



MODULE 2: ROBOT OPERATING SYSTEM

Nicholas Ho, PhD
Institute of System Science, NUS



Robots within HRI Topic



Human-Robot Interaction (HRI) is a field of study dedicated to understanding, designing, and evaluating robotic systems **for use by or with humans**. Interaction, by definition, requires **communication between robots and humans**



What is ROS?



- Flexible framework for writing robot software
- Tools, libraries and conventions
- Open-source





Why ROS?



- Creating truly robust, general-purpose robot software is **hard**
 - From the robot's perspective, problems that seem trivial to humans often vary wildly between instances of tasks and environments.
 - Dealing with these variations is so hard that no single individual, laboratory, or institution can hope to do it on their own





ROS Core Components



- **Executable**
 - ROS Nodes
- **Communication**
 - ROS Messages, ROS Topics
 - Publisher and Subscriber
 - ROS Services, ROS Actionlibs
- **Record and playback**
 - ROS Bags
- **Visualization**
 - rviz





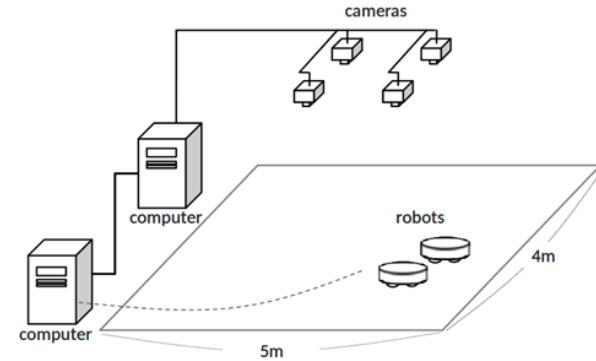
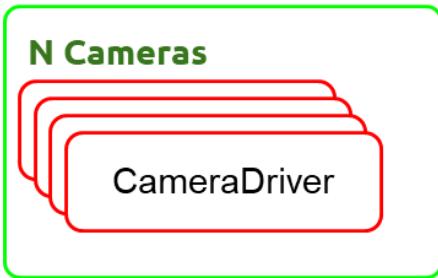
ROS Nodes



- A ROS Node is an executable process that uses ROS to communicate with other Nodes
- E.g., Magic Cards
 - The cameras need to see
 - CameraDriver
 - The “magic cards” need to be understood
 - Parser
 - The robots need to be localized at all time
 - Localization
 - The parsed card info needs to be aggregated as a plan for execution
 - Planner
 - The plan needs to be break down into low-level control commands
 - RobotDriver

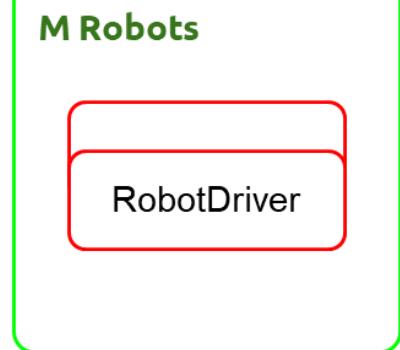


Magic Cards



hardware

ROS Node





ROS Messages



- **A ROS Message is a ROS data type for communication among the Nodes**
- **ROS built-in default message types**
 - **std_msgs**
 - basic data types, such as int, float, string
 - **geometry_msgs**
 - geometry-related data types, such as point, vector, pose
 - **sensor_msgs:**
 - sensory data types, such as batterystate, image, pointcloud
 - etc...
- **Customized message type**
 - Developer can define a new message type using the existing message types



std_msgs



- **Basic**

- Bool, Byte, Char
- Int8, Int16, Int32, Int64,
- UInt8, UInt16, UInt32, UInt64
- Float32, Float64
- String
- Time, Duration

- **Signal**

- Empty: a message usually used as a signal

- **Compound**

- **Header**: a message in sequence with timestamp
- **ColorRGBA**: a message to describe a pixel

std_msgs/Header.msg

uint32	seq
time	stamp
string	frame_id

std_msgs/ColorRGBA.msg

float32	r
float32	g
float32	b
float32	a



std_msgs (cont)

- **Array-related**
 - MultiArrayDimension

std_msgs/MultiArrayDimension.msg

```
string label
uint32 size
uint32 stride
```

- IntXXMultiArray, UIntXXMultiArray, FloatXXMultiArray

std_msgs/Int16MultiArray.msg

MultiArrayLayout	layout
uint8[]	data



std_msgs (cont)



- **Array-related**
 - MultiArrayLayout

std_msgs/MultiArrayLayout.msg

```
MultiArrayDimension[]    dim
uint32                  data_offset
```

E.g., a standard, 3-channel 640x480 image with interleaved color channels:

```
dim[0].label = "height"
dim[0].size = 480
dim[0].stride = 3*640*480
dim[1].label = "width"
dim[1].size = 640
dim[1].stride = 3*640
dim[2].label = "channel"
dim[2].size = 3
dim[2].stride = 3
```



geometry_msgs



- **Pose related**
 - **Point**, Point32, PointStamped
 - **Quaternion**, QuaternionStamped
 - **Pose**, Pose2D, **PoseStamped**, PoseWithCovariance, PoseWithCovarianceStamped
 - PoseArray
- **Control related**
 - Vector3, Vector3Stamped
 - Transform, TransformStamped
 - Twist, TwistStamped, TwistWithCovariance, TwistWithCovarianceStamped
 - Wrench, WrenchStamped
- **Etc.**



geometry_msgs (cont)



geometry_msgs/Pose.msg

Point	position
Quaternion	orientation

geometry_msgs/PoseStamped.msg

Header	header
Pose	pose

geometry_msgs/Point.msg

float64	x
float64	y
float64	z

geometry_msgs/Quaternion.msg

float64	x
float64	y
float64	z
float64	w



sensor_msgs

- **Camera related**
 - RegionOfInterest, CameraInfo
 - Image, CompressedImage
- **PointCloud related**
 - LaserEcho, LaserScan, MultiEchoLaserScan
 - PointCloud, PointCloud2
- **GPS related**
 - NavSatFix, NavSatStatus
- **Joystick related**
 - Joy, JoyFeedback, JoyFeedbackArray
- **Others**
 - BatteryState, Temperature, RelativeHumidity, FluidPressure, etc



Customized Message



- Create a msg file
- Follow the format of the default message
- Add in the fields using already defined message types
- E.g.

std_msgs/Header.msg

uint32	seq
time	stamp
string	frame_id

msgs/MagicCard.msg

uint32	card_id
string	card_txt
time	stamp
pose2D	pose
twist	control
...	



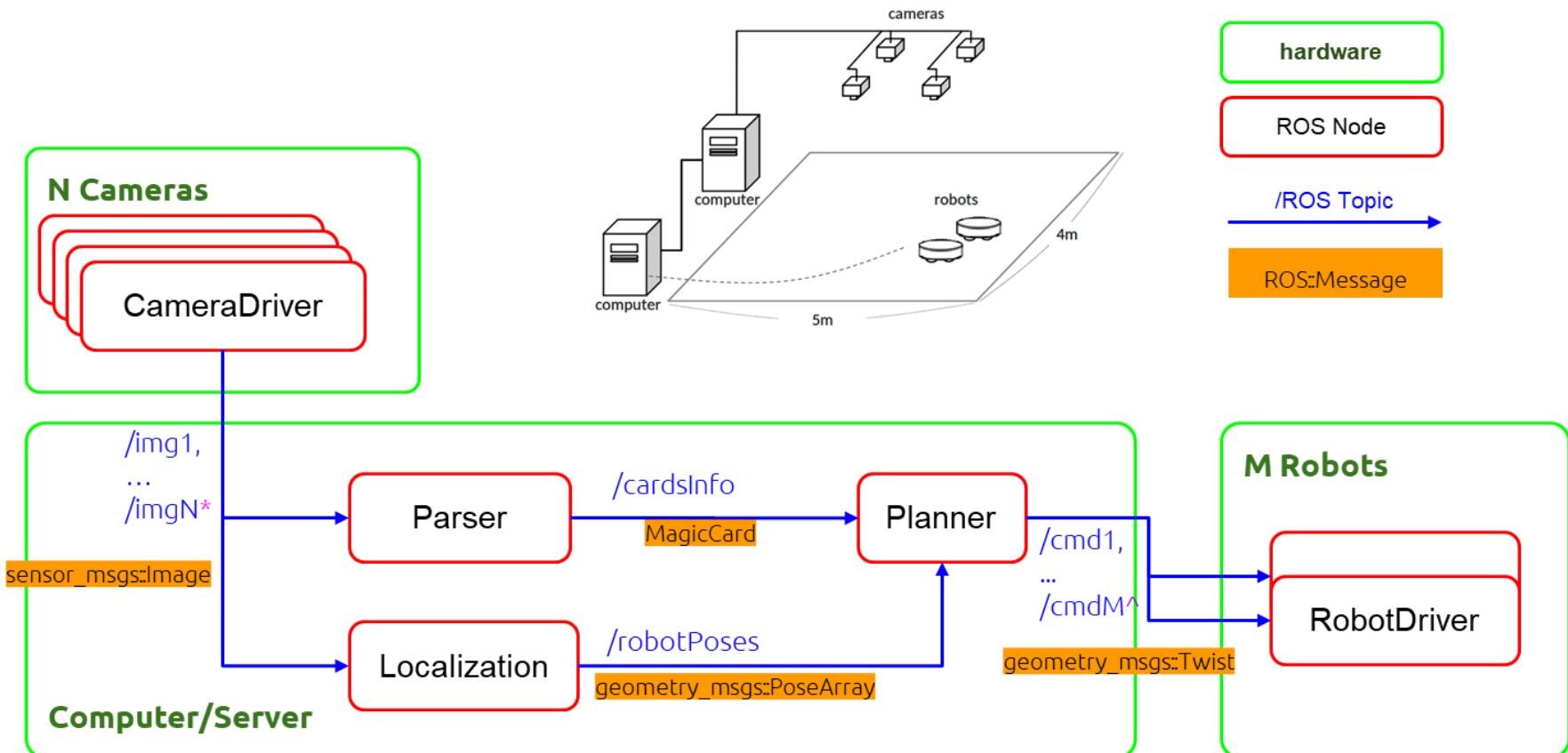
ROS Topics (RECAP)



- A **ROS Topic** is a channel for communicating Messages among the Nodes
- A ROS Topic has a fixed Message type
- A Node ***publishes*** on a Topic to broadcast Messages
- A Node ***subscribes*** to a Topic to get Messages from other Nodes
- *Many-to-many* communication
 - One Node can publish/subscribe multiple Topics
 - One Topic can be published/subscribed by different Nodes



Magic Cards (v1.0)

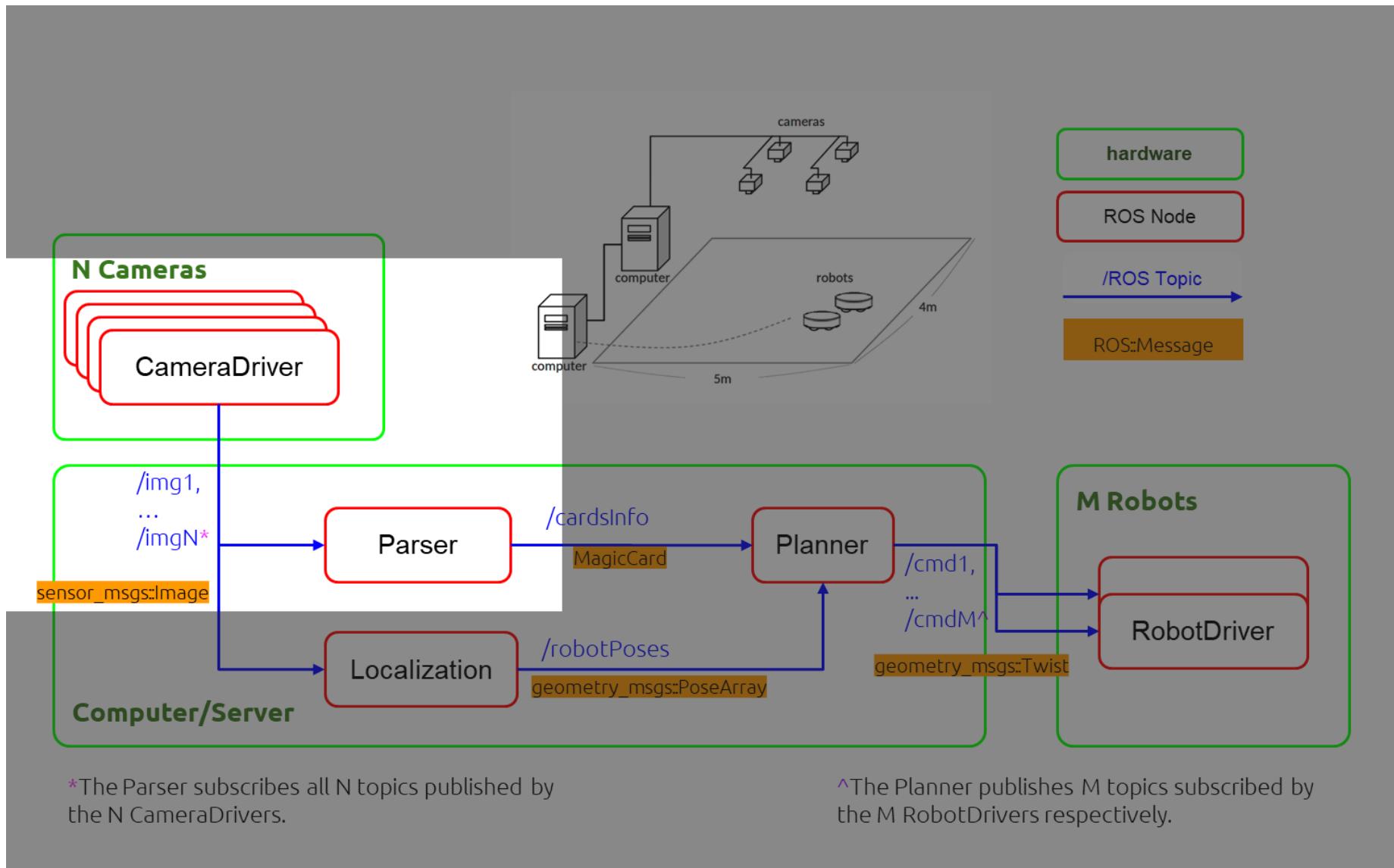


*The Parser subscribes all N topics published by the N CameraDrivers.

^The Planner publishes M topics subscribed by the M RobotDrivers respectively.

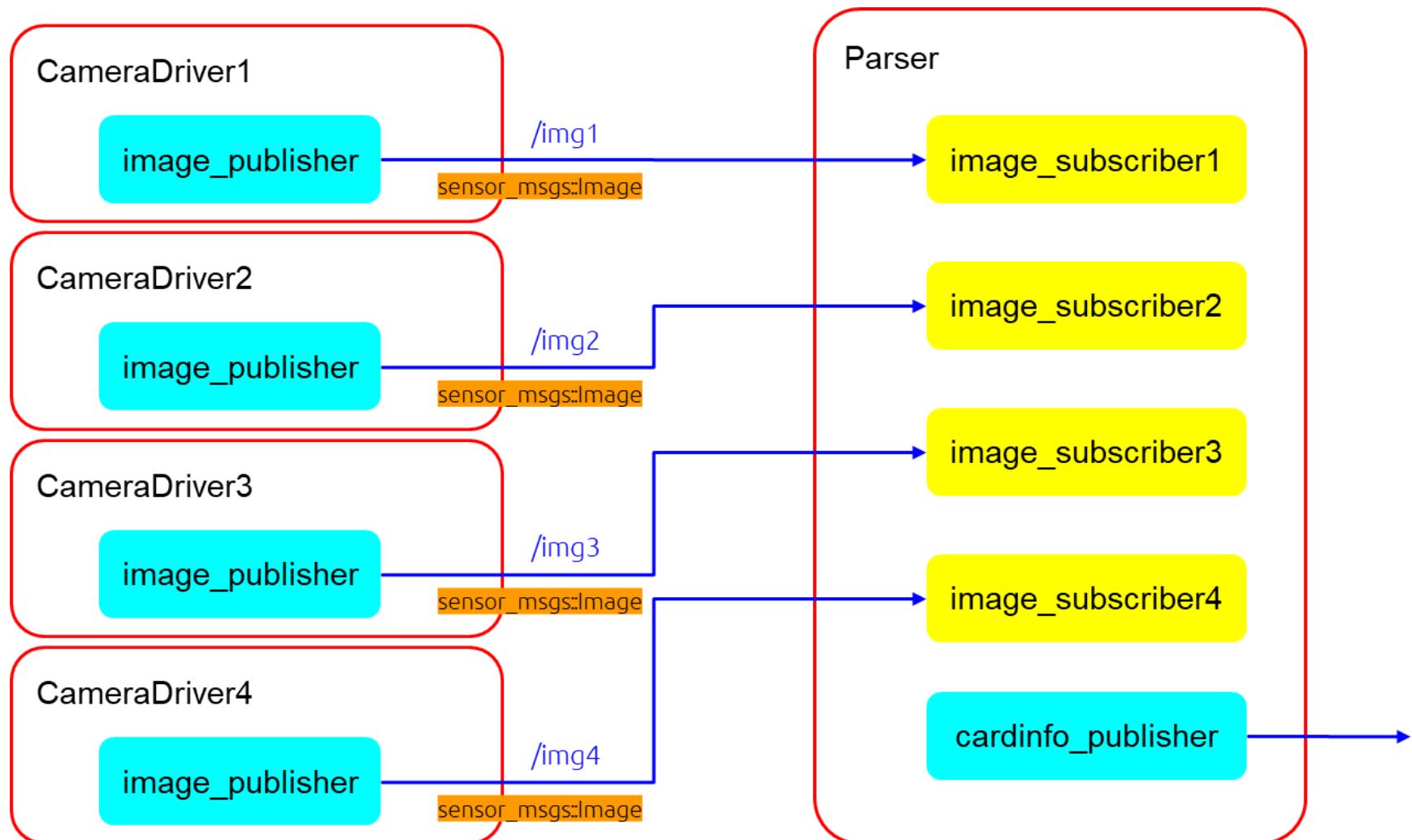


Magic Cards (v1.0)



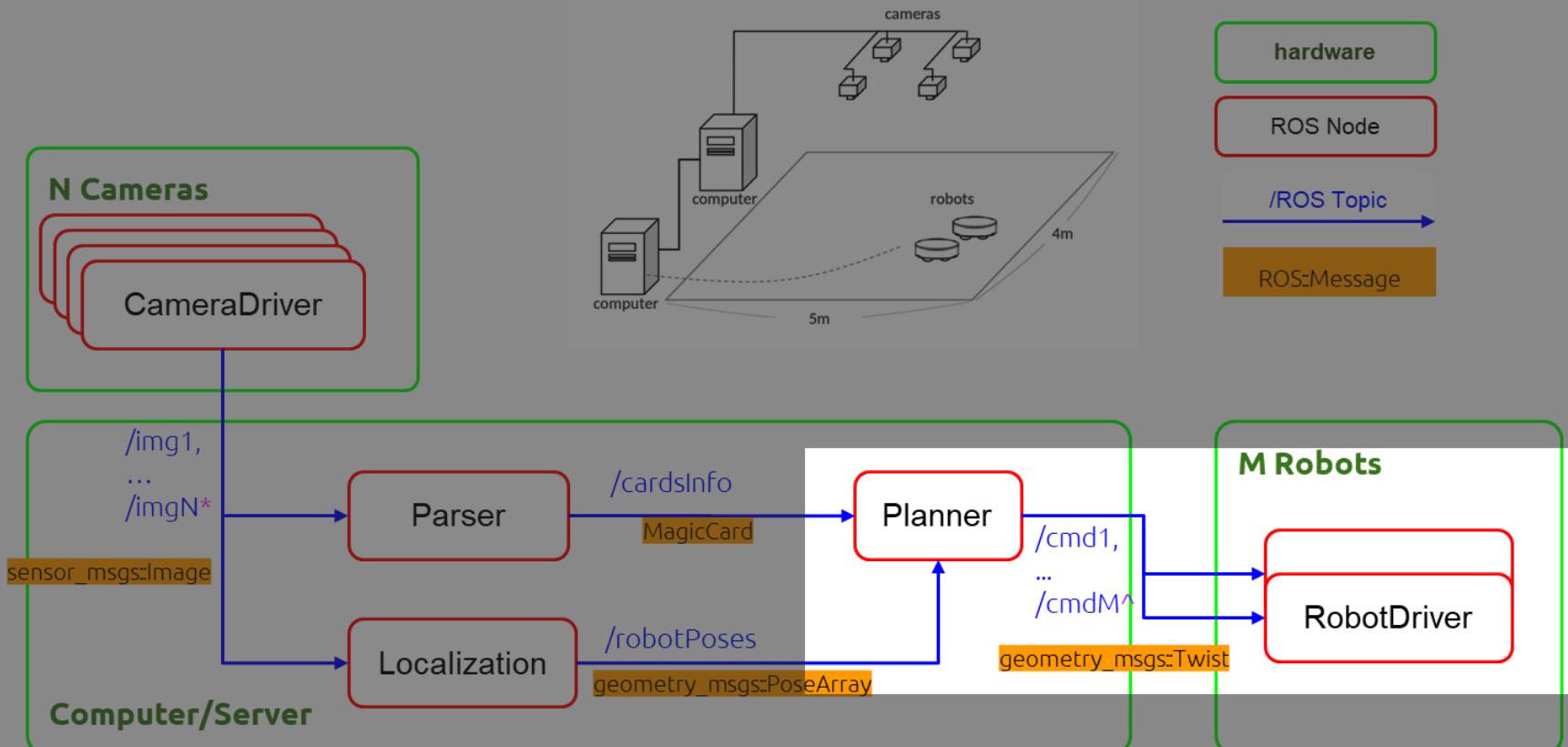


Publisher and Subscriber (v1.0)





Magic Cards (v1.0)

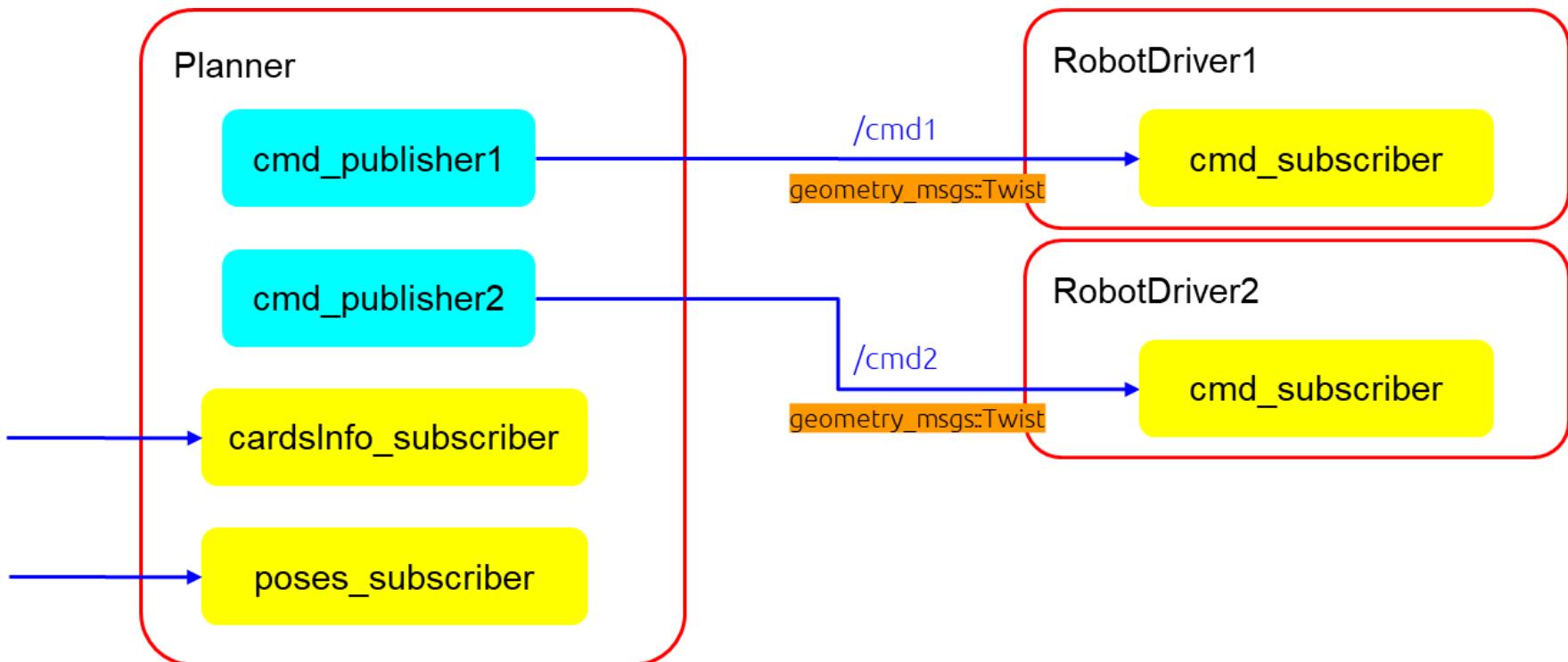


*The Parser subscribes all N topics published by the N CameraDrivers.

^The Planner publishes M topics subscribed by the M RobotDrivers respectively.

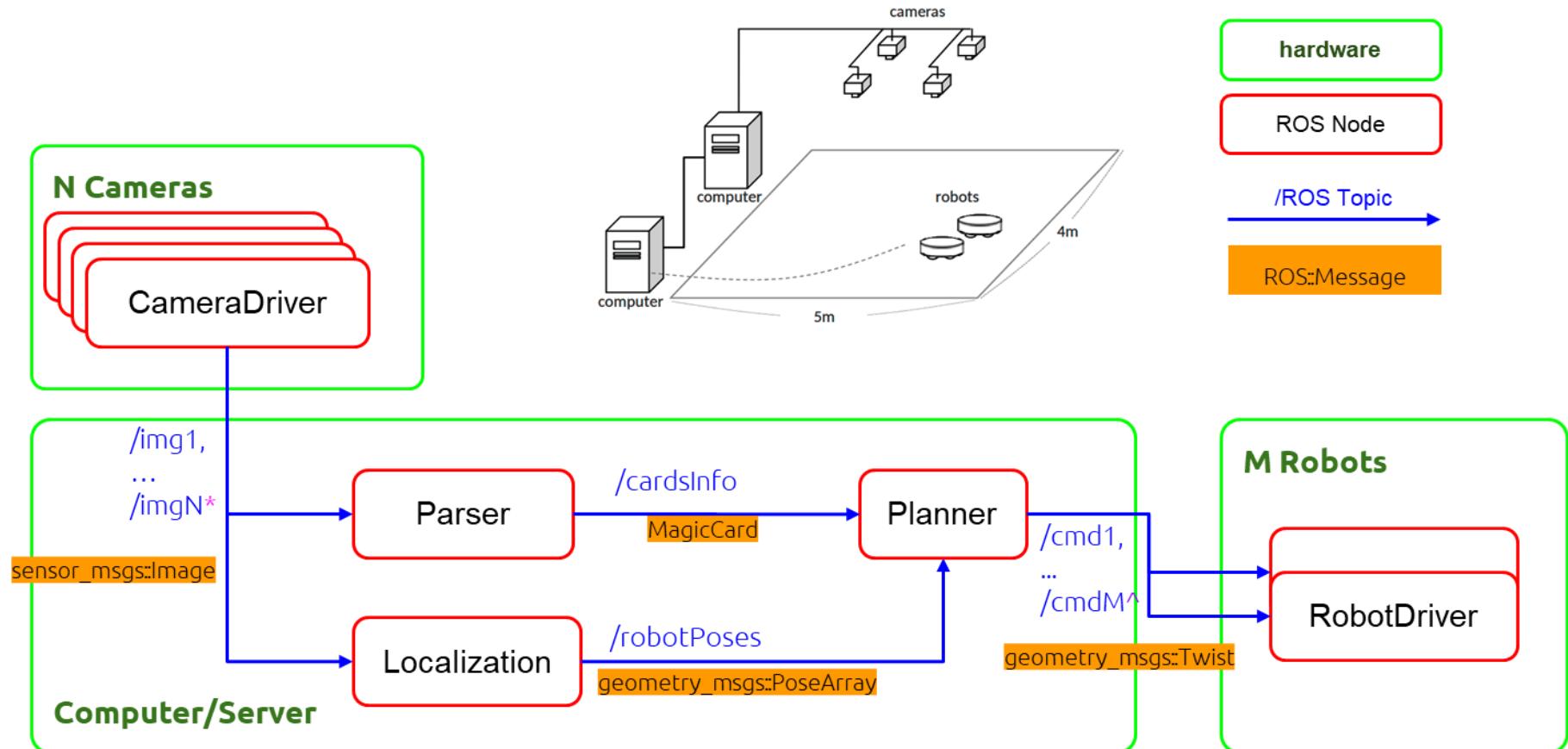


Publisher and Subscriber (v1.0) cont





Magic Cards (v1.0)

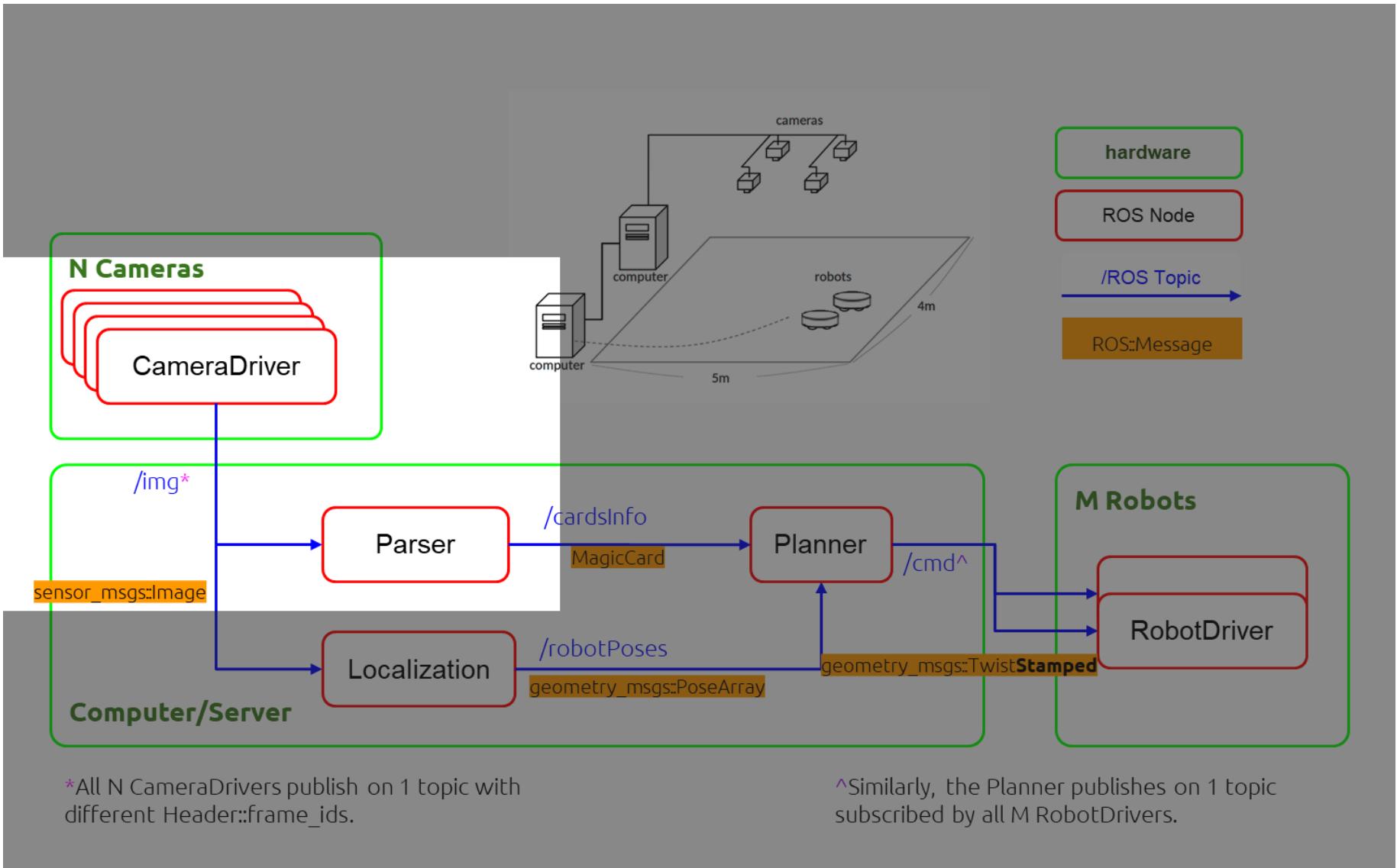


*The Parser subscribes all N topics published by the N CameraDrivers.

^The Planner publishes M topics subscribed by the M RobotDrivers respectively.



Magic Cards (v2.0)

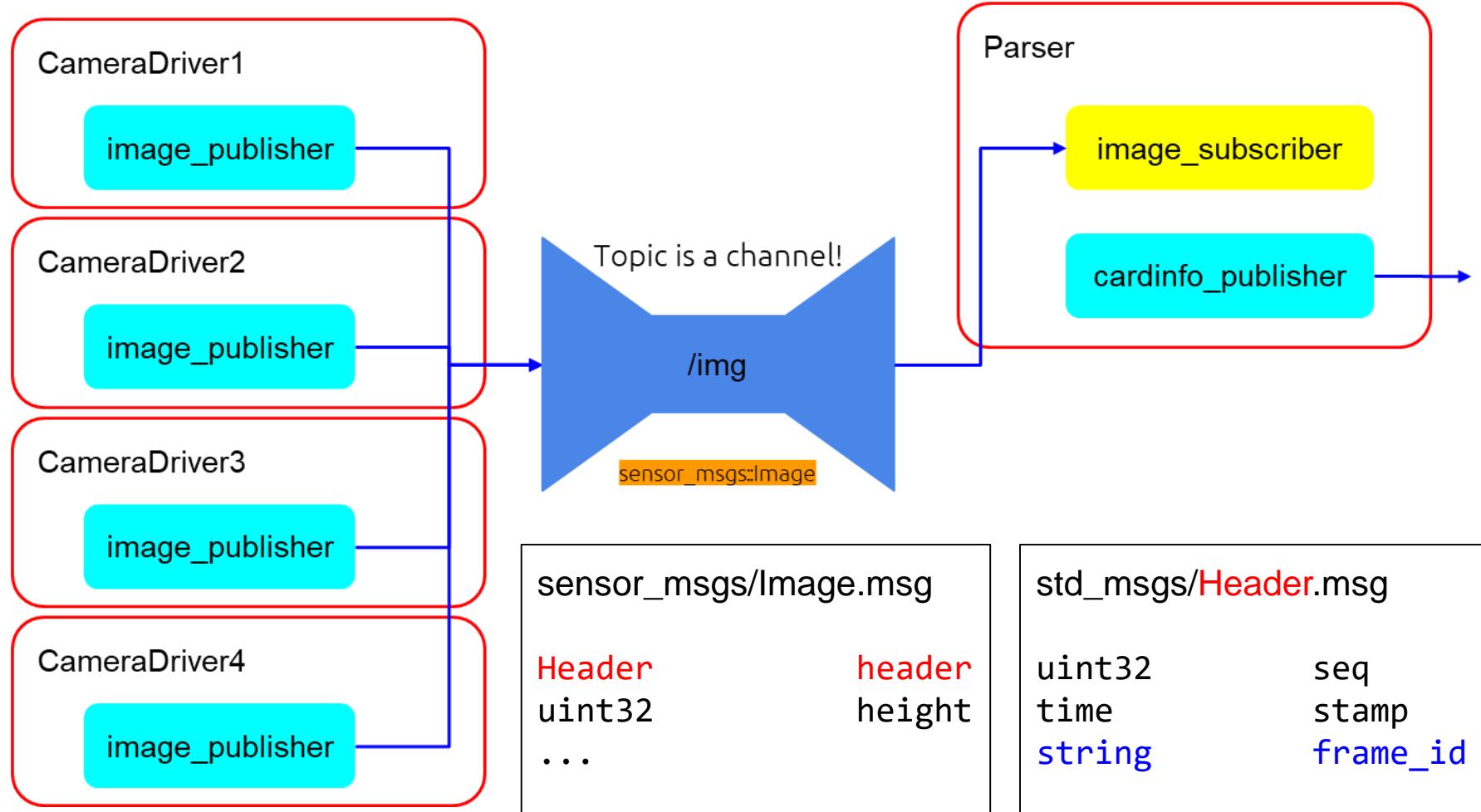


*All N CameraDrivers publish on 1 topic with different Header::frame_ids.

^Similarly, the Planner publishes on 1 topic subscribed by all M RobotDrivers.

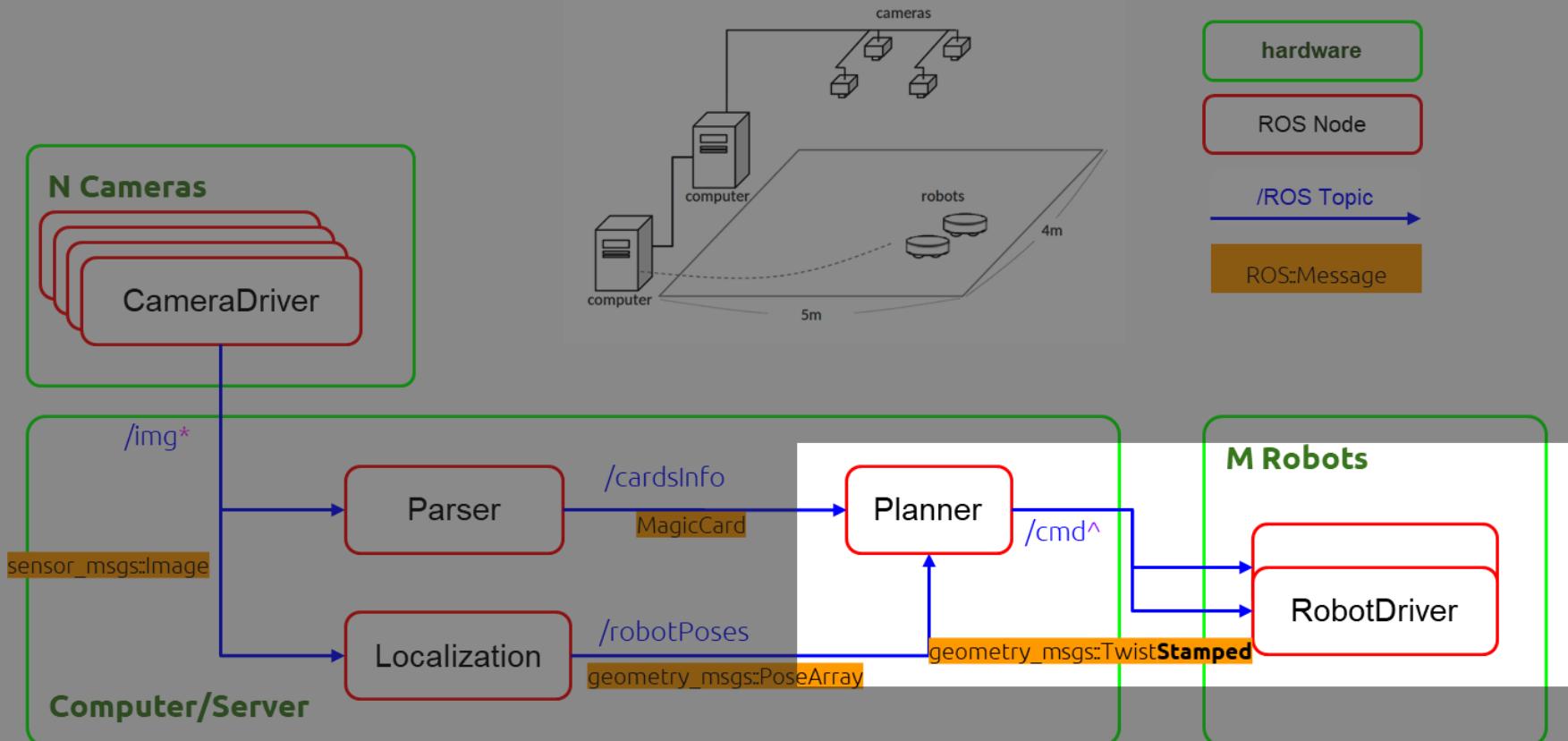


Publisher and Subscriber (v2.0)





Magic Cards (v2.0)

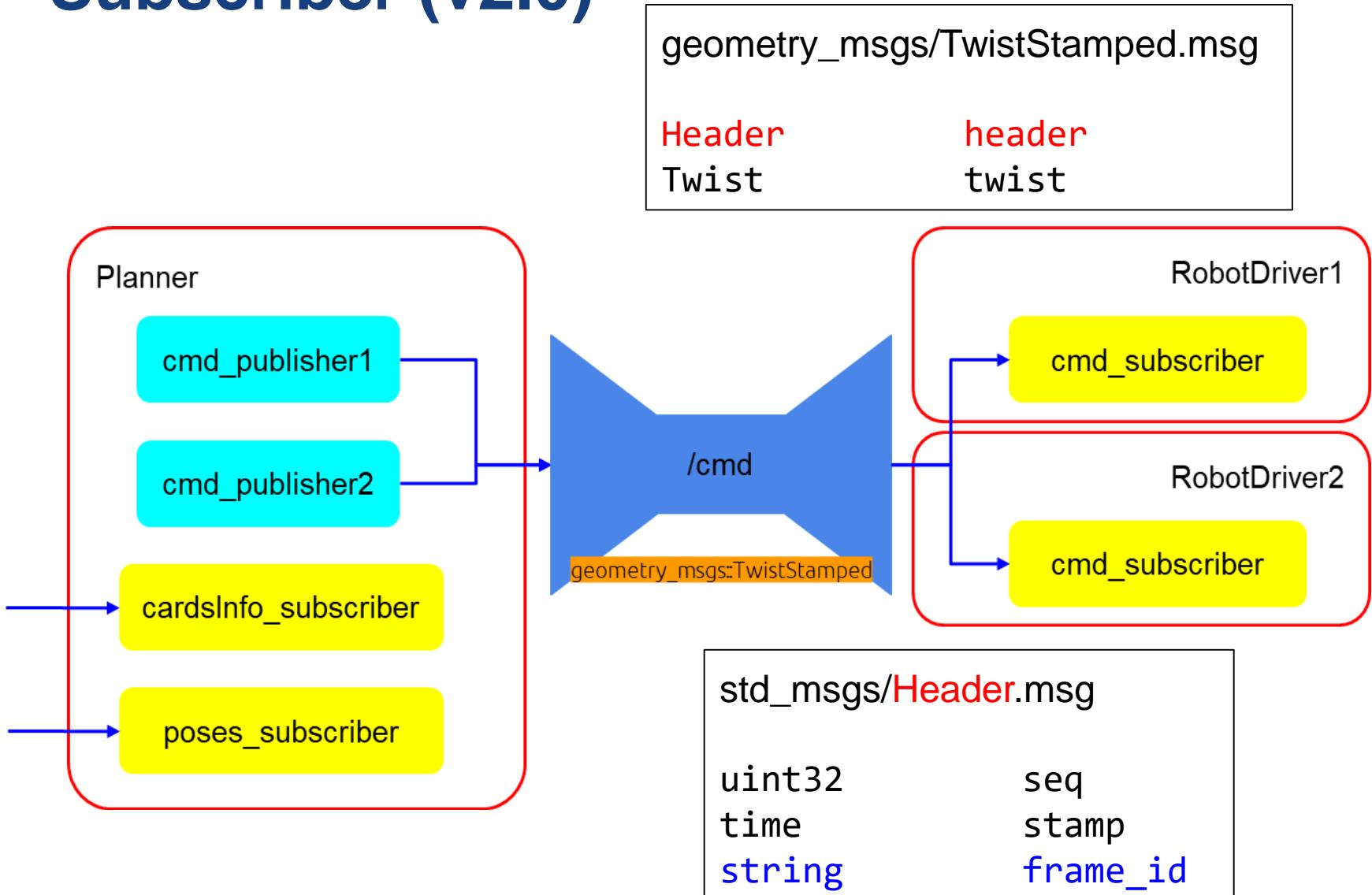


*All N CameraDrivers publish on 1 topic with different Header::frame_ids.

^Similarly, the Planner publishes on 1 topic subscribed by all M RobotDrivers.



Publisher and Subscriber (v2.0)





ROS Services

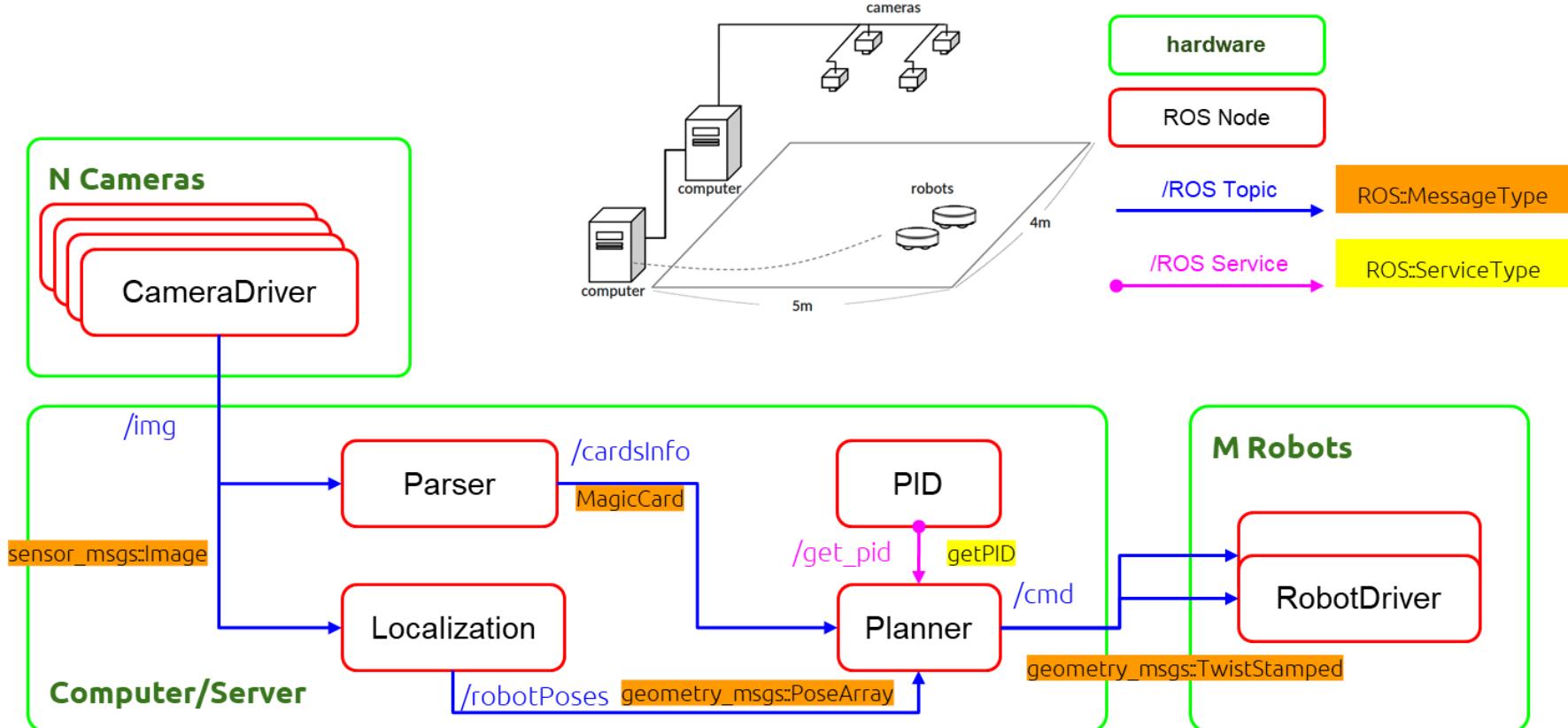


The **publish / subscribe** model is a very flexible communication paradigm, but its many-to-many one-way transport is not appropriate for **request / reply** interactions

- **ROS Services** are another way that nodes can communicate with each other (i.e. two-way)
- ROS Services allow nodes to send a request and receive a response
- **ROS built-in services**
- **Customized service**
 - Developer can define a new service call based on existing message types



Magic Cards (v3.0)





Built-in Service in sensor_msgs

sensor_msgs/setCameraInfo.srv

CameraInfo info

bool success
string status_message

Request

Response



Customized Service



sensor_msgs/setCameraInfo.srv

CameraInfo info

bool success

string status_message

srvs/getPID.srv

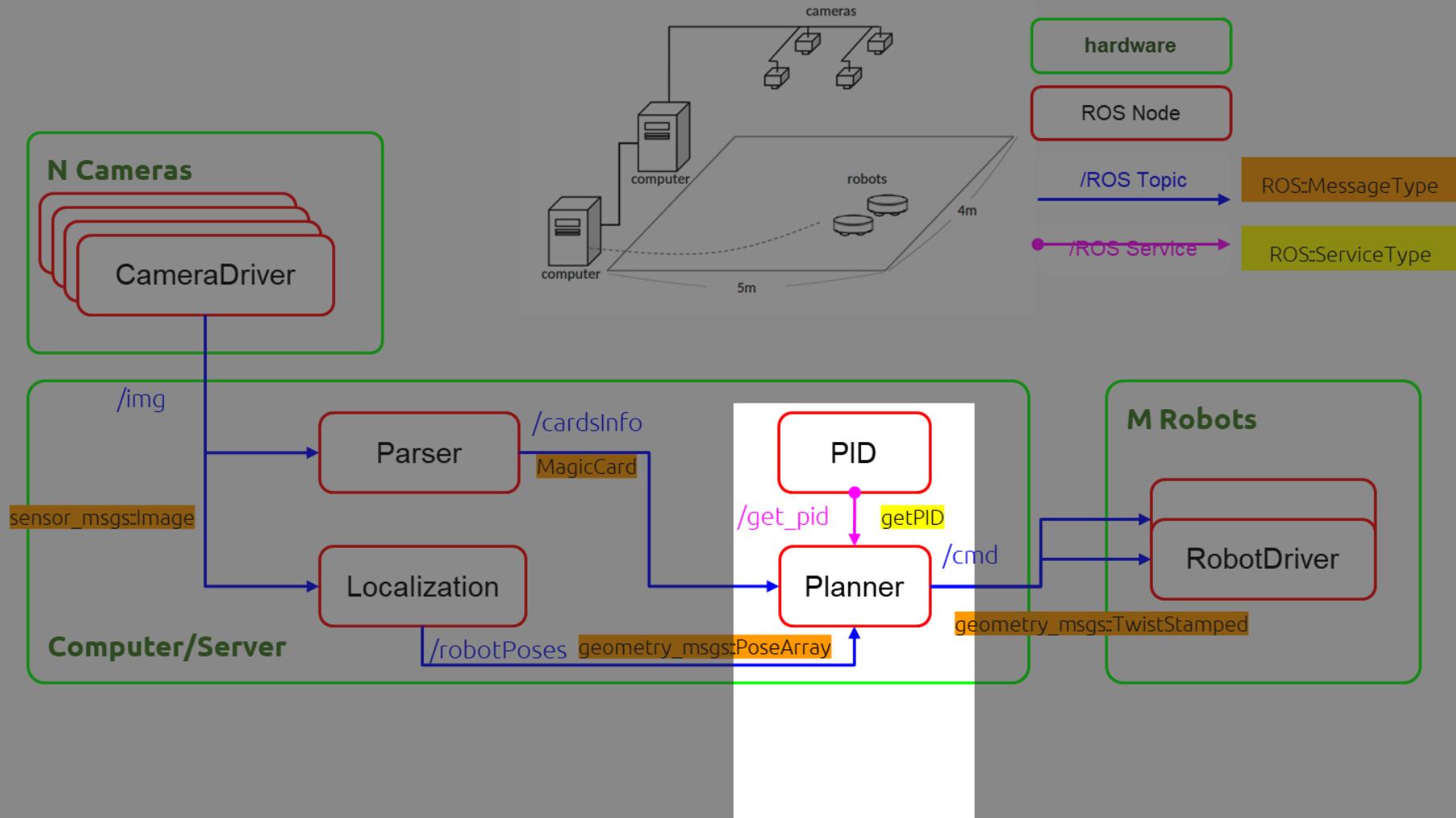
Pose goal_pose

Pose current_pose

Twist twist

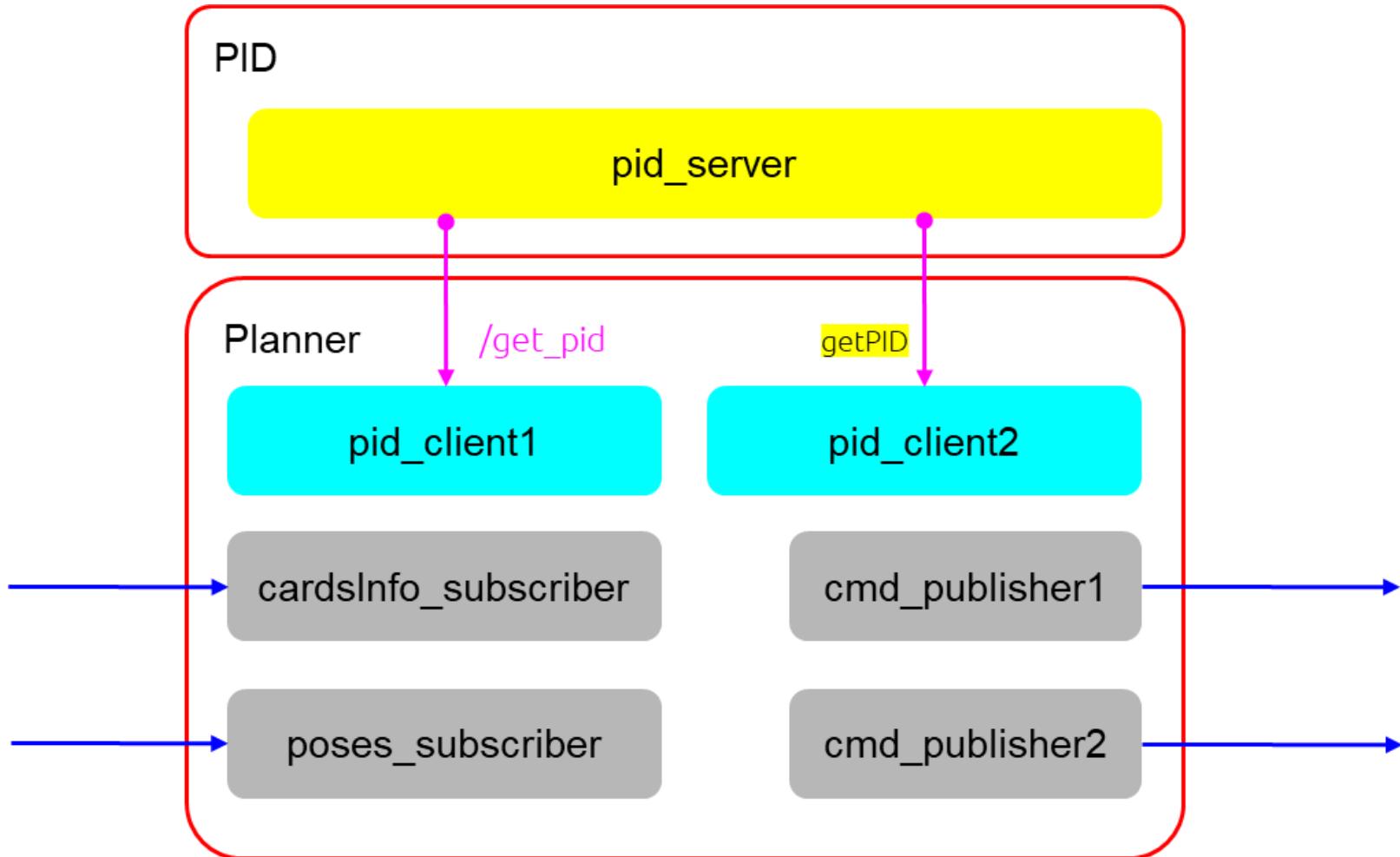


Magic Cards (v3.0)





Service and Client





ROS Architecture Construction (Practice)



Consider an autonomous drone that is in an environment with humans. You are required to **consider the following components and the respective features:**

1. One camera for it to see; its collected data aid in the localization of the robot and the machine vision system
2. A computer to identify things (be it humans or objects), to plan routes, and to localize the robot
3. The motors for the propellers



ROS Architecture Construction (Practice)



Based on the below information, **construct a ROS architecture** using the given template, “ROS Architecture Template”.

The various **node names** that are required for the ROS programme for this drone, together with their respective **topics and messages** involved are summarized in the table below (assume only topics are used):

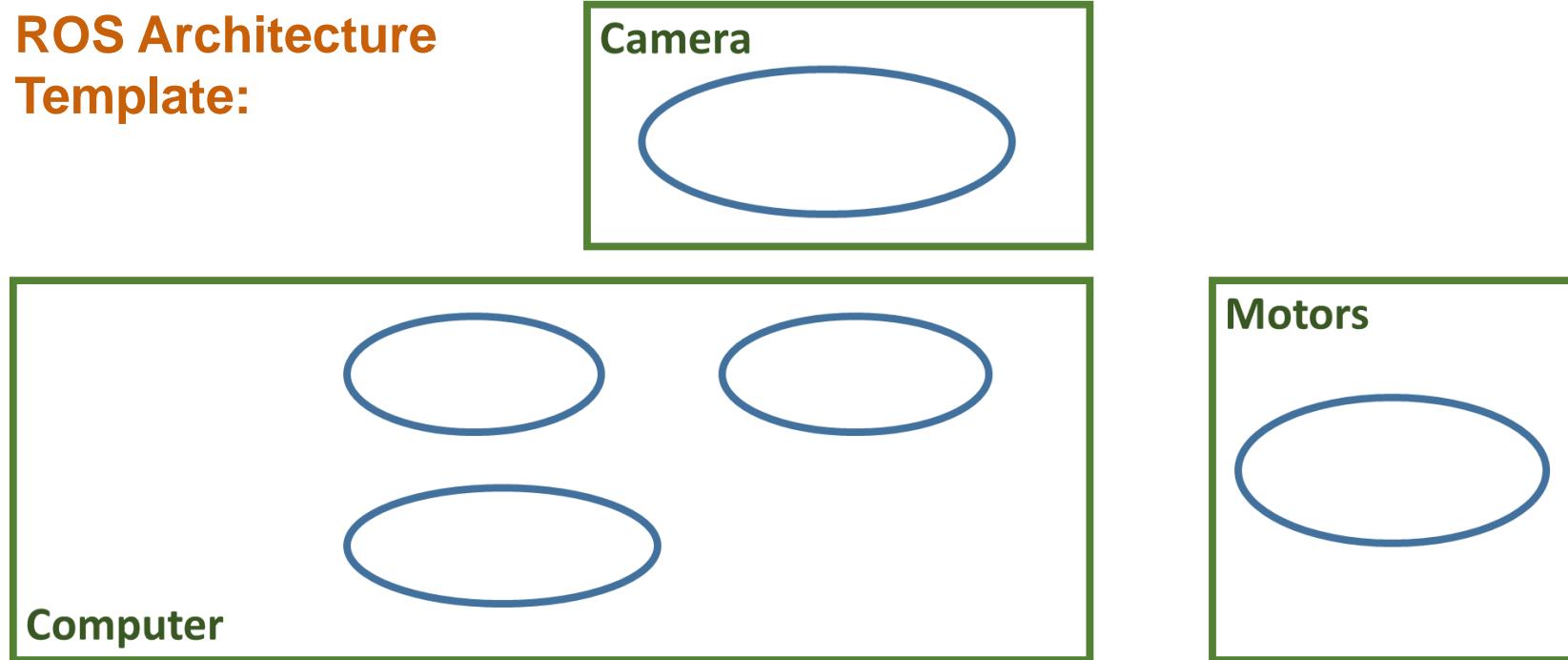
Hardware	ROS Node	/ROS Topic	ROS:Message
Camera	CameraDriver	/image	sensor_msgs::Image
Computer	MachineVision	/thingsData	Things
	Localization	/pose	geometry_msgs::PoseArray
	Planner	/cmd	geometry_msgs::Twist
Motors	MotorDriver	N.A.	N.A.



ROS Architecture Construction (Practice)



ROS Architecture
Template:



Legend:





Extra: ROS Actionlibs



In some cases, if the service takes a long time to execute, the developer might want the ability to cancel the request during execution or get periodic feedback about how the request is progressing.

- **ROS Actionlibs** create servers that execute long-running goals that can be preempted
- ROS Actionlibs provide a client interface to send requests to the server

action/CleanRooms.action

```
uint32[]      room_ids  
---  
uint32        total_rooms_cleaned  
---  
float32       percentage_completed
```

Goal

Result

Feedback



ROS Bags



Real-world robot data are generally hard and costly to acquire. **ROS Bags** enables easy record / playback for store, process, analysis and visualization, etc.

- **Record**

- A ROS Bag is a special Node that subscribes to one or more Topics, and stores the message data in a file as it is received.

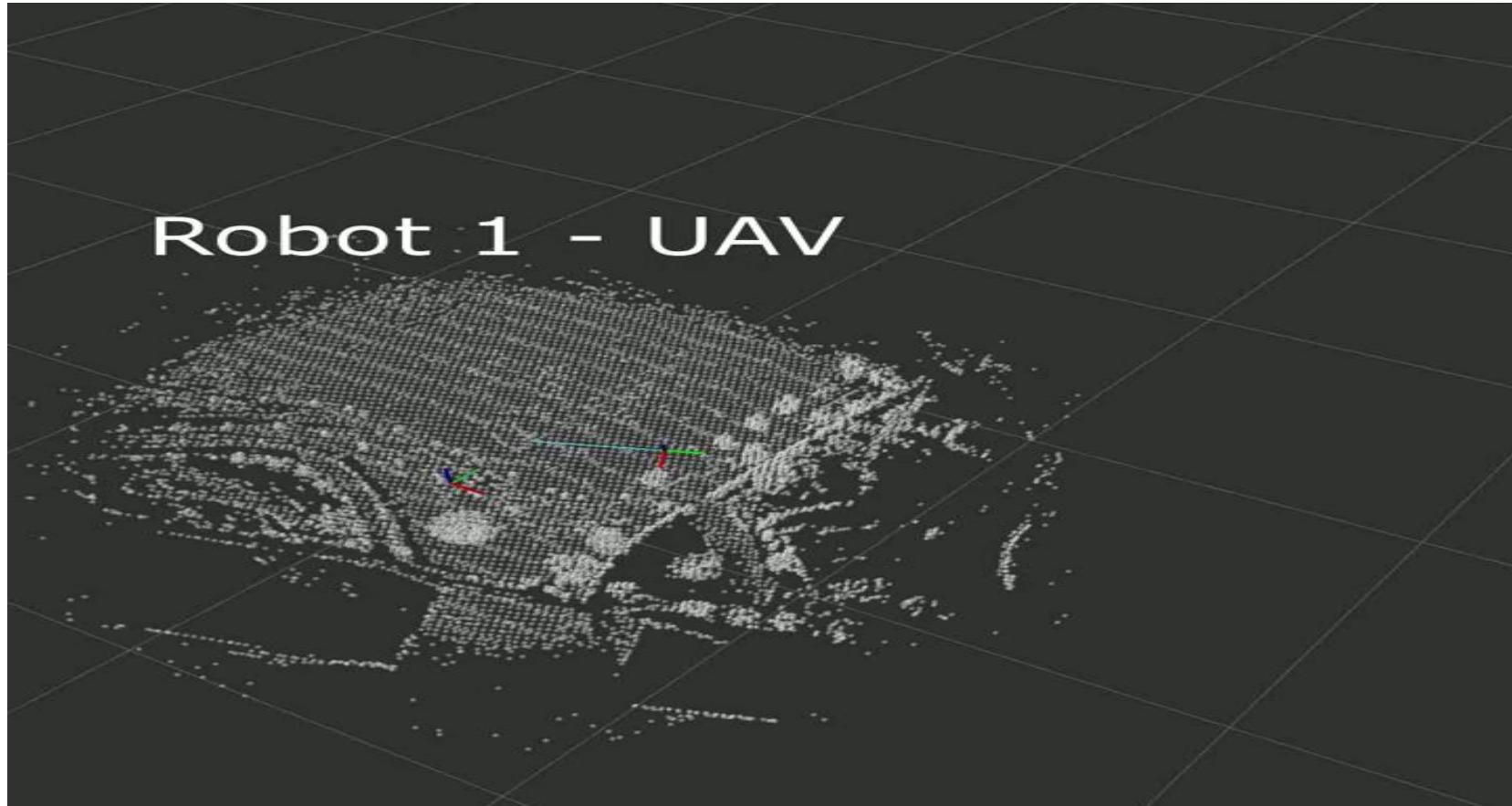
- **Playback**

- The bag files can also be played back in ROS to the same topics they were recorded from.



Data Playback on Collaborative 3D SLAM

Collaborative SLAM between UAV and UGV

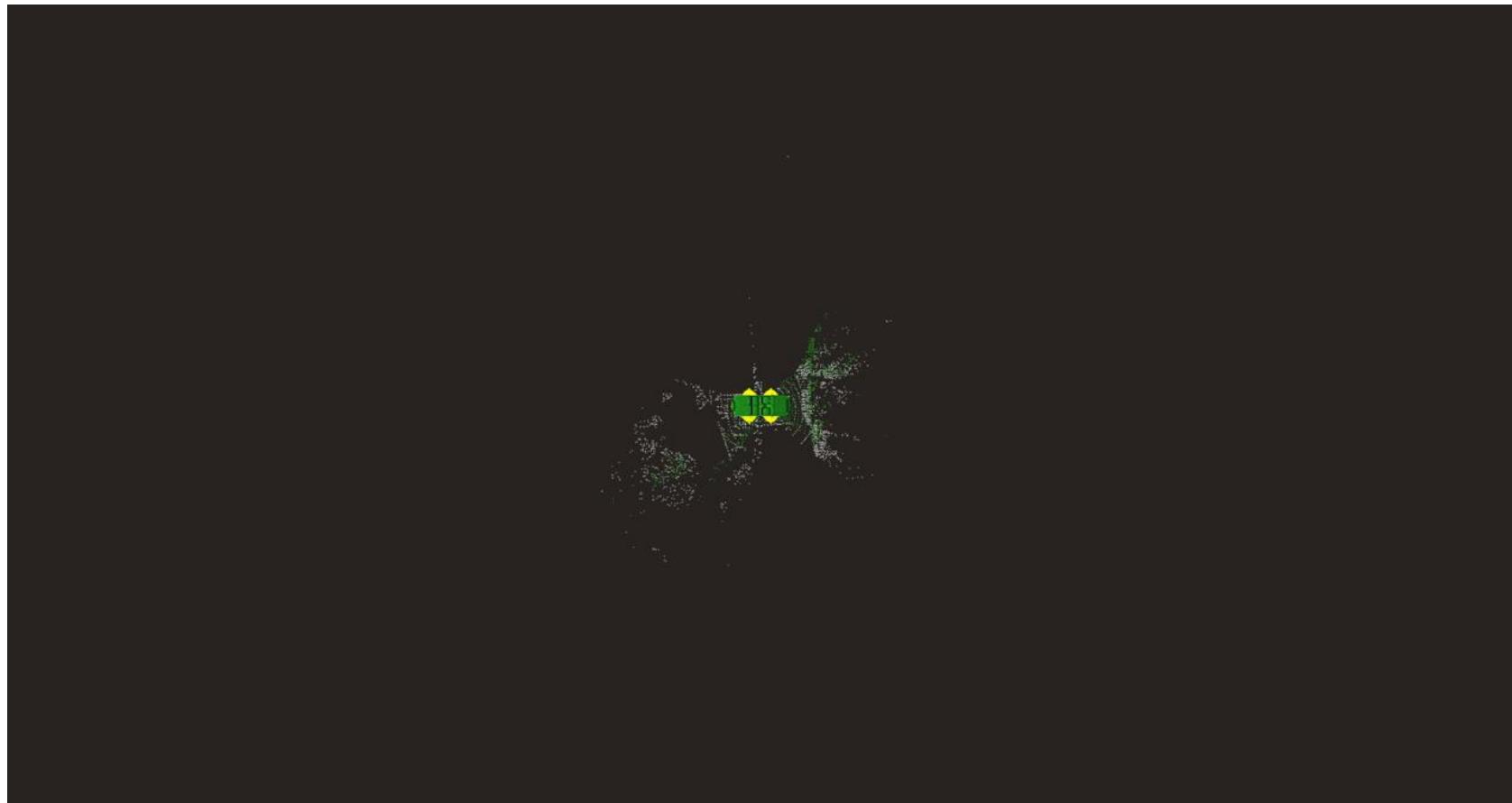


Source: https://www.youtube.com/watch?v=ZZQT_REkItU



Data Playback on Collaborative 3D SLAM (cont)

Collaborative SLAM in NUS forest



Source: <https://www.youtube.com/watch?v=1JoPp7GbN4E>



Extra: practical issues



ROS is a middleware which faces practical problems while passing data around

- **Data loss**
 - Network breaks down
- **Data delay**
 - Data may not be received in order
- **Data non-synchronization**
 - Messages played back from the Bags at the recorded pace

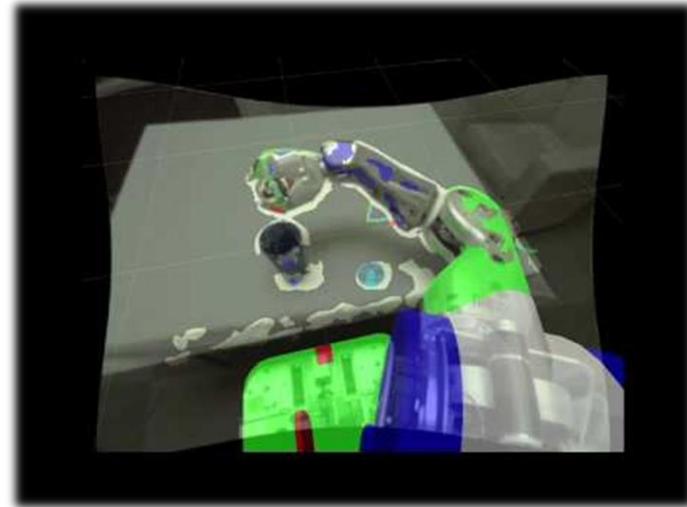


ROS Visualization



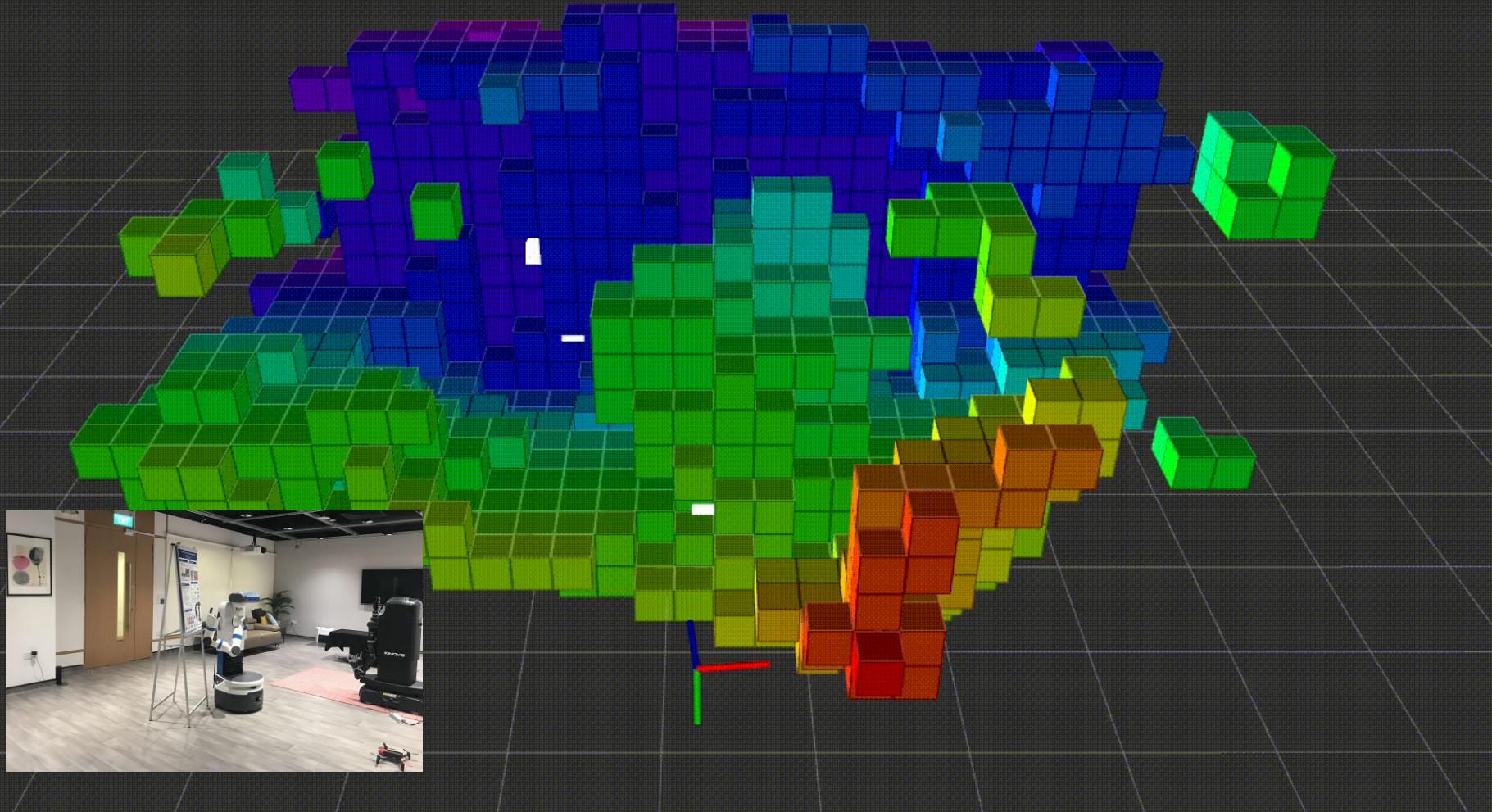
RViz is the goto (built-in) 3D visualizer for ROS

It is essentially a Node subscribing a fixed set of
Message types to be visualized





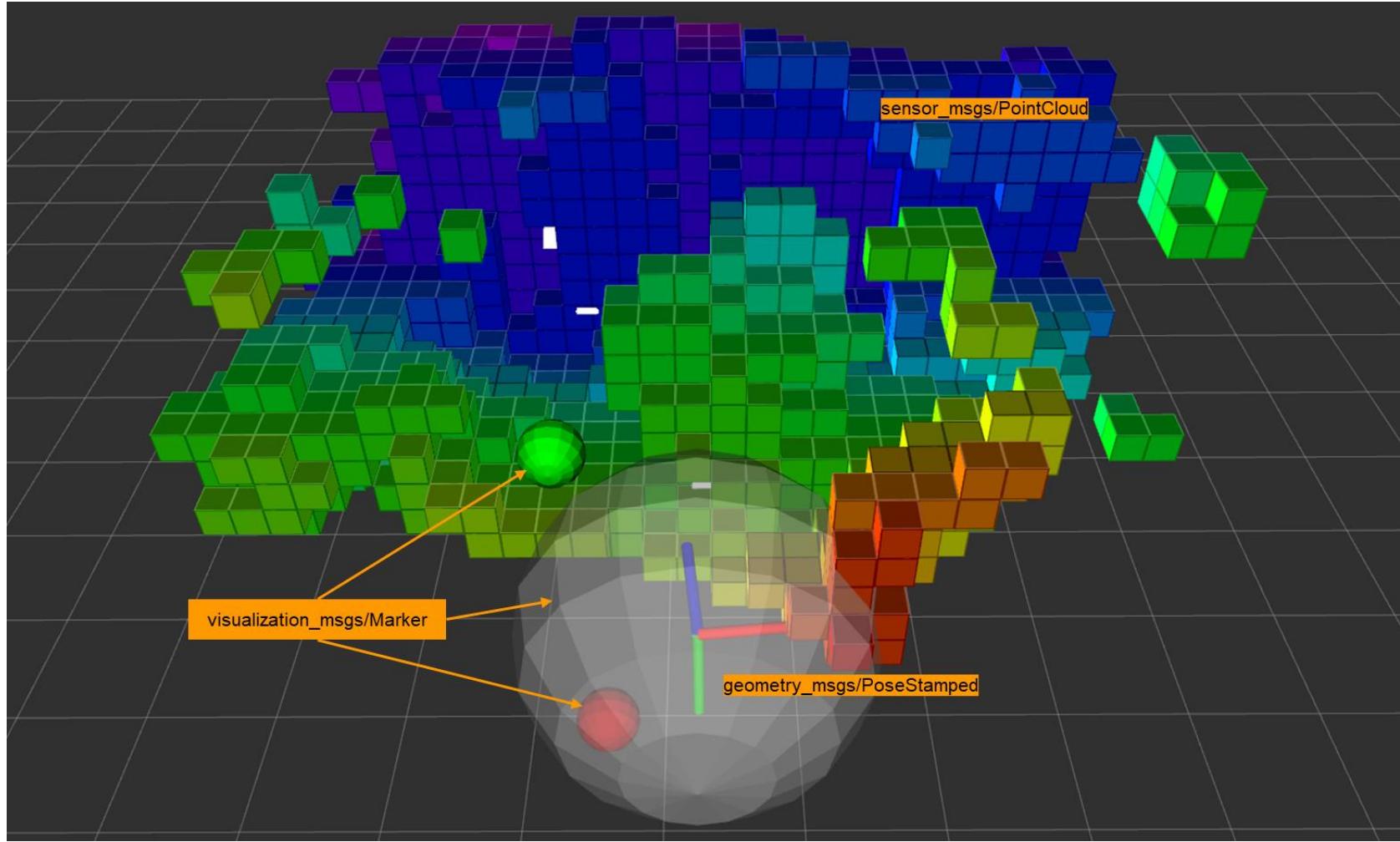
ROS Visualization (Example: Collision Avoidance)





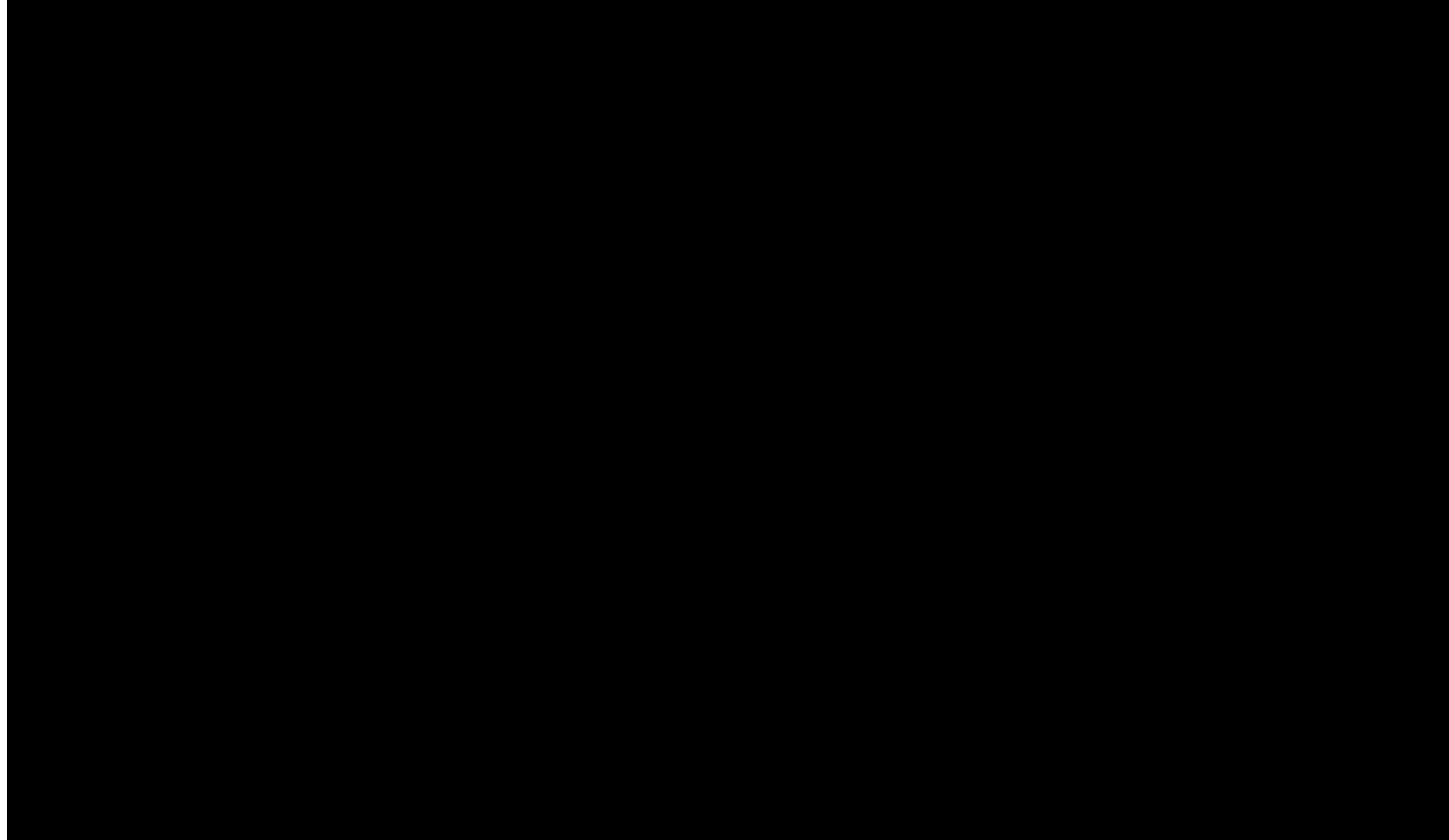
ROS Visualization

(Example: Collision Avoidance)





ROS Visualization (Example: Collision Avoidance)

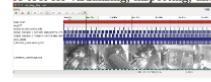
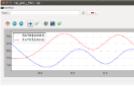


Source: <https://www.youtube.com/watch?v=pUOdwGLYmN8>



ROS cheatsheet



<p>Logging Tools</p> <p>rqt.console A tool to display and filtering messages published on rosout.</p>  <p>Usage: \$ rqt.console</p> <p>rqt.bag A tool for visualizing, inspecting, and replaying bag files.</p>  <p>Usage, viewing: \$ rqt.bag bag_file.bag</p> <p>Usage, bagging: \$ rqt.bag --press the big red record button.*</p> <p>rqt.logger.level Change the logger level of ROS nodes. This will increase or decrease the information they log to the screen and rqt.console.</p> <p>Usage: viewing \$ rqt.logger.level</p> <p>Introspection & Command Tools</p> <p>rqt.topic A tool for viewing published topics in real time.</p> <p>Usage: \$ rqt Plugin Menu->Topic->Topic Monitor</p> <p>rqt.msg, rqt.srv, and rqt.action A tool for viewing available msgs, svrs, and actions.</p> <p>Usage: \$ rqt Plugin Menu->Topic->Message Type Browser Plugin Menu->Service->Service Type Browser Plugin Menu->Action->Action Type Browser</p> <p>rqt.top A tool for ROS specific process monitoring.</p> <p>Usage: \$ rqt Plugin Menu->Introspection->Process Monitor</p> <p>rqt.publisher, and rqt.service.caller Tools for publishing messages and calling services.</p> <p>Usage: \$ rqt Plugin Menu->Topic->Message Publisher Plugin Menu->Service->Service Caller</p>	<p>rqt.reconfigure A tool for dynamically reconfiguring ROS parameters.</p> <p>Usage: \$ rqt Plugin Menu->Configuration->Dynamic Reconfigure</p> <p>rqt.graph, and rqt.dep Tools for displaying graphs of running ROS nodes with connecting topics and package dependencies respectively.</p>  <p>Usage: \$ rqt-graph \$ rqt.dep</p> <p>Development Environment</p> <p>rqt.shell, and rqt.ipc Two tools for accessing a respectively.</p> <p>Usage: \$ rqt Plugin Menu->Miscell Plugin Menu->Miscell</p> <p>Data Visualization</p> <p>view_frames A tool for visualizing the</p> <p>Usage: \$ roscore tf2.tools v \$ ariane.frames.pdf</p> <p>rqt.plot A tool for plotting data from bag files.</p>  <p>Examples: To graph the data in dir: \$ rqt.plot /topic1/f To graph the data all of: \$ rqt.plot /topic1/f To graph multiple fields: \$ rqt.plot /topic1/f \$ rqt.image.view A tool to display image topics.</p>  <p>Usage: \$ rqt.image.view</p>	<p>ROS Kinetic Catkin Workspaces</p> <p>Create a catkin workspace</p> <p>Setup and use a new catkin workspace from scratch.</p> <p>Example: \$ source /opt/ros/kinetic/setup.bash \$ mkdir -p ~/catkin_ws/src \$ cd ~/catkin_ws/src</p>	<p>ROS Kinetic Cheatsheet</p> <p>Filesystem Management Tools</p> <p>rospack A tool for inspecting packages.</p> <p>rospack profile Fixes path and pluginlib problems.</p> <p>roscd Change directory to a package.</p> <p>rospwd/rosd Pushd equivalent for ROS.</p> <p>rosis Lists package stack information.</p> <p>rosed Open requested ROS file in a text editor.</p> <p>roscp Copy a file from one place to another.</p> <p>rospack Installs package system dependencies.</p> <p>rostopic Displays a topic'summings about a running ROS system or launch file.</p> <p>catkin_create_pkgs Creates a new ROS stack.</p> <p>wstool Manage many repos in workspace.</p> <p>catkin_make Builds a ROS catkin workspace.</p> <p>rqt.dep Displays package structure and dependencies.</p> <p>Usage:</p> <pre>\$ rospack find [package] \$ rospack [package/]subdir] \$ rospack [package/]subdir +N -N] \$ rosd \$ rosis [package/]subdir] \$ rospack [file] \$ rospack [package/] [file] [destination] \$ rostopic info [topic] \$ rostopic echo [topic] \$ rostopic bw \$ rostopic hz \$ rostopic info \$ rostopic list \$ rostopic pub \$ rostopic type</pre> <p>Examples:</p> <pre>To graph the data in dir: \$ rqt.plot /topic1/f To graph the data all of: \$ rqt.plot /topic1/f To graph multiple fields: \$ rqt.plot /topic1/f</pre> <p>Start-up and Process Launch Tools</p> <p>roscore The basis nodes and programs for ROS-based systems. A roscore must be running for ROS nodes to communicate.</p> <p>Usage: \$ roscore</p> <p>rosrun Runs a ROS package's executable with minimal typing.</p> <p>Usage: \$ rosrun package_name executable_name</p> <p>Example (runs turtlesim): \$ rosrun turtlesim turtlesim_node</p> <p>rosaunch Starts a roscore (if needed), local nodes, remote nodes via SSH, and sets parameter server parameters.</p> <p>Examples: Launch a file in a package: \$ roslaunch package_name file.name.launch Launch on a different port: \$ roslaunch -p 1234 package_name file.name.launch Launch on the local nodes: \$ roslaunch --local package_name file.name.launch</p>	<p>Introspection and Command Tools</p> <p>rosnode Displays debugging information about ROS nodes, including publications, subscriptions and connections.</p> <p>Commands:</p> <pre>rosnode ping Test connectivity to node. rosnode list List active nodes. rosnode info Print information about a node. rosnode machine List nodes running on a machine. rosnode kill Kill a running node.</pre> <p>Examples:</p> <pre>Kill all nodes: \$ rosnode kill -a List nodes on a machine: \$ rosnode machine a.gy.local Ping all nodes: \$ rosnode ping --all</pre> <p>rostopic A tool for displaying information about ROS topics, including publishers, subscribers, publishing rate, and messages.</p> <p>Commands:</p> <pre>rostopic bw Display bandwidth used by topic. rostopic echo Print messages to screen. rostopic find Find topics by type. rostopic hz Display publishing rate of topic. rostopic info Print information about an active topic. rostopic list List all published topics. rostopic pub Publish data to topic. rostopic type Print topic type.</pre> <p>Examples:</p> <pre>Publish hello at 10 Hz: \$ rostopic pub -r 10 /topic.name std_msgs/String hello Clear the screen after each message is published: \$ rostopic echo -c /topic.name Display messages that match a given Python expression: \$ rostopic echo --filter "m.data=='foo'" /topic.name Pipe the output of rostopic to rosmsg to view the msg type: \$ rostopic type /topic.name rosmsg show</pre> <p>rosservice A tool for listing and querying ROS services.</p> <p>Commands:</p> <pre>rosservice list Print information about active services. rosservice node Print name of node providing a service. rosservice call Call the service with the given args. rosservice args List the arguments of a service. rosservice type Print the service type. rosservice uri Print the service ROSRPC uri. rosservice find Find services by service type.</pre> <p>Examples:</p> <pre>Call a service from the command-line: \$ rosservice call /add_two_ints 1 2 Pipe the output of rosservice to rosview to view the srv type: \$ rosservice type add_two_ints rosrv show Display all services of a particular type: \$ rosservice find rospy.tutorials/AddTwoInts</pre> <p>rosbag A set of tools for recording and playing back of ROS topics.</p> <p>Commands:</p> <pre>rosbag record Record a bag file with specified topics. rosbag play Play content of one or more bag files. rosbag compress Compress one or more bag files. rosbag decompress Decompress one or more bag files. rosbag filter Filter the contents of the bag.</pre> <p>Examples:</p> <pre>Record select topics: \$ rosbag record topic1 topic2 Replay all messages without waiting: \$ rosbag play -a demo.log.bag Replay several bag files at once: \$ rosbag play demo1.bag demo2.bag</pre> <p>tf.echo A tool that prints the information about a particular transformation between a source.frame and a target.frame.</p> <p>Usage: \$ rosrun tf tf.echo <source.frame> <target.frame></p> <p>Examples:</p> <pre>To echo the transform between /map and /odom: \$ rosrun tf tf.echo /map /odom</pre>
---	---	---	--	---

https://w3.cs.jmu.edu/spragunr/CS354_S19/handouts/ROSCheatsheet.pdf



MODULE 2 WORKSHOP: WORKSHOP DAY 1

HANDS-ON ANALYSIS OF STATE-OF-THE-ART HRI PROJECTS: INTERACTIONS WITH DOMESTIC ROBOTS



Introducing Internet-of-Things (IoT)



What is IoT?

- Internet connects all people, so it is called “the Internet of People”
- **IoT connects all things**, so it is called “the Internet of Things”
- Remember the key elements of IoT:
 1. **Connect** devices
 2. **Process** (i.e. data collection, analysis and management)
 3. **Act**



Internet-of-Things (IoT) plays which important role(s) in Human-Robot Interactions (HRI)? (you may choose more than one)



Storing Data in the Cloud
(i.e. online servers)

Data Analysis

Offsite Monitoring of
Robotic System

Offsite Control of Robotic
System (include actuators)

Offsite Alert Prompt to
Humans



IoT Supporting HRI



Offsite Real-time Monitoring via the Internet



Offsite Control via the Internet

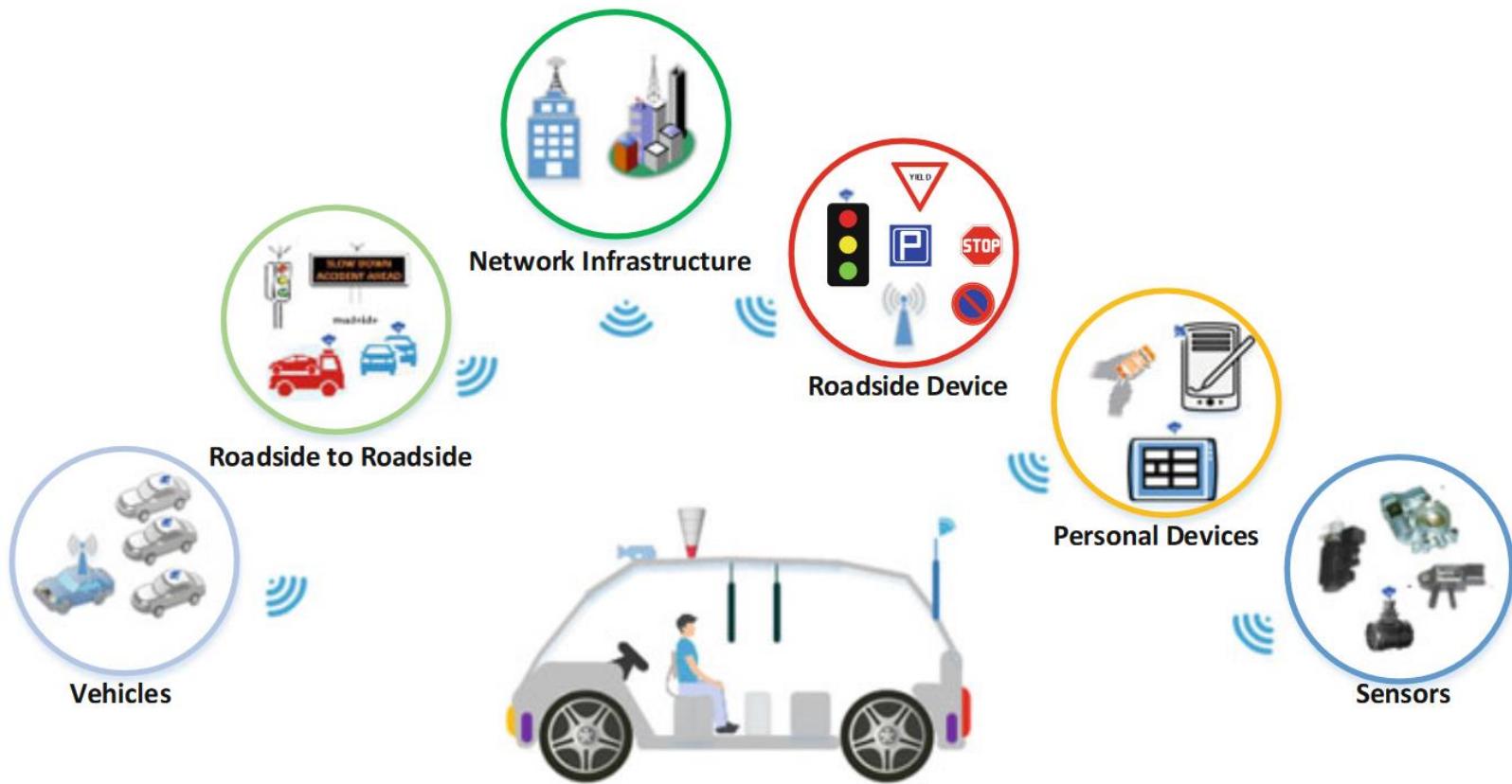


Offsite Alert Prompt to Humans





IoT can Help Shape the AV Industry



Interaction model for IoT-based ecosystem for an autonomous vehicle



IoT can Help Shape the AV Industry



V2X

Wireless tech
 $>> 200$ m

Radar/LiDAR

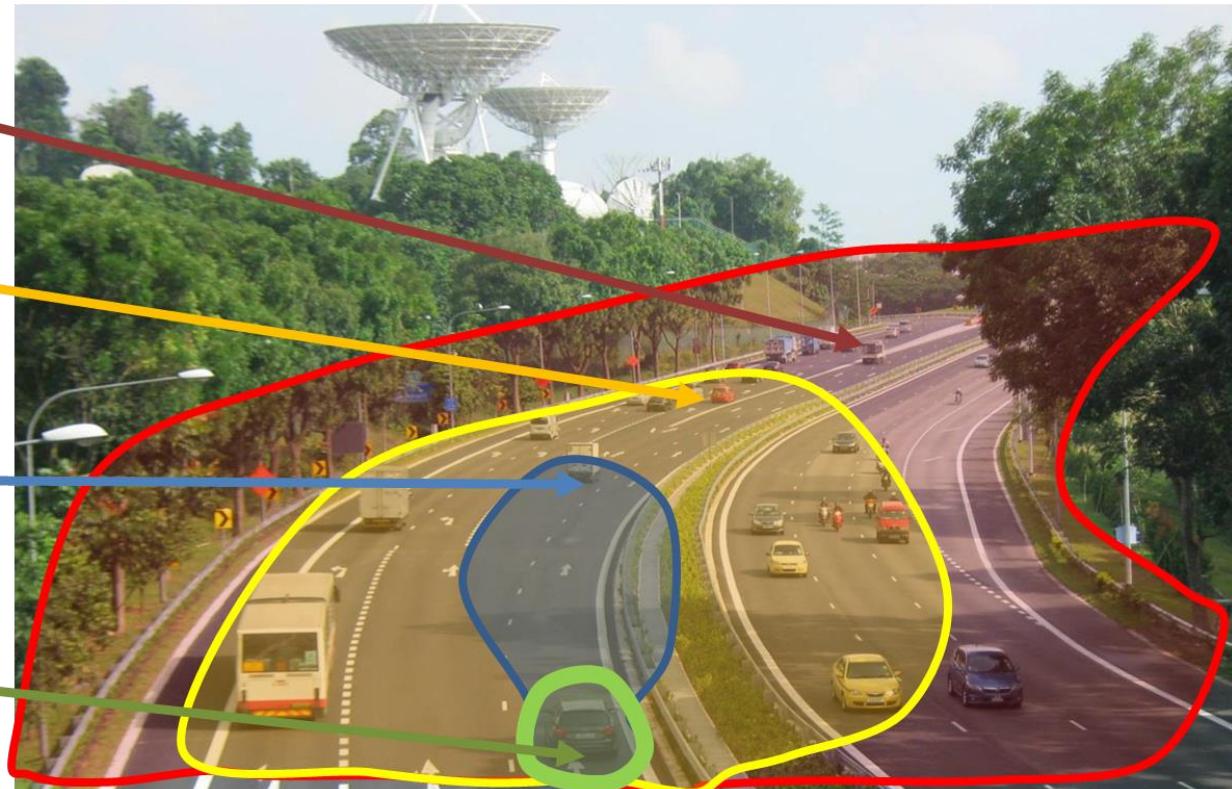
Radio waves, laser
 ≤ 200 m

Camera

Optical, infrared
 ≤ 80 m

Sonar

Ultra-sound
 ≤ 4 m

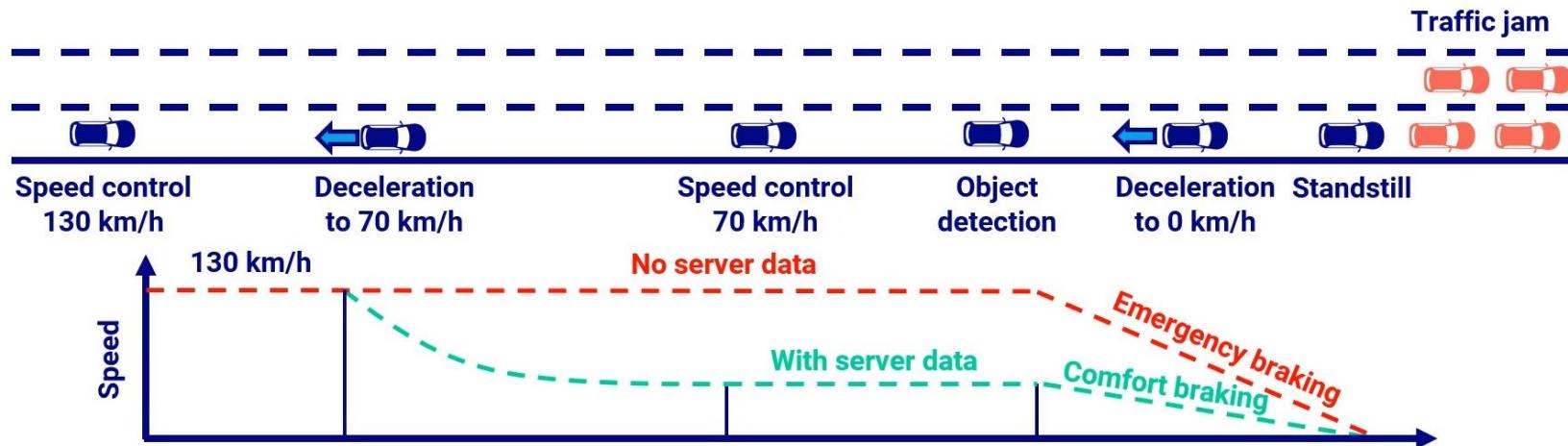




IoT can Help Shape the AV Industry



Comparison: With vs Without V2X communications

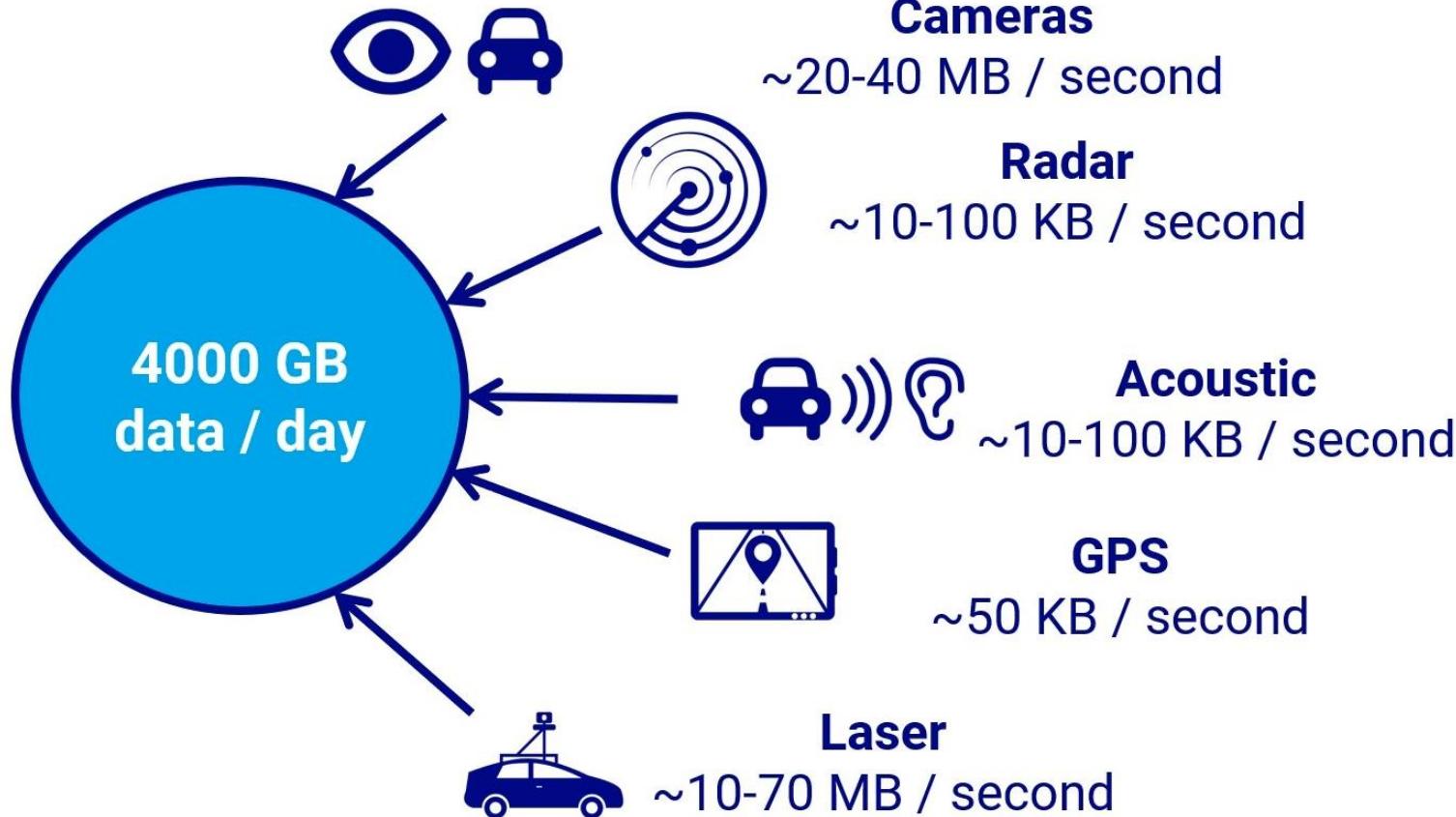




IoT can Help Shape the AV Industry

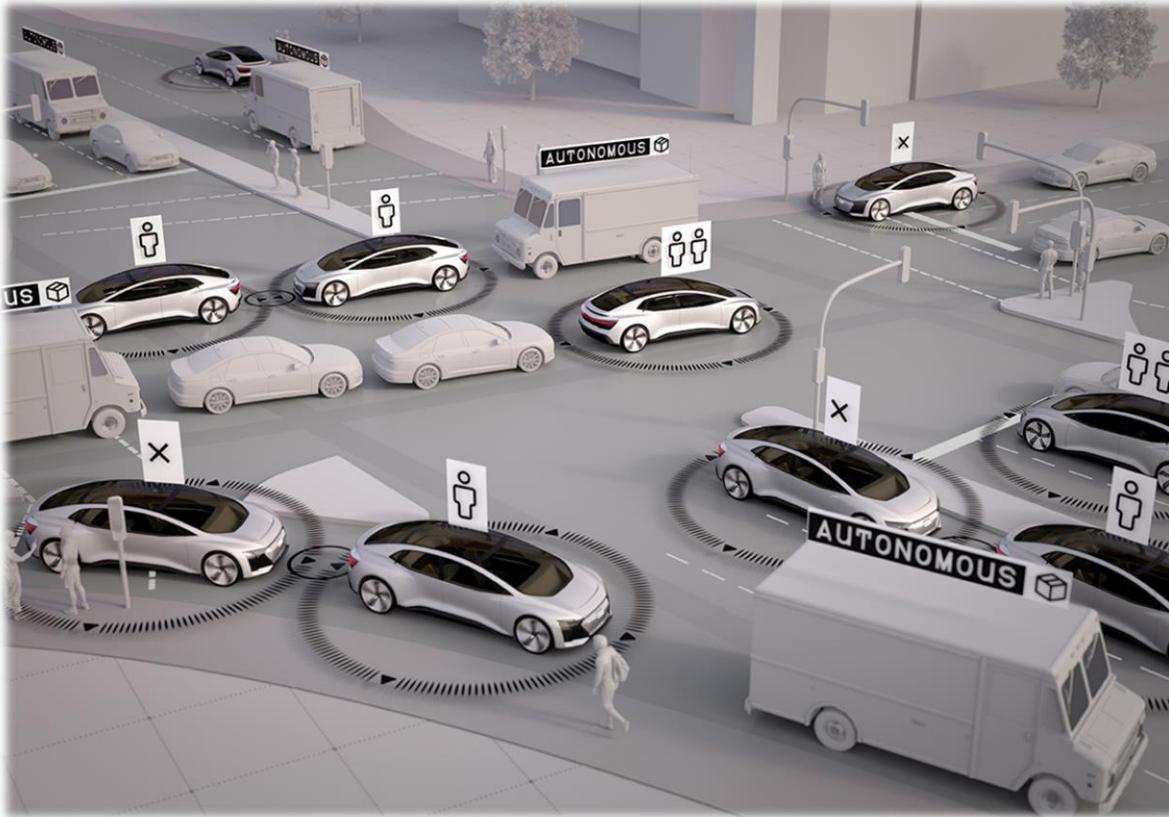


Big Data





IoT can Help Shape the AV Industry



Is the current technology good enough that we can implement this concept?

Why or Why Not?

What's the solution to this problem?



ThingSpeak (Intro)



- Main features: **Collect, Analyze, Act**
- An IoT analytics platform service that allows you to **aggregate, visualize, and analyze live data streams in the cloud**
- Able to **send data to ThingSpeak from your devices (e.g. sensors within robots)**, create instant visualization of live data, and send alerts (only Twitter for this Platform)
- **Free to public** but with limitations (limited 3 million messages and 4 channels; ok for this workshop)



Access to ThingSpeak to View Data Update

- You do not need to sign up any account for this process for now
- **Follow the steps below:**
 1. Using phone or laptop, navigate to <https://thingspeak.com/>
 2. On top of the site, click “Channels”, then point to and click on “Public Channels”
 3. Search for “Search by user ID” where there is an empty field box
 4. CopyPaste or Type my userID “**mwa0000018877967**” in this field
 5. Click “Submit” and you will see 2 boxes that you can select
 6. Depending on the Demo to be shown by the lecturer, click on the relevant demo to view the data update, i.e. click on the words “Demo for HRSE (TB3 Laser Scan)”

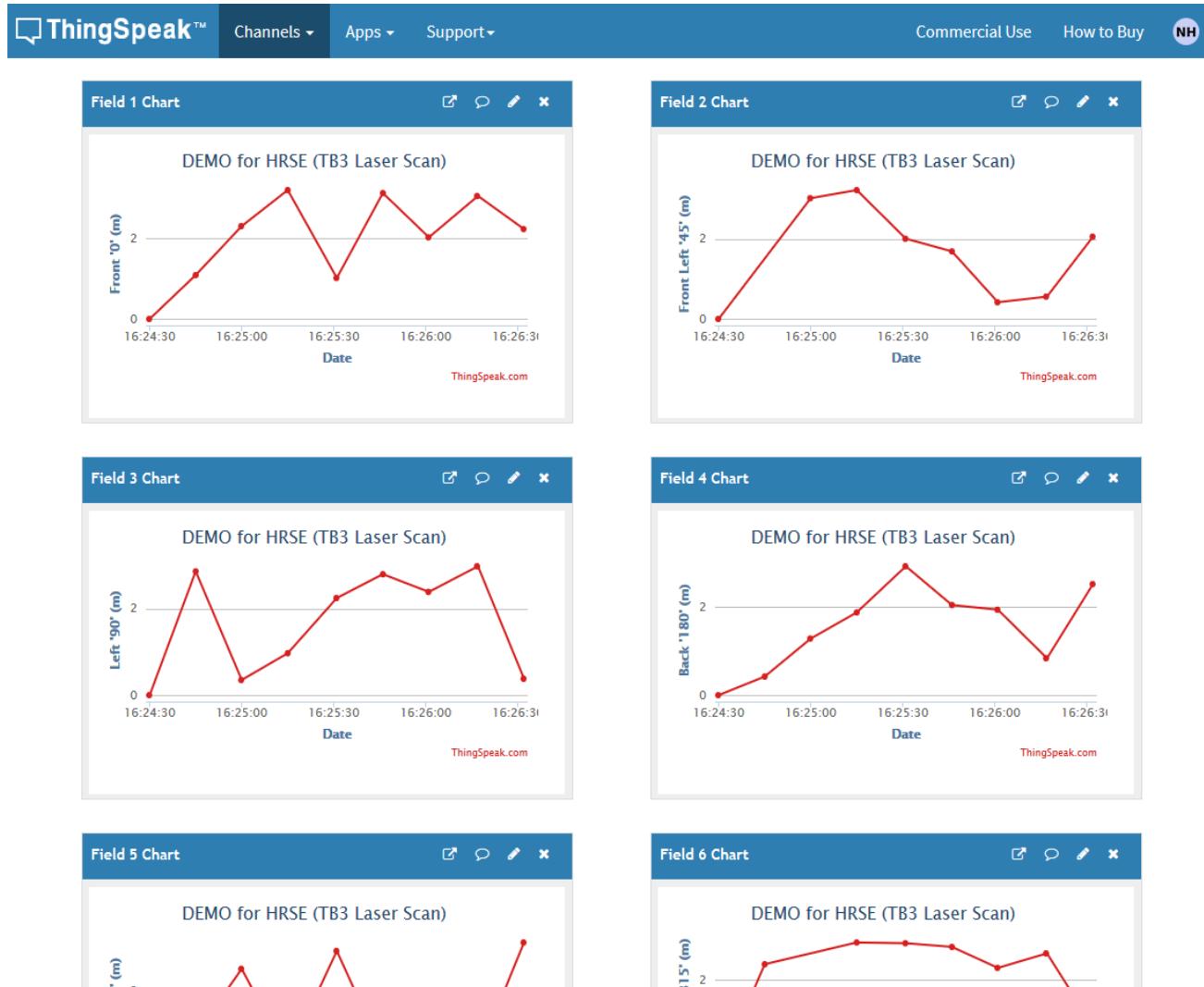




IoT Monitoring DEMO



Topic to monitor for this DEMO is **TB3 Laser Scan**





ThingSpeak (SignUp)



Follow the steps below:

- Navigate to
<https://thingspeak.com/>
- The lecturer will go through the steps on how to sign up
- Note that you will need an email at least to sign up an account (no need to be nus email)

Create MathWorks Account

Email Address

nicholasho.jh@u.nus.edu 

 To access your organization's MATLAB license, use your school or work email.

Location

Singapore 

First Name

Nicholas 

Last Name

Ho 

Continue

Cancel

Creating Channels on ThingSpeak:

1. Once your MathWorks account is created, click on ‘*Channels*’ (on top bar), and then click on ‘*My Channels*’
2. Click on the button ‘*New Channel*’
3. Type in the Name of your Channel (e.g. Monitoring of TB3 Laser Scan)
4. Click on the ticks under the field section; the number of ticks depends on how many topic messages you would like to publish in this platform (you may adjust this later on if needed; to be illustrated by lecturer)
5. Once done, scroll to bottom and click ‘*Save Channel*’



Download & Install hrse ROS Package



Steps:

1. Change to the source space directory of the catkin workspace:

```
$ cd ~/catkin_ws/src
```

2. Git Clone the hrse package:

```
$ git clone  
https://github.com/nicholashojunhui/hrse.git
```

3. Build the packages in the catkin workspace:

```
$ cd ~/catkin_ws && catkin_make
```

4. Go to *catkin_ws/src/hrse/src* and make all python files executable



Creating a new ROS Package (Optional)



Steps:

1. Change to the source space directory of the catkin workspace:

```
$ cd ~/catkin_ws/src
```

2. Use the catkin_create_pkg script to create a new package called '**new_package**' which depends on std_msgs, roscpp, and rospy:

```
$ catkin_create_pkg new_package std_msgs rospy roscpp
```

3. Build the packages in the catkin workspace:

```
$ cd ~/catkin_ws
```

```
$ catkin_make
```

4. Add the workspace to your ROS environment by sourcing the generated setup file:

```
$ . ~/catkin_ws/devel/setup.bash
```



Group Project 1

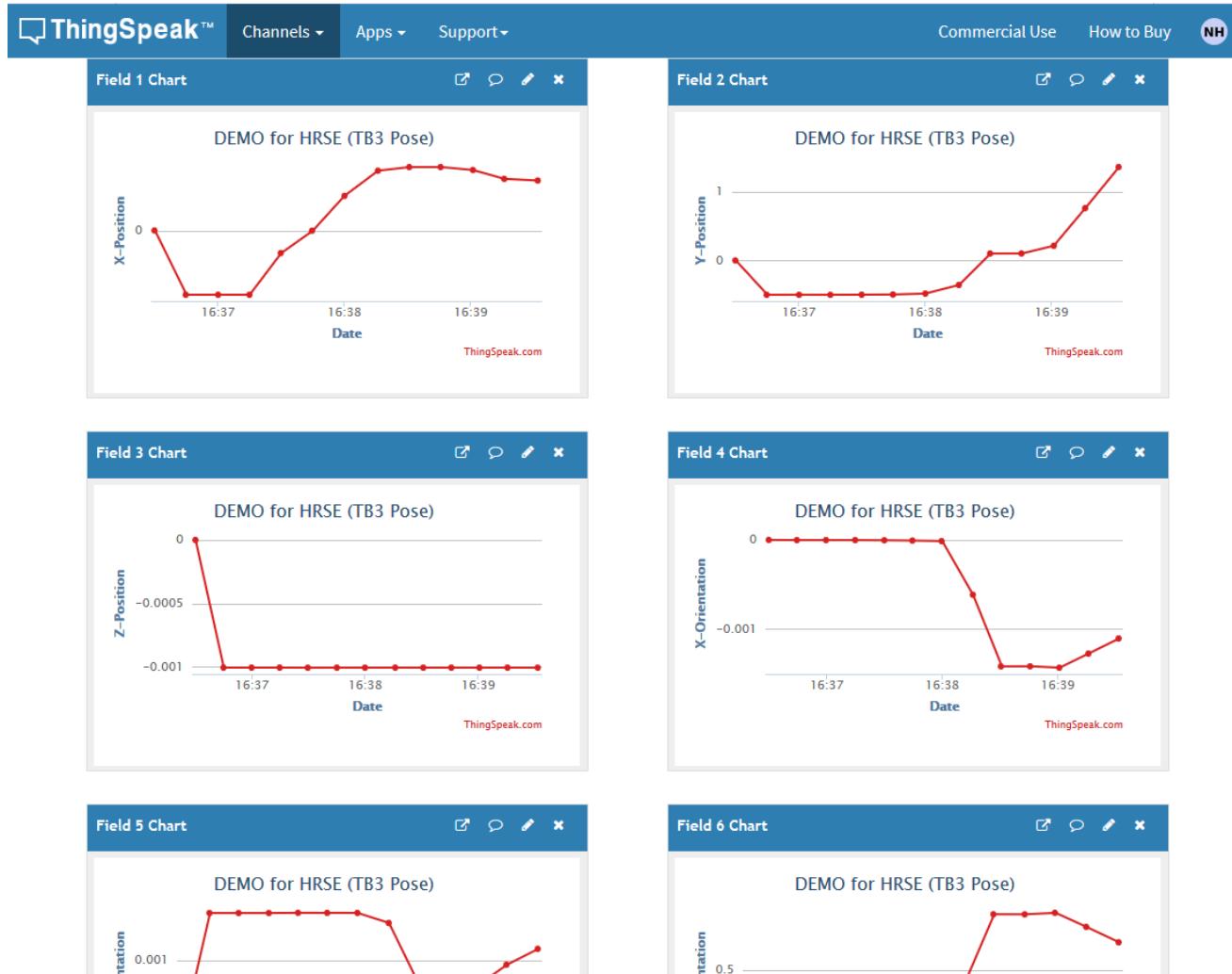


IoT plays an important role within systems that involve human-robot interactions. Be it monitoring, control, big data analysis, data management and storage, emergency alerts from the robots, etc, the key advantage of IoT is that it enables us humans to communicate with the robots via the Internet, hence we do not have to be onsite or physically near the robot for these communications.

For this project, we will be focusing on the real-time and retrospective IoT monitoring aspect. Instead of the example shown previously to monitor laser scans (i.e. LiDAR readings from the TB3), this time you have to **configure an IoT system that enables us to monitor the pose parameters taken from the /odom topic** (i.e. the x, y, z positions and x, y, z, w orientations of the TB3). Similarly like in the demo, **use rospy and ThingSpeak in your configurations.**



Group Project 1: Ideal Result for HRSE (TB3 Pose)





Group Project 1: Ideal Result for HRSE (TB3 Pose DEMO)

Follow the steps below:

1. Using phone or laptop, navigate to <https://thingspeak.com/>
2. On top of the site, click “*Channels*”, then point to and click on “*Public Channels*”
3. Search for “Search by user ID” where there is an empty field box
4. CopyPaste or Type my userID “**mwa0000018877967**” in this field
5. Click “Submit” and you will see 2 boxes that you can select
6. Depending on the Demo to be shown by the lecturer, click on the relevant demo to view the data update, i.e. click on the words “Demo for HRSE (TB3 Pose)”



Group Project 1: Setup Instructions



1. Create Channel and the required details (e.g. field section) in your ThingSpeak channel
2. In the created ThingSpeak channel, take note of your Channel ID and Write API Key; you may copy and paste these details into a notepad first
3. In your Ubuntu platform, in terminal, type the following commands (1 by 1) to install the thingspeak library; you need this to publish the data to the channel(s)
 - \$ sudo apt-get update
 - \$ sudo apt-get upgrade
 - \$ sudo pip install thingspeak



Group Project 1: RECAP on Sequences (Refer to README for detailed instructions)

1. Launch Gazebo in TB3 World
2. Run node to send required data to TS Channel (node to be developed in this project: Project1.py)
3. Check if your ThingSpeak Channel receives the data from your running node
4. Move TB3 around using the Teleop key node
5. The collected data should change as the TB3 moves around the map
6. Once done, export the collected data in CSV



Instructions



This is a group project. Each group submits one zip file of all your codes/files (i.e. py and CSV) into LumiNUS at the end of the workshop

A123456_A234567_A345678_P1.zip

- Download all files in the directory **/workshops/day1** for reference codes
- Refer to the README file for instructions



OPTIONAL TASKS

**PHYSICAL TB3 TEST, IOT MONITORING OF IMU,
IOT CONTROL OF TB3**



(A) Physical TB3 Test

RECAP on Sequences

(Refer to README for detailed instructions)

1. Do the OpenCR Setup (if have not done so)
2. Run roscore and do bring up procedures
3. Run node to send required data to TS Channel
(i.e. Project1.py)
4. Check if your ThingSpeak Channel receives the data from your running node
5. Move TB3 around using the Teleop key node
6. The collected data should change as the TB3 moves around the map





(B) IoT Monitoring of IMU

Just for practice ONLY if you have time

- Create a node (i.e. imu.py) that is able to send data to ThingSpeak for monitoring purposes
- You may create a new channel for this
- **Interested variables:** Angular Velocities and Linear Accelerations at the various directions
- **Hint:** you may use rqt's topic monitor to guide you to understand which topic and its respective parameters are relevant



(C) IoT Control



- Apart from IoT Monitoring, IoT Control enables us to **control the robot at an offsite location**
- We will explore using an Android App (i.e. **ROS Control**) for this example
- Note that you will require an **Android Device (i.e. phone, tablet)** to do this
- Although this example is not considered as a complete IoT system (using TCP/IP), it can be easily configured to be one



Setup:

1. Download and install the following app in your Android device:
<https://play.google.com/store/apps/details?id=com.robotca.ControlApp>
2. If you are using VirtualBox for your platform, go to Settings, Network, then change from 'NAT' to 'Bridged Adapter' mode
3. Ensure both your device and Ubuntu is connected to the same router
4. Within your Ubuntu, open terminal and type `$ ifconfig` to find out your IP address (i.e. 192.168.X.XXX)
5. Type command `$ nano ~/.bashrc` and press `alt+/` to go to end of line or you can scroll manually to the end of the line
6. Modify the IP address under `ROS_MASTER_URI` and `ROS_HOSTNAME` by replacing `localhost` with 192.168.X.XXX



Setup (Cont):

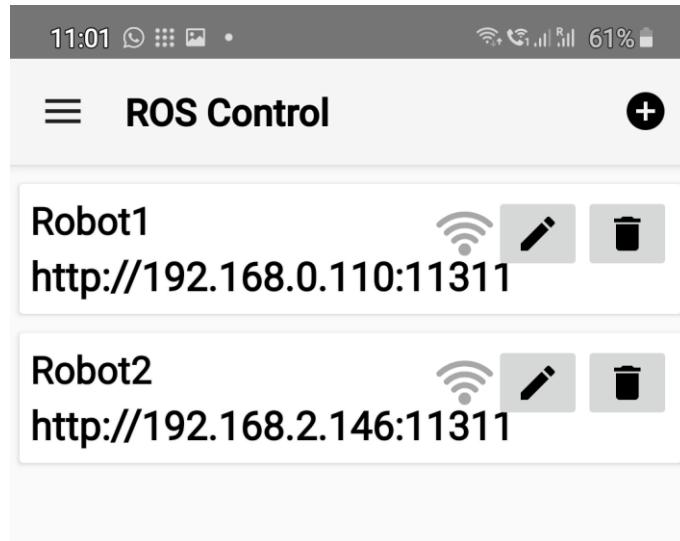
7. Once done, type ^X to exit and source the bashrc with \$ source ~/.bashrc
8. Once done, go to your downloaded app and add a robot (Robot name can be anything you want)
9. Refer to next slide for the other details to fill in (i.e. Master URI, Topic Names)



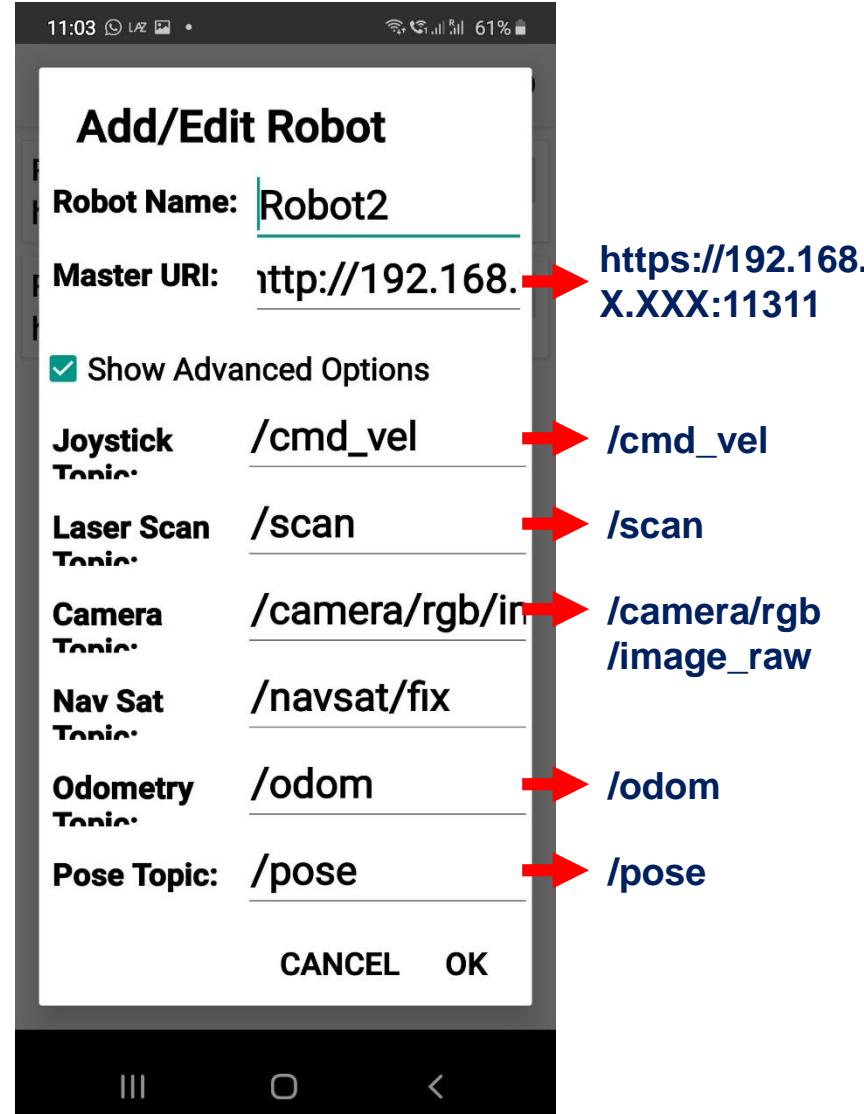
(C) IoT Control



Setup (Cont):



Can be used for both Virtual and Physical Robot; note that camera topic is different for virtual and physical robots





Execution (for Virtual TB3):

1. Launch Gazebo in TurtleBot3 world
2. Launch ROS Control Android App and under home, click on the robot that you have created (e.g. Robot1, Robot2)
3. A User Interface will appear which allows you to control the TB3 with the joystick (next slide)

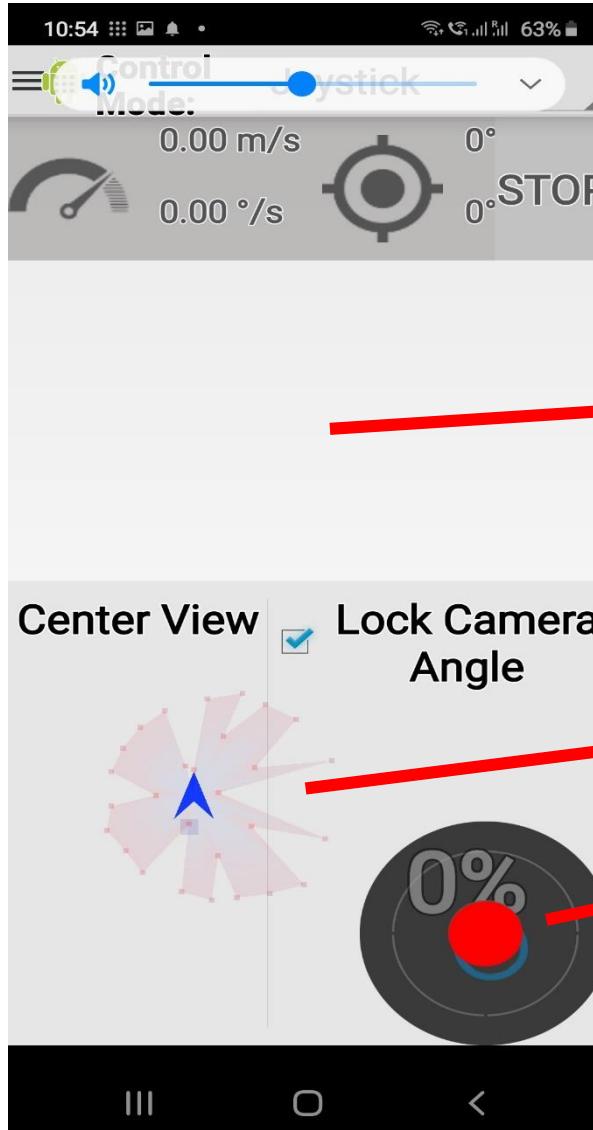
Take note that the app is very buggy; unable to load the camera data and will occasionally break connection. It is still not robust enough



(C) IoT Control



Execution (for Virtual TB3):





THANK YOU

Email: nicholas.ho@nus.edu.sg