ROS 2 Update

Deanna Hood, William Woodall
October 8, 2016
ROSCon 2016 Seoul

https://goo.gl/oCHR7H
Contents

ROS 2 overview
Overview of changes in the last year
  Details of select features
Experience porting Turtlebot
Roadmap

https://goo.gl/oCHR7H
ROS as we know it
Research applications
High-volume sensors
Complex kinematics
Lots of computation power
Ideal network connectivity
- Multi-robot system
- Small processors
- Battery power
- Unreliable network connectivity
Goals of ROS 2

Support multi-robot systems involving unreliable networks

Remove the gap between prototyping and final products

“Bare-metal” micro controller

Support for real-time control

Cross-platform support
ROS 2

Plumbing + Tools + Capabilities + Ecosystem
ROS 2

+ ROS usability

Plumbing

Tools

Capabilities

Ecosystem

less time spent here

means

more time to spend here
Architectural overview

- User code
- ROS client library API
Architectural overview

User code

ROS client library API

DDS implementation

= discovery + serialization + transport
ROSCon 2015 demos

Quality of Service demo for lossy networks using ROS 2

https://github.com/ros2/ros2/wiki/Tutorials

Bridge communication between ROS 1 and ROS 2

Efficient intra-process communication using ROS 2

Real-time safe code using ROS 2

ROS 2 on “bare-metal” microcontrollers
What’s new this year
Changes since ROSCon 2015: user-facing

- Windows feature parity (alpha 2)
- Fast RTPS added as a supported middleware (alpha 3)
- Partial port of tf2 including the core libraries (alpha 3)
- Python client library (alpha 4)
- 32-bit and 64-bit ARM added as experimentally supported platforms (alpha 5)
- Node "wait for service" functionality (alpha 6)
- Turtlebot demo using ported code from ROS 1 (alpha 7)
Changes since ROSCon 2015

- Windows feature parity (alpha 2)
- Fast RTPS supported as middleware (alpha 3)
- Partial port of core tf2 libraries (alpha 3)
- Python client library (alpha 4)
- 32- and 64-bit ARM experimentally supported platforms (alpha 5)
- Node “wait for service” functionality (alpha 6)
- Turtlebot demo using ported code from ROS 1 (alpha 7)
- Partial port of core tf2 libraries (alpha 3)
Changes since ROSCon 2015

- Support for C messages (as opposed to C++) (alphas 4, 5, 7)
- Improved support for large messages (images) in both Connext and Fast RTPS (alpha 6, alpha 7)
- ROS Client Library implementation (rcl) (from alpha 3, services alpha 5)
- Refactored C++ client library to use rcl (alpha 6)
- ROS graph events (alpha 6)
- Improved support for large messages (images) with Connext and Fast RTPS (alpha 6, alpha 7)
- Partial port of core tf2 libraries (alpha 3)
- Fast RTPS supported as middleware (alpha 3)
- Python client library (alpha 4)
- 32- and 64-bit ARM experimentally supported platforms (alpha 5)
- Windows feature parity (alpha 2)
- ROS graph events (alpha 6)
- Turtlebot demo using ported code from ROS 1 (alpha 7)
- Arduino feature parity (alpha 2)
Changes since ROSCon 2015

- Windows feature parity (alpha 2)
- Fast RTPS supported as middleware (alpha 3)
- Partial port of core tf2 libraries (alpha 3)
- Python client library (alpha 4)
- 32- and 64-bit ARM experimentally supported platforms (alpha 5)
- Partial port of core tf2 libraries (alpha 3)
- Node “wait for service” functionality (alpha 6)
- Turtlebot demo using ported code from ROS 1 (alpha 7)

**ROS Client Library**
- Implementation (rcl) (from alpha 3, services alpha 5)

**Support for C messages** (as opposed to C++) (alphas 4, 5, 7)

- Refactored C++ client library to use rcl (alpha 6)
- ROS graph events (alpha 6)
- Improved support for large messages (images) with Connext and Fast RTPS (alpha 6, alpha 7)
Changes since ROSCon 2015

- Windows feature parity (alpha 2)
- Fast RTPS supported as middleware (alpha 3)
- Partial port of core tf2 libraries (alpha 3)
- Python client library (alpha 4)
- 32- and 64-bit ARM experimentally supported platforms (alpha 5)
- Node “wait for service” functionality (alpha 6)
- Turtlebot demo using ported code from ROS 1 (alpha 7)
- ROS Client Library implementation (rcl) (from alpha 3, services alpha 5)
- Support for C messages (as opposed to C++) (alphas 4, 5, 7)
- Refactored C++ client library to use rcl (alpha 6)
- ROS graph events (alpha 6)
- Improved support for large messages (images) with Connext and Fast RTPS (alpha 6, alpha 7)
Changes since ROSCon 2015

- Windows feature parity (alpha 2)
- Fast RTPS supported as middleware (alpha 3)
- Improved support for large messages (images) in both Connext and Fast-RTPS (alpha 6, alpha 7)
- ROS Client Library implementation (rcl) (from alpha 3, services alpha 5)
- Partial port of core tf2 libraries (alpha 3)
- Refactored C++ client library to use rcl (alpha 6)
- ROS graph events (alpha 6)
- Support for C messages (as opposed to C++) (alphas 4, 5, 7)
- Improved support for large messages (images) with Connext and Fast RTPS (alpha 6, alpha 7)
- 32- and 64-bit ARM experimentally supported platforms (alpha 5)
- Python client library (alpha 4)
- Node “wait for service” functionality (alpha 6)
- Turtlebot demo using ported code from ROS 1 (alpha 7)
Architectural overview

User code

ROS client library API

DDS impl A or DDS impl B or ...

Architectural overview

User code

ROS client library API

ROS middleware API

DDS impl A  or  DDS impl B  or  ...

DDS agnostic

ROS agnostic
Architectural overview

User code

ROS client library API

ROS middleware API

DDS impl A or DDS impl B or ...

DDS agnostic

ROS agnostic
Architectural overview

User code

ROS client library API

ROS middleware API

RMW impl A

RMW impl B

... or

DDS impl A

or

DDS impl B

or

...
Architectural overview

User code

ROS client library API

ROS middleware API

Fast RTPS or DDS impl B or 
RMW impl B or 

DDS agnostic

ROS agnostic
Supported vendors until October 2016

User code

ROS client library API

ROS middleware API

eProsima  
Fast RTPS

RTI  
Connext

Connext Dynamic

PrismTech  
OpenSplice
Supported vendors since October 2016

- User code
- ROS client library API
- ROS middleware API

- eProsima Fast RTPS
- RTI Connext
- Connext Dynamic
- PrismTech OpenSplice

Default

Development paused
Why eProsima’s Fast RTPS?

- Changed the license June 2016:
  - LGPL -> Apache 2.0
- Code on GitHub
  - [https://github.com/eProsima/Fast-RTPS](https://github.com/eProsima/Fast-RTPS)
- Responsive to issues and pull requests
- Added features needed to support ROS 2
  - Fragmentation of large messages
  - Graph change notifications
- CMake buildsystem
Changes since ROSCon 2015

- Windows feature parity (alpha 2)
- Fast RTPS supported as middleware (alpha 3)
- Partial port of core tf2 libraries (alpha 3)
- Python client library (alpha 4)
- ROS Client Library implementation (rcl) (from alpha 3, services alpha 5)
- Support for C messages (as opposed to C++) (alphas 4, 5, 7)
- Refactored C++ client library to use rcl (alpha 6)
- ROS graph events (alpha 6)
- Improved support for large messages (images) with Connext and Fast RTPS (alpha 6, alpha 7)
- 32- and 64-bit ARM experimentally supported platforms (alpha 5)
- Turtlebot demo using ported code from ROS 1 (alpha 7)
ROS client libraries

talker.py

listener.cpp
ROS client libraries

talker.py
Python ROS client library

listener.cpp
C++ ROS client library
ROS client libraries

- User Code
- rclcpp
- rclpy
- ROS middleware interface (rmw)
- DDS vendor
ROS client libraries

- User Code
- rclcpp
- rclpy
- ROS middleware interface (rmw)
- DDS vendor

- Names & namespaces
- Time
- Parameters
- Console logging
- Threading model
- Intra-process communication
ROS client libraries

- User Code
  - rclcpp
  - rclpy
- ROS middleware interface (rmw)
- DDS vendor
- Names & namespaces
- Time
- Parameters
- Console logging
- Threading model
- Intra-process communication
ROS client libraries

- User Code
- rclcpp
- rclpy
- ROS client library (rcl)
- ROS middleware interface (rmw)
- DDS vendor
- Threading model
- Intra-process communication
  - Names & namespaces
  - Time
  - Parameters
  - Console logging
ROS client libraries

node.cpp  node.py  node.cs  node.java

rclcpp  rclpy  rclcs  rcljava

rcl  rmw

DDS vendor

https://github.com/firesurfer/rclcs
https://github.com/esteve/ros2_java
Tracing talker-listener

Consider this talker-listener example:

talker.py → /chatter → listener.cpp
Tracing talker-listener

```python
rclpy.init()

node = rclpy.create_node('talker')
chatter_pub = node.create_publisher(
    std_msgs.msg.String, 'chatter')
msg = std_msgs.msg.String()
i = 1

while True:
    msg.data = 'Hello World: {0}'.format(i)
i += 1
print('Publishing: "{0}"'.format(msg.data))
chatter_pub.publish(msg)
```
Tracing talker-listener

```python
rclpy.init()

node = rclpy.create_node('talker')
chatter_pub = node.create_publisher(
    std_msgs.msg.String, 'chatter')
msg = std_msgs.msg.String()
i = 1

while True:
    msg.data = 'Hello World: {0}'.format(i)
i += 1
    print('Publishing: "{}"'.format(msg.data))
    chatter_pub.publish(msg)
```
Tracing talker-listener

```python
static PyObject *
rclpy_publish(PyObject * Py_UNUSED(self), PyObject * args) {
    PyObject * pypublisher; // populated from args
    PyObject * pymsg; // populated from args
    // ...
    void * raw_ros_message = convert_from_py(pymsg);

    rcl_ret_t ret = rcl_publish(publisher, raw_ros_message);
    if (ret != RCL_RET_OK) {
        // error handling
    }
    // ...
}
```

rclpy

rclpy_publish

talker.py
Tracingtalker-listener

```python
static PyObject *
rclpy_publish(PyObject * Py_UNUSED(self), PyObject * args) {
    PyObject * pypublisher; // populated from args
    PyObject * pymsg; // populated from args
    // ...
    void * raw_ros_message = convert_from_py(pymsg);

    rcl_ret_t ret = rcl_publish(publisher, raw_ros_message);
    if (ret != RCL_RET_OK) {
        // error handling
    }
    // ...
}
```
Tracing talker-listener

```c
rcl_ret_t
rcl_publish(
    const rcl_publisher_t * publisher,
    const void * ros_message)
{
    // ...
    ret = rmw_publish(publisher->impl->rmw_handle, ros_message);
    if (ret != RMW_RET_OK) {
        // error handling
    }
    return RCL_RET_OK;
}
```
rcl_ret_t
rcl_publish(
    const rcl_publisher_t * publisher,
    const void * ros_message)
{
    // ...
    ret = rmw_publish(publisher->impl->rmw_handle, ros_message);
    if (ret != RMW_RET_OK) {
        // error handling
    }
    return RCL_RET_OK;
}
Tracing talker-listener

```
rmw_ret_t
rmw_publish(
    const rmw_publisher_t * publisher,
    const void * ros_message);
```
Tracing talker-listener

talker.py

rclpy

rcl

rmw

rmw impl

DDS Vendor

IPC

DDS Vendor

publish

/chatter

subscribe

listener.cpp
Tracing talker-listener

talker.py → publish → /chatter → subscribe → listener.cpp

rclpy

rcl

rmw

rmw_fastrtps_cpp

Fast RTPS

IPC

Fast RTPS
Tracing talker-listener

talker.py

rclpy

rcl

rmw

rmw_fastrtps_cpp

```c
rmw_ret_t
rmw_publish(
    const rmw_publisher_t * publisher, const void * ros_message)
{
    // ...
    eprosima::fastcdr::FastBuffer buffer;
    eprosima::fastcdr::Cdr ser(buffer);
    PublisherImpl * info = (PublisherImpl *)publisher->data;

    if(_serialize_ros_message(ros_message, ser, /* ... */)) {
        if(info->publisher_->write(&ser)) // Fast RTPS publisher
            return RMW_RET_OK;
        else
            // ... publish error handling
    }
    else
        // ... serialize error handling
}
```
Tracing talker-listener

```c
rmw_ret_t
rmwPublish(
    const rmw_publisher_t * publisher, const void * ros_message)
{
    // ...
    eprosima::fastcdr::FastBuffer buffer;
    eprosima::fastcdr::Cdr ser(buffer);
    PublisherImpl * info = (PublisherImpl *)publisher->data;

    if(_serialize_ros_message(ros_message, ser, /* ... */)) {
        if(info->publisher_->write(&ser)) // Fast RTPS publisher
            return RMW_RET_OK;
        else
            // ... publish error handling
    }
    else
        // ... serialize error handling
}
```
Tracing talker-listener

talker.py

rclpy

rcl

rmw

rmw_fastrtps_cpp

publish /chatter subscribe

listener.cpp

Fast RTPS

IPC

Fast RTPS

“Hello World”
void
chatter_callback(const std_msgs::msg::String::SharedPtr msg) {
    std::cout << "I heard: [" << msg->data << "]" << std::endl;
}

int
main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);

    auto node = rclcpp::Node::make_shared("listener");

    auto sub = node->create_subscription<std_msgs::msg::String>(
        "chatter", chatter_callback, rmw_qos_profile_default);

    rclcpp::spin(node);
}
void chatter_callback(const std_msgs::msg::String::SharedPtr msg) {
    std::cout << "I heard: [" << msg->data << "]" << std::endl;
}

int main(int argc, char * argv[]) {
    rclcpp::init(argc, argv);
    auto node = rclcpp::Node::make_shared("listener");
    auto sub = node->create_subscription<std_msgs::msg::String>(
        "chatter", chatter_callback, rmw_qos_profile_default);
    rclcpp::spin(node);
}
Tracing talker-listener

rclcpp::spin()

wait for “work”

Does subscription have data?

Yes

execute_subscription()
Tracing talker-listener

talker.py

rclpy

rcl

rmw

rmw_fastrtps_cpp

publish

/chatter

subscribe

listener.cpp

Subscription has data

“Hello World”

Fast RTPS

IPC

Fast RTPS
Tracing talker-listener

```
talker.py

rmw_fastrtps

rclpy

"Hello World"

IPC

Fast RTPS

listener.cpp

rclcpp

rclcpp::spin()

wait for “work”

Does subscription have data?

execute_subscription()

talker-listener

Does subscription have data?

wait for “work”

rclcpp::spin()

listener.cpp

rclcpp
```
Tracing talker-listener

```cpp
void execute_subscription(/* ... */ subscription) {
    std::shared_ptr<void> message =
        subscription->create_message();

    auto ret = rcl_take(
        subscription->get_subscription_handle(),
        message.get(), /* ... */);
    if (ret == RCL_RET_OK) {
        subscription->handle_message(message, /* ... */);
    } else { /* error handling */ }
}
```

Yes

execute_subscription()
Tracing talker-listener

```cpp
void execute_subscription(/* ... */ subscription)
{
    std::shared_ptr<void> message =
        subscription->create_message();

    auto ret = rcl_take(
        subscription->get_subscription_handle(),
        message.get(), /* ... */);

    if (ret == RCL_RET_OK) {
        subscription->handle_message(message, /* ... */);
    } else { /* error handling */ }
}
```
Tracing talker-listener

```c
rcl_ret_t
rcl_take(
    const rcl_subscription_t * subscription,
    void * ros_message, /* ... */)
{
    // ...
    bool taken = false;
    rmw_ret_t ret = rmw_take(
        subscription->impl->rmw_handle, ros_message, &taken);
    if (ret != RMW_RET_OK) {
        // ... error handling
    }
    if (!taken) {
        return RCL_RET_SUBSCRIPTION_TAKE_FAILED;
    }
    return RCL_RET_OK;
}
```
Tracing talker-listener

```cpp
rcl_ret_t
rcl_take(
    const rcl_subscription_t * subscription,
    void * ros_message, /* ... */
)
{
    // ...
    bool taken = false;
    rmw_ret_t ret = rmw_take(
        subscription->impl->rmw_handle, ros_message, &taken);
    if (ret != RMW_RET_OK) {
        // ... error handling
    }
    if (!taken) {
        return RCL_RET_SUBSCRIPTION_TAKE_FAILED;
    }
    return RCL_RET_OK;
}
```
Tracing talker-listener

```c
rmw_ret_t
rmw_take(
  const rmw_subscription_t * subscription,
  void * ros_message,
  bool * taken);
```
Tracing talker-listener

```c
rmw_ret_t
rmw_take(
    const rmw_subscription_t * subscription,
    void * ros_message, bool * taken)
{
    *taken = false;
    SubscriptionImpl * info = (SubscriptionImpl *)subscription->data;

    eprosima::fastcdr::FastBuffer buffer;
    SampleInfo_t sinfo;

    if(info->subscriber_->takeNextData(&buffer, &sinfo)) {
        if(sinfo.sampleKind == ALIVE) { // actually contains data
            _deserialize_ros_message(&buffer, ros_message, /* ... */);
            *taken = true;
        }
    }
}
```
Tracing talker-listener

```c
rmw_ret_t
rmw_take(
    const rmw_subscription_t * subscription,
    void * ros_message, bool * taken)
{
    *taken = false;
    SubscriptionImpl * info = (SubscriptionImpl *)subscription->data;
    eprosima::fastcdr::FastBuffer buffer;
    SampleInfo_t sinfo;

    if(info->subscriber_->takeNextData(&buffer, &sinfo)) {
        if(sinfo.sampleKind == ALIVE) { // actually contains data
            _deserialize_ros_message(&buffer, ros_message, /* ... */);
            *taken = true;
        }
    }
}
```
void chatter_callback(const std_msgs::msg::String::SharedPtr msg) {
    std::cout << "I heard: [" << msg->data << "]" << std::endl;
}

void

Tracing talker-listener

rclpy

rcl

rmw

rmw_fastrtps_cpp

listener.cpp

rclcpp

rcl

rmw

rmw_fastrtps_cpp

Fast RTPS

IPC

Fast RTPS

“Hello World”
Changes since ROSCon 2015

- Support for C messages (as opposed to C++) (alphas 4, 5, 7)
- Improved support for large messages (images) in both Connext and Fast-RTPS (alpha 6, alpha 7)
- ROS Client Library implementation (rcl) (from alpha 3, services alpha 5)
- 32- and 64-bit ARM experimentally supported platforms (alpha 5)
- ROS graph events (alpha 6)
- Partial port of core tf2 libraries (alpha 3)
- Refactored C++ client library to use rcl (alpha 6)
- Improved support for large messages (images) with Connext and Fast-RTPS (alpha 6, alpha 7)
- Turtlebot demo using ported code from ROS 1 (alpha 7)
- Windows feature parity (alpha 2)
- Fast RTPS supported as middleware (alpha 3)
- ROS graph events (alpha 6)
- Partial port of core tf2 libraries (alpha 3)
- Python client library (alpha 4)
Porting of Turtlebot to ROS 2

- Minimum viable demo (https://github.com/ros2/turtlebot2_demo)
  - Kobuki driver
  - Astra driver
  - Joystick driver
  - Follower node

https://orbbec3d.com/

http://kobuki.yujinrobot.com/
Porting of Turtlebot to ROS 2

- Kobuki driver
  - Used existing non-ROS dependencies
  - Replaced ROS 1 wrapper with ROS 2 wrapper
- Astra driver
  - Forked and ported existing ROS 1 driver to ROS 2
- Joystick driver
  - Wrote a simple joystick program from scratch (no porting)
- Follower node
  - Forked and ported existing ROS 1 node
- ROS 1 ⇔ ROS 2 bridge for visualization
Porting Experiments

- ROS 1 “shim” ([https://github.com/codebot/ros1_shim](https://github.com/codebot/ros1_shim))
  - Some things (like the astra driver) needed some deep features (e.g. custom serialization)
  - Hard to find the right strata in the interfaces to shim
Porting Experiments

  - Find ways to modify each to make them more similar
    - In order to minimize conversion effort
  - Mixing catkin (ROS 1) and ament (ROS 2)
    - To avoid converting unless necessary
  - Non-homogeneous workspace
    - Building catkin and ament packages at the same time
  - Ideal: one build tool for both
    - ament vs catkin not unlike catkin vs plain cmake
Porting Experiments

- catment continued...
  - Conceptual details to work out:
    - setup.*sh files in root of workspace
      - Currently required by catkin
      - Optional for ament
    - devel-space
      - ament uses “symlink install” instead
      - Avoiding confusion in documentation
    - Make catkin more like ament? (and vice versa?)
Roadmap

● Beta 1 - End of the Year
  ○ Composition
    ■ may use pluginlib and class_loader from ROS 1 for C++
  ○ QoS benchmarks
    ■ for example: unreliable comms, illustrated by wifi out-and-back
  ○ Design documents
  ○ Tutorials and examples
  ○ "rostopic list", "rostopic echo", and friends
  ○ Bridging services to/from ROS1 (in addition to topics)

● Nice to have by Beta 1:
  ○ Console logging
    ■ think “rosconsole”
  ○ Orchestration
    ■ think “roslaunch + verification & dynamic behavior”
Pointers

● ROS 2 wiki: https://github.com/ros2/ros2/wiki
  ○ Installation instructions
  ○ Tutorials
  ○ How to contribute
  ○ Current status
  ○ Roadmap

● Developer docs (work in progress):
  ○ https://github.com/ros2/ros_core_documentation/blob/master/source/developer_overview.rst
  ○ Architecture overview
  ○ Links to API docs

● Design documents: http://design.ros2.org/
  ○ Articles about various subjects
  ○ On going discussions on the issue tracker:
    https://github.com/ros2/design
Questions

https://goo.gl/oCHR7H
“Hour Glass” Pattern

API size

LOC

{client libraries}

rcl

rcl impl

rmw

{rmw impl}

{DDS vendor}
“Hour Glass” Pattern - C++ with Fast RTPS

- **rmw**
- **rmw_fastrtps_cpp**
- **Fast RTPS**
- **rcl_impl**
- **rcl**
- **rclcpp**

API size

LOC
"Hour Glass" Pattern - Python with RTI Connext

API size

LOC

rclpy
rcl
rcl impl
rmw
rmw_connext_cpp
RTI Connext
“Hour Glass” Pattern - Python with RTI Connext

API size

LOC

Common

rclpy

rcl

rcl impl

rmw

rmw_connext_cpp

RTI Connext
"Hour Glass" Pattern - Python with RTI Connext