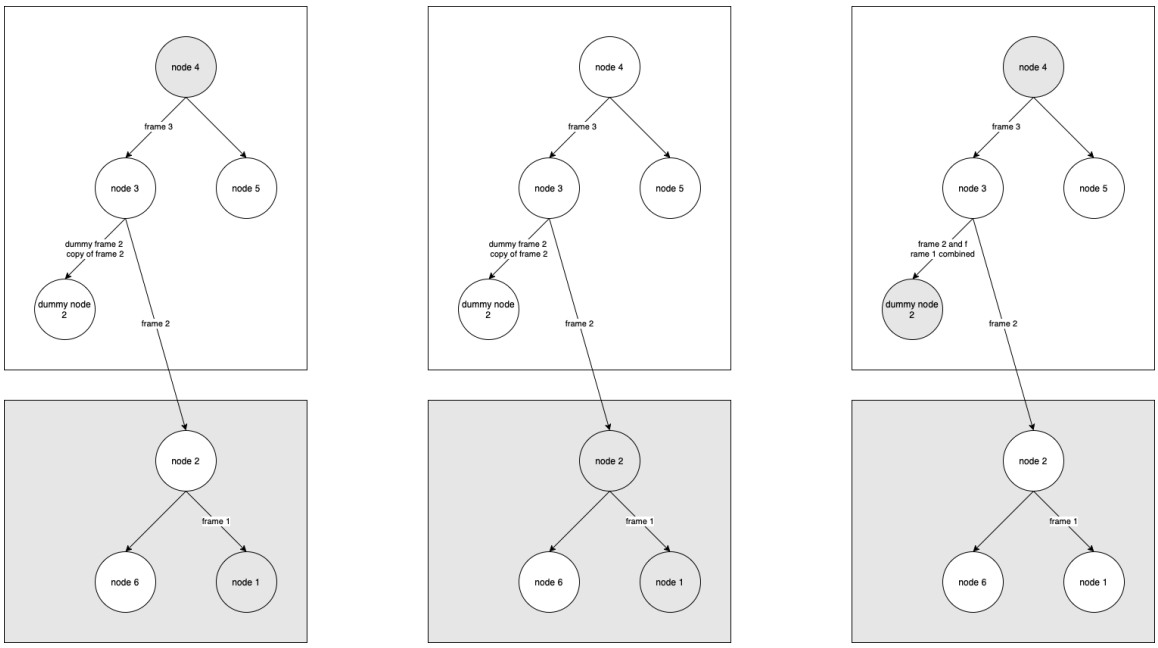

Figure 3.4: Get_path function for two processes with start node 1 in one process and end node 4 in another. Left: start and end of the searched frame. Middle: Start and end in the grey process. Right: the last step, start and end in the white process with an updated frame.

Chapter 4

Evaluation

After the explanation of the way the code works, we need to conduct tests on those functions. The first Section of this Chapter will give a short overview of the methods used to test the program.

4.1 Methodology of Tests

This thesis investigates the feasibility of using a decentralized structured program to manage a tree-structured graph. This goal makes it necessary to evaluate whether this is possible and if it is more efficient than currently implemented structures with the same functionality. These functions include storing the data needed to perform transformations between coordinate frames and calculating these transformations. After the implementation, I collected data from the finished code to analyze the project using qualitative and quantitative analysis methods. To compare the collected data, I also conducted the same tests on an old implementation of the *tf*-dude, implemented using a centralized server-client structure. This comparison makes it possible to determine the success of this project in several categories, including the program's overall function, the gain in computational speed, and the potential decrease of sent data. Chapter 4 and Section 5.1 further handle the conducted tests and representation of the collected data. The data can be divided into the two categories mentioned before. The qualitative data was collected using testing functions which return an error message if the tested functionality wasn't executed correctly or just behaved differently from what I expected. This testing was done in the code in C++ by implementing unit tests.

Similarly, the quantitative data was collected by executing functions multiple times. The two quantitative tests consist of the space and time complexity tests. The first of those two, the space, was tested by a theoretical approach, analyzing the package size of data sent and how many of those are transmitted via the inter-process communication during

various scenarios for each of the functions. The time complexity test, also determined by evaluating a quantitative variable, can be tested by writing a test code that executes the same method multiple times and then create a box plot of the resulting data set. This way of collecting data over a more extended period makes it easier to check the code's time efficiency more stable since the averaging of data can neutralize the influence of fluctuations induced by programs running in the background. Those time data sets are then compared to another data set created on the old system that was implemented before. This old version implemented the same functionality as my program but in a centralized way. Since both data sets are collected by running the code on the same machine, they are comparable.

My used data collection and analysis methods are advantageous since the tests for functionality cover a wide field of different scenarios testing the code and also in corner cases, which makes predictions for the code easier. The main functions are comparable to the old version since they work nearly the same way. The space complexity tests are valid in the way that they are collected over a multitude of runs. The same holds for the time complexity tests. Since I included the fluctuation of result data, I can identify how much the data varies and the median value of time the code needs to compute the new tree or output frame.

4.2 Backgrounds of Test Scenarios

A division of the described test into three subgroups is possible, and this Section lists and explains these. To understand why I conducted a specific test, I will explain the use-case of those tests.

4.2.1 Function of Code

The function of the code is the most basic test. If the code does not behave in the desired way without knowledge of the different behavior, it is impossible to conduct the subsequent tests. To do so, I test the implemented main CRUD operation, which stands for create, read, update and delete. The create function, called add, is used to create a new node in the tree structure and add the corresponding transformation. The read or path function can determine the path between a starting node and an end node and return the transformation between those two nodes. The update function can change the transformation between two adjacent nodes. And lastly, the delete or remove function can be used to delete nodes and the children of that nodes.
It is vital to test whether the functions work in a decentralized way or if any problems occur when the trees in different processes do actions that affect the operation executed in another process.

4.2.2 Space Complexity

If the function is correct, other essential factors can define the program. The first of those measurements is space complexity. The space complexity is not of great importance on the systems at *DLR* right now but may be more critical on more miniature mobile robots in the future. I only focus on the amount of data that has to get sent over the network. The data can be stored on other drives outside the robot, making more data space available for the other programs running on the robot's systems. If the robot is mobile, it may have to send the data over a connection with smaller bandwidth. This slower connection can lead to a bottleneck since this can lead to a time delay. That is the reason why the data amount sent has to get reduced.

4.2.3 Time Complexity

Another vital metric to describe the efficiency of code is the time complexity or time to calculate the result of a function. The computational time figure is crucial since a shorter computation time directly affects how many functions can be evaluated in a defined time frame. Additionally, a shorter time to change or read something in the tree can reduce the number of wrong outputs made by the code. The code can make such errors since it has a race condition, in which the output of the code depends on the timed function of other program sequences. For instance, in the case that a joint is moving, the process updates a frame in more than one step over a while. During one of the steps, the data is locked for access by a Mutex. If simultaneously, while the program finished one step of the update of a node, a request for a transformation accesses the data, it can give back an old state because the process still needs to be finished with all updates of the joint. Even worse, it could return invalid data as parts of the data structure are updating, and parts are still old. In the case of the *tf*-dude, the updating speed has a required limit, which was set to 1 kHz, which means that the program should update the frames in under one-thousandth of a second.

4.3 Expected Outcome

As I described the background of why I conduct a specific test, it is also essential to give an overview of the expected outcomes. This foresight is essential since it provides a base to compare the actual output to the expected result and makes it possible to discuss differences in the actual outcome of the experiments. In the Section "function of code" 4.3.1, I will also describe which output is expected and why using a description of how the code works.

4.3.1 Function of Code

The functions of how the code works were evaluated in Chapter 3. The tests conducted on these functions are the main functionality by calling the function in an intended way and corner cases. Since my implementation already includes such corner cases, undefined behavior is not expected to occur. The add function should only add elements where the parent exists in the tree. Otherwise, it should just not do anything. The remove function should remove elements as well as their children, even if they include deleting a subtree in another process. The update function's output is an update of frames inside a tree and connectors between processes, saved inside the special list managing connections. The last function, the get_path function should return the right path. Since the manager for the tree is already implemented, I expect that the results of functions that only perform a change to the trees and do not need unique managing by the decentralized process structure will perform their task in an intended way.

4.3.2 Space Complexity

Since the decentralized processes need to communicate with each other to work correctly, I assume that the data transferred via the internet is higher than on the centralized approach, which requires one call to the server per function. But the amount could also be decreased due to functions that calling processes can apply directly to their tree. Therefore these processes do not need to send it to another process since the function is already executed.

4.3.3 Time Complexity

The last conducted experiment is concerned with the speed at which the functions work. Since this is the most important reason for decentralization, I require that this program is faster in its main functions compared to the non-decentralized version of the *tf*-dude. However, this depends on the function's functionality, which is visible by looking at how the code works. To eliminate redundancy in the check, if an operation is possible in the tree of the process in which the function is initially called, the process always sends out the command to start in the main tree. How the program works makes the code slightly slower since the command always has to be sent except if it is called in the main tree. An exception is the update function. This function is especially time critical to achieving a higher rate at which the robot can update during a change in position. The update function checks if it is applied on a node owned by the process in which it is called. If not, it sends a command to the main tree. This could lead to repeated try to update the element in the tree owned by the process that called the function initially. This should make the processes slower compared to the *tf*-dude that has a centralized approach. The tree depicted in Figure 4.1

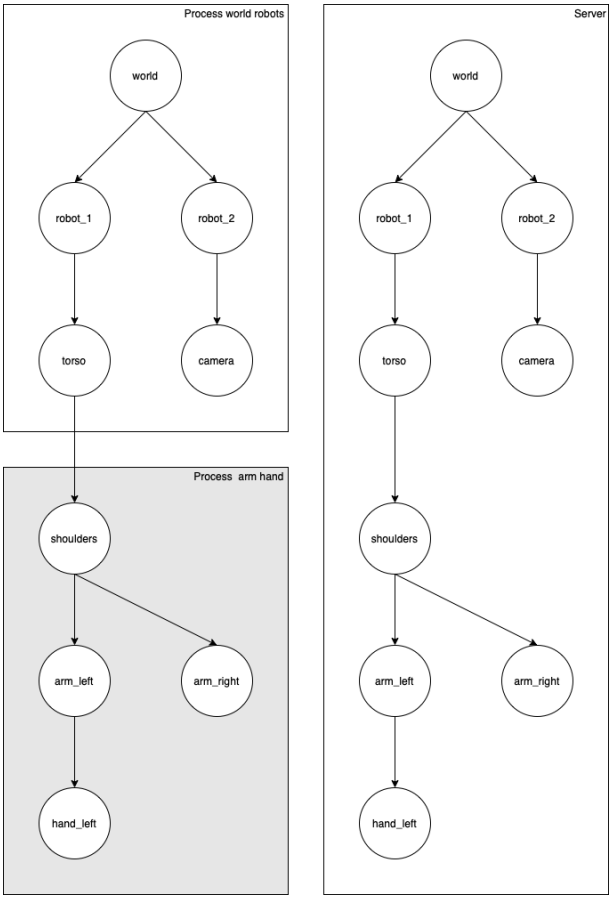

Figure 4.1: Tree used for the additional time tests on the left in the decentralized version, on the right in a server-client structure.

Chapter 5

Experimental Evaluation

The necessity of the different tests and what the results are expected to be was explained before. After the definition of the tests, I conducted them on the code in order to retrieve data. This data is outlined in this chapter and analyzed.

5.1 Evaluation Metrics

To measure the outcome of the tests, we first need to define the metrics by which the tests are conducted: This is necessary because it is only possible to determine the correct outcome with a descriptive way to express the results.

5.1.1 Function of Code

A quantitative variable can not describe the function of code. To determine the success, we compare the functions of the new code to the already existing implementation of the *tf*-dude with a centralized approach. The outcome is compared by directly accessing the tree structure in the processes and using standard methods implemented beforehand to ensure the right results. The tests are executed on the example tree, shown in Figure 4.1.

5.1.2 Space Complexity

Opposite to the function of the code, the space complexity is measurable with a numeric value. The space complexity was determined by looking at the definition of the messages sent by the process and then analyzing the number of requests necessary to do a specific functionality in the example tree, Figure 4.1. An upscaling of the results to more than two

trees is possible since the bigger one would take more time since it has more connections, which results in more sent data.

5.1.3 Time Complexity

The time complexity can is measurable by the use of the metric time. To do so, I conducted a series of variations of the add command, in which nodes are added in different scenarios. Box plots show the data in representative ways. These also contain the same process done in the standard version of the *tf*-dude, that does not utilize a decentralized structure, as a comparison. The data for the box plots were collected over three different testing periods. Each testing period consists of $100,000$ executions of the same functionality, where the execution was timed. These tests resulted in a total sample size of $300,000$ results in the unit microseconds. Conducting tests multiple times eliminates the error that results from other programs running in the background. Another important note for these tests is that I remove elements after they get added and vice versa. In the tests for removal, I added them again. Adding and removing the same element ensures that the same functionality is timed and not influenced by a bigger graph size from the tests before.

5.2 Evaluation Results

These defined metrics are now used to create outputs for the different tests. I will now describe the specific outputs and explain the meaning and possible reasons for the outputs.

5.2.1 Function of Code

The function tests created an output text file that shows the results. I will discuss the different experiments and their output in this subsection.

Add

The add functionality is the first function that the tests addressed. The add function should work exactly like the centralized program, and to test this, I added the elements in the test tree, Figure 4.1, except the camera, the hand_left, and the arm_right node. After this, the elements got inserted by calling the function add in the main process for the hand_left and the camera node. This test ensures that the function works when called in the main process and can add to the same process as the other client. The function worked as expected, and there was little room for unexpected behavior. Additionally, the handed-over frames were inserted correctly. I also tried inserting a node with a parent with

no element in the tree, so it cannot get added. The processes ignored the add function, and no element without a parent, a root, was created, which was how the functions should work.

Remove

The opposite of the add function, remove, was tested next. The tests for the remove function are similar to the ones for the add function, resulting in the right nodes getting removed. In addition, I remove the nodes robot_2 and arm_left node to check if their children also get removed, a functionality adapted from the centralized *tf*-dude. This should also work if the sub-tree of the removed element is in another process. The tree of the other process should be removed, too. Testing this case is possible by calling remove robot_1, and it removes the whole tree in the process with ownership of the arms. The occurring problem is that the process is not stopped and remains working without any usage since a node can only be added to it when creating the Client_Decentralized object.

Get Path

The path calculation is critical since it involves more logic than the remove and add function if the path reaches from one process into another. A variation of paths from different start and end nodes was called in the main process, and the other process was executed to test the code of the get_path function. To compare it, I used the same frames between nodes in the centralized tree and evaluated the same paths. The results are as expected; the processes calculate the right frame objects and inverse frames if we reverse the path start and end. A difference to the standard version is the introduction of a "no element found" response code in addition to those that only describe the start node and end node missing in order to enable the possibility of using more than two clients and a path ranging over three processes. This new response also makes it possible to find paths with non-existing nodes in the tree; this returns the corresponding response message.

Update

The last functionality is the update function. The test for this function was to update a node. The function always checks first in the calling process, so I also tried to update an element in the hand process called in the same process. The update function can update elements in the same process, in which the function call occurred, and also in a different process, as well as the connectors, which is important since the connector's frames are not saved in the normal tree structure but in an additional vector managing the connections.

5.2.2 Space Complexity

As described above, the space complexity describes the Bytes sent over the network while processing a called function. Table 5.1 presents the function and the corresponding data size of the request and response of the function.

Table 5.1: Size of sent packages per function.[1]

Function	Request data size	Response data size
Add	364 Byte	1 Byte
Remove	10 Byte	1 Byte
Get_path	20 Byte	345 Byte
Update	354 Byte	1 Byte

The centralized graph always uses one call to the tree, which means that it has the size in the table. On the other hand, the decentralized version needs more than one transfer per function call under bad circumstances. The difference is in the update function and when the call occurs in the main process. Those functions check the tree owned by the process first and therefore need no call of the function in a different process over the network. The number of processes is called n. Since the data structure works in a recursive-like way and returns an error message, if the element cannot be found in the lowest process, the highest amount of calls over the inter-process communication is equal to n. We define a lowest process as a process not having any processes in the list of child processes. The decentralization of the program leads to a range of 0 Bytes transferred up to $n * 365$ Bytes, which is the number of processes multiplied by the biggest transferred data. The add and the get_path functions send the biggest data package. The centralized approach needs at least one call of functions in the server over the inter-process communication, which leads to a range of 11 to 356 Bytes.

5.2.3 Time Complexity

As explained above, the time complexity was tested in different scenarios on the example tree, seen in Figure 4.1.

Add

The first of the add tests was to add the node tool as a child of the existing node camera. In the decentralized version, this function was called in the main process, which has ownership of the world and robot nodes. The resulting plots of those tests, Figure 5.1, show that the

[1] Ids are strings and therefore variable in size; since average words have five characters, I assume a naming of two words, equaling ten characters.

tree in decentralized form is much faster than the tree in the standard *tf*-dude. This result is the same as expected and is a consequence of the smaller tree size in the decentralized processes. The difference is significant since the standard version takes approximately three times as long as the decentralized program.

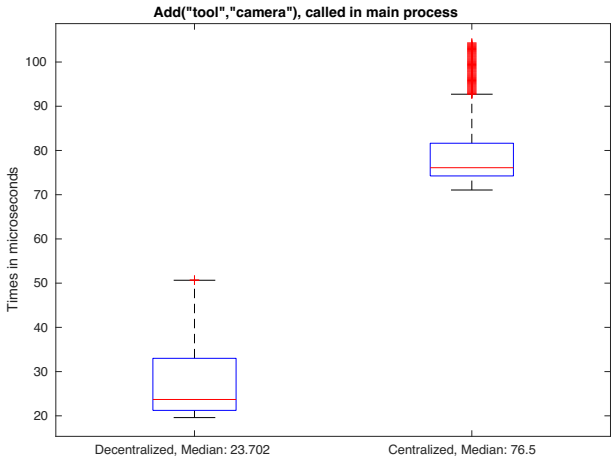

Figure 5.1: Box plot of time data received from adding an element in the same process as the calling one.

In contrast to the first test stands the second add test, in which the element that got added was in the other process. Here the main process calls an add function, which adds an element to the other client's tree. Take, for example, a call for adding the hand to the arm_2 node in the world robot process. This call is slower than the standard version since it has to send the element over the network. The box plot is shown in Figure 5.2. The add function also takes around six times longer if it has to send the command to another process than if it can do it directly in the calling process, whereas it takes about the same amount of time in the centralized tree. It can be inferred that the biggest reason for longer computation time is the necessity of sending data over the network, not the insertion into the tree structure. This also means that the process takes even longer if the tree consists of more than two processes and has to be sent more than once.

Remove

The remove functionality works similarly to the add function. This similarity already explains the outcome of the experiments, which are similar to the add function tests and the prediction in the last chapter. The remove function operates faster than the add function, even though the difference is marginal, resulting from it having to remove the linkage between elements and not copy frames into the memory. As seen in Figure 5.3 and Figure 5.4, the time for sending the request to another process outweighs the time saved by the smaller tree. As mentioned above, this could be reduced by checking in the same process first and not sending it to the main process, with the downside of possible

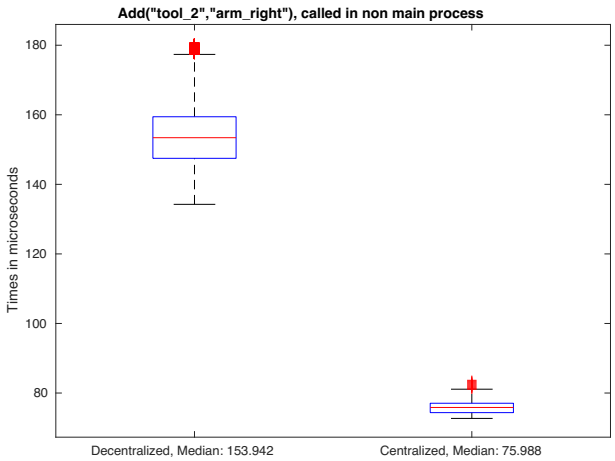

Figure 5.2: Box plot of time data received from adding an element in another process than the calling one.

repetition of checks in some processes. The remove functions were applied with the same nodes and in the same processes as the add tests.

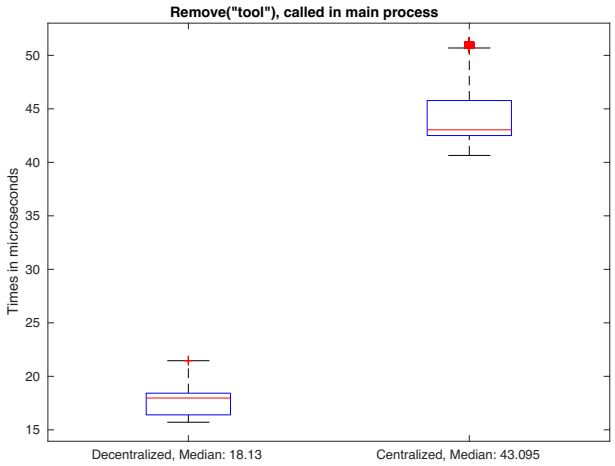

Figure 5.3: Box plot of time data received from removing an element in the same process as the calling one.

Get Path

After the remove function, the next tests evaluated the get_path function, which works differently than the two functions above. This function is more complex than the before-tested functions and sometimes needs data from other processes. It also has to call the update function on a tree if it has to connect two nodes in different processes. This was tested by letting the program find a path from the world to the hand_left node. The result

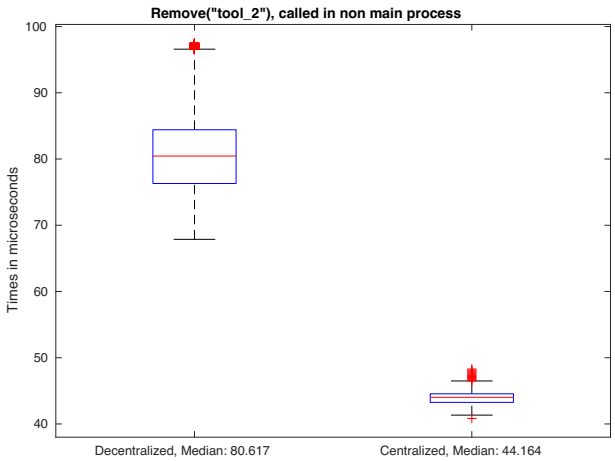

Figure 5.4: Box plot of time data received from removing an element in another process than the calling one.

of this test is the box plot in Figure 5.5. The decentralized tree takes about double the time that the centralized tree needs for this task. Since it has to send the data from the process containing the world to the other process, receive the result and modify the path, and calculate the path in the main process, which is called the function, this took a relatively small amount of time.

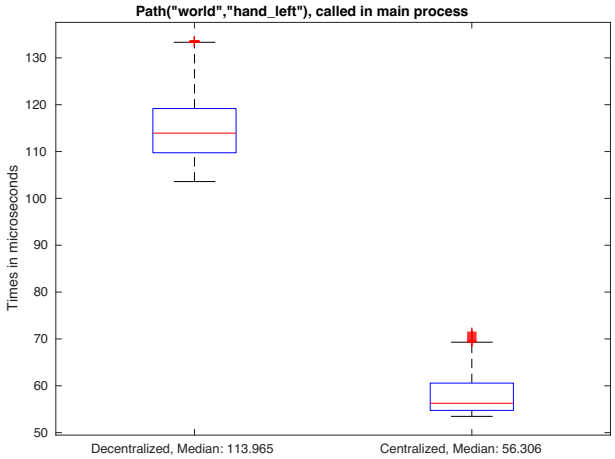

Figure 5.5: Box plot of time data received from getting the path from an element in a different process than the one it was called in to one in the same process as the call.

In the next test, we apply the search function for a path between two nodes in the tree that are stored in the same process, called in a different process. For example, the get_path function is called in the arm process, searching for a path from the world to robot_1. The data plot, Figure 5.7, shows the advantage of smaller trees over the time needed to send data over the network. The path length that the process has to search is the same, so the time the centralized version needs more is only due to the different tree sizes. The

decrease in time for the decentralized tree of 15us compared to the centralized standard version is subtle, but the example tree is only a fraction of the size of trees in a real robotic system. This increase in size would make the advantage even more significant in a practical application. The same effect can be seen if the get_path function searches for a path between elements in the same process and calls in the main process since this does require sending data over the network. In Figure 5.7, the path was evaluated between the world and robot_nodes, but this time the function was called in the main process. The time difference, which is nearly 80us, is significant. This result again shows the advantage of checking the own tree before sending the data over the network.

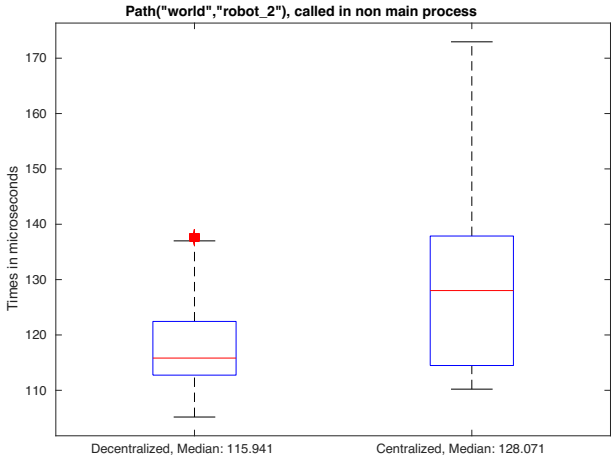

Figure 5.6: Box plot of time data received from getting the path between elements in the same process, called in the arm process.

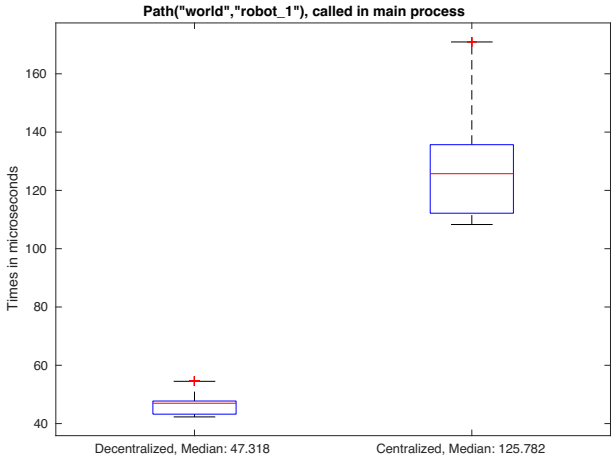

Figure 5.7: Box plot of time data received from getting the path between elements in the same process, called in the main process.

Update

The most time-critical process is the update function since this function should change the transformations between nodes as close to the speed at which the robot moves as possible. This is also why in this function, the processes check first if the element is in their own process and ignore the possibility of repeated search after sending it. The scenario in which the element is not in the same process in which the function was called has a resulting time depicted in Figure 5.8. Compared to the case in the process owning the element had the initial function call, Figure 5.9, it is about seven times slower due to the communication between the processes over the network. It is, however, not much slower than the centralized version, which is only 12us slower.

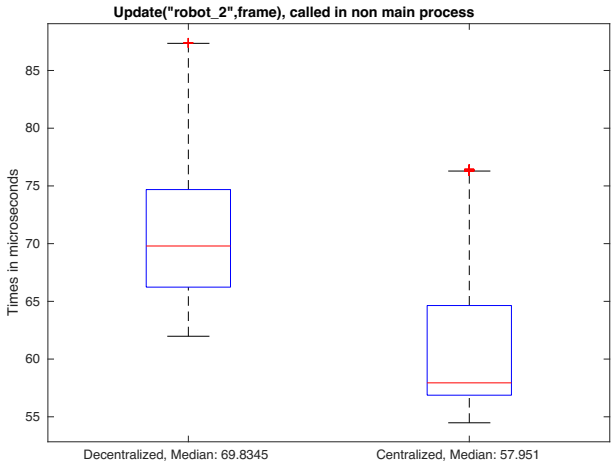

Figure 5.8: Box plot of time data received from updating the frame of an element in a different process than the one it was called in, called in the main process.

The result of an update function on the own process does not depend on if it is the main process or the other processes if executed directly, and the process does not have to send an element to another process. This is an effect of the before-mentioned different ways the update function works. Evidence for this is the comparison between the medians of Figures 5.9 and 5.10. Those are nearly equal even though they are called in processes main and the client process because they can update directly in the tree the processes own.

An even better result can be achieved by updating the connector between processes, which is the frame between the leaf of one process and the root of the next one. This improvement is the result of the data structure that manages these connections. It is a vector with a smaller length, which makes it faster to check the elements in it for the frame that has to get updated.

The update function generally achieves the requested update time of 1 kHz. It even exceeds it in the example, where the program executed the update in another process with an update rate of about 14 kHz. This rate, however, is only representative in the example

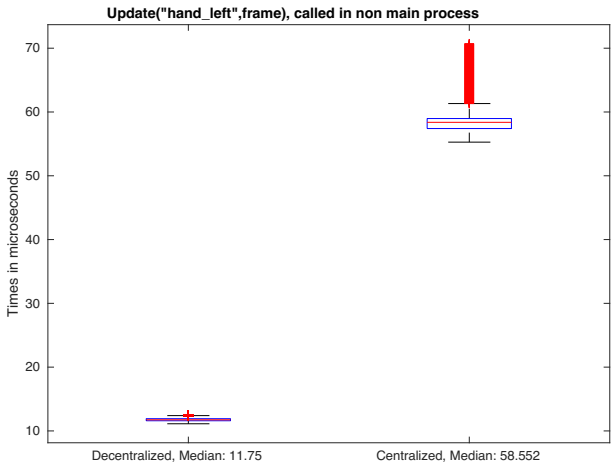

Figure 5.9: Box plot of time data received from updating the frame of an element in the same process it was called in, the arm process.

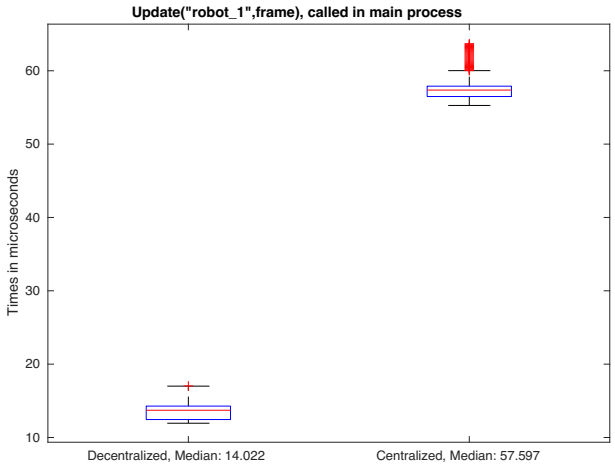

Figure 5.10: Box plot of time data received from updating the frame of an element in the same process it was called in, the main process.

tree, with only two clients in a bigger structure. This rate depends on the number of nodes within processes and the total number of processes involved in the processing of the update function.

In conclusion, the time efficiency of the function in the decentralized tree structure decreased compared to the centralized version if the function call could be processed in the same process that received the initial call. This method is a possible way to improve the program. On the other hand, it may lead to a higher computation time if the element is not inside the tree owned by the initially called process. This is the consequence of the program repeatedly trying to perform a function on the tree owned by a process. For example, if more than two programs are connected consecutively and the middle process

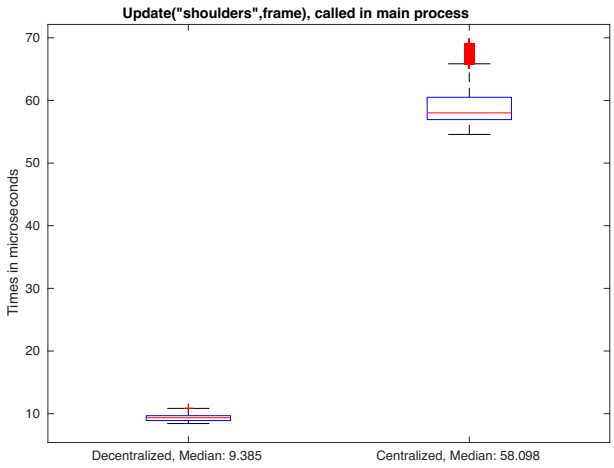

Figure 5.11: Box plot of time data received from updating a connection element.

tries to add an element to itself. After a failed attempt, it sends it to the first process, and it will propagate through the first process. Then the middle process tries again to add it to its own tree.

5.3 Discussion

This section critically discusses the findings presented in the first part of the chapter.

Functionality

In the functionality part, I evaluate the program's functionality in as many cases as possible. In reality, it is not possible to predict every outcome right since it is still possible that a smaller bug still needs to be discovered that did not come up in the limited trials I conducted. Another point is that the functionality was only tested on two processes, and there is an intention for an extension on more than two processes. The functions already work, but they still need to be tested to a full extent. The problem with this extension is the synchronization of the processes since they do not run in an infinite loop and these processes try to send a call to another process while it performs a task, which results in errors.

Space Complexity

The space complexity is evaluated in a theoretical rather than a practical way. The data that the program has to calculate in the processes, due to the complicated methods to

evaluate the exact data amount, was left out of the tests. This amount of data also highly fluctuates in dependency on the size of the tree. On the other hand, the data sent over the network is always the same when a process uses inter-process communication. The only partially certain part is the size of different identifiers, which I assumed to be around 10 Bytes, as explained in a footnote. This variation in identifier length can make a difference in the data size if the identifier is longer or shorter, but the change is still small compared to the size of the frame data. In addition to this, the processes may need a different amount of transfers depending on the function and where in the tree it can be performed.

Time Complexity

Lastly, the tests for the time complexity and possible errors in evaluating it are addressed. The first is the sample size. $300,000$ samples are already representative, but with a larger size, it may be possible that more outliers show up, which alters the median result. The samples for the decentralized version and the centralized program were taken at different times, allowing different programs to run in the background. If, by chance, the program that runs in the back while the centralized program is running is complex, the time for the centralized version may be higher than it would normally be. The times evaluated might be wrong in the way that they are evaluated on the same machine, which makes them a result of the processing speed of the computer on which the program was running. If the processor were stronger than the machine I used, the times would be shorter and vice versa. This dependency on the machine the code runs on makes the values just valuable for comparison to the old program and not a comparison in general, which might be tempting but not possible.

Chapter 6

Conclusion

This last chapter gives a short overview of the program this thesis is about and closes off this thesis by giving a short outlook.

6.1 Summary

After the possibility and performance of the decentralized *tf*-dude were now analyzed, I want to summarize the main aspects of this thesis.

A robotic system needs coordinate frames, which are three unit vectors $x, y, z \in R^3$, right-angled to each other. These coordinate frames are located in space, and every part of the robot, which are joints, sensors, actuators, objects, and even the robot itself, has its own coordinate system. These systems can be described by a translation and rotation from one coordinate frame to another. This description makes it possible to perform transformations on points in 3D space between coordinate frames. These multiple frames are useful since it makes it easier to describe the position of, for instance, the distance of an object to the camera since we can measure its distance as one coordinate if the frame is defined correctly. To manage the transformations between these frames, it is useful to implement an efficient way to calculate them. The *tf*-dude in a server-client structure, where the server manages the tree, implements this already. This tree saves each of the frames as nodes and can calculate the transformation from one node to another. This structure also shows the connection between the nodes. This thesis evaluates if decentralization of the process is possible and if it results in a performance boost. The implementation was done using the tree manager, which manages a tree and divides the server's tree into smaller sub-trees, each managed by its process. These processes communicate over inter-process communication. The client, in which one of the create, read, or delete functions is called, sends the command to the main process, which has ownership of the highest root. It then manages by searching through all processes for one in which it can perform the operation.

The update function first tries to update the tree owned by the process in which it was called. If this is impossible, it also sends it to the main process. The functions work in the way expected. A call of delete also deletes the children of the deleted function, if one of the children is a different process's root, it deletes the whole sub-tree, and the get_path function works over more than one process. Comparing the data sent between the two versions, we can see that the data decreases in the decentralized version. The decrease results from not needing to send data every time a function is called. In time comparison, the decentralized version is noticeably faster than the centralized version if the function can perform its task in the process in which it was called if it is the main tree. This improvement can be improved by always checking if the function can be done in the process it was called in and not sending the function call directly to the main tree. The problem with this is that the time might be longer when searching for a place to apply the function in a different process. The reason for that is that the calling processes tree gets visited twice. This principle, as mentioned before, was used in implementing the updated function and showed an increase in speed. In the future, the decentralized *tf*-dude still has room for improvement, but it was proven that decentralization decreases sent data and computation time.

6.2 Outlook

The decentralized *tf*-dude is in an experimental state and is, therefore, not ready to be used in applications. Another not yet tested feature is the expansion to more than two processes, which works in theory. Tests on implementing three or more clients are already conducted on the path function, testing the function over more than two processes. These tests already show success but still need development time to solve problems with the timing of the processes. In general, the decentralization of the processes increases time efficiency. It is even possible to increase it more if the repetitive call of functions in a process is accepted. In the future, the decentralized approach might be the best way to calculate transformation between different robotic parts and joints, but this will still take some development work.

List of Figures

List of Tables

Bibliography

[Ead13] Ethan Eade. Lie groups for 2d and 3d transformations. http://ethaneade.com/lie.pdf, 2013.

[Ele21] Schneider Electric. Shifting, rotating, and scaling the coordinate system. https://olh.schneider-electric.com/Machine\%20Expert/V2.0/en/codesys_softmotion/codesys_softmotion/modules/_sm_cnc_din66025_coordinate_shift.html, 2021.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[Mey06a] Walter Meyer. Chapter 4 - transformation geometry i: Isometries and symmetries. In Walter Meyer, editor, *Geometry and Its Applications (Second Edition)*, pages 143–197. Academic Press, Burlington, second edition edition, 2006.

[Mey06b] Walter Meyer. Chapter 6 - transformation geometry ii: Isometries and matrices. In Walter Meyer, editor, *Geometry and Its Applications (Second Edition)*, pages 257–310. Academic Press, Burlington, second edition edition, 2006.

[Pre22] Cambridge University Press. Cambridge business english dictionary. https://dictionary.cambridge.org/de/worterbuch/englisch/robot, Nov 2022.

[Sch22] Florian Schmidt. Links_and_nodes / links_and_nodes gitlab. https://gitlab.com/links_and_nodes/links_and_nodes, Nov 2022.

[SLL00] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. The boost graph library (bgl). https://www.boost.org/doc/libs/1_80_0/libs/graph/doc/index.html, 2000.

[Sta18] Stanford Artificial Intelligence Laboratory et al. Robotic operating system. https://www.ros.org, May 2018.

[Web19] Wolfgang Weber. *Industrieroboter*. Carl Hanser Verlag GmbH & Co. KG, München, 4., aktualisierte auflage edition, 2019.