ROS 2 Cheat Sheet

Robotics Content Lab

www.roboticscontentlab.com

ROS 2 Command Line Interface

All ROS 2 Command Line Interface (CLI) commands start with the **ros2** command which is followed by a **verb** or **noun** and possible <*arguments*>.

For any CLI command you can use the `--help` or `-h` arguments to receive further usage documentation. E.g.:

```
$ ros2 --help
$ ros2 run --help
```

Finally, auto-completion for any command and argument should be enabled via the ⇥ (tab) key:

```
$ ros2 param ⇥ ⇥
```

ROS 2 Node Management

A node is a process that performs computation. It can publish and subscribe to topics, provide and use services, and send and receive actions [1].

Commands

List all active nodes:

```
$ ros2 node list
```

Display detailed information about a node:

```
$ ros2 node info <node_name>
```

Set the lifecycle state of a node (e.g., inactive to active):

```
$ ros2 lifecycle set <node> <state>
```

Get the current lifecycle state of a node:

```
$ ros2 lifecycle get <node>
```

Examples

Get info about the turtlesim node:

```
$ ros2 node info /turtlesim
```

ROS 2 Run/Launch Management

Nodes can be run individually or launched together using a launch file utilizing the **ros2 run** and **ros2 launch** commands [2, 3].

Commands

Run a single node from a package:

```
$ ros2 run <package> <executable>
```

Launch nodes using a launch file:

```
$ ros2 launch <package> <launch_file>
```

Show expected arguments for a launch file:

```
$ ros2 launch <package> <launch_file> --show-arguments
```

Examples

Run the turtlesim node of the turtlesim package:

```
$ ros2 run turtlesim turtlesim_node
```

Run and rename the turtlesim node:

```
$ ros2 run turtlesim turtlesim_node --ros-args -r __node:=my_turtlesim
```

Launch the multisim.launch.py file from the turtlesim package:

```
$ ros2 launch turtlesim multisim.launch.py
```

ROS 2 Topic Management

A topic is a named channel over which nodes can exchange messages via the publish-subscribe communication model [4].

Commands

List all active topics:

```
$ ros2 topic list
```

Echo messages published to a topic:

```
$ ros2 topic echo <topic>
```

Publish a message to a topic:

```
$ ros2 topic pub <topic> <message_type> <message>
```

Get detailed information about a topic:

```
$ ros2 topic info <topic>
```

Display the rate (Hz) of messages published to a topic:

```
$ ros2 topic hz <topic>
```

Show the message type of a topic:

```
$ ros2 topic type <topic>
```

Examples

Publish a Twist message to the cmd_vel topic with 10 hz:

```
$ ros2 topic pub cmd_vel geometry_msgs/msg/Twist "{linear: {x: 0.1}}" -r 10
```

ROS 2 Package Management

Packages are the primary unit of ROS 2 code organization and reusability. A package may contain libraries, executables, and configuration files. [9]

Commands

List all installed ROS 2 packages:
`$ ros2 pkg list`

Create a new ROS 2 package with dependencies[a]:
`$ ros2 pkg create <package_name> --build-type <ament_type> --dependencies <dep1 dep2>`

Show the installation path of a package:
`$ ros2 pkg prefix <package_name>`

Show the exececutables of a package:
`$ ros pkg executables <package_name>`

Get detailed information about a package:
`$ ros2 pkg xml <package_name>`

Examples

Create a new package with a C++ build type:
`$ ros2 pkg create my_pkg --build-type ament_cmake`

Create a new package with a Python build type:
`$ ros2 pkg create my_pkg --build-type ament_python`

Get the maintainer of a package:
`$ ros2 pkg xml -t maintainer pkg_name`

[a]Dependencies must be resolveable by rosdep https://docs.ros.org/en/jazzy/Tutorials/Intermediate/Rosdep.html

ROS 2 Parameter Management

Parameters are key-value pairs that are bound to a node and can be used to configure the behavior of a node on startup and during runtime [5].

Commands

List all parameters of a node:
`$ ros2 param list <node>`

Get the value of a parameter from a node:
`$ ros2 param get <node> <param_name>`

Set a parameter on a node:
`$ ros2 param set <node> <param_name> <value>`

Describe all parameters and their types:
`$ ros2 param describe <node>`

Load parameters from a YAML file:
`$ ros2 param load <node> <yaml_file>`

Dump parameters to a YAML file:
`$ ros2 param dump <node> > < path/to/-file.yaml>`

Examples

Get the use_sim_time[a] parameter:
`$ ros2 param get /node use_sim_time`

Set the use_sim_time for a node to debug:
`$ ros2 param set /node luse_sim_time true`

Dump the parameters to a file:
`$ ros2 param dump /node > ./file.yaml`

[a]The use_sim_time parameter is used to control whether the node should use simulation time or wall-clock time: https://design.ros2.org/articles/clock_and_time.html

ROS 2 Interface Inspection

Interfaces are used to define the structure of data exchanged between nodes (messages, services, and actions) [6].

Commands

List all available message, service, and action interfaces:
`$ ros2 interface list`

List all packages that contain interfaces:
`$ ros2 interface packages`

List all interfaces of a package:
`$ ros2 interface package <package_name>`

Show the definition of a specific message, service, or action interface:
`$ ros2 interface show <interface_type>`

Examples

Show the definition of the geometry_msgs/msgs/Twist message:
`$ ros2 interface show geometry_msgs/msg/Twist`

Get all the actions of the example_interfaces package:
`$ ros2 interface package example_interfaces --only-actions`

ROS 2 Service Management

A service is a named entity that provides a way to request work to be done and receive a response [7].

Commands

List all available services:
$ **ros2 service list**
Show the type of a specific service:
$ **ros2 service type** <*service*>
Call a service with arguments:
$ **ros2 service call** <*service*> <*service_type*> <*arguments*>

Examples

Check if the service AddTwoInts is available:
$ **ros2 service list** | grep AddTwoInts
Call the turtlesim teleport_absolute service:
$ **ros2 service call** /turtle1/teleport_absolute turtlesim/srv/TeleportAbsolute {x: 2.0, y: 3.0, theta: 0.0}

ROS 2 Action Management

Actions are used to define goal-oriented behaviors that can be preempted and provide feedback [8].

Commands

List all available actions:
$ **ros2 action list**
Send a goal to an action server:
$ **ros2 action send_goal** <*action*> <*goal*>
Cancel a previously sent goal:
$ **ros2 action cancel** <*goal_id*>
Get feedback from an ongoing action:
$ **ros2 action feedback** <*action*>
Show the type of an action:
$ **ros2 action type** <*action*>

Examples

Send a goal to the turtlesim rotate_absolute action:
$ **ros2 action send_goal** /turtle1/rotate_absolute turtlesim/action/RotateAbsolute {theta: 1.57}
Sending a goal to an action and receiving feedbak:
$ **ros2 action send_goal** /turtle1/rotate_absolute turtlesim/action/RotateAbsolute {theta: 1.57} –feedback

ROS 2 Bag File Management (Data Recording and Playback)

A bag file is a data format used to store ROS message data. Bag files are useful for recording and replaying data, which can help with debugging and testing [10].

Commands

Record messages to a bag file from specified topics:
$ **ros2 bag record** <*topics*> *--output <file_name>*
Play messages from a bag file:
$ **ros2 bag play** <*bagfile*>
Show information about a bag file (e.g., size, duration):
$ **ros2 bag info** <*bagfile*>
List all recorded topics in a bag file:
$ **ros2 bag info** <*bagfile*> *--topics*

Examples

Record data from `/cmd_vel` and `/scan` topics:
$ **ros2 bag record** /cmd_vel /scan *--output my_bagfile*
Replay a recorded bag file at double speed:
$ **ros2 bag play** my_bagfile *--rate 2.0*

ROS 2 Diagnostics and Troubleshooting

ROS 2 offers several tools to diagnose and resolve issues related to node communication, topic connections, and system health.

Commands

Run diagnostics to check for common issues in the ROS 2 environment:
$ **ros2 doctor**
Inspect a node's publishers, subscribers, and services:
$ **ros2 node info** <*node_name*>
Get detailed information about a topic (e.g., type, publishers, subscribers):
$ **ros2 topic info** <*topic*>
Visualize the ROS 2 computation graph to identify connections between nodes:
$ **rqt_graph**
Restart the ROS 2 daemon to resolve discovery issues:
$ **ros2 daemon stop**
$ **ros2 daemon start**

Resolving Topic Name/Type Mismatches

Mismatches between topic names or message types can cause communication issues between nodes. Follow these steps to resolve them:

- **Inspect Node Information**: Use `ros2 node info` to list all topics a node publishes or subscribes to. This will help confirm if the node is using the correct topic name and message type.
  - Example: Inspect the `turtlesim_node`:
    $ **ros2 node info** /turtlesim
- **Check Topic Details**: Use `ros2 topic info` to inspect the message type of a topic and compare it with the expected type.
  - Example: Get information about the `/cmd_vel` topic:
    $ **ros2 topic info** /cmd_vel
- **Visualize with rqt_graph**: Run `rqt_graph` to visualize the node-topic connections and identify possible issues with topic mismatches or missing connections.
  - Example: Launch `rqt_graph`:
    $ **rqt_graph**

Example Workflow for Troubleshooting Topic Issues

- Inspect the publishing node:
  $ **ros2 node info** /publisher_node
- Inspect the subscribing node:
  $ **ros2 node info** /subscriber_node
- Verify that both nodes are using the same topic and message type:
  $ **ros2 topic info** /common_topic
- Use `rqt_graph` to visualize the connections and confirm proper node-topic relationships.

If the issue persists, restart the ROS 2 daemon to ensure proper discovery of nodes:

$ **ros2 daemon stop**
$ **ros2 daemon start**

Additional Tools

- **Check resource usage**: Tools like `top` or `htop` can be useful to identify memory or CPU bottlenecks.
- **Examine log files**: Review logs stored in `/.ros/log` for detailed error information.

ROS 2 Network and Security Tools

ROS 2 provides various networking and security tools to manage discovery, communication, and encryption between nodes.

Commands

Send a multicast message for node discovery (e.g., for debugging network discovery issues):

$ **ros2 multicast send**

Receive multicast messages:

$ **ros2 multicast receive**

Start the ROS 2 daemon (manages discovery, keeps the ROS environment active):

$ **ros2 daemon start**

Stop the ROS 2 daemon:

$ **ros2 daemon stop**

Generate a security keystore for encrypted communication between nodes:

$ **ros2 security generate_keystore** <*directory*>

Enable security for a node using environment variables:

$ `export ROS_SECURITY_ENABLED=1`

Examples

Create a security keystore in the directory `keystore_dir`:

$ **ros2 security generate_keystore** keystore_dir

Environment Variables

Environment variables play a crucial role in configuring the behavior of ROS 2 systems. Below are some of the most important ROS 2 environment variables:

ROS_SECURITY_ENABLE: Enables or disables ROS 2 security features. Set to `1` to enable security or `0` to disable it.

$ `export ROS_SECURITY_ENABLE=1`

ROS_SECURITY_KEYSTORE: Specifies the path to the directory containing security keys and certificates for encrypted communication between nodes.

```
$ export ROS_SECURITY_KEYSTORE=path_to_keystore
```

ROS_PACKAGE_PATH: Defines the search paths for ROS 2 packages. It contains multiple directories separated by colons.

```
$ export ROS_PACKAGE_PATH=/path/to/your/package
```

RMW_IMPLEMENTATION: Specifies the middleware implementation being used by ROS 2 (e.g., `rmw_fastrtps_cpp`, `rmw_cyclonedds_cpp`).

```
$ export RMW_IMPLEMENTATION=middleware
```

COLCON_DEFAULTS_FILE: Points to a file that contains default settings for Colcon commands, allowing for custom build configurations.

```
$ export COLCON_DEFAULTS_FILE=path_to_defaults_file
```

ROS_ETC_DIR: Specifies the directory where ROS 2 configuration files (such as launch and parameter files) are located.

```
$ export ROS_ETC_DIR=path_to_etc_dir
```

ROS_DISTRO: Defines the active ROS 2 distribution (e.g., `foxy`, `galactic`, `humble`).

```
$ export ROS_DISTRO=humble
```

ROS_AUTOMATIC_DISCOVERY_RANGE: Configures the automatic discovery range for ROS 2 nodes, which can help control the scope of node discovery. Values can be `local` or `global`.

```
$ export ROS_AUTOMATIC_DISCOVERY_RANGE=local
```

ROS_DOMAIN_ID: Sets the domain ID used by ROS 2, ensuring that only nodes within the same domain can communicate. This is useful for isolating multiple ROS systems on the same network.

```
$ export ROS_DOMAIN_ID=id_number
```

ROS_VERSION: Specifies the version of ROS in use. For ROS 2, this variable is set to 2.

```
$ export ROS_VERSION=2
```

ROS_PYTHON_VERSION: Defines the Python version used by ROS 2. Common values are 3.

```
$ export ROS_PYTHON_VERSION=3
```

ROS_WORKSPACE: Points to the current ROS 2 workspace, where packages are built and managed.

```
$ export ROS_WORKSPACE=path_to_workspace
```

Colcon Build System

The Colcon build system is used to build and manage ROS 2 workspaces. It supports building multiple packages in a single workspace and provides features like parallel builds, dependency management, and task execution.

Commands

Build all packages in the current workspace:

```
$ colcon build
```

Build the workspace whitout stopping on the first error:

```
$ colcon build --continue-on-error
```

List all installed packages in the workspace:

```
$ colcon list
```

Run tests for all packages in the workspace:

```
$ colcon test
```

Generate a report of test results:

```
$ colcon test-result
```

Source the workspace setup files after building without including underlay workspaces:

```
$ source install/local_setup.bash
```

Source the workspace setup files after building with including the underlay workspaces:

```
$ source install/setup.bash
```

Options

Build a specific package in the workspace:

```
$ colcon build --packages-select <package_name>
```

Build packages while skipping dependencies:

```
$ colcon build --packages-ignore <package_name>
```

Build with additional verbosity for debugging:

```
$ colcon build --event-handlers console_direct+
```

Parallelize the build process (increase speed):

```
$ colcon build --parallel-workers <number>
```

Examples

Build a single package called `my_package`:

```
$ colcon build --packages-select my_package
```

Build the workspace and run all tests:

```
$ colcon build && colcon test
```

Clean the workspace and rebuild everything:

```
$ colcon build --clean && colcon build
```

Check test results after running tests:

```
$ colcon test-result
```

Must Know Flags

Use 'symlinks' instead of installing (copying) files where possible (works for .py files):

```
$ --symlink-install
```

Continue other packages when a package fails to build. Packages recursively depending on the failed package are skipped:

```
$ --continue-on-error
```

Show output on console:
`$ --event-handlers console.direct+`
Show output on console after a package has finished:
`$ --event-handlers console.cohesion+`
Build only specific package(s):
`$ --packages-select`
Build specific package(s) and its/their recursive dependencies:
`$ --packages-up-to`
Build specific package(s) and other packages that recursively depend on it:
`$ --packages-above`
Skip package(s):
`$ --packages-skip`
Skip a set of packages that have finished building previously:
`$ --packages-skip-build-finished`
Pass arguments to CMake projects:
`$ --cmake-args`
Remove CMake cache before the build (implicitly forcing CMake configure step):
`$ --cmake-clean-cache`
Build target 'clean' first, then build (to only clean use '–cmake-target clean'):
`$ --cmake-clean-first`
Force CMake configure step:
`$ --cmake-force-configure`

Environment Variables

The full path to the CMake executable:
`$ CMAKE_COMMAND`
Flag to enable all shell extensions:
`$ COLCON_ALL_SHELLS`
Set the logfile for completion time:
`$ COLCON_COMPLETION_LOGFILE`
Set path to the yaml file containing the default values for the command line arguments (default: $COLCON_HOME/defaults.yaml):
`$ COLCON_DEFAULTS_FILE`
Select the default executor extension:
`$ COLCON_DEFAULT_EXECUTOR`
Set the configuration directory (default: /.colcon):
`$ COLCON_HOME`
The full path to the PowerShell executable:
`$ POWERSHELL_COMMAND`

Bibliography

References

[1] About Nodes, https://docs.ros.org/en/jazzy/Concepts/Basic/About-Nodes.html

[2] Starting single nodes, https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html#ros2-run

[3] Starting multiple nodes, https://docs.ros.org/en/jazzy/Tutorials/Intermediate/Launch/Launch-Main.html

[4] About topics, https://docs.ros.org/en/jazzy/Concepts/Basic/About-Topics.html

[5] About parameters, https://docs.ros.org/en/jazzy/Concepts/Basic/About-Parameters.html#parameters

[6] About interfaces, https://docs.ros.org/en/jazzy/Concepts/Basic/About-Interfaces.html

[7] About services, https://docs.ros.org/en/jazzy/Concepts/Basic/About-Services.html

[8] About actions, https://docs.ros.org/en/jazzy/Concepts/Basic/About-Actions.html

[9] Package layout, https://docs.ros.org/en/jazzy/The-ROS2-Project/Contributing/Developer-Guide.html#package-layout

[10] Recording and playing back data, https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Recording-And-Playing-Back-Data/Recording-And-Playing-Back-Data.html

[11] ROS 2 Command Line Arguments, https://design.ros2.org/articles/ros_command_line_arguments.html