indigo | kinetic | lunar | melodic *Show EOL distros:* ☐

Documentation Status

ros_comm (/ros_comm?distro=melodic)*: message_filters | ros (/ros?distro=melodic) | rosbag (/rosbag?distro=melodic) | rosconsole (/rosconsole?distro=melodic) | roscpp (/roscpp?distro=melodic) | rosgraph (/rosgraph?distro=melodic) | rosgraph_msgs (/rosgraph_msgs?distro=melodic) | roslaunch (/roslaunch?distro=melodic) | roslisp (/roslisp?distro=melodic) | rosmaster (/rosmaster?distro=melodic) | rosmsg (/rosmsg?distro=melodic) | rosnode (/rosnode?distro=melodic) | rosout (/rosout?distro=melodic) | rosparam (/rosparam?distro=melodic) | rospy (/rospy?distro=melodic) | rosservice (/rosservice?distro=melodic) | rostest (/rostest?distro=melodic) | rostopic (/rostopic?distro=melodic) | roswtf (/roswtf?distro=melodic) | std_srvs (/std_srvs?distro=melodic) | topic_tools (/topic_tools?distro=melodic) | xmlrpcpp (/xmlrpcpp?distro=melodic)*

Package Links

- **Code API (http://docs.ros.org/melodic/api/message_filters/html)**
- FAQ (http://answers.ros.org/questions/scope:all/sort:activity-desc/tags:message_filters/page:1/)
- Changelog (http://docs.ros.org/melodic/changelogs/message_filters/changelog.html)
- Change List (/ros_comm/ChangeList)
- Reviews (/message_filters/Reviews)

Dependencies (5)
Used by (59)
Jenkins jobs (10)

Package Summary

✓ Released **Continuous Integration: 1381 / 1382** **✓ Documented**

A set of message filters which take in messages and may output those messages at a later time, based on the conditions that filter needs met.

- Maintainer status: maintained
- Maintainer: Dirk Thomas <dthomas AT osrfoundation DOT org>
- Author: Josh Faust, Vijay Pradeep
- License: BSD
- Source: git https://github.com/ros/ros_comm.git (https://github.com/ros/ros_comm) (branch: melodic-devel)

Contents

1. Overview

`message_filters` is a utility library for use with roscpp (/roscpp) and rospy (/rospy). It collects commonly used message "filtering" algorithms into a common space. A message filter is defined as something which a message arrives into and may or may not be spit back out of at a later point in time.

An example is the time synchronizer, which takes in messages of different types from multiple sources, and outputs them only if it has received a message on each of those sources with the same timestamp.

2. Filter Pattern

All message filters follow the same pattern for connecting inputs and outputs. Inputs are connected either through the filter's constructor or through the `connectInput()` method. Outputs are connected through the `registerCallback()` method.

Note that the input and output types are defined per-filter, so not all filters are directly interconnectable.

For example, given two filters `FooFilter` and `BarFilter` where `FooFilter`'s output is compatible with `BarFilter`'s input, connecting foo to bar could be (in C++):

Toggle line numbers

```
FooFilter foo;
BarFilter bar(foo);
```

or:

Toggle line numbers

```
FooFilter foo;
BarFilter bar;
bar.connectInput(foo);
```

in Python:

Toggle line numbers

```
bar(foo)
```

Toggle line numbers

```
bar.connectInput(foo)
```

To then connect bar's output to your own callback function:

Toggle line numbers

```
bar.registerCallback(myCallback);
```

The signature of `myCallback` is dependent on the definition of `BarFilter`.

2.1 registerCallback()

You can register multiple callbacks with the `registerCallbacks()` method. They will get called in the order they are registered.

C++

In C++ `registerCallback()` returns a message_filters::Connection (http://www.ros.org/doc/api/message_filters/html/c++/classmessage__filters_1_1Connection.html) object that allows you to disconnect the callback by calling its `disconnect()` method. You do not need to store this connection object if you do not need to manually disconnect the callback.

3. Subscriber

See also: C++ message_filters::Subscriber API docs (http://www.ros.org/doc/api/message_filters/html/c++/classmessage__filters_1_1Subscriber.html) Python message_filters.Subscriber (http://www.ros.org/doc/api/message_filters/html/python/#message_filters.Subscriber)

The `Subscriber` filter is simply a wrapper around a ROS subscription that provides a source for other filters. The `Subscriber` filter cannot connect to another filter's output, instead it uses a ROS topic as its input.

3.1 Connections

Input

No input connections

Output

C++: `void callback(const boost::shared_ptr<M const>&)` **Python**: `callback(msg)`

3.2 Example (C++)

Toggle line numbers

```
message_filters::Subscriber<std_msgs::UInt32> sub(nh, "my_topic", 1);
sub.registerCallback(myCallback);
```

is the equivalent of:

Toggle line numbers

```
ros::Subscriber sub = nh.subscribe("my_topic", 1, myCallback);
```

3.3 Example (Python)

Toggle line numbers

```
sub = message_filters.Subscriber("pose_topic", robot_msgs.msg.Pose)
sub.registerCallback(myCallback)
```

4. Time Synchronizer

See also: C++ message_filters::TimeSynchronizer API docs (http://www.ros.org/doc/api/message_filters/html/c++/classmessage__filters_1_1TimeSynchronizer.html), Python message_filters.TimeSynchronizer (http://www.ros.org/doc/api/message_filters/html/python/#message_filters.TimeSynchronizer)

The `TimeSynchronizer` filter synchronizes incoming channels by the timestamps contained in their headers, and outputs them in the form of a single callback that takes the same number of channels. The C++ implementation can synchronize up to 9 channels.

4.1 Connections

Input

C++: Up to 9 separate filters, each of which is of the form `void callback(const boost::shared_ptr<M const>&)`. The number of filters supported is determined by the number of template arguments the class was created with. **Python**: N separate filters, each of which has signature `callback(msg)`.

Output

C++: For message types `M0`..`M8`, `void callback(const boost::shared_ptr<M0 const>&, ..., const boost::shared_ptr<M8 const>&)`. The number of parameters is determined by the number of template arguments the class was created with.
Python: `callback(msg0.. msgN)`. The number of parameters is determined by the number of template arguments the class was created with.

4.2 Example (C++)

Suppose you are writing a ROS node that needs to process data from two time synchronized topics. Your program will probably look something like this:

Toggle line numbers

```
#include <message_filters/subscriber.h>
#include <message_filters/time_synchronizer.h>
#include <sensor_msgs/Image.h>
#include <sensor_msgs/CameraInfo.h>

using namespace sensor_msgs;
using namespace message_filters;

void callback(const ImageConstPtr& image, const CameraInfoConstPtr& cam_info)
{
  // Solve all of perception here...
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "vision_node");

  ros::NodeHandle nh;

  message_filters::Subscriber<Image> image_sub(nh, "image", 1);
  message_filters::Subscriber<CameraInfo> info_sub(nh, "camera_info", 1);
  TimeSynchronizer<Image, CameraInfo> sync(image_sub, info_sub, 10);
  sync.registerCallback(boost::bind(&callback, _1, _2));

  ros::spin();

  return 0;
}
```

(Note: In this particular case you could use the CameraSubscriber (http://www.ros.org/doc/api/image_transport/html/classimage__transport_1_1CameraSubscriber.html) class from image_transport (/image_transport), which essentially wraps the filtering code above.)

4.3 Example (Python)

Toggle line numbers

```
import message_filters
from sensor_msgs.msg import Image, CameraInfo

def callback(image, camera_info):
  # Solve all of perception here...

image_sub = message_filters.Subscriber('image', Image)
info_sub = message_filters.Subscriber('camera_info', CameraInfo)

ts = message_filters.TimeSynchronizer([image_sub, info_sub], 10)
ts.registerCallback(callback)
rospy.spin()
```

5. Time Sequencer

See also: C++ message_filters::TimeSequencer API docs (http://www.ros.org/doc/api/message_filters/html/c++/classmessage__filters_1_1TimeSequencer.html)

Python: the `TimeSequencer` filter is not yet implemented.

The `TimeSequencer` filter guarantees that messages will be called in temporal order according to their header's timestamp. The `TimeSequencer` is constructed with a specific delay which specifies how long to queue up messages before passing them through. A callback for a message is never invoked until the messages' time stamp is out of date by at least delay. However, for all messages which are out of date by at least the delay, their callback are invoked and guaranteed to be in temporal order. If a message arrives from a time prior to a message which has already had its callback invoked, it is thrown away.

5.1 Connections

Input

C++: `void callback(const boost::shared_ptr<M const>&)`

Output

C++: `void callback(const boost::shared_ptr<M const>&)`

5.2 Example (C++)

The C++ version takes both a delay an an update rate. The update rate determines how often the sequencer will check its queue for messages that are ready to be pass through. The last argument is the number of messages to queue up before beginning to throw some away.

Toggle line numbers

```
message_filters::Subscriber<std_msgs::String> sub(nh, "my_topic", 1);
message_filters::TimeSequencer<std_msgs::String> seq(sub, ros::Duration(0.1), ros::Duration(0.01), 10);
seq.registerCallback(myCallback);
```

6. Cache

See also: C++ message_filters::Cache API docs (http://www.ros.org/doc/api/message_filters/html/c++/classmessage__filters_1_1Cache.html) Python message_filters.Cache (http://www.ros.org/doc/api/message_filters/html/python/#message_filters.Cache)

Stores a time history of messages.

Given a stream of messages, the most recent N messages are cached in a ring buffer, from which time intervals of the cache can then be retrieved by the client. The timestamp of a message is determined from its `header` field.

If the message type doesn't contain a `header`, see below for workaround.

The Cache immediately passes messages through to its output connections.

6.1 Connections

Input

C++: `void callback(const boost::shared_ptr<M const>&)` **Python**: `callback(msg)`

Output

C++: `void callback(const boost::shared_ptr<M const>&)` **Python**: `callback(msg)`

In C++:

Toggle line numbers

```
message_filters::Subscriber<std_msgs::String> sub(nh, "my_topic", 1);
message_filters::Cache<std_msgs::String> cache(sub, 100);
cache.registerCallback(myCallback);
```

In Python:

Toggle line numbers

```
sub = message_filters.Subscriber('my_topic', sensor_msgs.msg.Image)
cache = message_filters.Cache(sub, 100)
```

In this example, the `Cache` stores the last 100 messages received on `my_topic`, and `myCallback` is called on the addition of every new message. The user can then make calls like `cache.getInterval(start, end)` to extract part of the cache.

If the message type does not contain a `header` field that is normally used to determine its timestamp, and the `Cache` is contructed with `allow_headerless=True`, the current ROS time is used as the timestamp of the message. This is currently only available in Python.

Toggle line numbers

```
sub = message_filters.Subscriber('my_int_topic', std_msgs.msg.Int32)
cache = message_filters.Cache(sub, 100, allow_headerless=True)
# the cache assigns current ROS time as each message's timestamp
```

7. Policy-Based Synchronizer [ROS 1.1+]

The `Synchronizer` filter synchronizes incoming channels by the timestamps contained in their headers, and outputs them in the form of a single callback that takes the same number of channels. The C++ implementation can synchronize up to 9 channels.

The `Synchronizer` filter is templated on a policy that determines how to synchronize the channels. There are currently two policies: `ExactTime` and `ApproximateTime`.

C++ Header: `message_filters/synchronizer.h`

7.1 Connections

Input

C++: Up to 9 separate filters, each of which is of the form `void callback(const boost::shared_ptr<M const>&)`. The number of filters supported is determined by the number of template arguments the class was created with. **Python**: N separate filters, each of which has signature `callback(msg)`.

Output

C++: For message types `M0`..`M8`, `void callback(const boost::shared_ptr<M0 const>&, ..., const boost::shared_ptr<M8 const>&)`. The number of parameters is determined by the number of template arguments the class was created with. **Python**: `callback(msg0.. msgN)`. The number of parameters is determined by the number of template arguments the class was created with.

7.2 ExactTime Policy

The `message_filters::sync_policies::ExactTime` policy requires messages to have exactly the same timestamp in order to match. Your callback is only called if a message has been received on all specified channels with the same exact timestamp. The timestamp is read from the `header` field of all messages (which is required for this policy).

C++ Header: `message_filters/sync_policies/exact_time.h`

Example (C++)

Toggle line numbers

```
#include <message_filters/subscriber.h>
#include <message_filters/synchronizer.h>
#include <message_filters/sync_policies/exact_time.h>
#include <sensor_msgs/Image.h>
#include <sensor_msgs/CameraInfo.h>

using namespace sensor_msgs;
using namespace message_filters;

void callback(const ImageConstPtr& image, const CameraInfoConstPtr& cam_info)
{
  // Solve all of perception here...
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "vision_node");

  ros::NodeHandle nh;
  message_filters::Subscriber<Image> image_sub(nh, "image", 1);
  message_filters::Subscriber<CameraInfo> info_sub(nh, "camera_info", 1);

  typedef sync_policies::ExactTime<Image, CameraInfo> MySyncPolicy;
  // ExactTime takes a queue size as its constructor argument, hence MySyncPolicy(10)
  Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), image_sub, info_sub);
  sync.registerCallback(boost::bind(&callback, _1, _2));

  ros::spin();

  return 0;
}
```

7.3 ApproximateTime Policy

The `message_filters::sync_policies::ApproximateTime` policy uses an adaptive algorithm (/message_filters/ApproximateTime) to match messages based on their timestamp.

If not all messages have a `header` field from which the timestamp could be determined, see below for a workaround.

C++ Header: `message_filters/sync_policies/approximate_time.h`

Example (C++)

Toggle line numbers

```
#include <message_filters/subscriber.h>
#include <message_filters/synchronizer.h>
#include <message_filters/sync_policies/approximate_time.h>
#include <sensor_msgs/Image.h>

using namespace sensor_msgs;
using namespace message_filters;

void callback(const ImageConstPtr& image1, const ImageConstPtr& image2)
{
  // Solve all of perception here...
}

int main(int argc, char** argv)
{
  ros::init(argc, argv, "vision_node");

  ros::NodeHandle nh;
  message_filters::Subscriber<Image> image1_sub(nh, "image1", 1);
  message_filters::Subscriber<Image> image2_sub(nh, "image2", 1);

  typedef sync_policies::ApproximateTime<Image, Image> MySyncPolicy;
  // ApproximateTime takes a queue size as its constructor argument, hence MySyncPolicy(10)
  Synchronizer<MySyncPolicy> sync(MySyncPolicy(10), image1_sub, image2_sub);
  sync.registerCallback(boost::bind(&callback, _1, _2));

  ros::spin();

  return 0;
}
```

If some messages are of a type that doesn't contain the `header` field, `ApproximateTimeSynchronizer` refuses by default adding such messages. However, its Python version can be constructed with `allow_headerless=True`, which uses current ROS time in place of any missing `header.stamp` field:

Toggle line numbers

```
import message_filters
from std_msgs.msg import Int32, Float32

def callback(mode, penalty):
  # The callback processing the pairs of numbers that arrived at approximately the same time

mode_sub = message_filters.Subscriber('mode', Int32)
penalty_sub = message_filters.Subscriber('penalty', Float32)

ts = message_filters.ApproximateTimeSynchronizer([mode_sub, penalty_sub], 10, 0.1, allow_headerless=True)
ts.registerCallback(callback)
rospy.spin()
```

8. Chain [ROS 1.1+]

See also: C++ API Docs (http://www.ros.org/doc/api/message_filters/html/c++/classmessage__filters_1_1Chain.html)

The `Chain` filter allows you to dynamically chain together multiple single-input/single-output (simple) filters. As filters are added to it they are automatically connected together in the order they were added. It also allows you to retrieve added filters by index.

`Chain` is most useful for cases where you want to determine which filters to apply at runtime rather than compile-time.

8.1 Connections

Input

C++: `void callback(const boost::shared_ptr<M const>&)`

Output

C++: `void callback(const boost::shared_ptr<M const>&)`

8.2 Examples (C++)

Simple Example

Toggle line numbers

```
void myCallback(const MsgConstPtr& msg)
{
}

Chain<Msg> c;
c.addFilter(boost::shared_ptr<Subscriber<Msg> >(new Subscriber<Msg>));
c.addFilter(boost::shared_ptr<TimeSequencer<Msg> >(new TimeSequencer<Msg>));
c.registerCallback(myCallback);
```

Bare Pointers

It is possible to pass bare pointers in. These will **not** be automatically deleted when Chain is destructed.

Toggle line numbers

```
Chain<Msg> c;
Subscriber<Msg> s;
c.addFilter(&s);
c.registerCallback(myCallback);
```

Retrieving a Filter

Toggle line numbers

```
Chain<Msg> c;
size_t sub_index = c.addFilter(boost::shared_ptr<Subscriber<Msg> >(new Subscriber<Msg>));

boost::shared_ptr<Subscriber<Msg> > sub = c.getFilter<Subscriber<Msg> >(sub_index);
```

Wiki: message_filters (last edited 2018-08-14 13:56:11 by Martin Pecka (/Martin%20Pecka))