VIETNAMESE – GERMAN UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

Frankfurt University of Applied Sciences

Faculty 2: Computer Science and Engineering

Supporting Goods Movement In Warehouses Using Decentralized Swarm Robotics

Full name: Nguyen Bao Hoang Chuong

Matriculation number: 17098

First supervisor: Dr. Le Lam Son

Second supervisor: Dr. Vo Bich Hien

BACHELOR THESIS

Submitted in partial fulfillment of the requirements for the degree of Bachelor Engineering in study program Computer Science, Vietnamese - German University, 2024

Binh Duong, Viet Nam

Declaration

I am Nguyen Bao Hoang Chuong. I declare that this bachelor thesis which is submitted to Vietnamese-German University and Frankfurt University of Applied Sciences, has been done after the registration for the Bachelor's degree thesis in Computer Science at Vietnamese-German University. Furthermore, I assert that all of my views, findings, conclusions and suggestions are entirely mine. The thesis has not been published or submitted to any other universities or institutions.

I certify that this work is similar in both its digital and physical forms. I am aware that a specialized piece of software will be used to verify my thesis' digital form for plagiarism.

Signature

Nguyen Bao Hoang Chuong
Student at Computer Science Department
Vietnamese – German University

Table of contents

Title: Supporting Goods Movement in Warehouses Using Decentralized Swarm Robotics

Abstract

Decentralized swarm robotics is an emerging field of research since it can support in many other fields such as searching and rescue, logistic and warehouse automation. While improvements in hardware have improved the performance of robotic systems, decentralized decision-making, security, and trust remain critical topics. Blockchain technology, a fast growing field of study since the birth of Bitcoin, offers potential advancements in these fields by providing enhanced security and consensus mechanisms. Additionally, Ethereum's smart contracts demonstrate promise in facilitating decentralized decision-making, particularly in task allocation, management, and validation. This project aims to integrate blockchain and smart contracts into a decentralized swarm of robots to support goods movement in warehouses, exploring the capability of smart contracts as a commander in a decentralized robot system.

Keywords: decentralized swarm robotics, blockchain, Ethereum smart contract, task allocation.

Acknowledgement

I am writing to express my deep gratitude for the support and care I earned from my teachers during the completion of this last chapter in my university life, which formed a memorable academic experience.

My foremost sincere appreciation goes to my supervisors, Dr. Le Lam Son and Dr. Vo Bich Hien. Dr Son suggested a marvellous area of research to me, which allowed me to work with hardware and robots for the first time in my four years at the university. During my exploration, his knowledge and experience were a lighthouse that guided me through the turning points of my research on my blockchain system. His readiness to share and discuss academic topics provided me with tons of insights into blockchain technology. Dr. Vo Bich Hien provided me not only with helpful insights into robotics and swarm robotics but also inspired me with his great enthusiasm towards academic research. Under his supervision, I was immersed in the study atmosphere and was blessed with the mindset of inheritance, sharing and contribution spirit of the academic world.

Finally, it would be a thing to regret if I ever missed out on the great and constant support from my teacher, Tran Hoang Huy, from the robotics laboratory. Without his ardour to guide me through the early struggles of dealing with hardware and his sharing of deep knowledge in working with Turtlebots, I could have had a much harder time with my thesis.

List of figures

List of Listings

List of tables

1. Introduction

Swarm robotics is a subfield of robotics that studies large groups of robots to work collaboratively towards a common goal. The field shows potential applications in areas such as search and rescue, environmental monitoring, and warehouse automation. However, ensuring the efficient coordination of a decentralized swarm system is still challenging, particularly in terms of task allocation and security. Blockchain technology is an emerging field of study known for its decentralized, tamper-proof ledgers. Integrating these new technologies offers a promising solution for a secure, transparent, and trustless environment for swarm robotics and robust data-driven global control with smart contracts.

The decentralized system offers robotics swarm scalability and resilience to failures but brings with it the struggles of global synthesis, decision-making, data consistency, trust issues between robots and security. This project aims to partly resolve the above obstacles with blockchain technology and Ethereum smart contracts. While the problem of data consistency and security is mostly resolved by blockchain technology, the issues in global synthesis, decision-making and trust between robots will be tackled in this project using smart contracts.

This project explores the application of Ethereum smart contracts as a decentralized task allocator, and manager in a swarm robotics system. By exploiting the transparency and security of blockchain, the project seeks to develop a robust solution for managing robot swarms in tasks such as warehouse goods movement. Additionally, the project investigates how smart contracts can validate the completion of tasks in a secure and tamper-proof manner, ensuring accountability and accuracy within the swarm.

The scope of this project involves simulating a swarm of Turtlebot Burger robots performing a list of goods movement tasks in a warehouse environment. The robots communicate with a private Ethereum blockchain, where a smart contract acts as a central commander in a high-level design to allocate tasks, track progress, and validate task completion. The system is developed with the Robot Operating System (ROS) for robot control, Ethereum for blockchain network, and Solidity for smart contracts. The simulation tests the accuracy, availability, efficiency, error tracking, autonomy, and security of this decentralized approach.

This project aims to contribute to the growing field of decentralized robotics by demonstrating the potential of integrating blockchain technology with swarm robotics. By decentralizing task allocation and validation using smart contracts, the system improves performance, security and trust in the autonomous system while preserving the natural strengths of a decentralized system. The findings of this project could have implications for real-world applications in logistics and warehouse automation, where secure and scalable multi-robot systems are essential.

This writing goes through these major points: chapter 2 reviews existing knowledge in the fields of swarm robotics, blockchain technology, and their integration. Chapter 3 discusses the system design and architecture of this project, while chapter 4 covers the details of the implementation of the navigator, the blockchain client, the web user interface, and the robot navigation system. Then, chapter 5 demonstrates the results of the practical simulations, and chapter 6 discusses the findings, limitations, and future directions of the project. Eventually, chapter 7 concludes the thesis by summarizing this writing.

2. Literature Review

In this project and paper, existing knowledge in robotic swarm, blockchain, Ethereum, and their overlapping research areas is used. Therefore, this part is dedicated to the existing knowledge that supports or is relevant to the project.

2.1. Swarm Robotics

2.1.1. Definition of Swarm Robotics

Robotics swarms can be viewed as groups of robots cooperating with each other to work towards mutual goals. The size of such groups needs to be equal to or larger than 3. The swarms may consist of other units such as drones, vehicles… The units in swarms are capable of operating based on pre-programmed instructions and logic. Finally, it is important that the swarms consist of units that are not directly controlled by humans (Arnold et al., 2019).

There are numerous existing systems of robotics swarms, which follow various styles and methodologies. The system designs can be separated into two fields: centralized and decentralized; each has advantages and drawbacks.

2.1.2. Centralized System

The typical system of a centralized swarm robotics consists of a central planner and multiple robots. The planner is responsible for collecting and summarizing information from robots. From that information, a plan of action is created, and the robots are dictated to work towards the goals. The first benefit of centralized systems is that the swarm and each of the robots within it act in sync based on a complete history of knowledge derived from every active robot. Therefore, the movement of the system is easy to predict, which could ease the process of programming and debugging. However, the use of a central commander suffers from having a bottleneck and an apparent single point of failure - the commander. These result in having limited scalability since the command point cannot process an excessive number of connections and is sensitive to the situation of losing, being captured or malfunctioning the commander (Barca & Sekercioglu, 2013).

2.1.3. Decentralized System

Decentralized systems are considered more popular in the field of swarm robotics, as they can resolve some natural pain points of the centralized system. The decentralized design may have robots that are able to communicate with each other in order to convey and try their best to summarize the global status. The wonderful thing about decentralization is that the system does not have a central processor. Therefore, it is free from the bottleneck and the ‘single point of failure’ problem. Besides allowing much larger swarm size, the system can naturally use all robots' parallel processing power. As an exchange for that many strengths, decentralized systems struggle to synthesize the system's overall status. This results in the system making locally maximal decisions instead of globally maximal ones. That is why decentralized systems tend to have more energy and time overheads than centralized ones. Moreover, predictions of behaviors of such systems are generally more challenging, which complicates the development process of decentralized systems (Barca & Sekercioglu, 2013).

2.2. Blockchain Technology

In 2008, Nakamoto introduced the concepts of Bitcoin and blockchain to the world. Since then, Bitcoin has become one of the most popular digital currencies, while blockchain turned into a lucrative field of academia (Pilkington, 2016). The applications of blockchain are soon discovered in other fields of study: finance, legal, social media, supply chain, games, etc…

Structure and Characteristic

Blockchain's base architecture is a list of blocks of information. The blocks are chained together in a linear manner, where one block stores the hash value of its previous block. The hash value is generated from the critical values of one block. Therefore, using the hash value as a link ensures the data in the blockchain cannot be tampered with since it requires the re-calculation of all the blocks that come after it, which consumes an excessively long time, especially if the blockchain network is using the proof-of-work mechanism. Because in this mechanism, finding an appropriate nonce to meet the network rules of hash values is a tremendous challenge (Pilkington, 2016; Zheng et al., 2017).

In blockchain networks using proof-of-work, there are many users who compete to solve the calculation of hash value and add it to the chain for rewards. This means the data is constructed and shared by different users from different places and is decentralized. That is a powerful protection against any attempt to attack the system because it requires control over more than 50% of the computational power in the network to take over the system itself. A miner could not add a block with invalid transactions because after a node finishes calculating the hash value and adding a block to the chain, the new status of the chain is spread to other nodes. Afterwards, other nodes check the validity of the new blocks. If there is a block that is not valid, it will be rejected. This consensus mechanism ensures the newly added blocks are agreed upon by most nodes in the network (Pilkington, 2016; Zheng et al., 2017).

Proof-of-stake is proposed later in order to replace the resource-intensive proof-of-work. Instead of relying on excessive energy and time to produce a block as proof of credit as the proof-of-work method, this method requires miners to stake an amount of currency on its collateral. The selection of which miners or validators will do the calculation and add a block to the chain could be based on the amount of currency those miners stake and random probability. This implies the higher the value a miner stakes, the higher the chance it is chosen to do the calculation and get rewards. However, there are chances that the lower-stake miners will do the mining so that the decentralization of the network is kept intact. The above is just one method to implement the proof-of-work mechanism. In practice, there are many different ways of implementing proof-of-stake using various criteria to prevent one miner (or validator) from being selected repeatedly. In theory,

proof-of-stake is more vulnerable to attack than proof-of-work as the time to produce a block is greatly decreased. However, due to the decentralized nature of the network, the situation of being successfully attacked remains highly impossible (Zheng et al., 2017).

2.3. Ethereum

Ethereum was created to simplify the process of building decentralized applications with blockchain by providing 'an abstract foundational layer', which consists of a blockchain and Solidity - a fully capable programming language. As claimed in the Ethereum document, the tool can facilitate fast and simple developments of smart contracts and blockchain networks. Some of the concepts of Ethereum are now to be discussed based on the Whitepaper (*Ethereum Whitepaper*, n.d.) and the Yellow paper of Ethereum (Wood, 2024).

2.3.1. Accounts

Ethereum accounts are the cornerstones of the network. Together, the accounts decide the current global state of the system. In Ethereum, accounts are identified by an address of twenty bits in size. When an account commits a transfer of information or value, the state changes.

Accounts in Ethereum can be divided into two types: external owned accounts and contract accounts. External owned accounts are processed by users or used by external applications. Each external owned account has a unique private key which is kept exclusively to the owner of the account, and can be used to sign transactions. Contract accounts are specified by their codes, which are written by humans before being deployed in the network. In a typical scenario, external owned accounts can send digital currency to each other and can send messages to the contract accounts to run pieces of code on it, which then can modify the storage of the contract accounts and may lead to chained actions affecting other external owned accounts (Wood, 2024).

Contracts or smart contracts in Ethereum are different from their name and do not have the characteristics of a real-world contract. They do not include a list of rules for other accounts to follow. It is, however, considered a program residing inside the blockchain network. The instructions in that program could not be altered after deployment. The program would execute a matching piece of instructions when it is invoked correctly by other accounts (Wood, 2024).

2.3.2 Transactions

A transaction in Ethereum can be defined as data that follows a pre-designed format. It includes essential information such as the recipient (address), the signature of the sender, the amount of ether to be transferred, an optional data field, a “gasLimit” - the upper limit for the number of steps allowed for the resolving of the transaction, and a “gasPrice” - the amount of gas that the sender is charged for each step of the computation (Wood, 2024).

The data field is used in case the transaction is sent to a contract account. This field then consists of an opcode, which indicates the function to be operated and the value of the parameters. (Wood, 2024)

2.3.3. Block

A block in Ethereum is made up of four parts. A header (H), a list of transactions (T), an array which is left empty, this part used to be occupied by uncle block headers, which is now deprecated; and finally, a newly added feature after the Shanghai fork: a withdrawals collection. In the review for this project, only the header and transactions series are focused on. (Wood, 2024)

The header of a block consists of about 17 fields; some are deprecated since the replacement of the old proof-of-work and may be removed in the future. (Wood, 2024) Some important ones are:

- parentHash: the hash value of the header of the previous (parent) block. This part makes the chain highly tamper-proof. This field has a size of 256-bit
- beneficiary: the address that is rewarded the priority fee for mining or validating the block. This field is 160-bit in size.
- number: an index value of the block which starts from 0.
- gasLimit: the maximum gas value to be used per block.
- gasUsed: the value of gas used for the current block.
- timestamp: the Epoch timestamp by second at the start of this block.

2.4. Swarm Robotic and Blockchain

The two rising areas of study meet each other and are combined in many ways. As decentralization is considered the future of swarm robotics, its natural weaknesses need to be addressed. And blockchain appeared to be the solution (Castelló Ferrer, 2019).

One problem with swarms is security, as it hinders the deployment of larger swarms. Because, in more enormous autonomous swarms, ensuring security appears to be much more challenging. The inclusion of malfunctioning members in a swarm can possibly drive the whole swarm off the designated track; therefore, much work has been dedicated to the search for systems that can provide mutual trust between members of the swarm (Higgins et al., 2009).

Blockchain uses public key cryptography. This can ensure the authority over sent data if the owner uses its private key to encrypt messages. Moreover, if a message needs to be sent secretly to only one recipient, it can be encrypted by the recipient's public key. Therefore, only the one who possesses the private key can decrypt that message (Wood, 2024).

Blockchain can also support distributed decision-making, which is an important function of a swarm in coping with global situations. For example, whenever a robot finds it is required for the swarm to make a critical decision, it issues a transaction describing the situation and options to vote for. After the transaction is validated and published as a block, all robots in the swarm can vote for a choice based on their local observations. There are ways to vote. For example, in Ethereum, they can send ETH to a certain address corresponding to an option or a spending virtual credit point stored in a smart contract to vote. Blockchain also provides multi-signature techniques. This demands more than one digital signature from different robots to commit a transaction, such as confirming that work is done or that a state is reached (Castelló Ferrer, 2019).

Using smart contracts as a commander is already applied in multi-agent robot systems where there are heterogeneous devices with a 'board' system for work validation (Grey et al., 2020; Mallikarachchi et al., 2022). For homogeneous swarm robotics, there are projects using smart contracts to summarize work results with a credit system to sort out malicious robots based on detecting outliers in results (Pacheco et al., 2020; Strobel et al., 2020). This project explores a different and simpler approach to applying smart contracts as a form of task allocator, manager

and validator for robot swarms of homogeneous devices, which have a similar role, in a particular simulation situation of goods movement in warehouses. Moreover, the project proposes a credit system based on automatic random cross-validating with the swarm.

3. System Design and Architecture

The idea behind this project is to control a decentralized swarm of robots using a smart contract as a 'centralized commander' to take the strengths of both centralized and decentralized models. Although the smart contract is decentralized by nature, it is often mentioned as a 'central commander' in this project. This is because the smart contract runs on the Ethereum Virtual Machine (EVM), which operates on every full node in the network. Since each robot in the system uses a full node within a private network, and all nodes execute the same smart contract program with identical data, the smart contract gives the appearance of functioning as a single central commander, even though it operates in a decentralized manner. The goal of this project is to make it act like one.

In order to accomplish the goal, a natural approach is that each robot has one blockchain client running on it, which receives instructions from the smart contract and then controls the robot accordingly. After that, the robot's action is reported back to the smart contract.

3.1. System Overview

The swarm being developed in this project included three individual robots. Each robot has a Wi-fi adapter and connects to the same local network. The robots will not directly send any data to each other. They exclusively communicate with a private blockchain network that is deployed on the local network to which they are connected. Each robot runs a blockchain client using an Ethereum account and acts as a node in the private blockchain network.

In this project, Ethereum is used as the blockchain platform, and the local private blockchain is deployed using Kurtosis. A smart contract written in Solidity is deployed on the blockchain network. The smart contract stores the state of tasks and robots in the system. Each of the robots exclusively exchanges information with the contract account. There is an account for administrators, which is owned by humans, to view data from the blockchain and to add new tasks for the robot swarm.

Off-chain data is stored on the cloud server of Firebase, which provides the coordinates of the locations and details of goods stored in each location. The information on goods is used only for the simulation. In practical implementation, the goods should be physically recognized by robots' sensors.

The reason behind this design is to make the smart contract play the role of a central commander, which can synthesize the global situation from all robots and can make decisions based on a complete observation of the system. This design also ensures all communications are going through the smart contract. Therefore, the record of all interactions is stored in tamper-proof ledgers.

There are other programs that form the bridge between the smart contract and the physical movements of each robot. The brief functional description and structural diagram below will illustrate the overview of those programs.

Program	Tasks
Bringup robot node (turtlebot)	Initiate and connect all the hardware of a robot, listen to the master node (roscore) and perform accordingly
Navigation complex	Maintain a global map, costmaps, perform localization and plan the path towards goals, which are received from the navigator node.
Navigator node	Listen and publish to the navigation nodes and the blockchain client. Handle navigation errors, timeout, and success.
Rosbridge_web socket	Connect navigator (python) to blockchain client (javascript)
Blockchain client	Work as a brain of a robot, process tasks from the smart contract and give instructions to the navigator node. Call function/submit transaction to smart contract according to the status of the task reported by the navigator

Smart Contract	Work as a virtual central commander. Allocate tasks to robots based on the global state of the system. Manage tasks' progress. Manage robots' credits and status

Table 1. Functionalities of programs in the system

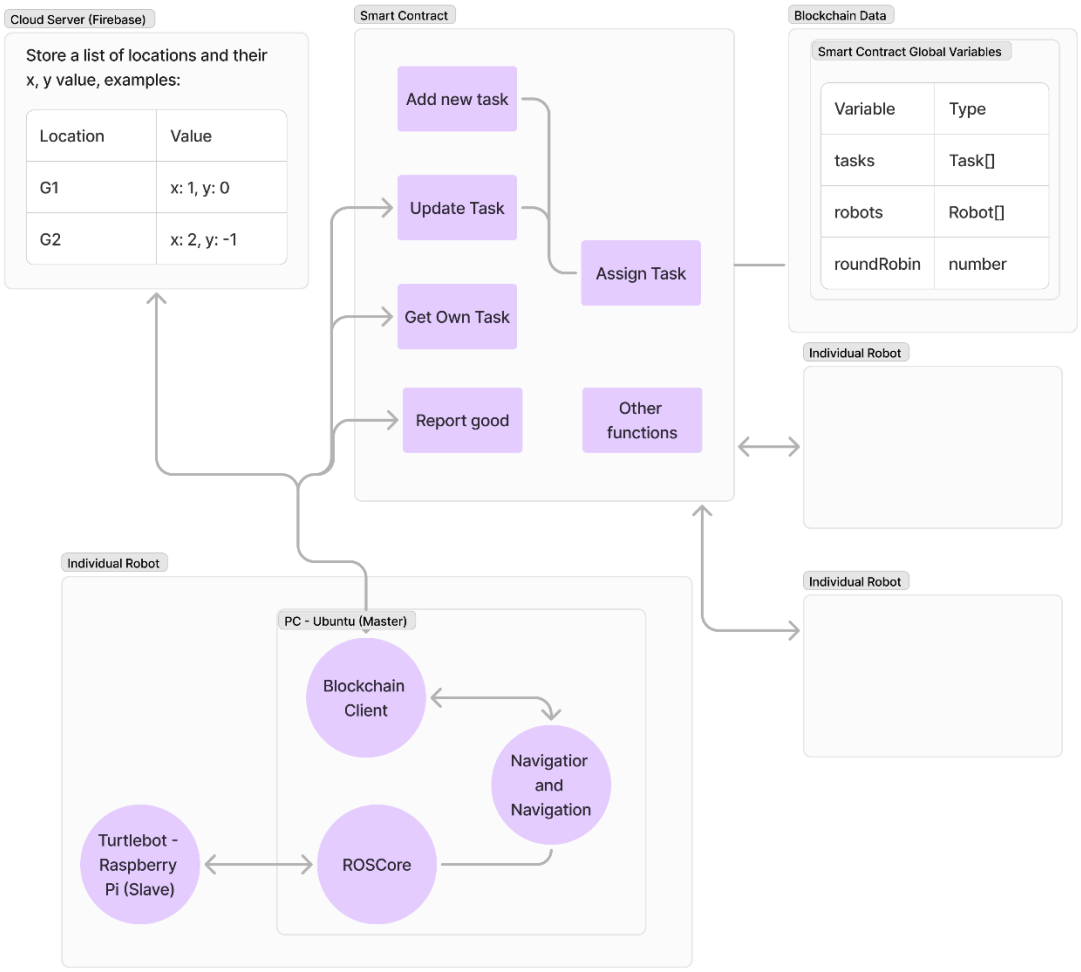

Figure 1. Overall Structure of the system

3.2. Smart Contract

3.2.1. Overview

The smart contract is vital in this system. The role of the smart contract is to receive, store, allocate, manage, and verify the completion of tasks. It is also responsible for storing robot information, communicating, coordinating, validating all the robots. In a traditional view, this smart contract perfectly plays the role of a central controller. However, there is a great difference between a smart contract and a server. That is, a smart contract is impossible to be hacked or taken over by attackers. Which preserves entirely the decentralization of the robot swarm, and enhances security significantly.

Because a deep understanding of functionalities of the smart contract greatly explains the system's architecture, the implementation of smart contract is discussed in this part instead of the later part of implementation.

The smart contract is written in Solidity and then deployed to the private blockchain. After deploying, the smart contract resides in an Ethereum contract account and has a virtual address of 160-bits (e.g. 0xb4B46bdAA835F8E4b4d8e208B6559cD267851051). The smart contract account is not a secret. Each robot's blockchain client will have the address as a constant in the source code and will use that account to communicate with the smart contract.

```
const contractAddress = "0x422A3492e218383753D8006C7Bfa97815B44373F";
//setup blockchain
 console.log("START BLOCKCHAIN SETUP");
 const TaskManager: ContractFactory = await ethers.getContractFactory(
   "TaskManager"
 );
 const accounts: Signer[] = await ethers.getSigners();

 const taskManager: BaseContract = await TaskManager.attach(contractAddress);
 const taskManagerRunner = taskManager.connect(accounts[nodeNumber]);

 taskManager.on("DoneFindingFreeBots", (freeBotsID: number) => {
   console.log("Free Bots IDs:", freeBotsID);
 });

 // Listen for DoneFindingNewTasks event
 taskManager.on("DoneFindingNewTasks", (unassignedTasksID: number) => {
   console.log("Unassigned Tasks IDs:", unassignedTasksID);
 });
 console.log("DONE BLOCKCHAIN SETUP");
```

Listing 1. Establishing connection with smart contract

3.2.2. Smart contract storage and on-chain data

In the smart contract account, there are variables storing a list of tasks and a list of robots. These variables reflect the current status of the system, and any call of function that changes those variables results in a transaction. This design allows all the important variables that determine the state to be accessed quickly in the smart contract. All the changes in such variables are stored securely in the blockchain.

```
struct Task {
       uint id;
       string good;
       string origin;
       string destination;
```

```
        uint assignee;
        uint validator;
        uint stage;

        uint timeIssued;
        uint timeStarted;
        uint timeDelivered;
    }

    struct Robot {
        address node_address;
        uint status; // 0 is free
        uint credit;
    }

    // Declare arrays
    Task[] public tasks;
    Robot[] public robots;
    uint public roundRobin = 0;
```

Listing 2. Variables stored in smart contract

For each task, ten fields of data are stored. They are

- id: a unique decimal number used to identify each task.
- good: a string identifying an item of good to be delivered in this task
- origin: a string which can identify a location where the robot needs to pick up the piece of goods; the client can use this string to get information about the coordinates of a location from cloud servers.
- destination: a string which can identify a location where the robot needs to deliver the piece of goods; the client can use this string to get information about the coordination of a location from cloud servers.

The above four fields could never be empty since it is the bare minimum information needed to determine a task in this setup.

- assignee: an unsigned integer that identifies a robot that is assigned to do this task. This default is 0 when the task is added.

- validator: an unsigned integer that identifies a robot, which validates the completion of this task. This default is 0 when the task is added.

-stage: an unsigned integer that illustrates the current state of the task. It equals to:

- 0, when the task is added
- 1, when the task is assigned to a robot
- 2, when the robot accepted and started the task
- 3, when the robot has successfully picked up the designated goods at pick up point and is on the way to the delivery location
- 4, previously reserved for error
- 5, when the robot reported successful delivery at the delivery point
- 6, when a validator confirms that the task is completed.
- 7, when a validator reports false completion of the task

Anything higher or equal to 400 means there is an error during the process, and the task is abandoned.

The above three fields reflect the current state of the task and all the robots participated in it.

- timeIssued: an Epoch timestamp in seconds, which records the time when the task is added to the list.

- timeStarted: an Epoch timestamp in seconds, which records the time when the task is started by a robot.

- timeDelivered: an Epoch timestamp in seconds, which records the time when the task is reported to be finished by a robot.

The last three fields support monitoring tasks and tracking purposes.

For each robot, there are three fields:

- node_address: a 160-bit address of the Ethereum external owned account that the robot is using.

- status: 0 is free and 1 is busy

- credit: an unsigned integer represents the credit point of the robot. When a robot accepts a task and starts working on it, one credit point is taken from that robot. When that robot finishes the task, and the result is validated by another robot, the robot is awarded an amount of credit.

3.2.3. Functions overview

As far as functions of the smart contract are concerned, there are eight functions and a constructor. The constructor function initiates three robots with the status of free with five credits each. It would always be the first transaction to be made after the deployment of the smart contract.

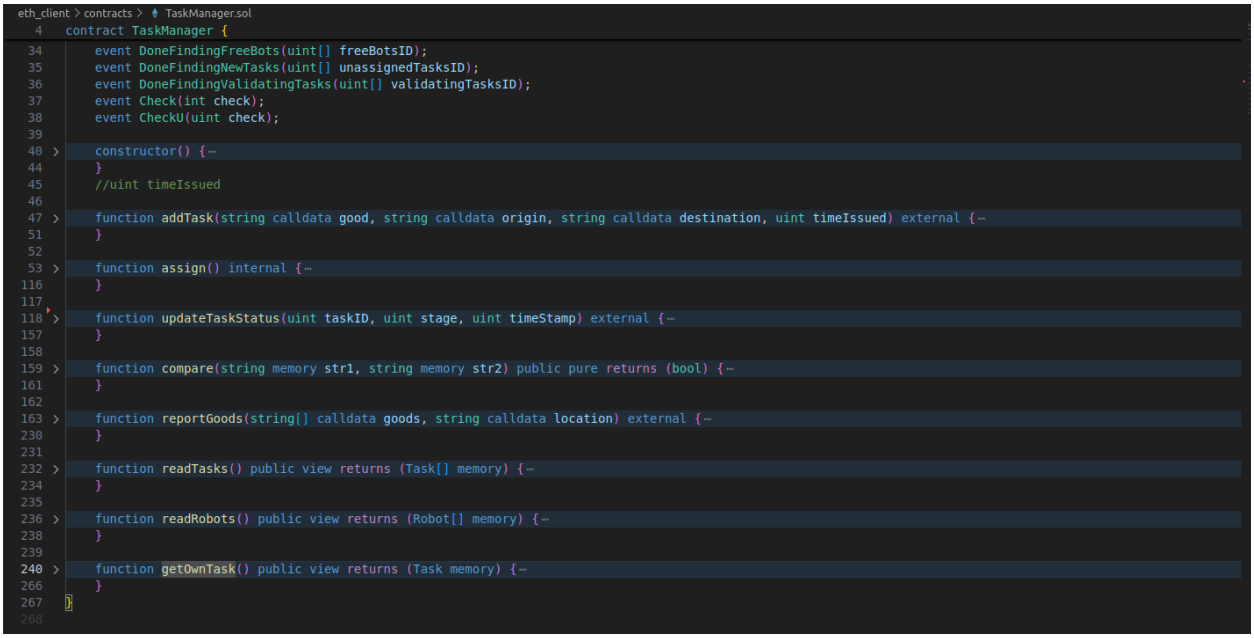

Figure 2. Functions in the Smart Contract

Among the eight functions, there are two functions defined to be used only internally; the other six are designed to be invoked externally. Within those six, there are three view functions which are used to get the current state of the storage without making any changes to it; thus, they create no transaction. For the last three, which commit transactions when they are called, two functions are exclusively used by the robots' 'blockchain client', and one is exclusive for the admin user interface.

3.2.4. Task adding functions

The first function to be discussed is 'addTask', which is usually the first one being used after initiation. This is the function to add a new task to the storage of the smart contract and log a record into the ledger. This function is the one that is exclusively created for the admin user interface, which means it is for humans to use. This function is relatively simple as it takes in three strings and one unsigned integer as arguments:

- 'good': which identifies the exact piece of goods to pick up;

- 'origin': where to pick the goods up;

- 'destination': where to deliver the goods to;

- 'timeIssued': the Epoch time in seconds, which is retrieved from the admin website's local clock.

As mentioned above, the three must-have values to create a task are included here as inputs alongside a timestamp for tracking in further monitoring. After acquiring the essential inputs, the function creates a Task object based on the structure shown in Figure 3 and then adds it to the 'tasks' list. Finally, this function calls the internal function 'assign', which allocates the unassigned tasks to eligible robots.

```
function addTask(string calldata good, string calldata origin, string calldata
destination, uint timeIssued) external {
        Task memory task = Task(tasks.length, good, origin, destination, 0, 0, 0,
timeIssued, 0, 0);
        tasks.push(task);
        assign();
    }
```

Listing 3. addTask function

If the function is successfully executed, a block that includes a transaction, whose data contains the operation code of this function and the values of the arguments, is added to the blockchain.

3.2.5. Task allocation technique and assign function

To continue, ‘assign’ function is picked out for analysis as it is called in the function above. The ‘assign’ function is different from the previous one since it is designed to be an internal function, that it can only be called by functions from within the same smart contract and cannot be invoked by external owned accounts. This mechanism is intentionally implemented as a shield to protect the critical function. Therefore, the function that is responsible for the main feature of task allocation is prevented from being invoked unexpectedly and causing unpredictable issues with the correctness and robustness of the system.

The first part of the ‘assign’ function’s work is to find all the free robots for the upcoming assignment. The result of this part is an array of free robots stored in the order of priority. This means, for example, if the array is [robot_3, robot_1, robot_0] and there are only two available tasks, then only robot_3 and robot_1 are assigned to the tasks. The priority of a robot is determined using the round-robin algorithm, which means that at one time, the robot that took the latest task has the lowest priority and vice versa. Moreover, as a credit system is used for robots, only robots that have a positive credit value are considered to be put into the array. Any robot with credit equal to zero or less is ineligible to receive any task.

```
        //#region find FreeBots
        uint[] memory freeBotsID = new uint[](robots.length);
        uint freeBotCount = 0;
        uint i = roundRobin;

        while (true) {
            i++;
            if (i >= robots.length) {
                i = 0;
            }
            if (robots[i].status == 0 && robots[i].credit > 0) {
                freeBotsID[freeBotCount] = i;
                freeBotCount++;
            }
            if (i == roundRobin) {
                break;
```

```
            }
        }

        uint[] memory freeBots = new uint[](freeBotCount);
        for (uint k = 0; k < freeBotCount; k++) {
            freeBots[k] = freeBotsID[k];
        }

        emit DoneFindingFreeBots(freeBots);
        //#endregion
```

Listing 4. Find eligible robots

After having the eligible robots in the order of priority, the function continues to find the unassigned tasks, which are defined as tasks with a stage number equal to zero. The algorithm for this is relatively simple, as it loops through the task with a conditional check to pick out tasks at stage zero.

```
        //#region Find unassigned tasks
        uint[] memory unassignedTasksID = new uint[](tasks.length);
        uint unassignedTaskCount = 0;
        uint j = 0;

        while (j < tasks.length) {
            if (tasks[j].stage == 0) {
                unassignedTasksID[unassignedTaskCount] = j;
                unassignedTaskCount++;
            }
            j++;
        }

        uint[] memory unassignedTasks = new uint[](unassignedTaskCount);
        for (uint l = 0; l < unassignedTaskCount; l++) {
            unassignedTasks[l] = unassignedTasksID[l];
        }
        emit DoneFindingNewTasks(unassignedTasks);
        //#endregion
```

Listing 5. Find unassigned tasks

As both code snippets are examined, it seems redundant when it takes two loops to select the eligible robots and the unassigned tasks. They are relevant to an issue with memory arrays in Solidity since it is required that the size be specified at the declaration of the array. And if the size of actual use is smaller than the declared size, the remaining empty slots are filled with zeros. For example, if an array ‘Anarray’ is declared with a size of 5 and Anarray[0] is assigned with 2, Anarray[1] with 1, and Anarray[2] with 3, then ‘Anarray’ will have the value of [2, 1, 3, 0, 0]. Therefore, in the code of the two above parts, the number of eligible robots and unassigned tasks are calculated in the first loops; the sizes are then used to declare the second arrays, which are actually used for task allocation.

There are events emitted after each part is done for the purpose of debugging. As logging into the console is not an option for smart contract operation, emitting events is one of a few ways to debug.

After the setup is complete, the ‘assign’ function will loop through two arrays produced by the last parts from start to end. During the looping, it matches the robot with the task until it reaches the end of the robot array or the end of the task array, whatever comes first. The assignment process includes:

1. Set the task stage to be 1, which means assigned.
2. Set the robot status to 1, which means busy.
3. Set the assignee of the task to be the robot ID.
4. Set the roundRobin variable to be the robot ID, indicating such a robot is the latest one to receive a task and has the lowest priority.

```
//#region assign tasks
uint tasksIter = 0;
uint botsIter = 0;
while (tasksIter < unassignedTasks.length && botsIter < freeBots.length) {
    tasks[unassignedTasks[tasksIter]].stage = 1; // Mark task as assigned
    tasks[unassignedTasks[tasksIter]].assignee = freeBots[botsIter];
    // tasks[unassignedTasks[tasksIter]].timeAssigned = block.timestamp;
    robots[freeBots[botsIter]].status = 1;
    roundRobin = freeBots[botsIter];
    botsIter++;
    tasksIter++;
}
//#endregion
```

Listing 6. Assignment of tasks to eligible robots

3.2.6. Task status management function

The following function to be discussed is 'updateTaskStatus'. This is an external function that is used by the robots to update the status of the tasks they are in charge of. The input of this function is the taskId, stage, and timestamp. The 'taskId' variable identifies the task to be updated; the 'stage' variable is the integer that identifies the reporting stage of the tasks, and the 'timestamp' is specified by the client to keep track of the time of the stage.

The first part of the function is to validate the identity of the sender. The algorithm is based on the storage of account addresses in the 'robots' list to know which robot is sending the request. Then, some checks are done to make sure the security:

1. Check if the sender's address is valid (be among the stored addresses)
2. Check if the sender is in charge of the task that it requests to update.
3. Check if the new stage is valid since updating from a later stage to the earlier one is not allowed

All the checks are done by using the ‘require’ function provided by Solidity. In the case of failing the checks, an error is returned to the client.

```
        //check robot id
        int robotID = -1;
        uint robotsIter = 0;
        while (robotsIter < robots.length){
            if (robots[robotsIter].node_address == msg.sender){
                robotID = int(robotsIter);
                break;
            }
            robotsIter++;
        }

        require(robotID != -1, '1:No such robot!');
```

Listing 7. Validate robot’s identity

The next part is to update the status of the task, there are two branches of instructions, one is for normal stage, another is for error stage. If the function receives a stage that is greater or equal to 400, it will run the error branch. Otherwise, the normal branch is executed. For the normal scenario, the new stage is simply assigned to the stage of the requested task in the array of tasks. However, if the error code is reported, the function would process the code to The next part is to update the status of the task; there are two branches of instructions: one is for normal stages, and the other is for the error stage. If the function receives a stage that is greater or equal to 400, it will run the error branch. Otherwise, the regular branch is executed. For the normal scenario, the new stage is

simply assigned to the stage of the requested task in the array of tasks. However, if an error code is reported, the function will process the code to include the current stage of the task in the error code. The purpose of this is to specify the stage in which the error occurs. In this way, the error code is able to illustrate more details about the error.

```
        //check task assignee
       require(tasks[taskID].assignee == uint(robotID), '0:This robot is not doing
this task!');
       require(tasks[taskID].stage  <  stage,  '2:  This  task  has  passed  this
stage!');
       //update task stage
       if (stage > 399){
           robots[uint(robotID)].status = 0;
           tasks[taskID].stage = stage + tasks[taskID].stage * 10;
       } else {
           tasks[taskID].stage = stage;
       }
```

Listing 8. Validate and update task status

The last part is for modifying the status and credit of the robot assigned to the task. When the stage is set to 2, the credit point of the assigned robot is deducted by 1 point as collateral for starting the task. Additionally, the timestamp from the input is assigned to timeStarted of the task. When the stage is five, and the task is reported to be completed, the robot status is set to be free, so it is ready to be assigned to a new task. Also, in this case, timeDelivered is modified to be the timestamp given by the client.

```
        //update robot status
        if (stage == 2){
            robots[uint(robotID)].credit -= 1;
            tasks[taskID].timeStarted = timeStamp;
        }
        if (stage == 5){
            robots[uint(robotID)].status = 0;
            tasks[taskID].timeDelivered = timeStamp;
        }
        assign();
```

Listing 9. Update time of task and status of assignee

If the function is successfully executed, a block that includes a transaction, whose data contains the operation code of this function and the values of the arguments, will be appended to the blockchain ledgers.

After updating the state of tasks and robots based on the report from the robot's 'blockchain client', this function invokes the 'assign' function to run the allocation algorithm immediately according to the new situation.

The 'compare' function is a utility function used to compare two strings since Solidity does not support native functions for that. The scheme is to compare the keccak256 hash value of two strings and return the result for the string itself.

```
function compare(string memory str1, string memory str2) public pure returns
(bool) {
  return keccak256(abi.encodePacked(str1)) == keccak256(abi.encodePacked(str2));
   }
```

Listing 10. The compare function

3.2.7. Task completion validating

The next one is ‘reportGoods’, which is a vital function created for the credit system. This is an external function that can be called by external owned contracts from robots’ clients. It takes in an array of strings of goods’ IDs and a string which indicates the location of the goods described.

This function is used to report goods at a specific location, and the smart contract uses this information to validate the completeness of tasks, which are declared to be completed by the assignee. Based on the result of the validation, the credit point of the assignee can be increased or reduced drastically.

The first part of this function checks the identity of the report sender. It is familiar with what is done in the task status update function and is used to protect the system from third-party assaults. The difference is, in this function, the robot with no credit cannot report inventory status because such a robot is determined as being malicious. Therefore, allowing it to validate other robots' deliveries is definitely unwise.

```
int robotID = -1;
uint robotsIter = 0;
while (robotsIter < robots.length){
    if (robots[robotsIter].node_address == msg.sender){
        robotID = int(robotsIter);
        break;
    }
    robotsIter++;
}

emit Check(robotID);
require(robotID > -1, '1:No such robot!');
require(robots[uint(robotID)].credit > 0, '10: This robot has no credit!');
```

Listing 11. Validate robot's identification and eligibility

The next step is to list out the completed tasks in the reporting location. This is done by filtering out the items in the 'tasks' list where:

1. It has a destination which is identical to the location in the input (using the 'compare' function)
2. It has a stage number of five, which indicates that this task is reported by the assignee to be completed but is not verified.

3. The reporting robot is not the assignee of this task. It is reasonable not to let the assignee validate its own work.

The tasks which satisfy the three conditions above are then added to a list, preparing for the next steps.

```
        //find done task at the location
        uint[] memory doneTasksID = new uint[](tasks.length);
        uint taskIter = 0;
        uint doneTaskCount = 0;
        while (taskIter < tasks.length){
            if (tasks[taskIter].stage == 5 && compare(tasks[taskIter].destination,
location) && tasks[taskIter].assignee != uint(robotID)){
                emit CheckU(tasks[taskIter].assignee);
                emit CheckU(uint(robotID));
                doneTasksID[doneTaskCount] = tasks[taskIter].id;
                doneTaskCount++;
            }
            taskIter++;
        }

        uint[] memory doneTasks = new uint[](doneTaskCount);
        uint t = 0;
        while (t < doneTaskCount){
            doneTasks[t] = doneTasksID[t];
            t++;
        }

        emit DoneFindingValidatingTasks(doneTasks);
```

Listing 12. Find completed task at the location

In this last part, the algorithm validates each satisfied task by browsing the list of the reported goods to see if the good specified in the task exists. If it does, the smart contract marks that task to be verified (stage = 6) and adds credit points to the assignee; otherwise, it marks the task as false complete (stage = 7) and reduces the credit score of the assignee. In both cases, the ID of the validating robot is recorded in the task for later checking.

```
        uint checkIter = 0;
        while (checkIter < doneTasks.length){
            uint goodIter = 0;
            bool isContained = false;
            while (goodIter < goods.length){
                if (compare(tasks[doneTasks[checkIter]].good, goods[goodIter])){
                    isContained = true;
                }

                goodIter++;
            }
            if (isContained){
                robots[tasks[doneTasks[checkIter]].assignee].credit += 2;
                tasks[doneTasks[checkIter]].validator = uint(robotID);
                tasks[doneTasks[checkIter]].stage = 6;
            } else {
                robots[tasks[doneTasks[checkIter]].assignee].credit -= 2;
                tasks[doneTasks[checkIter]].validator = uint(robotID);
                tasks[doneTasks[checkIter]].stage = 7;
            }
            checkIter++;
        }
```

Listing 13. Verify completed tasks and modify credit points

3.2.8. Information Reading

The next two tasks are ‘readTasks’ and ‘readRobots’. They are declared as a ‘public view’ function, which means it will return something to the external owned account that invokes it. This function is only used by the admin application to acquire the current state and illustrate it on the user interface.

```
    function readTasks() public view returns (Task[] memory) {
        return tasks;
    }
```

```
function readRobots() public view returns (Robot[] memory) {
    return robots;
}
```

Listing 14. Read tasks and read robots functions

The last function to be discussed is ‘getOwnTask’. This function is used by robots to get tasks which are assigned to them by the smart contract. The first part of this function validates the account sending the request and identifies which robot is getting its task. Afterwards, the function checks if the robot is assigned with a task; if not, it throws an error message and returns null.

```
    int robotID = -1;
    uint robotsIter = 0;
    while (robotsIter < robots.length){
        if (robots[robotsIter].node_address == msg.sender){
            robotID = int(robotsIter);
            break;
        }
        robotsIter++;
    }

    require(robotID != -1, '1:No such robot!');
    require(robots[uint(robotID)].status != 0, '0:The robot is free!');
```

Listing 15. Identifying robot and check its status in ‘getOwnTask’

Then, the list of tasks is searched to find one task which has the requesting robot as its assignee and is not completed. This is done by comparing the assignee of each task to the robotID of the requester while ensuring its stage is less than 4. After the loop, two tricky lines of code are added to pass the compiler's checks. The two lines of code make sure that the function returns null if no satisfied task is found, but it still seems to the compiler that it is returning a Task object.

```
        uint taskIter = 0;
       while (taskIter < tasks.length){
           if (tasks[taskIter].assigner == uint(robotID) && tasks[taskIter].stage
< 4){
               return tasks[taskIter];
           }
           taskIter++;
       }

       // assign();
       require(taskIter != tasks.length, '0:The robot is free!');
       return Task(0,'n','n', 'n', 0, 0, 0, 0, 0, 0);
```

Listing 16. Finding the current task of a robot

4. Implementation

In this chapter, the development environment is described, and the technical details of critical parts: the 'navigator', the 'blockchain client', and the robot's navigation configurations will be discussed. The smart contract details have already been discussed in the previous part, so they are not discussed in this part. The relevant parts of the web user interface are also mentioned.

4.1. Development Environment

The project is a combination of two fields, Robotics and Blockchain, so the stack of technologies in both fields is utilized and should be specified. For the development of the robotics side, ROS Noetic is used on a Ubuntu 20.04 LTS (Focal) laptop for system execution. On the blockchain side, the stack of Hard Hat, Kurtosis, and Docker are used to develop and deploy a private Ethereum blockchain network. Gazebo and Nviz are used for the physical simulation and illustration of robots' movements. In practical robot experiments, the system uses the Turtlebot 3 - Burger model with the default settings, which runs ROS Kinetic. The admin user interface for illustrating the status of robots and tasks is developed using Nextjs with the extension MetaMask.

There are multiple programming languages featured in the project. For robotics, launch files of move_base are developed using XML with the parameters written in YAML format. The navigator to coordinate and report the movement of a robot to the blockchain client is written in Python with the 'rospy' library. The blockchain client and admin user interface is written in Javascript and

Typescript. Finally, the smart contract is coded in Solidity. All the code is written using Visual Studio Code as the text editor.

4.2. The navigator

In this project, the navigator is written to be a node in the ROS system. This node is needed because the blockchain client is written in Javascript and would have numerous troubles interacting directly with the ROS system to control the robot's behaviours. The 'navigator' is created to be the one and only interface for the 'blockchain client' to control the robot. The two programs are connected through a websocket provided by ROS in the package rosbridge_server.

```
pub = rospy.Publisher('taskReport', String, queue_size=10)
navclient = actionlib.SimpleActionClient('move_base',MoveBaseAction)
isInterrupted = False
t1 = None

if __name__ == "__main__":
   isBusy = 0
   stage = 0
   startTime = 0
   endTime = 0
   rospy.init_node('tb3_0_navigator')
   rospy.Subscriber("taskAssign", String, callThread)
   print('tb3_0_navigator is listening!')

   rospy.spin()
```

Listing 17. Initiation of the 'navigator'

The first part to discuss about the navigator is the initialization. Here, the code creates the publisher and the subscriber to establish two-way communication with the 'blockchain client'. The subscriber listens to the topic 'taskAssign', which emits type String data, and the publisher writes String data to the topic 'taskReport'. The variable 'navclient' is declared for sending instructions to the 'move_base' node and control navigation of the robot. 't1' is a variable to store a thread. This program uses the multi-thread technique because it is required to listen to two external programs continuously at a time. The variable' isInterrupted' is used to interrupt ongoing navigation, while

the startTime and endTime prevent duplicate instructions from the 'blockchain client'. 'busy' and 'stage' variables are used to control tasks' operation. Finally, 'rospy.init_node()' and 'rospy.spin()' are used to start a ROS node and keep it looping, respectively.

According to the flow, the next function to be discussed shall be ‘callThread’. This function is invoked every time there is a message pushed to the topic ‘taskAssign’ by the ‘blockchain client’. As its name suggests, this function’s main job is to start a thread to react to the message. However, it also performs some conditional checks after taking the data from the message as input.

Prior to the discussion of the function, the format of data transferred from the ‘blockchain client’, which would significantly support the understanding of this function, is demonstrated. The standard format of the message consists of 5 variables separated by semicolons.

```
timeStamp;goalPosition.x;$goalPosition.y;stageNumber;taskId
```

Listing 18. Data Format 1 through taskAssign

- timeStamp: an integer specifies the Epoch timestamp when the message is sent
- goalPosition.x and goalPosition.y: the coordinate of the goal
- stageNumber: an integer indicates the stage of the ‘blockchain client’ when it issues this message
- taskId: the ID number of the current task.

Above is the format of the message when the ‘blockchain client’ commands the ‘navigator’ to move towards a coordinate. The second format is for interrupting the current navigation, mainly due to an error occurring in the ‘blockchain client’. In this case, the data is only a string ‘-1’.

```
def callThread(data):
    global t1, isInterrupted

    trimData = data.data.split(';')

    if (int(trimData[0]) == -1):
        print('Should interrupt!!!!')
        isInterrupted = True

        try:
            t1.join()
        except:
            print('no thread is running')
        return

    try:
        t1.join()
    except:
        print('no thread is running')
    finally:
        isInterrupted = False
        t1 = threading.Thread(target=doTaskCallback, args=(data,))
        t1.start()
```

Listing 19. ‘callThread’ function

The 'callThread' function, after splitting data into variables, checks if the first variable equals -1 or not. This conditional clause can separate the two formats of messages as the 'timeStamp' cannot be negative in the standard format.

In case it is the standard format, it first tries to join any running thread into the main thread where this function is running. This action ensures every previous navigation ends properly before the new one starts and prevents unintended task preemptions. If there is an ongoing task, the function will wait for it to end and execute the 'finally' section. Otherwise, if there is no thread performing any navigation task, it would print out a log and execute the 'finally' section. The 'finally' part starts a new thread to run and listen to a navigation task from the navigation nodes. It is worth noting that the variable 'isInterrupted' is set to False before starting a new navigation task.

In case the message is sent to interrupt the ongoing navigation task, the variable 'isInterrupt' is set to True. Afterwards, 't1.join()' is executed to end the thread and wait for the task to end.

The next function to be analyzed is the one that is executed by ‘callThread’ - ‘doTaskCallback’. The purpose of this function is to execute the navigation task and report results to the ‘blockchain client’. The first part of this function gets global variables and extracts values from the message’s data. It also checks if another task is being executed by using ‘isBusy’ and stops the current function if there is one. This is an old way of preventing preemption, and in this version, the thread technique prevented this situation thoroughly. However, this piece of code may help identify an unexpected runtime error, so it is not removed.

```
def doTaskCallback(data):
    taskId = -1
    try:
        global isBusy
        global stage
        global startTime
        global endTime
        trimData = data.data.split(';')

        if (isBusy == 1):
            print('robot is busy doing another task!!')
            return

        timeStamp = int(trimData[0])
        taskStage = int(trimData[3])
        taskId = int(trimData[4])
```

Listing 20. Variables in ‘doTaskCallback’

After determining all the variables, the function starts to check the timeStamp to ensure there is no repeating instruction. This piece of code cancels any message that comes in between the 'startTime' and the 'endTime' of the previous task. This makes sense because in this workflow, the 'blockchain client' is not supposed to send any new navigation instructions before the previous instructions are fulfilled or failed. This design also allows the 'blockchain client' to repeatedly send messages until receiving a confirmation, ruling out the chance of a message being missed. Missing messages

actually happened in the first tests due to an issue occurring only in a small while after the 'blockchain client' connected to the websocket.

```
        if (startTime != 0 and (timeStamp >= startTime and timeStamp <= endTime)):
            return
```

Listing 21. Time check in ‘doTaskCallback’

The next part is to prepare the variables to start the task. Firstly, the value of timeStamp is set to startTime, marking a new startTime of a new task. Then, set the value of isBusy and stage both to 1. The next step is to define the position and orientation of the goal. The 'x' and 'y' of 'position_goal' is taken from the input data. For 'orientation_goal', it defaults to one constant orientation. This orientation is necessary to execute the navigation, but its value is not critical to the system. The last thing to do before the actual execution is to message the 'blockchain client' that the instruction has actually been received and is now executed. The message includes the 'taskStage' and 'taskId' which can accurately identify which navigation of which task has been successfully received.

```
        startTime = timeStamp
        isBusy = 1
        stage = 1

        position_goal = {"x": float(trimData[1]), "y" : float(trimData[2]), "z":
0.0}
        orientation_goal = {"x": 0.0, "y":  0.0, "z":  0.0, "w": 0.1}



        rospy.loginfo('received task!')
        pub.publish(f'{taskStage};{taskId}')
```

Listing 22. Preparation to start a navigation task

Before starting the task, the sub-functions are discussed. These functions would be called inside the execution flow when a certain event happens or a condition is met. The first one is ‘timeoutResolve’, which is invoked when the navigation task takes longer than the time limit. This one is used to prevent the navigation nodes from looping forever to find a path to an unreachable

goal, which happens frequently in tests with real robots. The function sets ‘isBusy’ to 0, sets the endTime to the current time and informs the ‘blockchain client’ of the timeout event with the code ‘-3’ alongside the taskId. The second one is ‘done_nav_deliver_cb’, which is a callback that runs when a navigation task is successfully done. It checks if the status is equal to 3 (which means successful), sets the stage to 2, sets the endTime, and informs the success to the ‘blockchain client’ by sending back a stage it received but increased by 1.

```
    def timeoutResolve():
        global isBusy
        global endTime
        isBusy = 0
        endTime = round(time.time() * 1000)
        print('TIMEOUT! or FAILURE!')
        pub.publish(f'{-3};{taskId}')
        #-3 means timeout

     def done_nav_deliver_cb(status, result):
        if status == 3:
            rospy.loginfo("the goal is reached!")
            global stage
            global endTime
            stage = 2
            endTime = round(time.time() * 1000)
            pub.publish(f'{taskStage+1};{taskId}')
```

Listing 23. Function and callback in ‘doTaskCallback’

The next part is about the execution of the navigation task. It is a while loop running forever until the node is forcefully shut down or it runs into a break statement. With the initial stage of 1, the ‘navigation’ is called with position, orientation and a callback. It then waits for the result and checks if it is true or false. If it is true, the stage is set to 2 by the callback, and the while loop runs the branch where the stage equals two and breaks the loop. Otherwise, a False value indicates a failure in the navigation nodes, which triggers the ‘timeoutResolve’ function and breaks the loop.

```
        while not rospy.is_shutdown():
            if (stage == 1):
                result1= navigate(position_good, orientation_good,
done_nav_deliver_cb)
                if not result1:
                    timeoutResolve()
                    break
            elif (stage == 2):
                #add success block
                print('finished')
                isBusy = 0
                stage = 0
                break
```

Listing 24. While loop in ‘doTaskCallback’

The last part of this function is the exception part. This part is important because it is required for the ‘navigator’ to keep running continuously without being shut down by a runtime error. This part catches any error in this loop and reports it to the ‘blockchain client’ to log the error into the blockchain system. This part also resets the variables and marks the end of a task so the function can go on with the upcoming navigation task.

```
    except BaseException as err:
        print('navigator error in do task callback - should rebound',err)
        isBusy = 0
        endTime = round(time.time() * 1000)
        pub.publish(f'{-2};{taskId}')
        return
```

Listing 25. Exception catching in ‘doTaskCallback’

The ‘navigate’ function is used to exchange information directly with the move_base and is responsible for performing a single navigation task. This function takes two objects: position and orientation and a callback function. In this function, two global variables: ‘navclient’ and ‘isInterrupted’ are used. The line ‘navclient.wait_for_server()’ is written at the beginning to ensure

the connection to the ‘navclient’ action server is active. Afterwards, a goal object is created based on the input and is used in ‘navclient.send_goal()’ function as a parameter, along with callbacks to receive and print the status of the navigation task. Then, a start time and a timeout duration are declared to control the maximum execution time of the navigation. The while loop right after it is responsible for catching the result of the task. There are four possible outcomes of a navigation task: success, failure by errors, timeout, and being interrupted. While in success and failure outcomes, this function only listens to the report from the navigation nodes and acts upon it; this function forces the outcome in other cases using ‘navclient.cancel_all_goals()’. Specifically, it follows instructions from the ‘blockchain client’ in case of interruptions and makes a decision by itself to stop the task in case of timeout. Therefore, the ending of a task can be initiated from different parts of the system.

```
def navigate(position, orientation, done_cb):
    # navclient = actionlib.SimpleActionClient('move_base',MoveBaseAction)
    global navclient, isInterrupted
    navclient.wait_for_server()

    # Example of navigation goal
    goal = MoveBaseGoal()
    goal.target_pose.header.frame_id = "map"
    goal.target_pose.header.stamp = rospy.Time.now()

    goal.target_pose.pose.position.x = position['x']
    goal.target_pose.pose.position.y = position['y']
    goal.target_pose.pose.position.z = position['z']
    goal.target_pose.pose.orientation.x = orientation['x']
    goal.target_pose.pose.orientation.y = orientation['y']
    goal.target_pose.pose.orientation.z = orientation['z']
    goal.target_pose.pose.orientation.w = orientation['w']

    navclient.send_goal(goal, done_cb, active_cb, feedback_cb)

    start_time = rospy.Time.now()
    timeout = rospy.Duration(secs=180)  # 180 seconds timeout

    while not rospy.is_shutdown():
        # Check if interrupted
        if isInterrupted:
```

```
            rospy.loginfo("Navigation interrupted!")
            navclient.cancel_all_goals()  # Cancel all goals
            return False

        state = navclient.get_state()
        if state == 3:
            rospy.loginfo(navclient.get_result())
            return True
        elif state > 3:
            rospy.logerr(state)
            return False

        # Check for timeout
        if rospy.Time.now() - start_time > timeout:
            rospy.logwarn("Navigation timed out!")
            navclient.cancel_all_goals()
            return False
```

Listing 26. The navigate function

4.3. The blockchain client

The blockchain client is the most sophisticated part of the system. It is the brain of each robot; it makes decisions and communicates with the smart contract, the off-chain server, and the navigator. Similar to other programs in this system, the blockchain client uses an endless loop to keep itself always active. The blockchain client is designed following the state pattern, which means it acts upon the stage it is currently in, and there are transitions between states. In this program, states are actually called ‘stages’ because they were initially created based on stages of the tasks. Inside the endless loop, there are conditional clauses according to stages to decide which piece of code to be executed. The blockchain client has more stages than the task itself because, in the later phase of development, more stages were added to handle arising situations in the implementation.

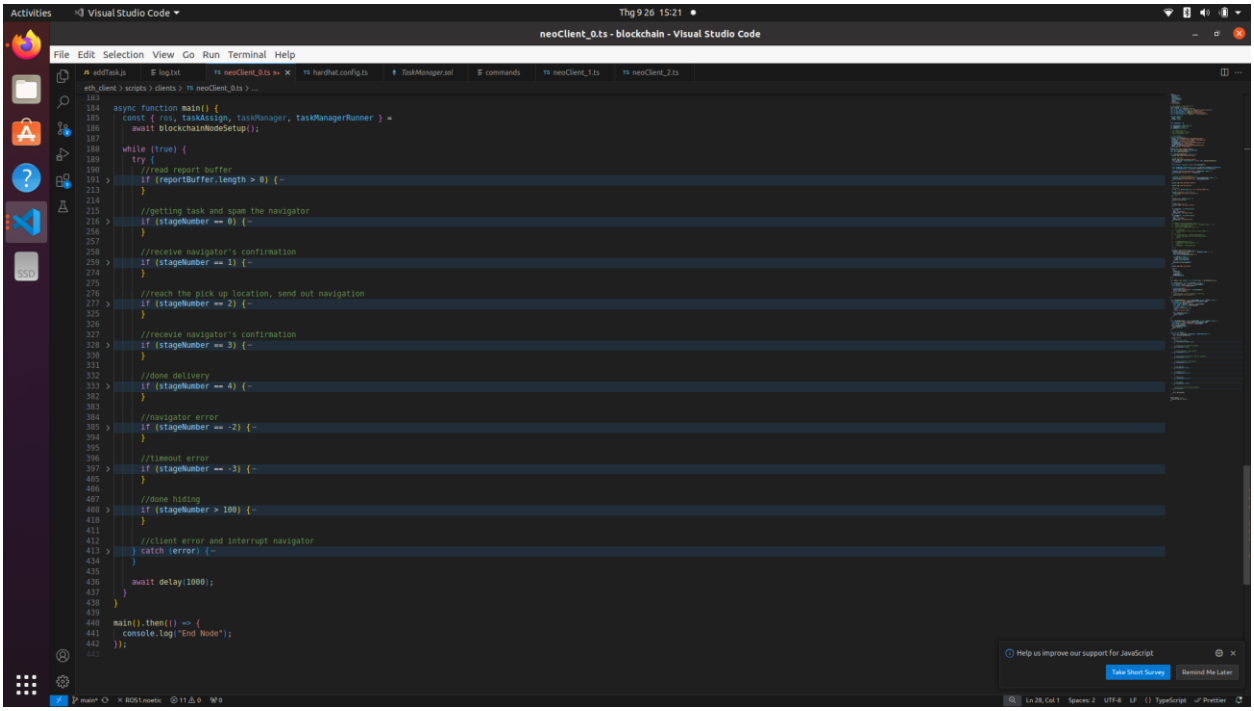

Figure 3. Overview of the blockchain client loop

Before going to the core loop of the blockchain client, the setup part of this program will be discussed. The setup part runs first and only once when the program starts, and the core loop is not executed until the setup part is done.

First is the connection setup to the off-chain database. In this project, Firebase's Firestore is used as off-chain storage because of its cloud-based nature, which provides constant availability, security, and ease of access. Firebase provides 'firebaseConfig,' which acts as a route to connect with the cloud server. The 'apiKey' in it is the secret key used to prove the authority of access to the database. The 'db' variable is initialized to be used later for interactions with off-chain data in the core loop.

```
// setup firebase
const firebaseConfig = {
 apiKey: "AIzaSyA9qKXOVt5Nyc_hZ5hAqBBHf6EuStFm2Bc",
 authDomain: "location-service-6d88a.firebaseapp.com",
 projectId: "location-service-6d88a",
 storageBucket: "location-service-6d88a.appspot.com",
```

```
 messagingSenderId: "126038330312",
 appId: "1:126038330312:web:a0895b39c0fab0cda2cefd",
 measurementId: "G-Y4X5Y6WTPE",
};
console.log("START FIREBASE SETUP");
const app = initializeApp(firebaseConfig);
const db = getFirestore(app);
```

Listing 27. Firebase Setup in the blockchain client

The next part is the initialization of the connection with the blockchain network. This part is written from the view of one robot. This part starts with creating a 'ContractFactory' with the smart contract prototype named 'TaskManager'. 'TaskManager' is the only smart contract used in this project. It is worth noticing that for the line 'await ethers.getContractFactory("TaskManager")', the result is only an undeployed prototype of the smart contract but not a deployed instance. A smart contract factory includes the bytecode, the interface, and, optionally, the account signer that may deploy the smart contract. In this function, the contract interface is essential for initializing the interactions with the deployed one. Then, the function gets the available accounts in the current environment, which includes the one account assigned to this robot. Afterwards, an object representing the deployed instance of the smart contract is created by binding the ContactFactory to the address of the contract account by the function 'attach'. Finally, a smart contract runner is created as the connection between the smart contract account and this external account, which this robot uses, by using the function 'connect()' of BaseContract. This 'taskManagerRunner' is used later to invoke functions of the smart contract on behalf of a specific account belonging to the robot this program is running on. Finally, the subscriptions to events of the smart contract are done for debugging purposes.

```
 console.log("DONE BLOCKCHAIN SETUP");

 //setup blockchain
 console.log("START BLOCKCHAIN SETUP");
 const TaskManager: ContractFactory = await ethers.getContractFactory(
   "TaskManager"
 );
 const accounts: Signer[] = await ethers.getSigners();
```

```
const taskManager: BaseContract = await TaskManager.attach(contractAddress);
const taskManagerRunner = taskManager.connect(accounts[nodeNumber]);

taskManager.on("DoneFindingFreeBots", (freeBotsID: number) => {
  console.log("Free Bots IDs:", freeBotsID);
});

// Listen for DoneFindingNewTasks event
taskManager.on("DoneFindingNewTasks", (unassignedTasksID: number) => {
  console.log("Unassigned Tasks IDs:", unassignedTasksID);
});

console.log("DONE BLOCKCHAIN SETUP");
```

Listing 28. Blockchain Setup

```
console.log("START ROS SETUP");

//ROS setup
let ros = new ROSLIB.Ros({ url: "ws://localhost:9090" });

ros.on("connection", function () {
  console.log("successfully connected!");
  // pulish();
});

ros.on("error", (error: any) => {
  console.error(error);
});

ros.on("close", () => {
  console.log("connection closed");
});

let taskAssign = new ROSLIB.Topic({
  ros: ros,
  name: `/taskAssign`,
  messageType: "std_msgs/String",
});
let taskReport = new ROSLIB.Topic({
```

```
  ros: ros,
  name: `/taskReport`,
  messageType: "std_msgs/String",
});

taskReport.subscribe((message: any) => {
  console.log("Received message on " + taskReport.name + ": ");
  console.log(message.data);
  const data = message.data.split(";");

  let newReport: IReport = {
    stage: Number(data[0]),
    taskId: Number(data[1]),
  };
  reportBuffer.push(newReport);
});

console.log("DONE ROS SETUP");
```

Listing 29. ROS Setup

This part of the function sets up the connection to the navigator node described in the part above. Firstly, the function connects to the websocket created by the 'ros_bridge' package. In the simulation, the port of the websocket may vary as different websockets are opened on the same laptop and are used to connect different robots to different 'roscores'. Afterwards, there are subscriptions to the websocket events. These lines are used exclusively for debugging and logging. Next are the declarations of the connections to the two topics, 'taskAssign' and 'taskReport'. Those two are the routes to send and receive messages from the navigator. The 'taskAssign' object is used in the core loop, while the 'taskReport' topic is continuously listened to using the function 'taskReport.subscribe()'. The subscription is used to handle the received message immediately by changing the stage of the blockchain client according to the message received from the navigator. However, that approach faced the tedious problem of not reacting to messages when they were coming at a rapid rate. Therefore, a buffer 'reportBuffer' is created to store a first-in-first-out stack of messages and process them one by one.

The failure scenario mentioned above was that the program received message M1, set the stage from idle (-1) to 2, and then ran a piece of code P1 according to stage 2. While executing P1, the

message M2 came, setting the stage to 3, which finally was changed to idle (-1) again as a part of P1's execution. As a result, the system missed the message M2 and never switched to stage 3.

The first part of the core loop is reading the report buffer. There are the appearances of some global variables in this part. 'taskValue' is a task object acquired from the smart contract storing the information of the task that this blockchain client is working on. 'stageNumber' is the variable that decides which state this function is in. It directly affects which piece of code is executing in this while loop.

The design of this reading buffer part ensures that the message is processed only when no execution is being performed. Therefore, it prevents the scenario above and any other race condition scenarios. This piece of code takes and removes the first message from the navigator in the buffer. A message from the navigator includes two parts: a stage number and a task ID. Task ID is included in the message to prevent the blockchain client from mistakenly applying a report of one task to another task. This issue happened in previous experiments when the navigator and blockchain client programs were not synchronized. About the stage number, the navigator sends stage numbers in messages based on the received stage number in the instructions from the blockchain client. When the blockchain client receives a message with stage number 'S' from the navigator, it understands that stage 'S' is done and shifts its stage number to S+1. However, it is a code of error if the received stage number is negative, and the 'stageNumber' of the blockchain is set to the exact error code informed by the navigator to resolve the error correctly.

```
    if (reportBuffer.length > 0) {
      let currentReport = reportBuffer.shift();

      if (!currentReport) {
        throw "shift error!";
      }

      if (!taskValue) {
        console.log("no current task to receive report!");
        continue;
      }
      if (currentReport.taskId != Number(taskValue[0])) {
        console.log("report not for the current task");
```

```
        continue;
      }

      if (currentReport.stage >= 0) {
        stageNumber = currentReport.stage + 1;
      } else {
        //error code
        stageNumber = currentReport.stage;
      }
    }
```

Listing 30. Read report buffer inside while loop

Stage	Meaning
-1	Transition condition: the program has finished dealing with an event, which instructs the program to wait for the next event without doing anything. Description: The program is idle, and no process is running except for the message buffer reading.
0	Transition condition: the program has ended the previous task/ has no task assigned to the Ethereum account. Description: the program is waiting for a task from the smart contract. It will send instructions to the navigator on moving towards the pickup point once it finds a new task.
1	Transition condition: The program receives a message from the navigator saying that it has started the navigation based on the instruction sent in stage 0. Description: The program informs the smart contract that the task is started. In the end, it set the stage number to -1.
2	Transition condition: The program receives a message from the navigator saying that it has done the navigation based on the instruction sent in stage 0. Description: The program performs the pickup and verification processes. Afterwards, it sends instructions on how to move to the delivery point to the navigator. In the end, it set the stage number to -1.
3	Transition condition: The program receives a message from the navigator saying that it has started the navigation based on the instruction sent in Stage 2. Description: The program does nothing specific except for the message buffer reading.
4	Transition condition: The program receives a message from the navigator saying that it has done the navigation based on the instruction sent in stage 2 Description: The program performs the delivery and verification processes.

	Afterwards, it sends an instruction of moving away from the delivery point to the navigator. In the end, it set the stage number to 0.
-2	Transition condition: The program receives a message from the navigator saying that it caught a runtime error. Description: The program messages the smart contract that the task is ended by an error in the navigator.
-3	Transition condition: The program receives a message from the navigator saying that the navigation exceeded the time limit. Description: The program messages the smart contract that the task is ended by timeout.

Table 2. Stages Overview

The next part is the code for stage number 0, which is responsible for finding a task assigned to this robot. This stage is transitioned into when the program first starts, when it has done the previous task, and when no task is assigned to the robot this program is running on. That means it will loop forever in this stage if no task is assigned to this robot.

The line 'await taskManagerRunner.getOwnTask();' is used to get the task assigned to this account from the smart contract. The line returns nothing and throws an error: '0:The robot is free!' (shown in the code of the smart contract) if there is no task assigned. In this case, the program would wait for 2 seconds before running again at stage number 0. In the case of getting a task, that task object is stored in the global variable taskValue. After that, an instruction message consisting of five elements is created and sent to the navigator through the topic 'taskAssign'. The function' getDocument()' is constructed and used to get the coordinates of a location based on its ID. The other three elements mentioned in detail above (page 36) in the navigator parts are the timeStamp, the stageNumber and the taskID. The reason why taskValue[0] is the taskID is that the result of getOwnTask() is a Task object in Solidity. But when it is returned in Javascript, it becomes an array with a size of 10. Each element in the array is a field in the Task structure, with the same order as that in the Task structure (page 20/figure 3)

```
    //getting task and spam the navigator
    if (stageNumber == 0) {
      //get task
```

```
      taskValue = null;
      try {
        taskValue = await taskManagerRunner.getOwnTask();
        console.log("get task result", taskValue);
      } catch (error: any) {
        console.error(error);
        console.log(error.message);
        const code = +error.message.split(":")[1];
        console.log("check code", code);
        if (code == 0) {
          await delay(2000);
        }
      }

      //start job
      if (taskValue == null) {
        continue;
      }
      const timeStamp = Date.now();
      const goodPosition = await getDocument(taskValue[2]);

      console.log(
        "message: ",
        `${timeStamp};${goodPosition.x};${goodPosition.y}`
      );
      let message = new ROSLIB.Message({
        data:
`${timeStamp};${goodPosition.x};${goodPosition.y};${stageNumber};${taskValue[0]}
`,});
      taskAssign.publish(message);
      console.log("done start publish!");
    }
```

Listing 31. Stage number 0

```
    //receive navigator's confirmation
    if (stageNumber == 1) {
      try {
        let tx = await taskManagerRunner.updateTaskStatus(
          Number(taskValue[0]),
          2,
          Math.round(Date.now() / 1000)
```

```
        );
        await tx.wait();
        console.log("Receiving Task was successful");

        stageNumber = -1;
      } catch (error) {
        console.log("Transaction failed:", error);
        stageNumber = 0;
      }
    }
```

Listing 32. Stage number 1

The program transitions to stage number 1 when it receives a message from the navigator confirming the receipt of the instruction in Stage 0 and the beginning of the navigation to the pick-up location. In this stage, the program reports the status to the smart contract that the current task has moved to stage 2 (Started). The code is put in a try-catch bracket, so if there is a runtime error, the task will be marked as failed with an error initiated from the blockchain client, and this program will transit to stage 0, waiting for a new task.

```
    //reach the pick up location, send out navigation
    if (stageNumber == 2) {
      try {
        const timeStamp = Date.now();
        const deliveryPosition = await getDocument(taskValue[3]);

        const goodPosition = await getDocument(taskValue[2]);

        await checkAndTakeGood(taskValue[2], taskValue[1]); //goodPosition and
goodID

        let tx = await taskManagerRunner.updateTaskStatus(
          Number(taskValue[0]),
          3,
          Math.round(Date.now() / 1000)
        );
        await tx.wait();

        try {
          let txValidate = await taskManagerRunner.reportGoods(
```

```
            goodPosition.good,
            taskValue[2]
          );
          await txValidate.wait();
        } catch (error) {
          console.log(
            "reporting good is not running, the system still continue",
            error
          );
        }

        let message = new ROSLIB.Message({
          data:
`${timeStamp};${deliveryPosition.x};${deliveryPosition.y};${stageNumber};${taskV
alue[0]}`,
        });
        taskAssign.publish(message);
        console.log("done delivery publish!");

        console.log("Receiving Good was successful");
        stageNumber = -1;
      } catch (error) {
        console.log("Receiving Good failed:", error);
        let tx = await taskManagerRunner.updateTaskStatus(
          taskValue[0],
          403,
          Math.round(Date.now() / 1000)
        );

        await tx.wait();
        stageNumber = 0;
      }
    }
```

Listing 33. Stage number 2

Stage number 2 is switched on when the blockchain client receives a message from the navigator that the navigation to the pick-up point is successfully done. The program then does four things, one after another. First, it takes the goods from the inventory, and then it updates the state to the smart contract. Afterwards, it reports the existing goods in the inventory (the state of the inventory before it takes the goods). Finally, it sends a message instructing the navigator to move the robot

to the delivery point. The stage is finally set to -1, and the program waits for the next message from the navigator. The error-catching process is applied similarly to the above stage.

```
    if (stageNumber == 3) {
      console.log("received delivery goal");
    }
```

Listing 34. Stage number 3

This stage means the blockchain client receives the confirmation of the navigation instruction in stage 2. Which indicates the robot is moving towards the delivery location.

```
    //done delivery
    if (stageNumber == 4) {
      try {
        const deliverPosition = await getDocument(taskValue[3]);

        await checkAndGiveGood(taskValue[3], taskValue[1]);

        let tx = await taskManagerRunner.updateTaskStatus(
          taskValue[0],
          5,
          Math.round(Date.now() / 1000)
        );
        await tx.wait();

        try {
          let txValidate = await taskManagerRunner.reportGoods(
            deliverPosition.good,
            taskValue[3]
          );
          await txValidate.wait();
        } catch (error) {
          console.log(
            "reporting good is not running, the system still continue",
            error
          );
        }

        console.log("Transaction was successful");
        stageNumber = 0;
```

```
        const timeStamp = Date.now();

        const deliveryPosition = await getDocument(taskValue[3]);
        let message = new ROSLIB.Message({
          data: `${timeStamp};${deliveryPosition.x - 0.5};${
            deliveryPosition.y + 0.5
          };${100};${-1}`,
        }); // get away from the delivery spot
        taskAssign.publish(message);
        console.log("done delivery publish!");
      } catch (error) {
        console.log("Delivery failed:", error);
        let tx = await taskManagerRunner.updateTaskStatus(
          taskValue[0],
          403,
          Math.round(Date.now() / 1000)
        );
        await tx.wait();
        stageNumber = 0;
      }
    }
```

Listing 35. Stage number 4

Stage number 4 marks the end of one task, as the navigator informs that the robot has successfully reached the delivery goal with the goods. Firstly, this program adds the goods to the inventory of the delivery location. Secondly, it updates the status of the task to the smart contract. Thirdly, it reports the existing goods in the inventory before the current good is added to the smart contract to verify other tasks ended in this location. Finally, it sends a message to the navigator to move the robot away from the location. This action prevents collisions with other robots going to this location in the near future.

```
    //navigator error
    if (stageNumber == -2) {
      console.log("Task Error in Navigator");
      let tx = await taskManagerRunner.updateTaskStatus(
```

```
        Number(taskValue[0]),
        402,
        Math.round(Date.now() / 1000)
      );
      await tx.wait();
      stageNumber = 0;
    }

    //timeout error
    if (stageNumber == -3) {
      let tx = await taskManagerRunner.updateTaskStatus(
        Number(taskValue[0]),
        405,
        Math.round(Date.now() / 1000)
      );
      await tx.wait();
      stageNumber = 0;
     }
```

Listing 36. Error handling stage

The stages of negative two and negative three occur when those error codes are sent from the navigator, informing the runtime error and timeout error, respectively. This program sends the error message to the smart contract with the error code. The full error code will be shown in the Appendix.

```
  } catch (error) {
    console.log("global error:", error);
    let message = new ROSLIB.Message({
      data: `${-1}`,
    });
    console.log("published error!");

    taskAssign.publish(message);
    console.log("client error!");
    try {
      let tx = await taskManagerRunner.updateTaskStatus(
        Number(taskValue[0]),
        400,
        Math.round(Date.now() / 1000)
```

```
      );
      await tx.wait();
    } catch (error) {
      console.log("The error occurred not related to any task!");
    }

    stageNumber = 0;
  }
```

Listing 37. Global error handling

In case an error is not caught by the error handlers in each stage, it will be caught by this global error handler. This piece of code interrupts any working task in the navigator by sending '-1' to the program. Then, it informs the smart contract of an unknown error and changes the stage to 0, waiting for a new task.

4.4. The navigation system

The navigation system includes some nodes that are responsible for the Turtlebot's localization and pathfinding. For localization, the default Adaptive Monte Carlo Localization (AMCL) is used. For the path calculation, the 'move_base' node is used with some modification. The 'move_base' node in ROS serves as the interface between the higher-level navigation goals and the underlying control and perception. It works with the data from the Turtlebot lidar to plan a path to a goal. However, the default parameters and planner of move_base are not optimal for the practical simulation of swarm robotics because other robots become dynamic obstacles in the map. So, the Time Elastic Band local planner is used to improve pathfinding in a dynamic environment. The rate of updating the local cost map is increased to the hardware maximum to allow robots to react to sudden obstacles. The area of the local map has also been increased to 4 meters to catch obstacles from a further distance. The cost of passing near an obstacle is set to be higher and is drastically increased when getting closer to the obstacle. All three modifications above prevent collisions between robots on their path but make the robots move more slowly and conservatively. The recovery behaviours list is set to empty, as in this version (Noetic) of ROS, the 'move_base' has an occasional issue when trying to cancel a goal when it is performing a recovery behaviour. Other parameters can be inspected in the Appendix.

```
<launch>
   <arg name="model" default="$(env TURTLEBOT3_MODEL)" doc="model type [burger,
waffle, waffle_pi]"/>
   <arg name="cmd_vel_topic" default="/cmd_vel" />
   <arg name="odom_topic" default="odom" />
   <arg name="move_forward_only" default="false"/>


   <node pkg="move_base" type="move_base" respawn="false" name="move_base"
output="screen">
       <param name="base_local_planner"
value="teb_local_planner/TebLocalPlannerROS"/>

       <rosparam file="$(find
ros_multi_tb3)/newparams/costmap_common_params.yaml" command="load"
ns="global_costmap" />
       <rosparam file="$(find
ros_multi_tb3)/newparams/costmap_common_params.yaml" command="load"
ns="local_costmap" />
       <rosparam file="$(find
ros_multi_tb3)/newparams/local_costmap_params.yaml" command="load" />
       <rosparam file="$(find
ros_multi_tb3)/newparams/global_costmap_params.yaml" command="load" />

       <rosparam file="$(find ros_multi_tb3)/newparams/move_base_params.yaml"
command="load" />

       <rosparam file="$(find ros_multi_tb3)/newparams/base_global_planner.yaml"
command="load" />
       <rosparam file="$(find
ros_multi_tb3)/newparams/base_local_planner_teb.yaml" command="load" />

       <remap from="cmd_vel" to="$(arg cmd_vel_topic)"/>
       <remap from="odom" to="$(arg odom_topic)"/>
   </node>
</launch>
```

Listing 38. Move_base launch file

4.5. The web user interface

The web user interface is used to monitor tasks, robots and location. The web is built with NextJS and uses MetaMask to connect with the private blockchain network. The web app uses functions

provided by the smart contract to query current data of tasks and robots and uses the connection to Firebase to get data on locations and inventory. There is a function to add tasks in the web app, but it runs relatively slow because the transactions must go through MetaMask.

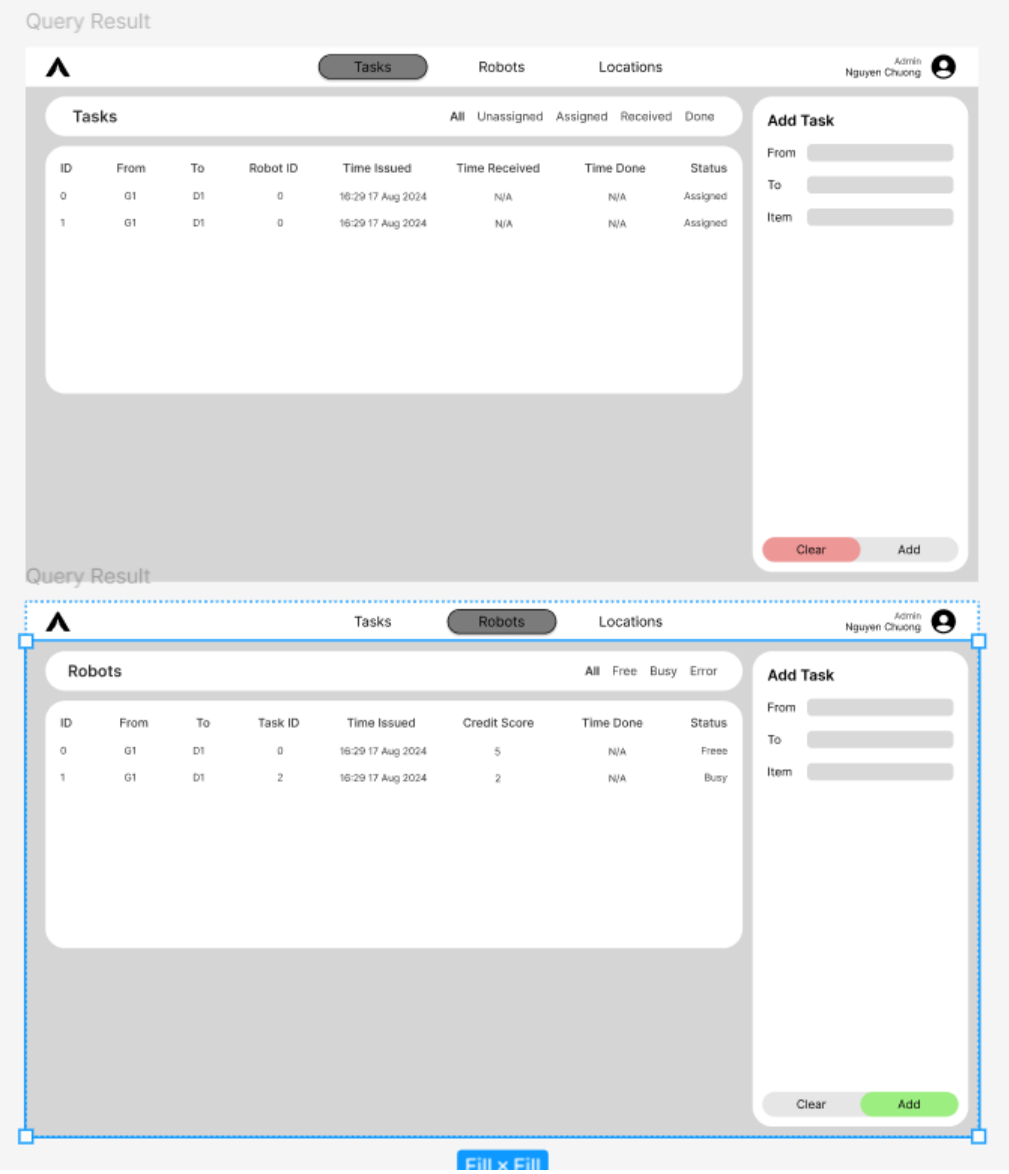

Figure 4. Figma Design of UI web

5. Result

5.1. Setup and Environment

The test run is carried out in the university lab room with three Turtlebots and four carton boxes as a simulation of pick-up and delivery locations. The carton boxes also improve the robots' mapping, localization, and navigation, considering that the testing field was originally big and empty.

The blockchain setup is done with the deployment of a private blockchain network with Kurtosis and Docker, and then a smart contract is deployed, and some tasks are added by messaging the deployed smart contract. The robot setup begins with one robot running alone to draw a map of the testing field. That map is later used by all robots. Then, each robot is initiated with the navigation package, the navigator, and the blockchain client. Right after the blockchain client of a robot runs, that robot starts the task.

Figure 5. Experiment environment

5.2. The process

The three robots run independently of each other in parallel; they first navigate to their pickup point. They wait there for seconds to simulate the behaviour of goods loading and unloading, as well as exchange information with the blockchain network. Afterwards, the robots move to the delivery points and wait there again for some seconds before moving away from the points to avoid potential collisions. The robots move slowly and conservatively due to the tunning of the navigation parameters algorithm to prevent collisions on the way.

The most common problem is the one happening with the navigation package that causes the robot to be unable to perform localization and, therefore, cannot plan a path. In this case, or any other errors that cause the robot not to respond after 180 seconds, the task assigned to that robot is marked timeout, and the robot can move to the next task. After marking five consecutive tasks as timeouts, the robot ran out of credit and could not take any task. These errors often happen in the first tasks of the robot and are usually caused by mistakes in setting up the robot. They can be avoided with careful setup. Therefore, in some of the later tests, the situation of five consecutive errors only appears if they are created intentionally to provide a situation to test the error handling in the system.

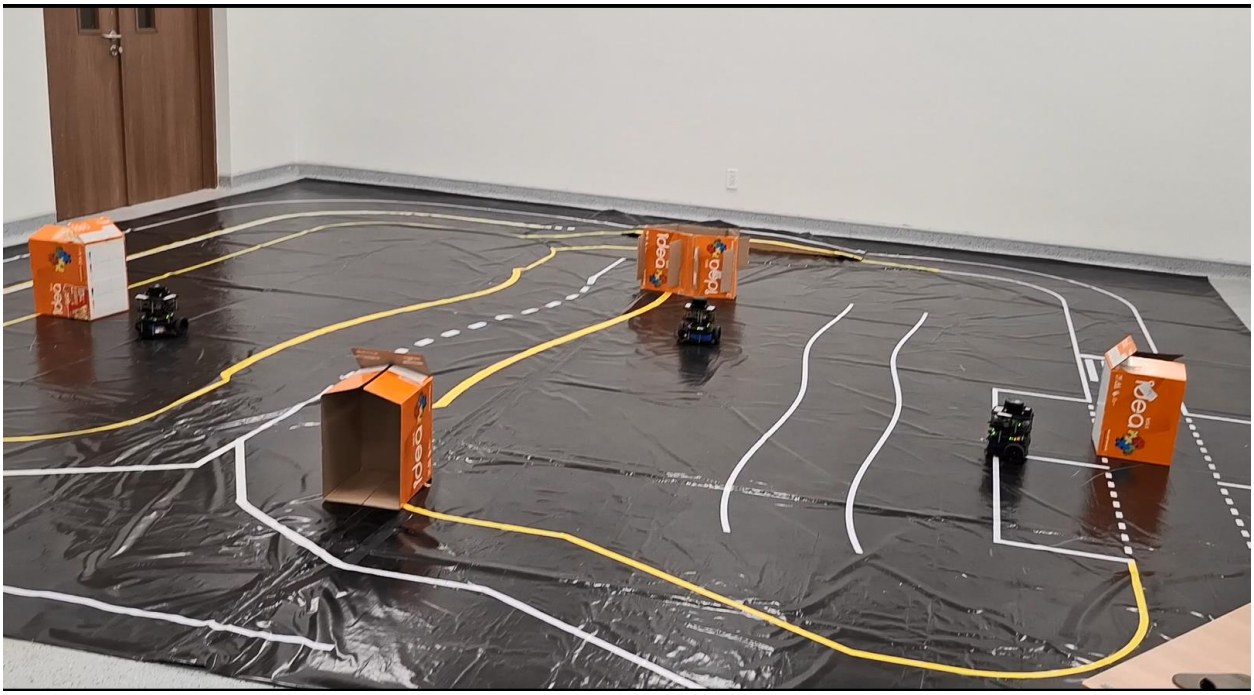

Figure 6. Robots at pickup points

5.3. Performance

Firstly, a comparison between the performances of a single robot and a swarm of three robots is made. The average time for a single robot to complete three tasks in a row is 233 seconds, with an average of 78 seconds for each task. Meanwhile, the three robots swarm complete three tasks in parallel on an average of 77.5 seconds with a standard deviation of 4.034 (seconds). This means the swarm runs 3.02 times faster than one robot system. This result of 3.006 logically may show a little bias caused by the difference in the start point of the robots and the tasks' positions because the number of tests is small (ten times for each). As the number of tests increases, the ratio should fluctuate closer to 3 as the three robot systems in the three task tests are simply three single robots running one task each. However, when the number of tasks increases, the average time for each task of the single robot remains stable, while the average time of a set of three tasks for the swarm increases by 9 seconds. This is because after submitting delivery time to the smart contract, the robots move away from the position, and this action takes approximately 8 seconds for each robot (this number is measured by an external timer). This collision avoidance action is indeed not needed for a single robot system. So, in the long run, the speed ratio between the three robots swarm and the single robot is approximately 2.693.

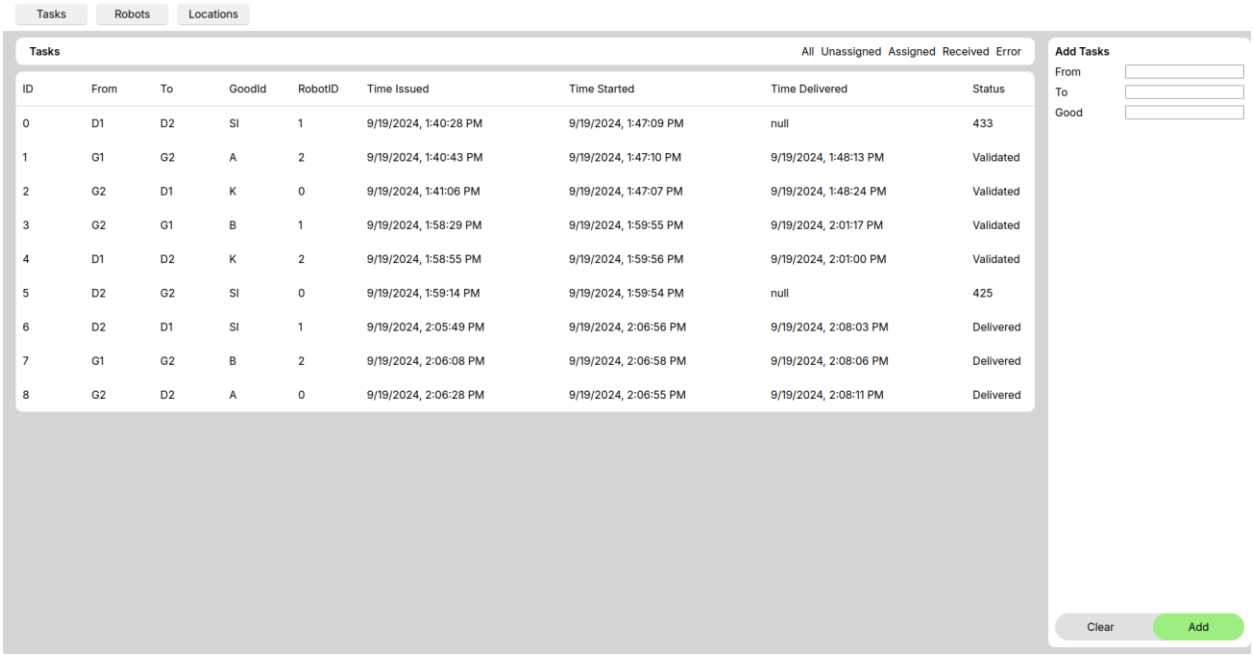

Figure 7. UI Tasks Monitor shows the result of one 9-task experiment

Secondly, the success rate of tasks is discussed. The action of adding new tasks to the network has a success rate remaining 100% until the time of this writing. The same success rate is shown when assigning tasks to eligible robots and when receiving and starting the assigned tasks. However, the rate of the tasks that are started to be completed is only guaranteed at more than 75%. The main reasons for failed tasks are timeout (~80%) and exceeding the path planning retry limit (~20%). The root of those errors comes from the physical interaction of robots, which is largely unpredictable, even though a lot of effort is made to prevent runtime interferences between robots.

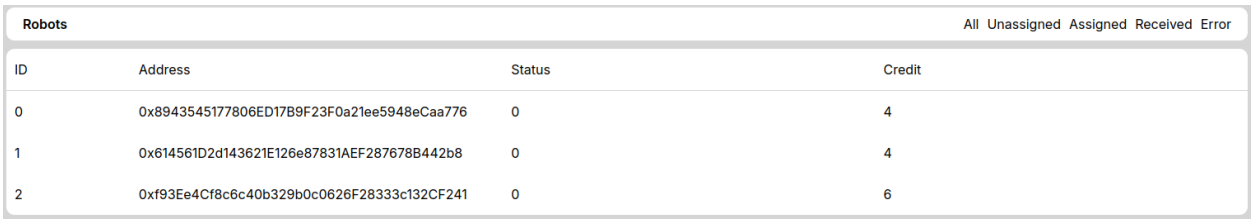

Figure 8. UI Robot monitor shows status of robots after the 9 tasks

ID	X	Y	Goods
D1	0.2	0	SI,
D2	1	1.5	K,A,
G1	1	-2	
G2	2	-0.5	B,
T1	4	3	LIVE,

Figure 9. UI Inventory monitor shows status of inventories after the 9 tasks

5.4. Validation Performance

The validation of complete tasks and the accounting of robots' credit witness great accuracy with a zero percent of false positives, which means no false completion report is accepted, and the robot responsible for the false information is penalized with the correct credit point. False negatives, on the other hand, have never happened, though it is logically possible. This is a logical failure in the verification system that allows false negatives. The issue is found late in the development, so even if the solution is known, the limited time prevents its implementation.

The scenario that allows false negatives is as follows:

Robot 1 delivers goods A to G1. Then, before any other robots reach G1, robot number 1 is assigned with the task of delivering goods A from G1 to D1. It comes to G1 and reports the inventory to the smart contract, but the smart contract does not allow robot number 1 to validate its own task. Afterwards, Robot 1 still picks up the goods A and brings it to D1. After that, Robot 2 comes to G1 and reports that good A is nowhere to be found in the inventory. This leads to the smart contract accusing Robot 1 of reporting false information and deducting its credit.

The solution to this could be to prevent a robot from being assigned to a task involving goods from its previous non-verified tasks. This algorithm could be implemented in the smart contract.

6. Discussion

The project marks the viability of the application of a smart contract as a task allocator, manager, and validator in swarm robotics for supporting warehouse goods movement and other purposes. And further research into this approach has certain potential.

The application of smart contracts as a way for robots to synthesize the global situation is a simple, safe, and accurate protocol, as it has some similarities in high-level logic with centralized designs. The similarity lies in the way the blockchain client sends messages and then gets data-driven instructions from a smart contract instead of a central server. This high-level implication significantly simplifies the development process.

The findings show that the processing and communication of the blockchain client and the smart contract can reach an absolute rate of success with details. This ensures the errors of tasks mostly lie in the function of robots themselves. Therefore, when these errors arise, the blockchain system has an absolute rate of success in catching, recording and resolving them. This implies that in terms of global information summary and decision-making, the smart contract can play the equivalent role of a central server in a decentralized system. Therefore, smart contracts' potential to affect the detailed behaviours of individual robots can be exploited further to optimize swarm movement and performance in a global context.

The credit system in this project is to simulate an equal society, where each individual has the authority to judge the completion of others. Although a specialized test is not set up to test this system's resilience against different types of intentional attacks, the credit system works fine during the test of performance to rule out malfunctioning robots. Compared to the system proposed by Grey et al. (2020), despite a different environment and task settings, this system theoretically shows a potential improvement in decentralization and objectivity because the robot to validate a task is the first one to reach the completion location of that task. The arrival of such a robot is not intentionally to validate any task; that robot is reaching such location as a part of its own task. This implies the validator for any task based on the task allocation algorithm, the order of the task, the location of robots at a certain time, and other objective factors. The combination of those factors makes the choice of the validators predominantly random and unpredictable in the long run. Moreover, the validation method is that the robot reports the inventory in the location without the knowledge of the tasks involved or the assignees. Therefore, bias in this system is hard to happen as the validators do not know which task they are validating, and the smart contract makes the final decision. However, this system needs more research to ensure the witnesses of inventories are correct and penalize dishonest validators for false reports.

Limitations and future research

Speed is a natural weakness of any blockchain system since an exchange between blockchain clients and the smart contract takes time to process. It takes approximately 3 seconds for a transaction to be committed, while information queries are much faster. Even though it is acceptable, as not many interactions of that kind are made during the process, it is not known if the processing time increases dramatically if the swarm is expanded. This leaves the scalability level of this design uncertain.

The simulation is not completely decentralized, as the Turtlebot used in the simulation needs more memory to install and run a blockchain node on it. Even though the simulation uses Docker containers and fully separated programs run on different cores to reach a virtual decentralization, a complete physical decentralization system should be tested in the future with a more powerful processor and a larger SSD and RAM storage Raspberry 5 for the installation of a blockchain client and node on Turtlebots.

The system design relies on campus wi-fi as a communication medium. That is not optimal because the availability of the system relies on the availability of the wi-fi. Different decentralized means of communication should be developed. Installing a wireless access point on each robot to generate local networks can be a solution. This issue was mentioned in Pacheco's research in 2020, and his approach to this is worth referring to.

There is a large space to improve the smart contract algorithm. The performance of the system can be boosted by assigning tasks to the nearest robot to the pickup location. The smart contract can be programmed to calculate the capture status of each location to avoid collisions there. Moreover, the failed tasks can be reassigned to other robots a few times before being considered undoable.

Regarding the consensus algorithm, the application of the proof-of-authority technique can be considered and compared with proof-of-stake in terms of performance and security in this approach. Proof-of-authority was implemented in Pacheco's paper in 2020. However, the technique is applied in a system different from this project.

7. Conclusion

The purpose of this project is to explore the application of Ethereum smart contracts for allocating, managing and validating tasks in a decentralized swarm robotics system, focusing on warehouse goods movement. By exploiting the power of smart contracts, the project investigated the feasibility of using blockchain to enhance security, reliability, and autonomy in a robotic swarm.

The results illustrate the smart contract's efficiency in dealing with task allocation and validation, achieving a nearly flawless success rate in robot communication and error tracking. Despite these successes, limitations in the speed of blockchain transactions and the imperfect decentralization in the current simulation show chances for further improvements.

The project provides useful insights for future research into the use of Ethereum smart contracts in swarm robotics in practical fields such as warehouse automation or logistics. The integration of blockchain and smart contracts into swarm robotics can ensure accuracy, security, global synthesis, and trust within the robot swarm.

However, the system's dependency on external infrastructure, such as Wi-Fi, and the long time to process blockchain transactions indicate room for improvement. Future reviews of this work may focus on completely decentralized communication methods and improving the speed of blockchain interactions. In this way, the system can not only inherit fully the advantageous characteristics of blockchain technology but also cover the natural weakness of this technology.

This project reveals the potential of the integration between blockchain, especially Ethereum smart contract, into a robot swarm. A successful combination of those technologies has the potential to create a scalable swarm of independent robots with fast, secure communication, finesse global synthesis and decision-making, and a trustworthy credit system. That swarm system could be a game-changer technology in many industries.

References

Arnold, R., Carey, K., Abruzzo, B., & Korpela, C. (2019). What is A Robot Swarm: A Definition for Swarming Robotics. *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 0074–0081. https://doi.org/10.1109/UEMCON47517.2019.8993024

Barca, J., & Sekercioglu, A. (2013). Swarm robotics reviewed. *Robotica*, *31*. https://doi.org/10.1017/S026357471200032X

Castelló Ferrer, E. (2019). The Blockchain: A New Framework for Robotic Swarm Systems. In K. Arai, R. Bhatia, & S. Kapoor (Eds.), *Proceedings of the Future Technologies Conference (FTC) 2018* (pp. 1037–1058). Springer International Publishing. https://doi.org/10.1007/978-3-030-02683-7_77

Ethereum Whitepaper. (n.d.). ethereum.org. Retrieved September 26, 2024, from https://ethereum.org/vi/whitepaper/

Grey, J., Godage, I., & Seneviratne, O. (2020). Swarm Contracts: Smart Contracts in Robotic Swarms with Varying Agent Behavior. *2020 IEEE International Conference on Blockchain (Blockchain)*, 265–272. https://doi.org/10.1109/Blockchain50366.2020.00040

Higgins, F., Tomlinson, A., & Martin, K. M. (2009). Survey on Security Challenges for Swarm Robotics. *2009 Fifth International Conference on Autonomic and Autonomous Systems*, 307–312. https://doi.org/10.1109/ICAS.2009.62

Mallikarachchi, S., Dai, C., Seneviratne, O., & Godage, I. (2022). Managing Collaborative Tasks within Heterogeneous Robotic Swarms using Swarm Contracts. *2022 IEEE International*

Conference on Decentralized Applications and Infrastructures (DAPPS), 48–55. https://doi.org/10.1109/DAPPS55202.2022.00014

Pacheco, A., Strobel, V., & Dorigo, M. (2020). *A Blockchain-Controlled Physical Robot Swarm Communicating via an Ad-Hoc Network* (pp. 3–15). https://doi.org/10.1007/978-3-030-60376-2_1

Pilkington, M. (2016). *Blockchain Technology: Principles and Applications*.

Strobel, V., Castelló Ferrer, E., & Dorigo, M. (2020). Blockchain Technology Secures Robot Swarms: A Comparison of Consensus Protocols and Their Resilience to Byzantine Robots. *Frontiers in Robotics and AI*, *7*. https://doi.org/10.3389/frobt.2020.00054

Wood, D. G. (n.d.). *ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER*.

Zheng, Z., Xie, S., Dai, H., Chen, X., & Wang, H. (2017). *An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends*. 557–564. https://doi.org/10.1109/BigDataCongress.2017.85

Appendices

Demo Video: https://youtu.be/QLP8PCj19-Q

Full source code, network configuration, navigation parameters, operating commands, deployment instructions are detailed in the repository.

GitHub public repository: https://github.com/DyzilestarHorches/thesisblockchainrobots

Error code

Format: ‘4 s c’, in which:

- 4 indicates an error,
- ‘s’ shows the stage where the error occurred,
- ‘c’ shows the cause of the error.

Code of stage (s)	Meaning
0	The stage where the error happened is unknown.
1	The error happened before the task started.
2	The error happened after the task started but before the robot could reach the pickup point.
3	The error happened after the robot received goods

Table I. Error code – code of stage

Code of causes (c)	Meaning
0	The cause of the error is unknown.
1	The error is caused by problem with the robot or the navigation part.
2	The error is caused by the navigator.
3	The error is caused by the blockchain client
4	The error is caused by the off-chain cloud server.
5	The error is caused by exceeding the limited time (timeout)

Figure II. Error code – code of causes