interval (std::chrono::milliseconds (10))
| map ([&](int i)
    { return mk_msg(hello + std::to_string(i)); })
| tap ([])(const std_msgs::String& msg)
    { ROS_INFO_STREAM (msg.data); })
| publish_to_topic<std_msgs::String>
    ("/chatter", 1000);
The Listener Example

```c
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{ 
  ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```
The Listener Example

```c
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{ ROS_INFO("I heard: [%s]", msg->data.c_str()); }

int main(int argc, char **argv) {
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
    ros::spin();
    return 0;
}
```
The Listener Example

```cpp
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{ ROS_INFO("I heard: [%s]", msg->data.c_str()); }

int main(int argc, char **argv) {
  ros::init(argc, argv, "listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
    ros::spin();
    return 0;
}
The Listener Example

```c
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    ROS_INFO("I heard: [%s]", msg->data.c_str());
}

int main(int argc, char **argv) {
    ros::init(argc, argv, "listener");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("chatter", 1000, chatterCallback);
    ros::spin();
    return 0;
}
```

```c
int main(int argc, char **argv) {
    rxros::init(argc, argv, "listener");
    rxros::observable::from_topic<std_msgs::String>("/chatter", 1000)
        .subscribe([](const std_msgs::String& msg)
        {
            ROS_INFO_STREAM("I heard: [" << msg.data << "]");
        });
    rxros::spin();
    return 0;
}
```
The Listener Example

Key points

- Problem
The Listener Example

Key points

- Problem
  - We have a simple mental model in ROS: a flow graph of messages
The Listener Example

Key points

- **Problem**
  - We have a simple mental model in ROS: a **flow graph of messages**
  - We think about **callbacks** when we realize it
The Listener Example

Key points

- **Problem**
  - We have a simple mental model in ROS: a flow graph of messages
  - We think about callbacks when we realize it
  - Among the most complex control-flow constructs
The Listener Example

Key points

- **Problem**
  - We have a simple mental model in ROS: a **flow graph of messages**
  - We think about **callbacks** when we realize it
  - Among the most complex control-flow constructs

- **Solution**
The Listener Example

Key points

- **Problem**
  - We have a simple mental model in ROS: a flow graph of messages
  - We think about callbacks when we realize it
  - Among the most complex control-flow constructs

- **Solution**
  - Reactive programming gives simple control-flow
The Listener Example

Key points

- **Problem**
  - We have a simple mental model in ROS: a **flow graph of messages**
  - We think about **callbacks** when we realize it
  - Among the most complex control-flow constructs

- **Solution**
  - Reactive programming gives **simple control-flow**
  - Flow of information is **explicit in the code**
The Talker Example

```c
int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker");
    ros::NodeHandle n;
    ros::Publisher chatter_pub =
        n.advertise<std_msgs::String>("chatter",10);
    ros::Rate loop_rate(10);
    int count = 0;
    while (ros::ok())
    {
        std_msgs::String msg;
        std::stringstream ss;
        ss << "hello world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}
```
The Talker Example

```cpp
int main(int argc, char **argv)
{
    ros::init(argc, argv, "talker");
    ros::NodeHandle n;
    ros::Publisher chatter_pub =
        n.advertise<std_msgs::String>("chatter", 10);
    ros::Rate loop_rate(10);
    int count = 0;
    while (ros::ok())
    {
        std_msgs::String msg;
        std::stringstream ss;
        ss << "HELLO world " << count;
        msg.data = ss.str();
        ROS_INFO("%s", msg.data.c_str());
        chatter_pub.publish(msg);
        ros::spinOnce();
        loop_rate.sleep();
        ++count;
    }
    return 0;
}
```

```cpp
int main(int argc, char **argv)
{
    rxros::init(argc, argv, "talker");
    const std::string hello = "hello world ";

    rxcpp::observable<>:
    interval (std::chrono::milliseconds (10))
    | map ([&](int i) {
        return mk_msg(hello + std::to_string(i)); })
    | tap ([](const std_msgs::String& msg) {
        ROS_INFO_STREAM (msg.data); })
    | publish_to_topic<std_msgs::String>
        ("/chatter", 1000);
    rxros::spin();
    return 0;
}
```
The Talker Example

Key points

- Problem
The Talker Example

Key points

- **Problem**
  - We have a simple mental model in ROS: a *flow graph of messages*

- **Solution**
  - Functional programming raises the abstraction level
  - We think about an *incremented stream with a frequency*
  - And we *transform* this stream (or messages in it)

In RxRos publisher and subscriber look similar: both are pipelines

In classic ROS they are very different: callback vs a loop

RxROS parallelizes pipeline processing

When you are avoiding callbacks, and remain pure (no side effects) as much as possible, the need for locks decreases, and with them concurrency problems
The Talker Example

Key points

- Problem
  - We have a simple mental model in ROS: a flow graph of messages
  - We think about loops, intervals, counters incremented when we realize it
The Talker Example

Key points

■ Problem
  ■ We have a simple mental model in ROS: a flow graph of messages
  ■ We think about loops, intervals, counters incremented when we realize it

■ Solution
The Talker Example

Key points

■ Problem
  ■ We have a simple mental model in ROS: a flow graph of messages
  ■ We think about loops, intervals, counters incremented when we realize it

■ Solution
  ■ Functional programming raises the abstraction level
The Talker Example

Key points

Problem
- We have a simple mental model in ROS: a flow graph of messages
- We think about loops, intervals, counters incremented when we realize it

Solution
- Functional programming raises the abstraction level
- We think about a incremented stream with a frequency
The Talker Example

Key points

Problem
- We have a simple mental model in ROS: a flow graph of messages
- We think about loops, intervals, counters incremented when we realize it

Solution
- Functional programming raises the abstraction level
- We think about a incremented stream with a frequency
- And we transform this stream (or messages in it)
The Talker Example

Key points

- **Problem**
  - We have a simple mental model in ROS: a flow graph of messages
  - We think about loops, intervals, counters incremented when we realize it

- **Solution**
  - Functional programming raises the abstraction level
  - We think about a incremented stream with a frequency
  - And we transform this stream (or messages in it)
  
- In RxRos publisher and subscriber look similar: both are pipelines

© Andrzej Wasowski, IT University of Copenhagen
The Talker Example

Key points

Problem
- We have a simple mental model in ROS: a flow graph of messages
- We think about loops, intervals, counters incremented when we realize it

Solution
- Functional programming raises the abstraction level
- We think about a incremented stream with a frequency
- And we transform this stream (or messages in it)
- In RxRos publisher and subscriber look similar: both are pipelines
- In classic ROS they are very different: callback vs a loop
The Talker Example

Key points

■ Problem
  ■ We have a simple mental model in ROS: a flow graph of messages
  ■ We think about loops, intervals, counters incremented when we realize it

■ Solution
  ■ Functional programming raises the abstraction level
  ■ We think about a incremented stream with a frequency
  ■ And we transform this stream (or messages in it)
  ■ In RxRos publisher and subscriber look similar: both are pipelines
  ■ In classic ROS they are very different: callback vs a loop
  ■ RxROS parallelizes pipeline processing

© Andrzej Wasowski, IT University of Copenhagen
The Talker Example

Key points

- **Problem**
  - We have a simple mental model in ROS: a flow graph of messages
  - We think about loops, intervals, counters incremented when we realize it

- **Solution**
  - Functional programming raises the abstraction level
  - We think about a incremented stream with a frequency
  - And we transform this stream (or messages in it)

- In RxRos publisher and subscriber look similar: both are pipelines
- In classic ROS they are very different: callback vs a loop
- RxROS parallelizes pipeline processing
- When you are avoiding callbacks, and remain pure (no side effects) as much as possible, the need for locks decreases, and with them concurrency problems
The Talker Example

Marble diagram
The Talker Example

Marble diagram

```cpp
interval (std::chrono::milliseconds (10))
| map ([&](int i)
    { return mk_msg(hello + std::to_string(i)); })
| tap ([&](const std_msgs::String& msg)
    { ROS_INFO_STREAM (msg.data); })
| publish_to_topic<std_msgs::String>
    ("/chatter", 1000);
```
The Talker Example

Marble diagram

interval (std::chrono::milliseconds (10))
The Talker Example

Marble diagram

```cpp
interval (std::chrono::milliseconds (10))

| map ([&](int i) { return mk_msg (hello+std::to_string(i)); })

"hello world 5" "hello world 4" "hello world 3" "hello world 2" "hello world 1"
```

© Andrzej Wasowski, IT University of Copenhagen
The Talker Example

Marble diagram

```
interval (std::chrono::milliseconds (10))
```

```
| map ([&](int i) { return mk_msg (hello+std::to_string(i)); })
```

```
| tap ([](const std_msgs::String& msg) { ROS_INFO_STREAM (msg.data); })
```

© Andrzej Wasowski, IT University of Copenhagen
The Talker Example

Marble diagram

```
interval (std::chrono::milliseconds (10))
```

```
| map ([&](int i) { return mk_msg (hello + std::to_string(i)); })
```

```
| tap ([](const std_msgs::String& msg) { ROS_INFO_STREAM (msg.data); })
```

```
| publish_to_topic<std_msgs::String>("/chatter", 1000);
```

The stream is published (string messages) to /chatter
RxROS

- RxROS is a very thin library (326 lines of C++ header file)

Extends RxCPP, a reactive programming library for C++. Adds several ROS-specific operators: advertiseService, from_topic, from_device, from_yaml, sample_with_frequency, publish_to_topic, call_service.

Available in melodic and kinetic:
APT INSTAll ROS-MElOdIC-RxROS

Available on GitHub: HTTPS://gITHub.COM/ROSIN-PROJECT/RxROS

Some examples: HTTPS://gITHub.COM/ROSIN-PROJECT/RxROS_ExAMPlES

© Andrzej Wąsowski, IT University of Copenhagen
RxROS

- RxROS is a **very thin library** (326 lines of C++ header file)
- Extends RxCPP, a reactive programming library for C++
**RxROS**

- **RxROS is a very thin library** (326 lines of C++ header file)
- **Extends RxCPP**, a reactive programming library for C++
- **Adds several ROS-specific operators**: advertiseService, from_topic, from_device, from_yaml, sample_with_frequency, publish_to_topic, call_service
RxROS

■ RxROS is a **very thin library** (326 lines of C++ header file)
■ Extends RxCPP, a reactive programming library for C++
■ Adds **several ROS-specific operators**: advertiseService, from_topic, from_device, from_yaml, sample_with_frequency, publish_to_topic, call_service
■ Available in **melodic** and **kinetic**: apt install ros-melodic-rxros

---

[GitHub Repository](https://github.com/ROSIN-Project/RxROS)

Some examples: [GitHub Examples](https://github.com/ROSIN-Project/RxROS_Examples)
RxROS

- RxROS is a **very thin library** (326 lines of C++ header file)
- Extends RxCPP, a reactive programming library for C++
- Adds several **ROS-specific operators**: advertiseService, from_topic, from_device, from_yaml, sample_with_frequency, publish_to_topic, call_service
- Available in **melodic** and **kinetic**: apt install ros-melodic-rxros
- Available on **GitHub** [https://github.com/rosin-project/rxros](https://github.com/rosin-project/rxros)
RxROS

- RxROS is a **very thin library** (326 lines of C++ header file)
- Extends RxCPP, a reactive programming library for C++
- Adds several **ROS-specific operators**: advertiseService, from_topic, from_device, from_yaml, sample_with_frequency, publish_to_topic, call_service
- Available in **melodic** and **kinetic**: apt install ros-melodic-rxros
- Available on **GitHub** https://github.com/rosin-project/rxros
- Some **examples** https://github.com/rosin-project/rxros_examples
auto joyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Joystick>("/joystick") //
| map([](rxros_teleop_msgs::Joystick joy) { return joy.event; });
auto keyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Keyboard>("/keyboard") //
| map([](rxros_teleop_msgs::Keyboard key) { return key.event; });
joyObsrv.merge(keyObsrv) // merge the joystick and keyboard messages;
| scan(std::make_tuple(0.0, 0.0), teleop2VelTuple) // turn the teleop stream into a linear velocity
| map(velTuple2TwistMsg) // turn the linear and angular velocities
| sample_with_frequency(frequencyInHz) // take latest Twist msg and populate
| publish_to_topic<geometry_msgs::Twist>("/cmd_vel"); // publish the Twist messages to the
VelocityPublisher / TeleOp

```c++
auto joyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Joystick>("/joystick") //
    | map([](rxros_teleop_msgs::Joystick joy) { return joy.event; });
auto keyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Keyboard>("/keyboard") //
    | map([](rxros_teleop_msgs::Keyboard key) { return key.event; });
joyObsrv.merge(keyObsrv) // merge the joystick and keyboard messages
    | scan(std::make_tuple(0.0, 0.0), teleop2VelTuple) // turn the teleop stream into a linear velocity
    | map(velTuple2TwistMsg) // turn the linear and angular velocity into a Twist message
    | sample_with_frequency(frequencyInHz) // take latest Twist msg and populate
    | publish_to_topic<geometry_msgs::Twist>("/cmd_vel"); // publish the Twist messages to the cmd_vel topic
```
VelocityPublisher / TeleOp

```cpp
auto joyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Joystick>("/joystick") //
    | map([](rxros_teleop_msgs::Joystick joy) { return joy.event; });
auto keyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Keyboard>("/keyboard") //
    | map([](rxros_teleop_msgs::Keyboard key) { return key.event; });
joyObsrv.merge(keyObsrv) // merge the joystick and keyboard messages
    | scan(std::make_tuple(0.0, 0.0), teleop2VelTuple) // turn the teleop stream into a linear and angular velocity
    | map(velTuple2TwistMsg) // turn the linear and angular velocity into a Twist msg
    | sample_with_frequency(frequencyInHz) // take latest Twist msg and populate
    | publish_to_topic<geometry_msgs::Twist>("/cmd_vel"); // publish the Twist messages to the topic
```

```
from map topic /joystick Joystick.event
```
VelocityPublisher / TeleOp

```cpp
auto joyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Joystick>("/joystick") // 
    | map([](rxros_teleop_msgs::Joystick joy) { return joy.event; });
auto keyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Keyboard>("/keyboard") // 
    | map([](rxros_teleop_msgs::Keyboard key) { return key.event; });
joyObsrv.merge(keyObsrv) // merge the joystick and keyboard messages
| scan(std::make_tuple(0.0, 0.0), teleop2VelTuple) // turn the teleop stream into a linear tuple
| map(velTuple2TwistMsg) // turn the linear and angular velocity into a twist
| sample_with_frequency(frequencyInHz) // take latest Twist msg and populate
| publish_to_topic<geometry_msgs::Twist>("/cmd_vel"); // publish the Twist messages to the topic
```

```
from map topic Joystick.event /joystick
from topic /keyboard
```
VelocityPublisher / TeleOp

```cpp
auto joyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Joystick>("/joystick") //
    | map([](rxros_teleop_msgs::Joystick joy) { return joy.event; });
auto keyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Keyboard>("/keyboard") //
    | map([](rxros_teleop_msgs::Keyboard key) { return key.event; });
joyObsrv.merge(keyObsrv) // merge the joystick and keyboard messages
    | scan(std::make_tuple(0.0, 0.0), teleop2VelTuple) // turn the teleop stream into a linear
    | map(velTuple2TwistMsg) // turn the linear and angular velocities
    | sample_with_frequency(frequencyInHz) // take latest Twist msg and populate
    | publish_to_topic<geometry_msgs::Twist>("/cmd_vel"); // publish the Twist messages to the
```

```
from topic /joystick map Joystick.event

from topic /keyboard map Keyboard.event
```
auto joyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Joystick>("/joystick") //
| map([](rxros_teleop_msgs::Joystick joy) { return joy.event; });
auto keyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Keyboard>("/keyboard") //
| map([](rxros_teleop_msgs::Keyboard key) { return key.event; });
joyObsrv.merge(keyObsrv) // merge the joystick and keyboard messages
| scan(std::make_tuple(0.0, 0.0), teleop2VelTuple) // turn the teleop stream into a linear velocity tuple
| map(velTuple2TwistMsg) // turn the linear and angular velocities to a Twist message
| sample_with_frequency(frequencyInHz) // take latest Twist msg and populate
| publish_to_topic<geometry_msgs::Twist>("/cmd_vel"); // publish the Twist messages to the topic

from /joystick  map  Joystick.event  merge  teleop event

from /keyboard  map  Keyboard.event
VelocityPublisher / TeleOp

```cpp
auto joyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Joystick>("/joystick") // 
  | map([](rxros_teleop_msgs::Joystick joy) { return joy.event; });
auto keyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Keyboard>("/keyboard") // 
  | map([](rxros_teleop_msgs::Keyboard key) { return key.event; });
joyObsrv.merge(keyObsrv) // merge the joystick and keyboard messages
  | scan(std::make_tuple(0.0, 0.0), teleop2VelTuple) // turn the teleop stream into a linear and angular velocity tuple
  | map(velTuple2TwistMsg) // turn the linear and angular velocity tuple into a Twist message
  | sample_with_frequency(frequencyInHz) // take latest Twist msg and populate
  | publish_to_topic<geometry_msgs::Twist>("/cmd_vel"); // publish the Twist messages to the /cmd_vel topic
```
VelocityPublisher / TeleOp

```cpp
auto joyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Joystick>("/joystick") //
    | map([](rxros_teleop_msgs::Joystick joy) { return joy.event; });
auto keyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Keyboard>("/keyboard") //
    | map([](rxros_teleop_msgs::Keyboard key) { return key.event; });
joyObsrv.merge(keyObsrv) // merge the joystick and keyboard msg
| scan(std::make_tuple(0.0, 0.0), teleop2VelTuple) // turn the teleop stream into a linear
| map(velTuple2TwistMsg) // turn the linear and angular velocity
| sample_with_frequency(frequencyInHz) // take latest Twist msg and populate
| publish_to_topic<geometry_msgs::Twist>("/cmd_vel"); // publish the Twist messages to the topic
```
**VelocityPublisher / TeleOp**

```cpp
auto joyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Joystick>("/joystick")
    | map([](rxros_teleop_msgs::Joystick joy) { return joy.event; });
auto keyObsrv = rxros::observable::from_topic<rxros_teleop_msgs::Keyboard>("/keyboard")
    | map([](rxros_teleop_msgs::Keyboard key) { return key.event; });
joyObsrv.merge(keyObsrv) // merge the joystick and keyboard meses
    | scan(std::make_tuple(0.0, 0.0), teleop2VelTuple) // turn the teleop stream into a linear
    | map(velTuple2TwistMsg) // turn the linear and angular velocities
    | sample_with_frequency(frequencyInHz) // take latest Twist msg and populate
    | publish_to_topic<geometry_msgs::Twist>("/cmd_vel"); // publish the Twist messages to the
```

![Diagram](image)
Challenges Ahead

- **Copying** semantics and **de-allocation** of objects rather complex in C++
  (comparing to managed languages)
Challenges Ahead

- **Copying** semantics and **de-allocation** of objects rather complex in C++ (comparing to managed languages)
- Unclear **impact on performance**, more threads (cost) but huge opportunities for parallelization (gain)
Challenges Ahead

- **Copying** semantics and **de-allocation** of objects rather complex in C++ (comparing to managed languages)
- Unclear **impact on performance**, more threads (cost) but huge opportunities for parallelization (gain)
- Some **mental cost** in changing the programming paradigm, but there is no going back :)

© Andrzej Wasowski, IT University of Copenhagen
Challenges Ahead

- **Copying** semantics and **de-allocation** of objects rather complex in C++ (comparing to managed languages)
- Unclear **impact on performance**, more threads (cost) but huge opportunities for parallelization (gain)
- Some **mental cost** in changing the programming paradigm, but there is no going back :)
- Understand **how much of ROS-based code is feasible to write this way**
RoadMap Ahead

- RxROS py
RoadMap Ahead

- RxROS py
- Action Lib
RoadMap Ahead

- RxROS py
- Action Lib
- RxROS 2, DDS

© Andrzej Wasowski, IT University of Copenhagen
RoadMap Ahead

- RxROS py
- Action Lib
- RxROS 2, DDS
- RxROS Java? Scala? C#? F#?

We seek contributors!
RoadMap Ahead

- RxROS py
- Action Lib
- RxROS 2, DDS
- RxROS Java? Scala? C#? F#?
- We seek contributors!
interval (std::chrono::milliseconds (10))
| map ([&](int i)
  { return mk_msg(hello + std::to_string(i)); })
| tap ([](const std_msgs::String& msg)
  { ROS_INFO_STREAM (msg.data); })
| publish_to_topic<std_msgs::String>
  ("/chatter", 1000);