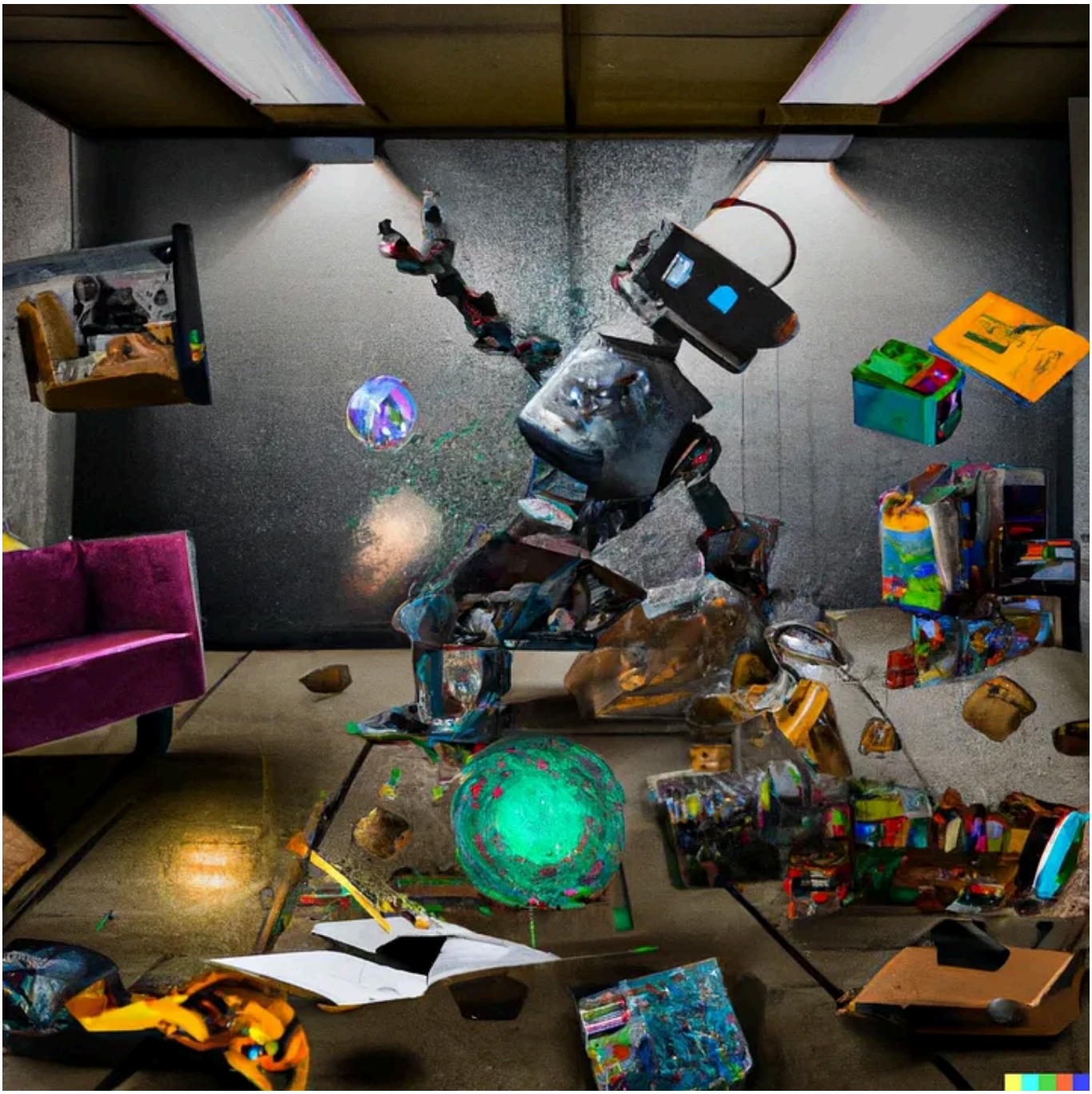
✨ Member-only story

Behavior Trees for ROS2 — Part 1: Unlocking Advanced Robotic Decision-Making and Control

Jegathesan Shanmugam · Follow

10 min read · May 14, 2023



In the complex and fascinating world of robotics, **Behavior Trees (BTs)** have emerged as an integral part of decision-making processes. They provide a **structured**, **modular**, and **efficient** approach to programming a robot's behavior. With roots in the video game industry where they were used to control non-player characters, BTs have found a home in robotics, where they excel at managing a myriad of tasks and conditions.

Robotic software like **Nav2**. used for robot navigation. and **MoveIt**. a

In this post, we're going to explore Behavior Trees to understand them thoroughly. We'll begin by breaking down the basic parts of Behavior Trees and gaining a clear understanding of how they work. Then, we'll step into the practical realm, writing code to create and use Behavior Trees, independent of any specific platform or framework.

But we won't stop there. Once we've mastered the basics, we'll take it a step further by integrating our Behavior Trees with ROS2 (Robot Operating System 2), a flexible framework for writing robot software. This will demonstrate how Behavior Trees can be utilized in a real-world robotics context, enhancing the decision-making capabilities of our autonomous systems.

Think about a behavior tree (BT) as a decision-making map for a video game character, a robot, or any other kind of autonomous agent that needs to decide what to do next.

Imagine that you're directing a play, but your actors are robots. As the director, you need a way to guide your actors, telling them what to do and when to do it. *A Behavior Tree (BT) is like your script*, detailing every move your robotic actors need to make. Just like in a real play, the show (or ***task***) begins when the director calls "action" (or, in this case, "***tick***").

So, what's a "tick"? In the context of a BT, a tick is a signal from the root node (or starting point) of the tree to its children nodes, urging them to perform their tasks. ***Nodes only "act" when they receive a tick.***

Now, the beauty of this system is in its feedback mechanism. Once a node starts performing its task, it communicates back to the director (its parent node) about its progress, using one of three statuses: **Running**, **Success**, or **Failure**.

Running: This status is a bit like saying, "I'm on it, but I'm not done yet."
Success: This is the equivalent of, "I've done my part, and it went well."
Failure: This is like saying, "I tried, but I couldn't do it."

Based on these statuses, the director decides where to send the next tick. It's a cycle that continues until the task is completed or no longer valid.

The Behavior Tree is composed of two main types of nodes: ***control flow nodes*** and ***execution nodes***. Let's dive into them a bit more.

Control Flow Nodes are like the director's assistants, who help make decisions about what the actors should do next. There are four main types:

Sequence (→) Node: This is like a strict assistant who demands that each task must be done perfectly, in order, before moving on to the next. If any task fails, the Sequence stops and reports failure.

Fallback (?) Node (also known as Selector): This is a more flexible assistant who has a list of tasks and tries them one by one until one succeeds. If all tasks fail, then the Fallback reports failure.

Parallel (⇒)Node: This is an assistant who can multitask, directing several actors to perform their tasks simultaneously. The Parallel node defines its success based on a set number of tasks that must succeed.

Decorator Node: This assistant is a bit special, as it deals with only one actor but can modify the actor's behavior in various ways. There are different kinds of Decorators, each providing a unique twist to the task.

Execution Nodes are the actors, the ones that perform the tasks (Action) or evaluate certain conditions (Condition).

Action Node: When this actor receives a tick, it starts its task, keeps the director updated with a "Running" status, and finally reports either "Success" or "Failure".

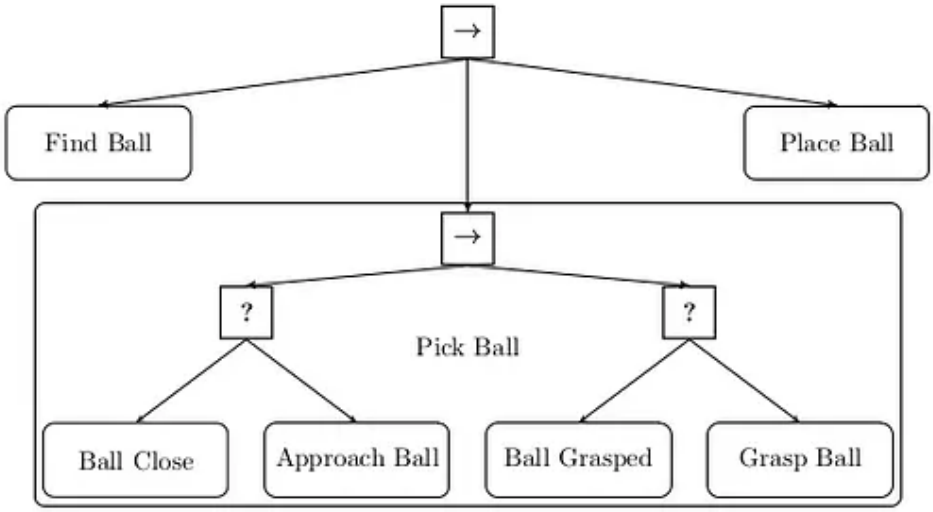
Condition Node: This actor is more of a spot-checker. It evaluates whether certain conditions are met and immediately reports back "Success" if the condition is true or "Failure" if it's false.

Now that you know the basics, you're ready to direct your robotic play using Behavior Trees. Remember, this framework is designed for flexibility and adaptability, allowing your robotic actors to respond swiftly to changes and perform complex, nuanced tasks.

Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◊	Custom	Custom	Custom

The node types of a BT

let's consider a behavior tree in the context of a robotic arm that can pick up a ball and place it at the goal position.



A high level BT carrying out a task consisting of first finding, then picking and finally placing a ball

1. Your BT begins with a Sequence node as the main controller. This node has two primary tasks for the robots: “Find a Ball” and “Place a Ball”.
2. The first task, “Find a Ball”, involves a single robot, which is assigned to locate a ball.
3. The second task, “Place a Ball”, is more complex. This task is managed by another Sequence node, which ensures that a series of actions are performed in order. This sequence involves the task “Pick a Ball”.
4. The “Pick a Ball” task is managed by two Fallback nodes. Each of these nodes has a different approach to guide the robot.

- The first Fallback node ensures that the robot checks if the “Ball is Close” and, if so, directs the robot to “Approach the Ball”.
- The second Fallback node is responsible for checking if the “Ball is Grasped” and, if not, directs the robot to “Grasp the Ball”.

Remember, in a BT, the nodes only act when they receive a signal, or “tick”. Once a task starts, the robot communicates its status back to its parent node. The cycle continues until all tasks are completed or cannot be completed.

So far, we've discussed the concepts and mechanisms of Behavior Trees at length. Now, it's time to put those ideas into action.

Before we delve into the actual coding, we need to set up the Behavior Tree package. In the following sections, we'll walk you through the process of setting up this package, after which we'll explore the code that brings our Behavior Tree to life.

To set up and run our first BehaviorTree.CPP example on Ubuntu 20.04, follow these steps:

```
# Install the required dependencies:
# Open a terminal and run the following commands to install the necessary
# dependencies:
sudo apt-get update
sudo apt-get install -y libzmq3-dev libboost-dev libboost-system-dev libboost-fi

# Build and install BehaviorTree.CPP:
# Clone the BehaviorTree.CPP repository, build, and install the library:
git clone https://github.com/BehaviorTree/BehaviorTree.CPP.git
cd BehaviorTree.CPP
mkdir build
cd build
cmake ..
make -j$(nproc)
sudo make install
```

Lets start with simple example using BehaviorTree.CPP will help you understand the fundamentals and get familiar with the syntax.

```
mkdir ~/example1
cd ~/example1
```

```
touch main.cpp && touch CMakeLists.txt
```

```
// main.cpp

#include <behaviortree_cpp/bt_factory.h>
#include <behaviortree_cpp/behavior_tree.h>

using namespace BT;

class SaySomething : public SyncActionNode
{
public:
    SaySomething(const std::string& name, const NodeConfiguration& config)
        : SyncActionNode(name, config)
    {
    }

    static PortsList providedPorts()
    {
        return { InputPort<std::string>("message") };
    }

    NodeStatus tick() override
    {
        std::string message;
        getInput("message", message);
        std::cout << "Robot says: " << message << std::endl;
        return NodeStatus::SUCCESS;
    }
};

static const char* xml_text = R"(
<root>
  <BehaviorTree>
    <SaySomething message="Hello, World!" />
  </BehaviorTree>
</root>
)";

int main()
{
    BehaviorTreeFactory factory;
    factory.registerNodeType<SaySomething>("SaySomething");

    auto tree = factory.createTreeFromText(xml_text);

    tree.tickWhileRunning();
```

```
    return 0;
}
```

```
# CMakeLists.txt
cmake_minimum_required(VERSION 3.5)
project(bt_example)

set(CMAKE_CXX_STANDARD 17)

find_package(behaviortree_cpp REQUIRED)

add_executable(bt_example main.cpp)
target_link_libraries(bt_example BT::behaviortree_cpp)
```

```
# Now, build and run the example:
mkdir build
cd build
cmake ..
make
./bt_example
```

Fantastic, you've got your first BehaviorTree.CPP code snippet! This script represents a basic Behavior Tree where the robot (your actor) has a simple task: to say something. Let's break it down:

The `SaySomething` class extends the `SyncActionNode` class, which is one of the types of execution nodes provided by BehaviorTree.CPP. The `SyncActionNode` is a synchronous action node, which means it does not return "running" status, but only "success" or "failure". In our case, it will always return "success".

The `tick()` method is the main method that gets called when this node is ticked. Here, it retrieves the input message and outputs it to the console. Since it successfully performs its task each time, it returns `NodeStatus::SUCCESS`.

The XML text defined as `xml_text` represents the structure of your Behavior Tree. It contains a single node, `SaySomething`, with the message "Hello, World!".

Finally, the `main` function creates a `BehaviorTreeFactory`, registers the `SaySomething` node type, creates the tree from the XML text, and ticks the tree while it's running. This makes the robot say "Hello, World!".

Now, armed with the basics, you are ready to move towards a more complex Behavior Tree such as the pick and place tree. In the pick and place tree, you'll introduce more control flow nodes and execution nodes, interweaving them to achieve the desired behavior. The complete code for this example are available here.

```
#include <iostream>

#include <behaviortree_cpp/bt_factory.h>
#include <behaviortree_cpp/behavior_tree.h>

using namespace BT;

// Define FindBall action node
class FindBall : public BT::SyncActionNode
{
  public:
    FindBall(const std::string& name) : BT::SyncActionNode(name, {})
    {
    }

    // Define the tick() function for the FindBall node
```

```
    NodeStatus tick() override
    {
        std::cout << "[⚓ FindBall ] => \t" << this->name() << std::endl;
        return BT::NodeStatus::SUCCESS;
    }
};

// Define PickBall action node
class PickBall : public BT::SyncActionNode
{
  public:
    PickBall(const std::string& name) : BT::SyncActionNode(name, {})
    {
    }

    // Define the tick() function for the PickBall node
    NodeStatus tick() override
    {
        std::cout << "[⚓ PickBall ] => \t" << this->name() << std::endl;
        return BT::NodeStatus::SUCCESS;
    }
};

// Define PlaceBall action node
class PlaceBall : public BT::SyncActionNode
{
  public:
    PlaceBall(const std::string& name) : BT::SyncActionNode(name, {})
    {
    }

    // Define the tick() function for the PlaceBall node
    NodeStatus tick() override
    {
        std::cout << "[⚓ PlaceBall ] => \t" << this->name() << std::endl;
        return BT::NodeStatus::SUCCESS;
    }
};

// Define GripperInterface class for interacting with the gripper
class GripperInterface
{
  private:
    bool _opened;

  public:
    GripperInterface() : _opened(true)
    {
    }
```

```
    NodeStatus open()
    {
        _opened = true;
        std::cout << "GripperInterface::open" << std::endl;
        return BT::NodeStatus::SUCCESS;
    }

    NodeStatus close()
    {
        std::cout << "GripperInterface::close" << std::endl;
        _opened = false;
        return BT::NodeStatus::SUCCESS;
    }
};

// Define BallClose condition function
BT::NodeStatus BallClose()
{
    std::cout << "[ Close to ball: NO ]" << std::endl;
    return BT::NodeStatus::FAILURE;
}

// Define BallGrasped condition function
BT::NodeStatus BallGrasped()
{
    std::cout << "[ Grasped: NO ]" << std::endl;
    return BT::NodeStatus::FAILURE;
}

// Define the behavior tree structure in XML format
static const char* xml_text = R"(
 <root main_tree_to_execute = "MainTree" >

     <BehaviorTree ID="MainTree">
        <Sequence name="root_sequence">
            <FindBall   name="found_ok"/>
                <Sequence>
                    <Fallback>
                        <BallClose   name="no_ball"/>
                        <PickBall    name="approach_ball"/>
                    </Fallback>
                    <Fallback>
                        <BallGrasped   name="no_grasp"/>
                        <GraspBall  name="grasp_ball"/>
                    </Fallback>
                </Sequence>
            <PlaceBall   name="ball_placed"/>
        </Sequence>
     </BehaviorTree>
 </root>
```

```
 )";

int main()
{
    BehaviorTreeFactory factory;

    // Register custom action nodes with the factory
    factory.registerNodeType<FindBall>("FindBall");
    factory.registerNodeType<PickBall>("PickBall");
    factory.registerNodeType<PlaceBall>("PlaceBall");

    // Register custom condition nodes with the factory
    factory.registerSimpleCondition("BallClose", std::bind(BallClose));
    factory.registerSimpleCondition("BallGrasped", std::bind(BallGrasped));

    // Create an instance of GripperInterface
    GripperInterface gripper;

    // Register a simple action node using the GripperInterface instance
    factory.registerSimpleAction("GraspBall", std::bind(&GripperInterface::close

    // Create the behavior tree using the XML description
    auto tree = factory.createTreeFromText(xml_text);

    // Run the behavior tree until it finishes
    tree.tickWhileRunning();

    return 0;
}
```

```
# Output
[⚓ FindBall ] =>       found_ok
[ Close to ball: NO ]
[⚓ PickBall ] =>       approach_ball
[ Grasped: NO ]
GripperInterface::close
[⚓ PlaceBall ] =>      cube_placed
```

Great, you've created a more complex Behavior Tree! This script represents a pick-and-place operation by a robot, composed of finding, picking, and placing a ball. Let's dissect it:

1. You've created three action nodes: `FindBall`, `PickBall`, and `PlaceBall`. Each of these classes extends `SyncActionNode` and implements the `tick()` function, which logs the action's name and returns `NodeStatus::SUCCESS`.
2. You've defined a `GripperInterface` class, which represents the robot's gripper. It can open and close the gripper, both operations returning `NodeStatus::SUCCESS`.
3. You've defined two condition functions: `BallClose` and `BallGrasped`. These functions represent checks the robot needs to make before picking up the ball. They currently always return `NodeStatus::FAILURE`, simulating scenarios where the ball is not close and the robot hasn't grasped the ball yet.
4. The XML string `xml_text` defines your Behavior Tree. It starts with a sequence node, which first tries to find the ball (`FindBall`). Once the ball is found, it tries to pick the ball, which is another sequence node composed of two fallback nodes. The first fallback node checks if the ball is close (`BallClose`), and if not, the robot approaches the ball (`PickBall`). The second fallback node checks if the ball is grasped (`BallGrasped`), and if not, the robot grasps the ball (`GraspBall`). Once the ball is picked up, the robot places the ball (`PlaceBall`).
5. Finally, the `main` function registers the custom action and condition nodes, creates an instance of the `GripperInterface`, registers the `GraspBall` action using the `GripperInterface`, creates the tree from the XML text, and ticks the tree while it's running.

This code provides a practical demonstration of how a robot can use a Behavior Tree to perform a sequence of tasks with checks along the way. However, note that the current conditions always return `NodeStatus::FAILURE`. To make this example more realistic, you might want to

update these conditions based on the actual state of the robot and its environment.

That's a wrap on this post! You've learned about Behavior Trees, their structure and components, and even implemented a complex tree using a pick-and-place example.

Stay tuned for our next post where we'll take things up a notch. We will introduce ROS2 (Robot Operating System 2) integration with Behavior Trees, opening up even more possibilities for creating complex, robust, and adaptive behaviors for your robotic applications. See you in the next post!

I hope this information was helpful and that you found it useful. If you have any questions or comments, please don't hesitate to let me know. I appreciate your feedback and are happy to help answer any questions you may have. Thank you for taking the time to read this.

I always share interesting articles and updates on LinkedIn, so if you want to stay informed, feel free to follow me on the platform.

Reference:

1. Behavior Tree official Documentation
2. Behavior Trees in Robotics and AI: An Introduction
3. Nav2 Behavior Trees

Robotics Cpp Behavior Trees Ros2 Robotics Automation

Written by Jegathesan Shanmugam

Follow

277 Followers

Talks about #robotics, #computervision, and #embeddedsystems

More from Jegathesan Shanmugam

Jegathesan Shanmugam

Docker and QEMU: A Powerful Combination for Accelerating Edg...

Edge computing development can be slow, especially when it comes to compiling large...

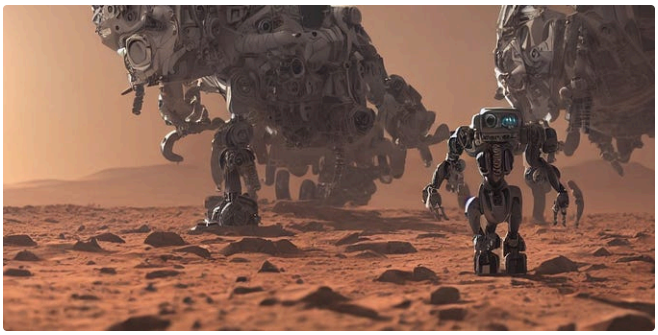
Jan 15, 2023 23 1

Jegathesan Shanmugam

ROS2 from the Ground Up: Part 5- Concurrency, Executors and...

We will explore some of the key concepts and interfaces related to concurrency in ROS2,...

Dec 27, 2022 40 1

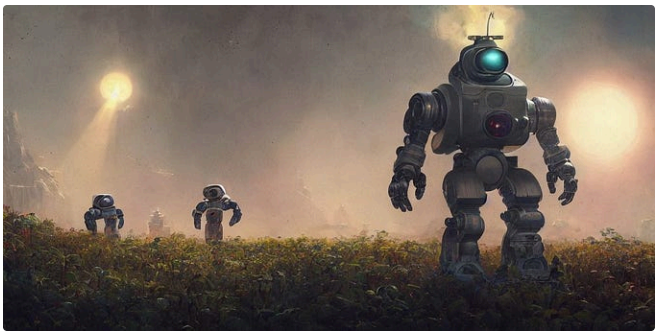


Jegathesan Shanmugam

ROS2 from the Ground Up: Part 1— An Introduction to the Robot...

ROS2, the robot's friend, A platform on which they depend, For communication,...

Dec 22, 2022 189 3



Jegathesan Shanmugam

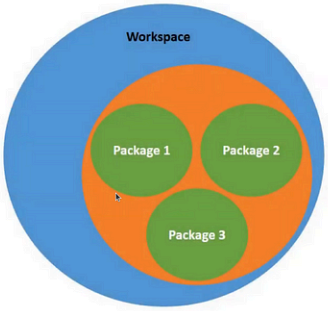
ROS2 from the Ground Up: Part 6- DDS in ROS2 for Reliable Robotics...

Modern robotics systems are complex due to the need to integrate various types of...

Dec 29, 2022 53

See all from Jegathesan Shanmugam

Recommended from Medium



Ghulam Hassan

Kubernetes is an open-source container orchestration

In this publication I have to share some important factors in distribution. As well as...

Sep 22 1K 10

Sagar Kumar in Spinor

Getting Started with ROS2: Overview of ROS2 Workspaces,...

Part 4 of our “Getting Started with ROS2” Series

Jul 2 10

Lists

Staff Picks

742 stories · 1337 saves

Stories to Help You Level-Up at Work

19 stories · 816 saves

Self-Improvement 101

20 stories · 2813 saves

Productivity 101

20 stories · 2399 saves

Michael McGuire in Godot Dev Digest

10 Proven GDScript Optimization Tips for Faster Game Performance

Top 10 GDScript Tips for Performance

Jun 15 5

George

Introduction to Data Structures: Lists, Trees, and Graphs

Sep 24 1.6K 46

Jason Bowling in Exploring ROS Robotics

How To Add A Motor Controller To Your ROS Robot

Give your robot straight, precise driving and odometry!

Sep 24, 2022 403

Saad Dogar

The Titans Clash: A Mythic Showdown in the Aeons of Culture

The term “Clash of the Titans” evokes images of battles of titans and great epochal clashes...

2d ago 1.3K 25

See more recommendations