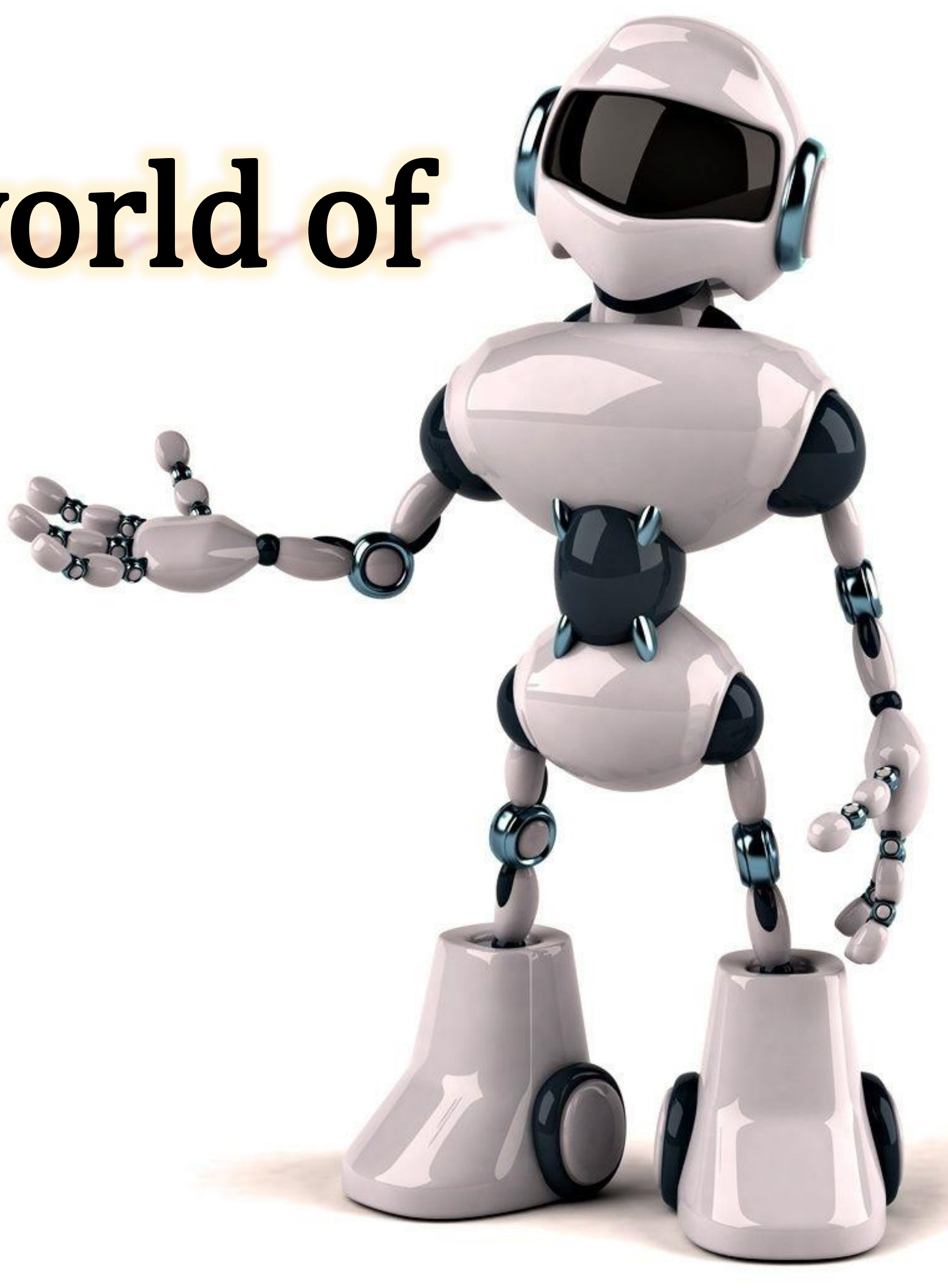
Stepping into the world of Robotics

Kanishkan M S
Phani Jayanth J
Sharanesh R

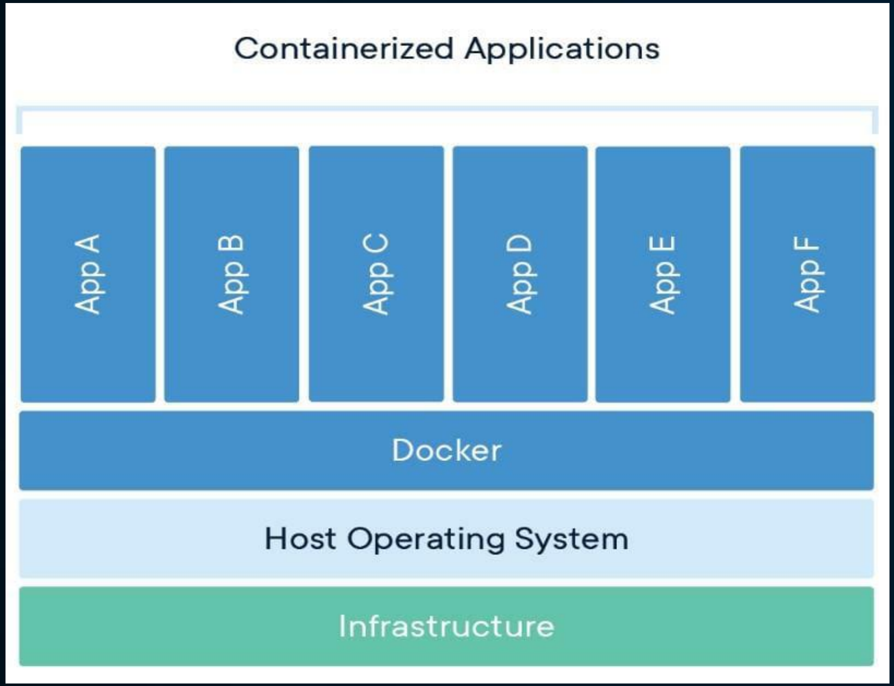


What is ROS?

- ROS stands for Robot Operating System
- As the name says, it’s actually not a real operating system since it runs on top of UNIX based systems (like Ubuntu)
- It’s framework allows abstraction of hardware from software – which implies that you can create robotic applications without having to deal with actual hardware!
- ROS is open-source (free to use) and comes with various tools to develop robotic systems

Docker

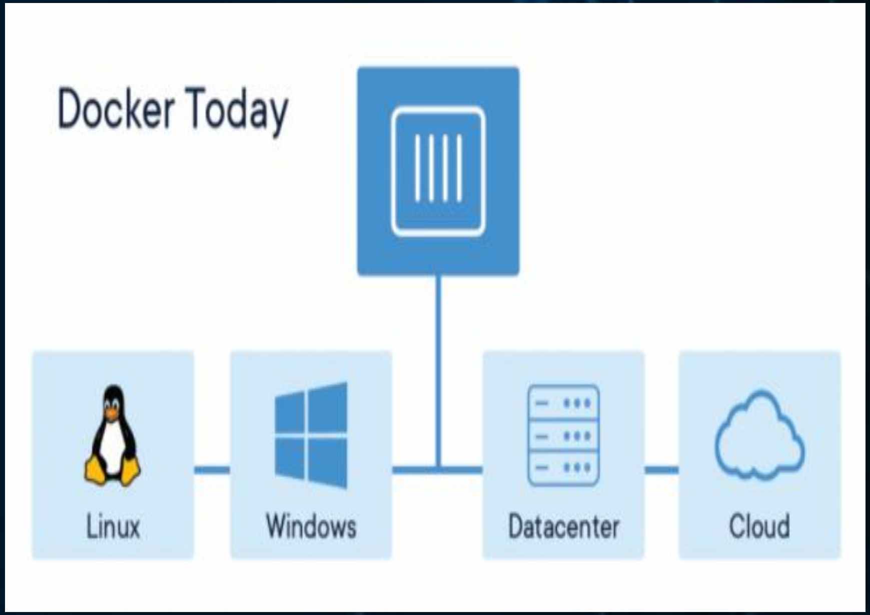
- What is Docker?
  - Docker is a software platform for building applications based on *containers* – small and lightweight execution environments that make shared use of operating system kernel, but otherwise run in isolation from one another.
  - Originally built for Linux, Docker now runs on Windows and MacOS as well.



- ***Containers***, by contrast, isolate applications' execution environments from one another, but share the underlying OS kernel.
- Each Docker container starts with a ***Dockerfile***. A Dockerfile is a text file written in an easy-to-understand syntax that includes the instructions to build a Docker ***image*** - a portable file containing the specifications for which software components the container will run and how.

Continued...

- ***Docker's run utility*** is the command that actually launches a container. Each container is an *instance* of an image. Containers are designed to be transient and temporary, but they can be stopped and restarted, which launches the container into the same state as when it was stopped. Further, multiple container instances of the same image can be run simultaneously (as long as each container has a unique name).
- ***Docker Hub***(https://hub.docker.com/) is a SaaS repository for sharing and managing containers, where you will find official Docker images from open-source projects and software vendors and unofficial images from the general public.
- ***Docker Engine*** is the core of Docker, the underlying client-server technology that creates and runs the containers.
- ***Docker containers*** keep apps isolated not only from each other, but from the underlying system. This not only makes for a cleaner software stack, but makes it easier to dictate how a given containerized application uses system resources—CPU, GPU, memory, I/O, networking, and so on.
- ***Repo Link for installing Docker:*** https://gist.github.com/aswinkumar1999/0cb7ed44e309e542aec0d144d9b0de93



Installing ROS using Docker

- In the traditional method of installing ROS on your local machine, you need to manage a lot of stuffs in regards to dependencies for the packages.
- This is where Docker comes into picture and runs each individual Docker image in separate containers so as to combat any sort of dependency conflicts.

- Using docker installing ROS is just a single command:
  - **$ docker pull osrf/ros-noetic-desktop-full**

- Now to setup the ROS environment the instructions can be found here: https://github.com/rsharanesh2002/Stepping-Into-the-World-of-Robotics
-

Setting up ROS Environment

- Creating a ROS Workspace:
  - A ROS workspace is a folder where you modify, build and install ROS packages.
  - **catkin** is the official build system of ROS.
  - To create a catkin workspace:

```
$ mkdir –p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

  - Now, your catkin workspace will have ***src***, ***build*** and ***devel*** folders
  - To make our workspace visible to ROS,we need to source the setup.*sh files in devel folder as follows:

```
$ source devel/setup.bash
```

Creating a catkin Package

- To create a catkin Package, change to the source space directory of catkin workspace we created earlier: **$ cd ~/catkin_ws/src**

  $ catkin_create_pkg pkg_ros_basics std_msgs rospy roscpp

- This will create a *pkg_ros_basics* folder containing a ***package.xml*** and a ***CMakeLists.txt.***
- Change to **~/catkin_ws** and run **catkin_make.** Now, source the generated setup file.
- Software is ROS is organized in ***packages***. Each package can contain libraries, executables, scripts, etc. Command ***rospack*** allows you to get information about a package.
- A ***manifest (package.xml)*** is a description of a package. This file contains package dependencies, licenses, etc.
- *CMakeLists.txt* has all the commands to build the ROS source code inside the package and create the executable.
- *src:* The source code of ROS packages are kept in this folder.

ROS Nodes

- A ROS Node is a piece of software/executable that uses ROS to communicate with other ROS Nodes. ROS Nodes are building blocks of any ROS application.
- For example, consider a robot with some sensor, motors, etc. One ROS Node could get the sensor values and another node can control the motors. These two nodes communicate with each other in order to move the robot.
- A ROS package can have multiple ROS Nodes
- You can have your entire application under a single ROS Node, but having multiple nodes ensures that if one node crashes, your entire application will not crash.
- Python and C++ are majorly used to write ROS Nodes
- ROS client libraries allow nodes written in different programming languages to communicate too!

Create a ROS Node

- Navigate to the *pkg_ros_basics* package we created:
  - **$ roscd pkg_ros_basics**
- Here, create a *scripts* folder for your python scripts:
  - **$ mkdir scripts**
  - **$ cd scripts**
- Create a Python script called *node_hello_ros.py*:
  - **$ touch node_hello_ros.py**
  - **$ gedit node_hello_ros.py**
- Copy and paste the code given in the repository, save and exit.
- Now, we have to make this script an executable:
  - **$ sudo chmod +x node_hello_ros.py**
- We have created a ROS Node – Let's continue to see how it is run…

ROS Master and Running ROS Nodes

- The role of the ROS Master is to enable individual ROS nodes to locate one another, so that they can communicate with each other peer-to-peer.
- So, communication between ROS Nodes is established by ROS Master.
- To start ROS Master: **$ roscore**

  - This will also start ROS Parameter server and **rosout** Logging Node

- The command **rosnode** displays information about ROS nodes that are currently running. Start ROS master and run **rosnode list**, you will se the output as **/rosout**

  - This is because only one node (**rosout)** is running. This is always running as it collects and logs nodes' output.

- **$ rosnode info [node_name]** - returns information about a specific node.

Continued…

- The command **rosrun** is used to run a ROS node/an executable in an arbitrary package from anywhere without having to give its full path, as follows:

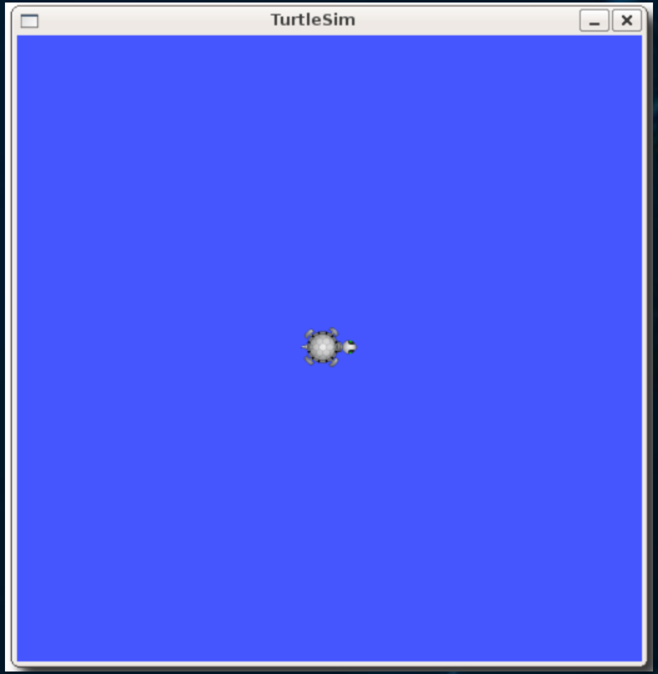
  - **$ rosrun [package_name] [executable]**

- Let us run the **turtlesim_node** in ***turtlesim*** package (Turtlesim is a lightweight simulator used for learning ROS):

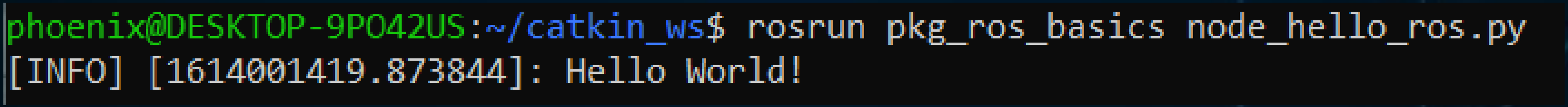
  - **$ rosrun turtlesim turtlesim_node**

  !!! Remember that ROS Master should be active and running in another terminal !!!

- You will now see the turtlesim window as shown in the picture



- Now, let us try running the *node_hello_ros.py* node we created earlier:
- First, make sure ROS Master is running:
  - **$ roscore**
- Now use rosrun to run the ROS Node:
  - **$ rosrun pkg_ros_basics node_hello_ros.py**
- You should get an output like this,



```
phoenix@DESKTOP-9PO42US:~/catkin_ws$ rosrun pkg_ros_basics node_hello_ros.py
[INFO] [1614001419.873844]: Hello World!
```

ROS Topics

- ROS Topics allow unidirectional communication between ROS nodes. When using ROS Topics, a ROS node can be a publisher, subscriber or both.
- Publisher Node – publishes data on a ROS Topic and a Subscriber Node – subscribes to a ROS Topic to get data.
- So, Publisher and Subscriber Nodes exchange ROS messages over ROS Topics
- ROS Message – a simple data structure which can hold data of various types (int, float, bool, etc.)
- **rostopic** - displays information about ROS Topics, publishers, subscribers, publishing rate and ROS Messages.
- **$ rostopic list** – shows a list of all topics currently subscribed to and published.
- **$ rostopic echo [topic]** – shows the data published on a topic.
- **$ rostopic type [topic]** – returns the message type of any topic being published.
- **$ rostopic pub [topic] [msg_type] [args]** – publishes data on a topic.

Understanding ROS Topics using TurtleSim

- Run the following in a new terminal: **$ rosrun turtlesim turtlesim_node**

  - Make sure ROS Master is running!

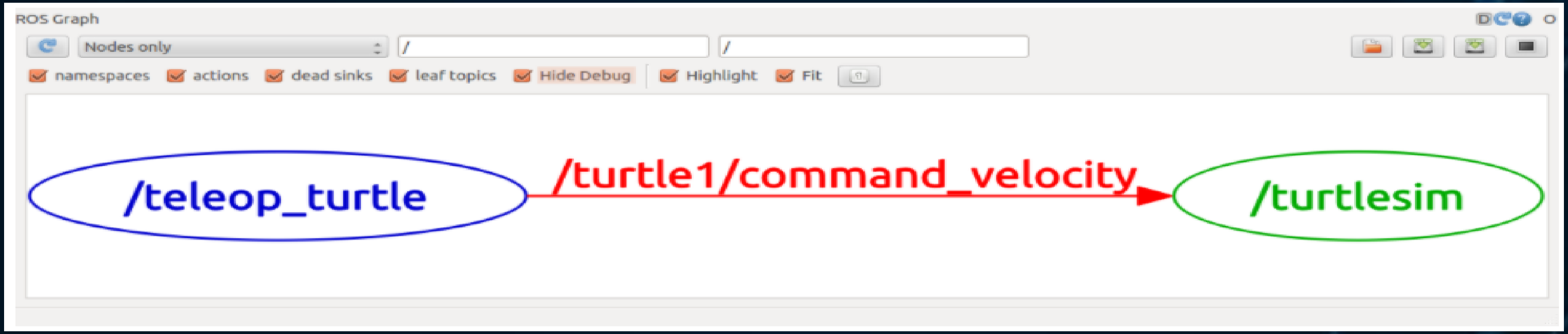
- Now in another terminal, run: **$ rosrun turtlesim turtle_teleop_key**
- Now, you can use the arrow keys of your keyboard to drive the turtle around. Select the terminal window of *turtle_teleop_key* and use the arrow keys and see the turtle moving in the TurtleSim window.
- So, What's happening here?
  - The *turtlesim_node* and *turtle_teleop_key* node are communicating over a ROS Topic. *turtle_teleop_key* node is publishing the key strokes on a topic, while *turtlesim_node* subscribes to the same topic to receive the key strokes, and the turtle moves.

  - ***rqt-graph:*** Creates a dynamic graph of what's going on in the system.

Rqt_graph and Using *rostopic pub*

- ***rqt_graph*** is a part of the ***rqt*** package. To install it, run:
  - **$ sudo apt-get install ros-<distro>-rqt**
  - **$ sudo apt-get install ros-<distro>-rqt-common-plugins**

  In a new terminal, run:
  - **$ rosrun rqt_graph rqt_graph** – You will see a window like the one below:



- ***rostopic pub*** – publishes data on a topic currently advertised
  - With the turtlesim running from before, let's try *rostopic pub*:
  - **$ rostopic pub -1 /turtle1/cmd_vel geometry_msgs/Twist -- '[2.0, 0.0, 0.0]' '[0.0, 0.0, 1.8]'**
  - This will send a message to turtlesim telling it to move with a linear velocity 2.0 and an angular velocity of 1.8

Some more Terms…

- **ROS Services -** Services are another way that nodes can communicate with each other. Services allow nodes to send a **request** and receive a **response.** A providing ROS node offers a service under a string name, and a client calls the service by sending the request message and awaiting the reply.
- **Parameter Server** is a space where all the necessary data that needs to be shared among various ROS Nodes is stored. It runs inside ROS Master. ROS Nodes can view and even modify data stored in the Parameter Server. Typically Parameter Server is used to store configuration parameters.
- ***roslaunch* -** tool for easily launching multiple ROS nodes locally and remotely via SSH.

A quick Demonstration on running ROS Nodes

- Launch two ROS Nodes:
  - Let's see how to use the *roslaunch* command to launch two nodes, namely *talker* and *listener* nodes.
  - Create a *chatter.launch* file and save in a folder named *launch,* in the pkg_ros_basics package we created earlier as follows:
    - **$ roscd pkg_ros_basics**
    - **$ mkdir launch**
    - **$ cd launch**
    - **$ touch chatter.launch**
    - **$ gedit chatter.launch**
  - Copy and paste the code from the repo into *chatter.launch*, save and exit.
- Now, change into the *scripts* folder we created in *pkg_ros_basics* and create two files – *talker.py* and *listener.py*
- Copy and paste the codes for these from the repo, save and exit.
- Now, to launch these two nodes, run: **$ roslaunch pkg_ros_basics chatter.launch**

In the next part of this session, we will be discussing:

- URDF (Unified Robot Description Format) and its applications:
  - Brushing through the syntax of URDF
  - Hands on building of 2/4 wheeled robot using URDF file
- Navigation of Robots:
  - Manually controlling the robot using keyboard
  - Automatic navigation of the built robot (example only)