COMP3431

Robot Software Architectures

UNSW
THE UNIVERSITY OF NEW SOUTH WALES

Week 2 – ROS Continued

ROS Continued

What we're doing today:

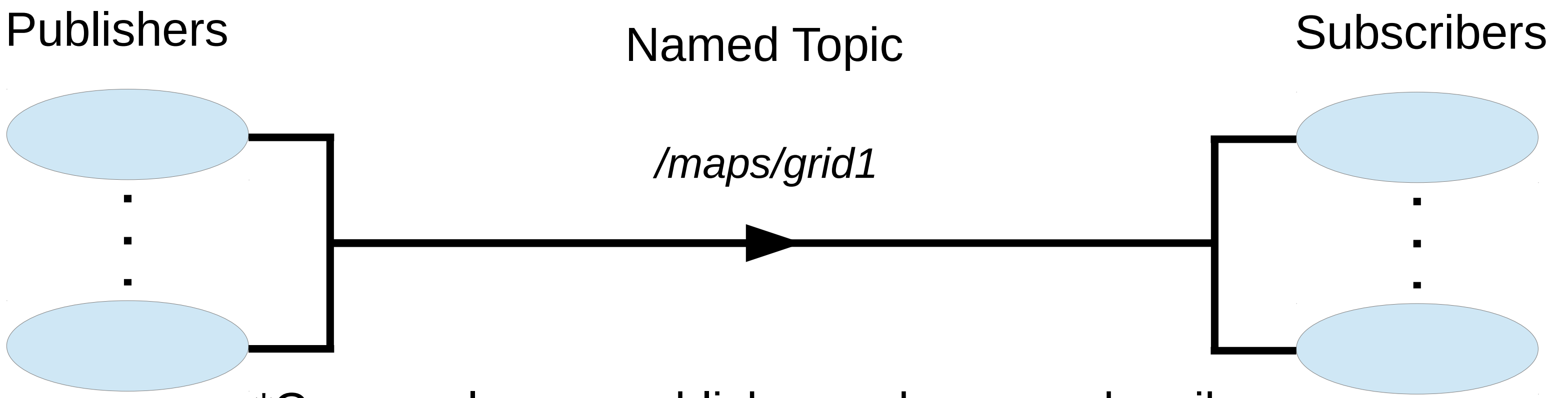
- Recap from last week
- Example of turtlebot setup
- Frames of Reference
- Closer look at different ROS tools
- Sensors
- In-class exercise

ROS Recap

- Peer-to-peer comms for distributed processes (*nodes*).
- Library of drivers, filters (e.g., mapping), behaviours (e.g., navigation).
- Not real-time.
- Multi-language support:
  - APIs for Python, C++, and Lisp; also support for Java, C#, and others.

ROS Recap – Basics

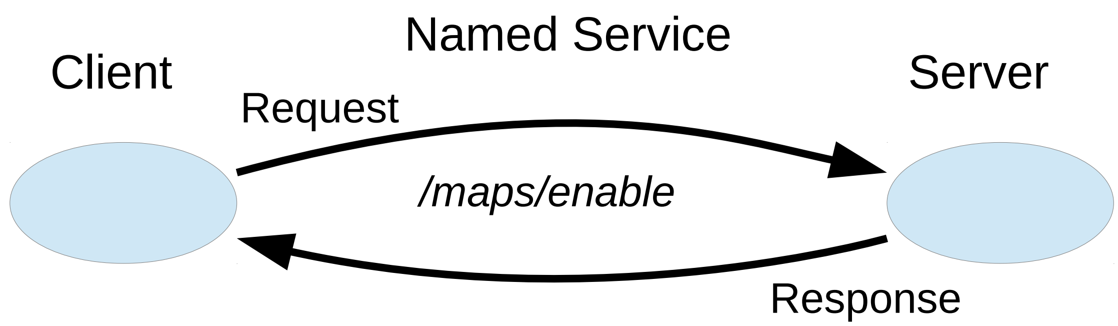
- ROS Nodes - registration at process startup.
- Two models of comms between nodes:
  - ROS Topics: Publisher-subscriber (many-to-many).



*Commonly: one publisher and many subscribers

ROS Basics

- ROS Nodes - registration at process startup.
- Two models of comms between nodes:
  - ROS Topics: Publisher-subscriber (many-to-many).
  - ROS Services: remote procedure call (one-to-one).



Nodes in a Distributed System

- Nodes can be on different computers.
- Requires some care:
  - Turn off local firewalls
  - Evironment variables to specify addresses of nodes and master:
    - ROS_MASTER_URI - location of the master.
    - ROS_IP/ROS_HOSTNAME - node registers with master using this value.
  - Safest to use IP addresses (not hostnames).

```
export ROS_MASTER_URI=http://192.168.1.2:11311
export ROS_IP=192.168.1.5
```

Turtlebot3 Basic Setup

The Turtlebot3's computer is limited so we want to off-load as much processing as possible to an external workstation (or VM).

Turtlebot3
IP: 192.168.1.10

Workstation/VM
IP: 192.168.1.20

Turtlebot3 Basic Setup – Step 1

Set ROS_MASTER_URI and ROS_IP (or ROS_HOSTNAME) for all terminals on each computer.

```
tb3/ws$ export ROS_MASTER_URI=192.168.1.200:11311
```

Turtlebot3
IP: 192.168.1.10

```
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.10
```

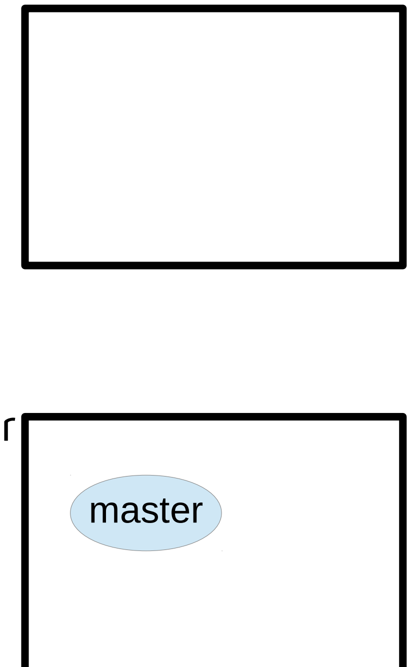
Workstation/VM
IP: 192.168.1.20

```
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.20
```

Turtlebot3 Basic Setup – Step 2

Spawn master in new terminal on workstation:

```
ws$ roscore
```



Turtlebot3
IP: 192.168.1.10

```
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.10
```

* **`roscore`** spawns master but also parameter server and logging outputs (not shown here).

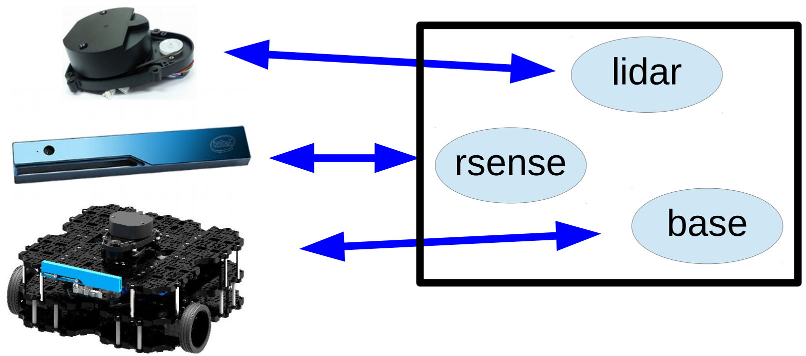
Workstation/VM
IP: 192.168.1.20

```
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.20
```

Turtlebot3 Basic Setup – Step 3

Run turtlebot3 startup in terminal on robot:

```
tb3$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

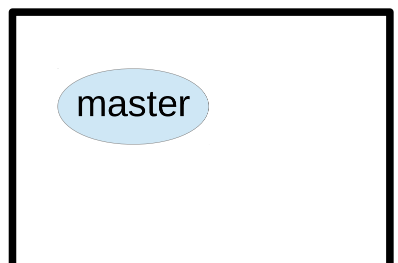


Turtlebot3
IP: 192.168.1.10

```
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.10
```

What this does:

- Spawns nodes to talk to hardware



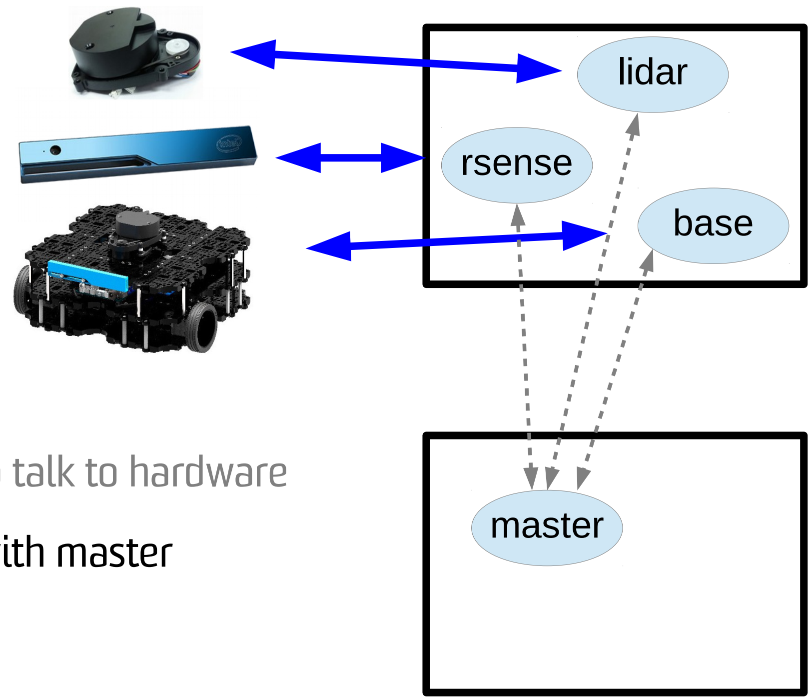
Workstation/VM
IP: 192.168.1.20

```
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.20
```

Turtlebot3 Basic Setup – Step 3

Run turtlebot startup in terminal on robot:

```
tb3$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```



Turtlebot3
IP: 192.168.1.10

```
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.10
```

Workstation/VM
IP: 192.168.1.20

```
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.20
```

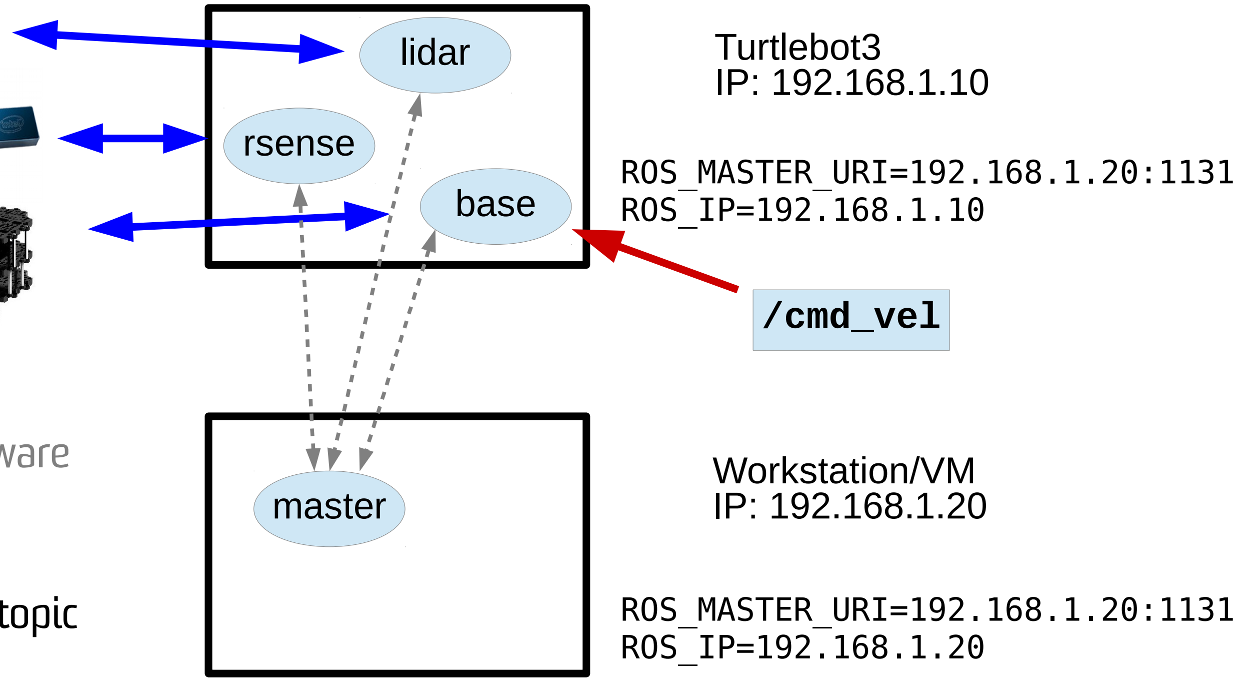
What this does:

- Spawns nodes to talk to hardware
- Nodes register with master

Turtlebot3 Basic Setup – Step 3

Run turtlebot startup in terminal on robot:

```
tb3$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

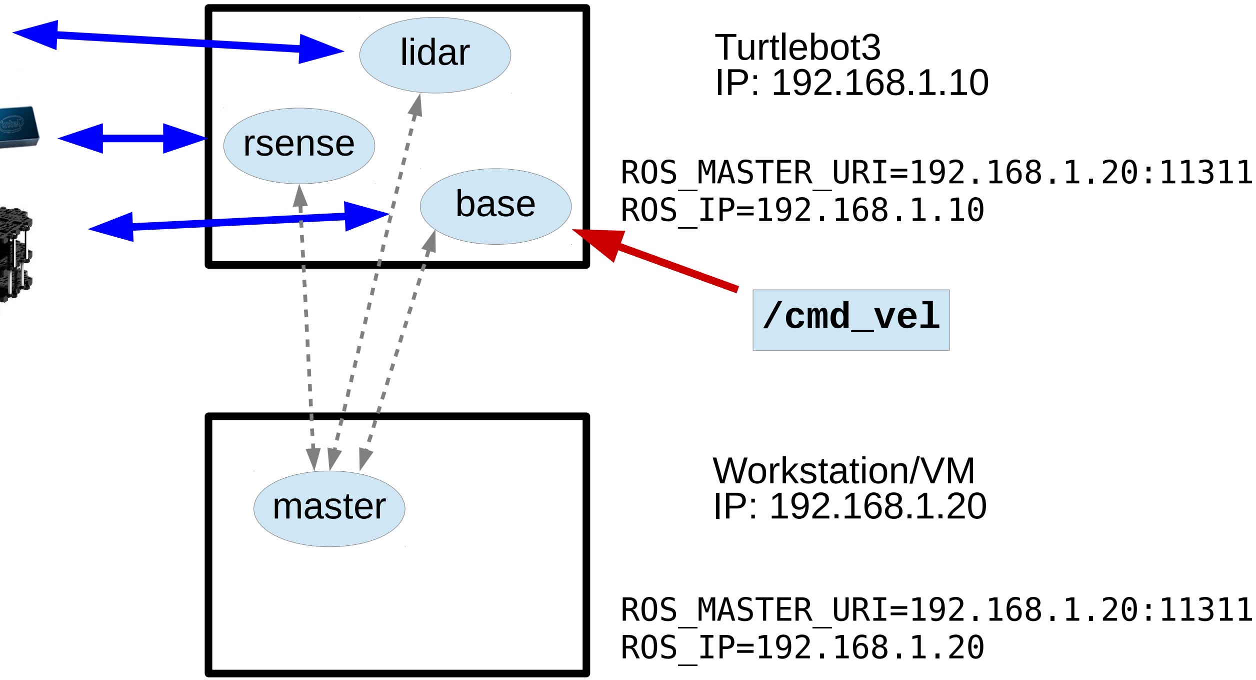


What this does:

- Spawns nodes to talk to hardware
- Nodes register with master
- **base** subscribes to **`/cmd_vel`** topic

Turtlebot3 Basic Setup

This is the basic setup. Everything else builds on this:

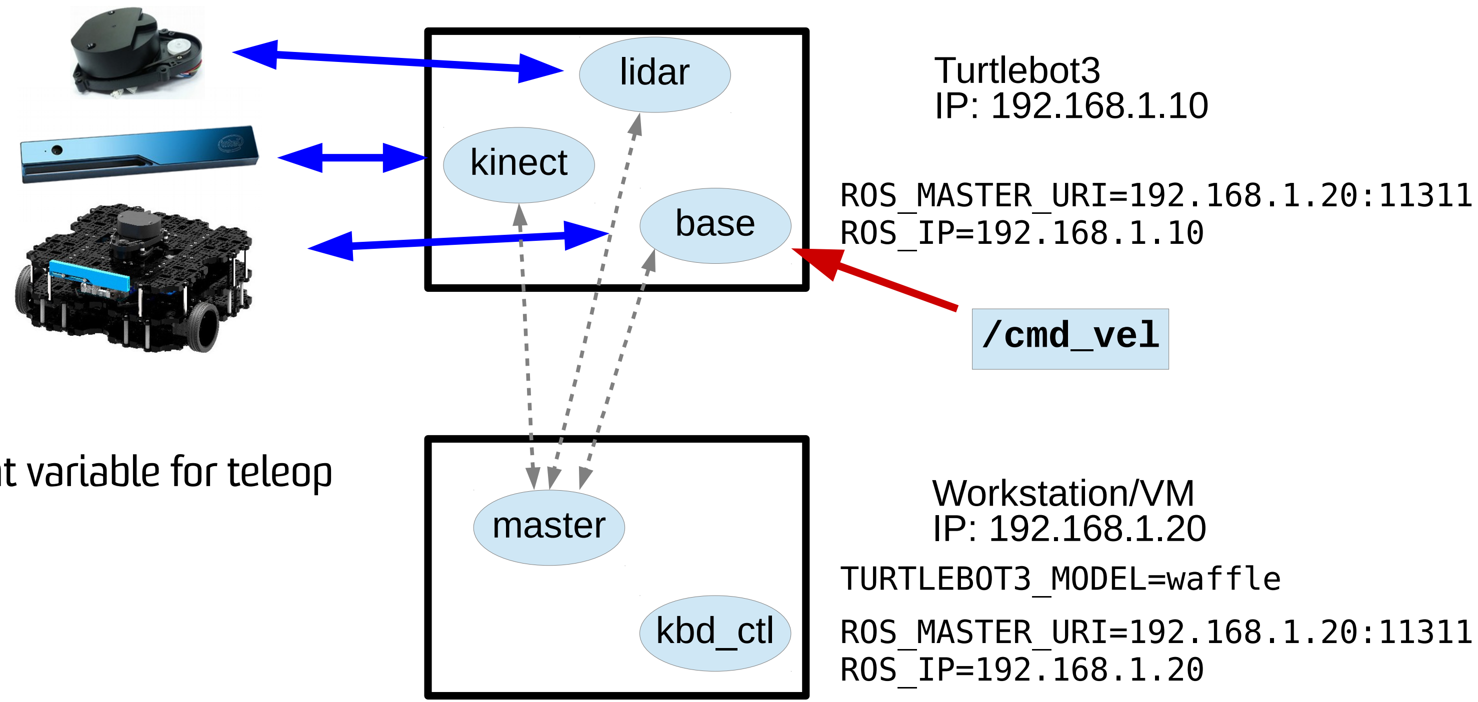


- Keyboard teleoperation
- Visualisation using rviz
- Mapping (SLAM)
- Autonomous operations

Turtlebot3 Teleop – Step 4

Set the turtlebot3 type on the workstation:

```
ws$ export TURTLEBOT3_MODEL=waffle
```



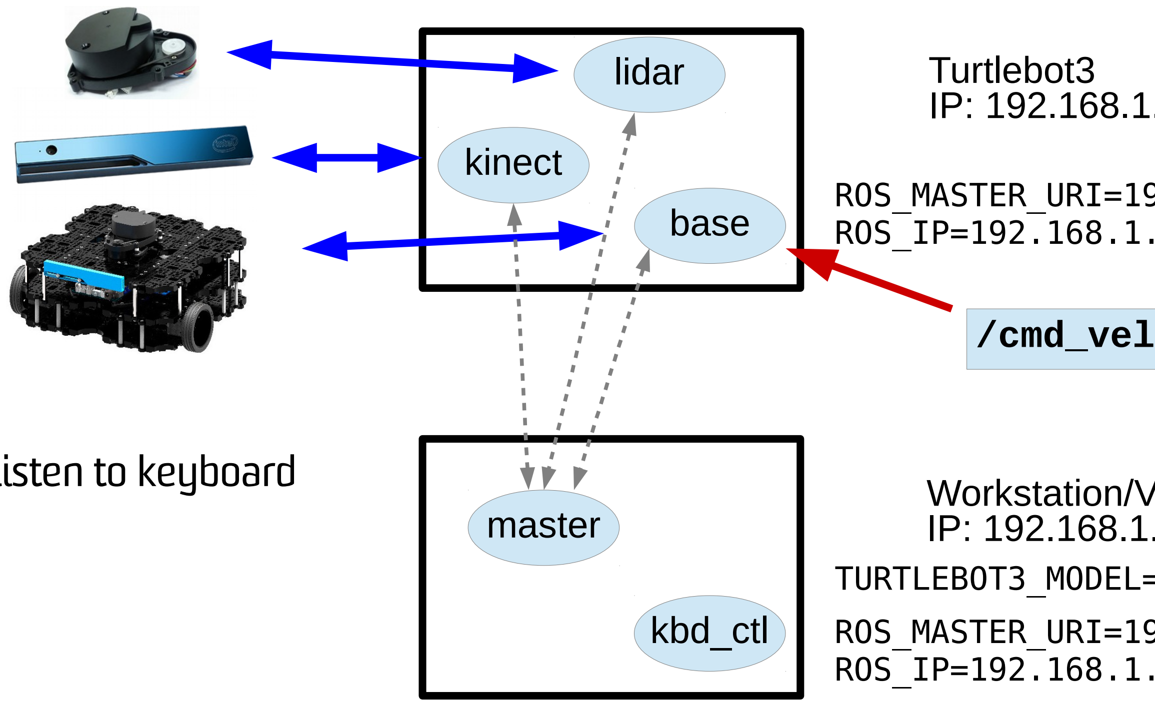
What this does:

- Sets environment variable for teleop

Turtlebot Teleop – Step 5

Run turtlebot teleop in workstation terminal:

```
ws$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```



Turtlebot3
IP: 192.168.1.10

```
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.10
```

Workstation/VM
IP: 192.168.1.20

```
TURTLEBOT3_MODEL=waffle
ROS_MASTER_URI=192.168.1.20:11311
ROS_IP=192.168.1.20
```

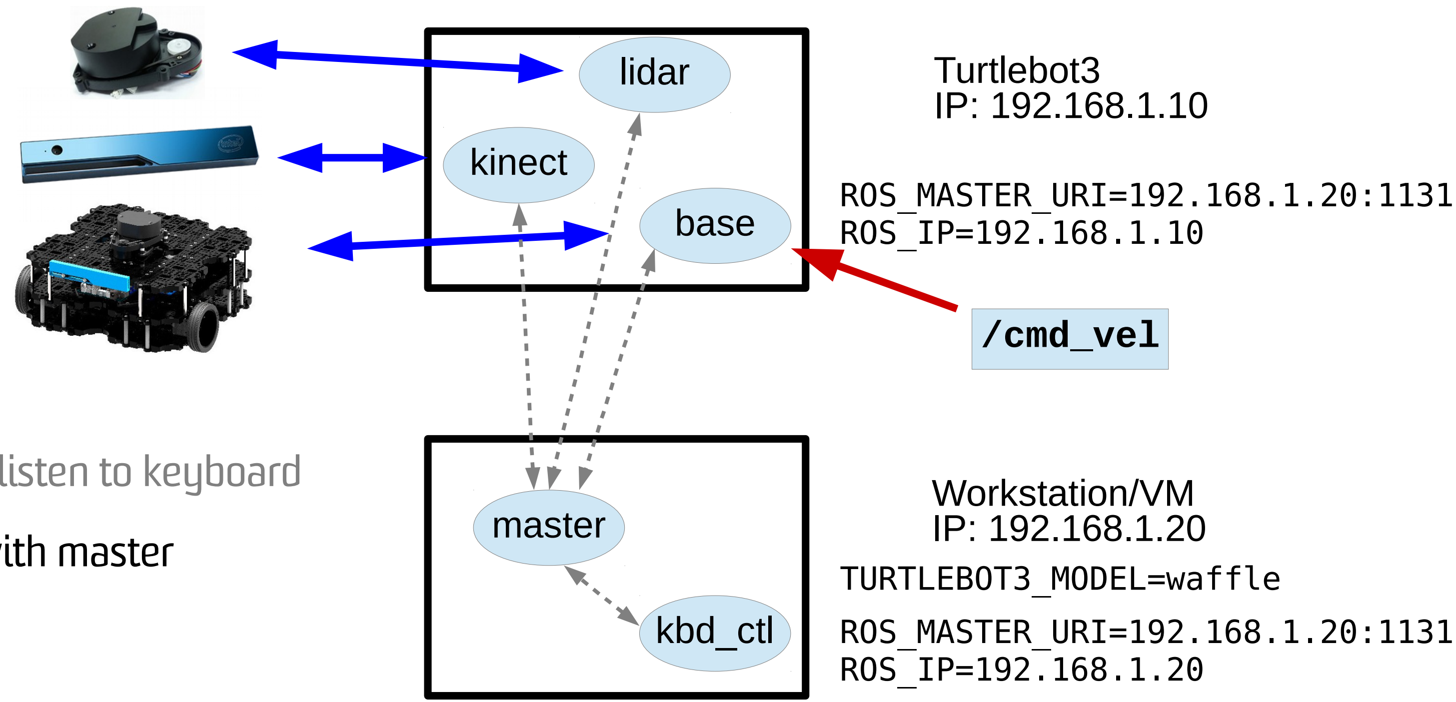
What this does:

- Spawns node to listen to keyboard

Turtlebot Teleop – Step 5

Run turtlebot teleop in workstation terminal:

```
ws$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```



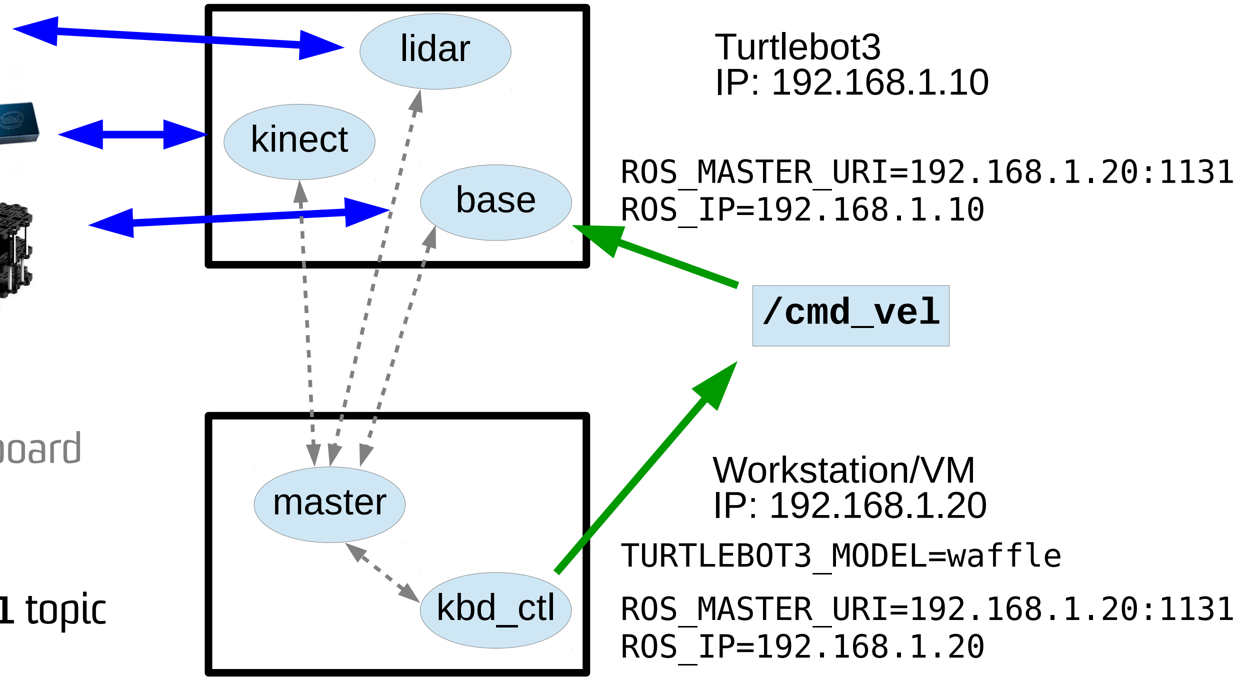
What this does:

- Spawns node to listen to keyboard
- Node registers with master

Turtlebot Teleop – Step 5

Run turtlebot teleop in workstation terminal:

```
ws$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

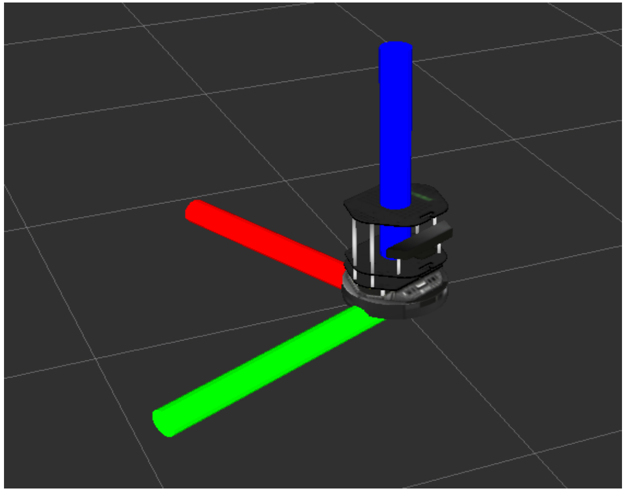


What this does:

- Spawns node to listen to keyboard
- Node registers with master
- **kbd_ctl** publishes to **`/cmd_vel`** topic

Frames of Reference

- ROS standardises the transformation model between different coordinate frames of reference.
- Right Hand Rule, X forward (XYZ ↔ RGB)
- Tree structure:
  - /map
    - /base_link
      - /base_footprint
      - /laser



- Example: laser detected object is relative to **laser** frame. Need to transform to **map** coordinate to know where it is on the map.

ROS Tools and Programs – 1

- Often first thing you run:

```
$ roscore
```

  - Spawns ROS master – already explained
  - Creates a logging node (listening on topic **/rosout**).
  - Parameter server (http://wiki.ros.org/Parameter%20Server):
    - Shared dictionary for storing runtime parameters
    - Provides flexibility for storing configuration data
    - Hierarchical structure (don't confuse with topic names or frames).
    - Allows private names – configuration specific to a single node.

ROS Tools and Programs – 2

- What is the difference between **roslaunch** and **rosrun?**

ROS Tools and Programs – 2

- What is the difference between **roslaunch** and **rosrun?**
- What is going on when I run:

```
$ roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

  - If ROS_MASTER_URI is local and no ROS master is running, then run **roscore**.
  - Execute instructions in **turtlebot3_robot.launch** in **turtlebot3_bringup/launch** directory (for syntax of launch file see http://wiki.ros.org/roslaunch/XML)
    - A weird mix of XML and shell scripting
    - … let's look at **turtlebot3_bringup/launch/turtlebot3_robot.launch**
    - **node** tag in **includes/lidar.launch** executes rosrun with appropriate parameters.

```
$ rosrun hls_lfcd_lds_driver hlds_laser_publisher _frame_id:=”base_scan” ...
```

- Note: the “_” - for private parameters.

ROS Tools and Programs – 3

- To debug the connections between nodes use:

```
$ rqt_graph
```

  – Visualises the node graph – and topic connections
- Rviz is the main visualisation tool for ROS:

```
$ rosrun rviz rviz
```

  – Provides plugins architecture for visualising different topics:
    - Videos
    - Map of environment and localised robot
    - Point cloud within the map
- Example: https://www.youtube.com/watch?v=25nnJ64ED5Q

ROS Tools and Programs – 4

- Possible to save the data produced by topics for later analysis and playback:

```
$ rosbag record -a
```

  - Creates a time stamped bag file in the current directory.
  - Warning: "-a" records all topics so will generate a lot of data.
- Often useful to only record only direct sensor inputs (e.g., laser scans and timing) because the other topics will be generated from processing sensor data.
- To replay:

```
$ rosbag play <bagfile>
```

- Useful if you are testing different interchangeable node (e.g., mapping with gmapping, hector SLAM, or different crosbot SLAM options).
- Note: SLAM (Simulataneous Localisation and Mapping) algorithms build a map while at the same time localising. Very widely used in robotics.

ROS Tools and Programs – 4

- Possible to save the data produced by topics for later analysis and playback:

```
$ rosbag record -a
```

  - Creates a time stamped bag file in the current directory.
  - Warning: "-a" records all topics so will generate a lot of data.
- Often useful to only record only direct sensor inputs (e.g., laser scans and timing) because the other topics will be generated from processing sensor data.
- To replay:

```
$ rosbag play <bagfile>
```

- Useful if you are testing different interchangeable node (e.g., mapping with gmapping, hector SLAM, or different crosbot SLAM options).
- Note: SLAM (Simulataneous Localisation and Mapping) algorithms build a map while at the same time localising. Very widely used in robotics.

Many Different Sensors

- Laser Scanner
- Camera
- IR Cameras
- Depth Cameras
- Motor
- Pressure Sensor
- Compass
- Accelerometer
- IMU (Inertial Measurement Unit) – detects linear acceleration using accelerometer and rotation using gyroscope
- Audio

ROS provides standardised data structures for some of these sensors.

Laser Scanners

- A laser is rotated through a plane
- Distance (& intensity) measurements taken periodically
- 180-270 degrees

sensor_msgs/LaserScan

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Cameras

- Stream images
- Various encodings used (RGB, Mono, UYVY, Bayer)
- ROS has no conversion functions

```
sensor_msgs/Image

std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step
uint8[] data
```

```
#include <sensor_msgs/image_encodings.h>
```

Depth Cameras

- Usually produce Mono16 images
- Typically turned into point clouds
- Depth measurements can be radial or axial

sensor_msgs/PointCloud

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
geometry_msgs/Point32[] points
  float32 x
  float32 y
  float32 z
sensor_msgs/ChannelFloat32[]
channels
  string name
  float32[] values
```

Motor Positions

- Many motors report their positions
- Used to produce transformations between frames of reference

sensor_msgs/JointState

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
string[] name
float64[] position
float64[] velocity
float64[] effort
```

In-Class Examples

- Simple publisher and subscriber:
  - Class member function callbacks.
  - Use Timer to publish at a specific rate.