ROS DOCUMENTATION FROM “*ROS wiki*”

QUICK OVERVIEW OF GRAPH CONCEPTS

- **Nodes**: A node is an executable that uses ROS to communicate with other nodes.
- **Messages**: ROS data type used when subscribing or publishing to a topic.
- **Topics**: Nodes can publish messages to a topic as well as subscribe to a topic to receive messages.
- **Master**: Name service for ROS (i.e. helps node to find each other)
- **Rosout**: ROS equivalent of stdout/stderr
- **Roscore**: Master + rousout + parameter server

Nodes

<u>A node really isn't much more than an executable file within a ROS package. ROS nodes use a ROS client library to communicate with other nodes. Nodes can publish or subscribe to a Topic. Nodes can also provide or use a Service.</u>

A node is a process that performs computation. Nodes are combined together into a graph and communicate with one another using streaming topics, RPC services, and the Parameter Server. These nodes are meant to operate at a fine-grained scale; a robot control system will usually comprise many nodes. For example, one node controls a laser range-finder, one Node controls the robot's wheel motors, one node performs localization, one node performs path planning, one node provides a graphical view of the system, and so on.

The use of nodes in ROS provides several benefits to the overall system. There is additional fault tolerance as crashes are isolated to individual nodes. Code complexity is reduced in comparison to monolithic systems. Implementation details are also well hidden as the nodes expose a minimal API to the rest of the graph and alternate implementations, even in other programming languages, can easily be substituted.

All running nodes have a graph resource name that uniquely identifies them to the rest of the system. For example, /hokuyo_node could be the name of a Hokuyo driver broadcasting laser scans. Nodes also have a node type, that simplifies the process of referring to a node executable on the fileystem. These node types are package resource names with the name of the node's package and the name of the node executable file. In order to resolve a node type, ROS searches for all executables in the package with the specified name and chooses the first that it finds. As such, you need to be careful and not produce different executables with the same name in the same package.

A ROS node is written with the use of a ROS client library, such as roscpp or rospy.

MESSAGES

<u>msg files are simple text files for specifying the data structure of a message. These files are stored in the msg subdirectory of a package.</u>

Message types use standard ROS naming conventions: the name of the package + / + name of the .msg file. For example, std_msgs/msg/String.msg has the message type std_msgs/String.
In addition to the message type, messages are versioned by an MD5 sum of the .msg file. Nodes can only communicate messages for which both the message type and MD5 sum match.

A message may include a special message type called '**<u>Header</u>**', which includes some common metadata fields such as a timestamp and a frame ID. The ROS Client Libraries will automatically set some of these fields for you if you wish, so their use is highly encouraged.
There are three fields in the header message shown below. The seq field corresponds to an id that automatically increases as messages are sent from a given publisher. The stamp field stores time

information that should be associated with data in a message. In the case of a laser scan, for example, the stamp might correspond to the time at which the scan was taken. The frame_id field stores frame information that should be associated with data in a message. In the case of a laser scan, this would be set to the frame in which the scan was taken.

Each **field** consists of a type and a name, separated by a space.
Field types can be:

1. a built-in type, such as "float32 pan" or "string name"
2. names of Message descriptions defined on their own, such as "geometry_msgs/PoseStamped"
3. fixed- or variable-length arrays (lists) of the above, such as "float32[] ranges" or "Point32[10] points"
4. the special Header type, which maps to std_msgs/Header

When embedding other Message descriptions, the type name may be relative (e.g. "Point32") if it is in the same package; otherwise it must be the full Message type (e.g. "std_msgs/String"). The only exception to this rule is Header.
NOTE: you must not use the names of built-in types or Header when constructing your own message types.

The **field name** determines how a data value is referenced in the target language. For example, a field called 'pan' would be referenced as 'obj.pan' in Python, assuming that 'obj' is the variable storing the message.
Field names must be translated by message generators to several target languages, so we restrict field names to be an alphabetical character followed by any mixture of alphanumeric and underscores, i.e. [a-zA-Z][a-zA-Z1-9_]*. It is recommended that you avoid using field names that correspond to keywords in common languages -- although those names are legal, they create confusion as to how a field name is translated.

ROS provides the special **Header** type to provide a general mechanism for setting frame IDs for libraries like tf. While Header is not a built-in type (it's defined in std_msgs/msg/Header.msg), it is commonly used and has special semantics. If the first field of your .msg is:

```
Header header
```

It will be resolved as "std_msgs/Header".

TOPICS
Topics are named buses over which nodes exchange messages. Topics have anonymous publish/subscribe semantics, which decouples the production of information from its consumption. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic; nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic.
Topics are intended for unidirectional, streaming communication. Nodes that need to perform remote procedure calls, i.e. receive a response to a request, should use services instead. There is also the Parameter Server for maintaining small amounts of state.

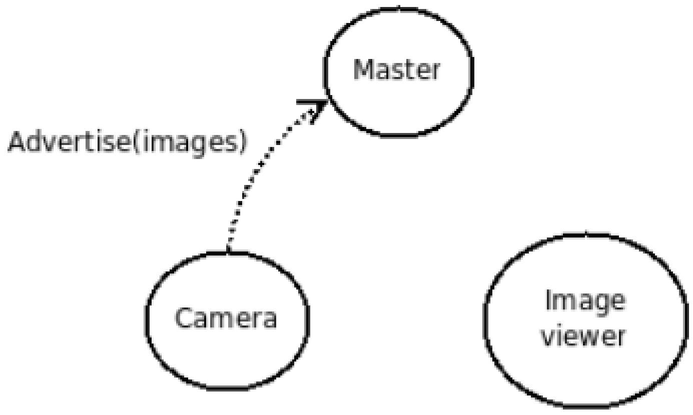
Each topic is strongly typed by the ROS message type used to publish to it and nodes can only receive messages with a matching **type**. The Master does not enforce type consistency among the publishers, but subscribers will not establish message transport unless the types match. Furthermore, all ROS clients check to make sure that an MD5 computed from the msg files match. This check ensures that the ROS Nodes were compiled from consistent code bases.

ROS currently supports TCP/IP-based and UDP-based **message transport**. The TCP/IP-based transport is known as TCPROS and streams message data over persistent TCP/IP connections. TCPROS is the default

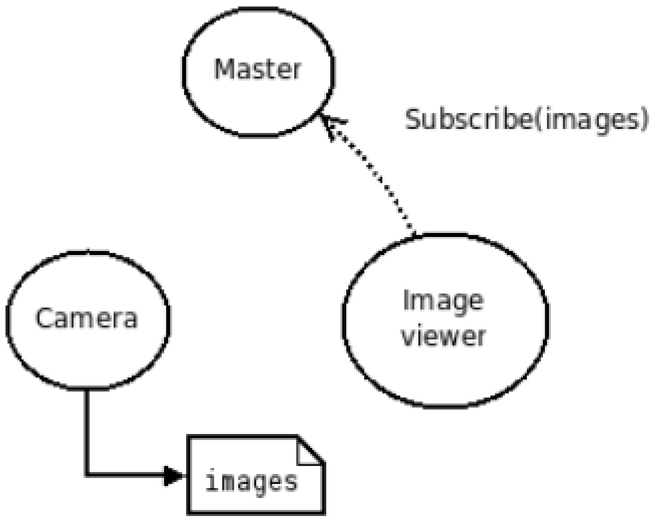
transport used in ROS and is the only transport that client libraries are required to support. The UDP-based transport, which is known as UDPROS and is currently only supported in roscpp, separates messages into UDP packets. UDPROS is a low-latency, lossy transport, so is best suited for tasks like teleoperation.
ROS nodes negotiate the desired transport at runtime. For example, if a node prefers UDPROS transport but the other Node does not support it, it can fallback on TCPROS transport. This negotiation model enables new transports to be added over time as compelling use cases arise.

MASTER
The ROS Master provides naming and registration services to the rest of the nodes in the ROS system. It tracks publishers and subscribers to topics as well as services. The role of the Master is to enable individual ROS nodes to locate one another. Once these nodes have located each other they communicate with each other peer-to-peer.
The Master also provides the Parameter Server.
The Master is most commonly run using the roscore command, which loads the ROS Master along with other essential components.

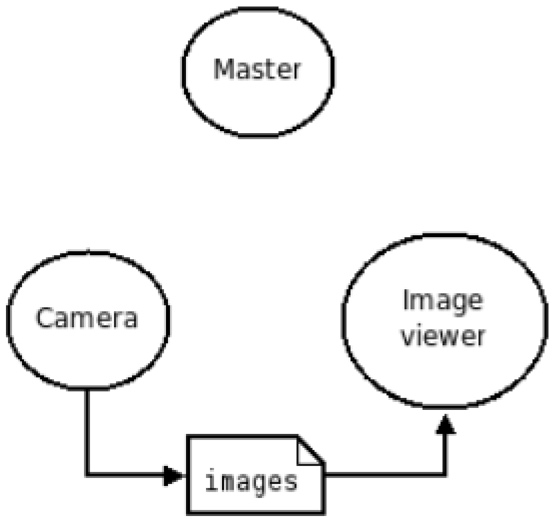
For instance, let's say we have two Nodes; a Camera node and an Image_viewer node. A typical sequence of events would start with Camera notifying the master that it wants to publish images on the topic "images":



Now, Camera publishes images to the "images" topic, but nobody is subscribing to that topic yet so no data is actually sent. Now, Image_viewer wants to subscribe to the topic "images" to see if there's maybe some images there:



Now that the topic "images" has both a publisher and a subscriber, the master node notifies Camera and Image_viewer about each others existence so that they can start transferring images to one another:



ROSOUT
rosout is the name of the console log reporting mechanism in ROS. It can be thought as comprising several components:

- The `rosout` node for subscribing, logging, and republishing the messages.
- The /rosout topic
- The /rosout_agg topic for subscribing to an aggregated feed
- rosgraph_msgs/Log message type, which defines standard fields as well as verbosity levels.
- client APIs to facilitate easy use of the rosout reporting mechanism
- GUI tools, like rqt_console, for viewing the console log messages.

The rosout package only provides the rosout node.

rosout subscribes to the standard /rosout topic, records these messages in a textual log file, and rebroadcasts the messages on /rosout_agg.
<u>Subscribed Topics</u>

- /rosout (rosgraph_msgs/Log)

Standard ROS topic for publishing logging messages.
<u>Published Topics</u>

- /rosout_agg (rosgraph_msgs/Log)

Aggregated feed of messages published to /rosout.
/rosout
ROS client libraries are required to publish console logging messages to the /rosout topic as a standard interface.
/rosout_agg
/rosout_agg is an aggregated feed for subscribing to console logging messages. This aggregated topic is offered as a performance improvement: instead of connecting to individual ROS nodes to receive their console messages, the aggregated message feed can instead be received directly from the rosout node.
rosout.log
rosout node by default logs all messages into rosout.log in the ROS log directory.

ROSCORE
roscore is a collection of nodes and programs that are pre-requisites of a ROS-based system. You must have a roscore running in order for ROS nodes to communicate. It is launched using the roscore command.
NOTE: If you use roslaunch, it will automatically start roscore if it detects that it is not already running (unless the --wait argument is supplied).

roscore will start up:

- a ROS Master
- a ROS Parameter Server
- a rosout logging node

There are currently no plans to add to roscore.
The roscore can be launched using the roscore executable.
It will also automatically be launched part of any roslaunch process if roslaunch detects that it is not running.
You can also specify a port to run the master on: `roscore -p 1234`

roscore.xml
The nodes launched as part of roscore are defined in roslaunch/roscore.xml. It is generally not recommended that new nodes be added to this file as it affects every roslaunch. These changes are also less portable to other people using ROS.
roslaunch treats nodes listed in roscore.xml differently: it searches for nodes running with the same name and will only launch ones that aren't already running.

PUBLISHING A TOPIC
You can create a handle to publish messages to a topic using the rospy.Publisher class. The most common usage for this is to provide the name of the topic and the message class/type of the topic. You can then call publish() on that handle to publish a message, e.g.:

```
pub = rospy.Publisher('topic_name', std_msgs.msg.String, queue_size=10)
pub.publish(std_msgs.msg.String("foo"))
```

rospy.Publisher initialization
rospy.Publisher(topic_name, msg_class, queue_size)
The only required arguments to create a rospy.Publisher are the topic name, the Message class, and the queue_size. Note: queue_size is only available in Hydro and newer. e.g:

```
pub = rospy.Publisher('topic_name', std_msgs.msg.String, queue_size=10)
```

There are additional advanced options that allow you to configure the Publisher:

- subscriber_listener=rospy.SubscribeListener
    - Receive callbacks via a rospy.SubscribeListener instance when new subscribers connect and disconnect.
- tcp_nodelay=False
    - Enable TCP_NODELAY on publisher’s socket, which disables Nagle algorithm on TCPROS connections. This results in lower latency publishing at the cost of efficiency.
- latch=False
    - Enable 'latching' on the connection. When a connection is latched, a reference to the last message published is saved and sent to any future subscribers that connect. This is useful for slow-changing or static data like a map. The message is not copied! If you change the message object after publishing, the change message will be sent to future subscribers.
- headers=None (dict)
    - Add additional key-value-pairs to headers for future connections.
- queue_size=None (int) [New in Hydro]
    - This is the size of the outgoing message queue used for asynchronous publishing.

Publisher.publish()
There are three different ways of calling publish() ranging from an explicit style, where you provide a Message instance, to two implicit styles that create the Message instance on the fly.

Explicit style
The explicit style is simple: you create your own Message instance and pass it to publish, e.g.:
`pub.publish(std_msgs.msg.String("hello world"))`

Implicit style with in-order arguments
In the in-order style, a new Message instance will be created with the arguments provided, in order. The argument order is the same as the order of the fields in the Message, and you must provide a value for all of the fields. For example, std_msgs.msg.String only has a single string field, so you can call:
`pub.publish("hello world")`
std_msgs.msg.ColorRGBA has four fields (r, g, b, a), so we could call:
`pub.publish(255.0, 255.0, 255.0, 128.0)`
which would create a ColorRGBA instance with r, g, and b set to 255.0 and a set to 128.0.

Implicit style with keyword arguments
In the keyword style, you only initialize the fields that you wish to provide values for. The rest receive default values (e.g. 0, empty string, etc...). For std_msgs.msg.String, the name of its lone field is data, so you can call:
`pub.publish(data="hello world")`
std_msgs.msg.ColorRGBA has four fields (r, g, b, a), so we could call:
`pub.publish(b=255)`
which would publish a ColorRGBA instance with b=255 and the rest of the fields set to 0.0.

queue_size: publish() behavior and queuing
publish() in rospy is synchronous by default (for backward compatibility reasons) which means that the invocation is blocking until:

- the messages has been serialized into a buffer
- and that buffer has been written to the transport of every current subscriber

If any of the connections has connectivity problems that might lead to publish() blocking for an indefinite amount of time. This is a common problem when subscribing to topics via a wireless connection.

As of Hydro it is recommended to use the new asynchronous publishing behavior which is more in line with the behavior of roscpp.
In order to use the new behavior the keyword argument queue_size must be passed to subscribe() which defines the maximum queue size before messages are being dropped.
While the serialization will still happen synchronously when publish() is being invoked writing the serialized data to each subscribers connection will happen asynchronously from different threads. As a result only the subscribers having connectivity problems will not receive new messages.

If you are publishing faster than rospy can send the messages over the wire, rospy will start dropping old messages.

Note that there may also be an OS-level queue at the transport level, such as the TCP/UDP send buffer.

Choosing a good queue_size
It is hard to provide a rule of thumb for what queue size is best for your application, as it depends on many variables of your system. Still, for beginners who do not care much about message passing, here we provide some guidelines:

- If you're just sending one message at a fixed rate it is fine to use a queue size as small as the frequency of the publishing.
- If you are sending multiple messages in a burst you should make sure that the queue size is big enough to contain all those messages. Otherwise it is likely to lose messages.

Generally speaking using a bigger queue_size will only use more memory when you are actually behind with the processing - so it is recommended to pick a value which is bigger than it needs to be rather than a too small value.
But if your queue is much larger than it needs to be that will queue up a lot of messages if a subscriber is lagging behind. This might lead to messages arriving with large latency since all messages will be delivered in FIFO order to the subscriber once it catches up.

queue_size Omitted

- If the keyword argument is omitted, None is passed or for Groovy and older ROS distributions the publishing is handled synchronously. As of Indigo not passing the keyword argument queue_size will result in a warning being printed to the console.

queue_size None

- Not recommended. Publishing is handled synchronously which means that one blocking subscriber will block all publishing. As of Indigo passing None will result in a warning being printed to the console.

queue_size Zero

- While a value of 0 means an infinite queue, this can be dangerous since the memory usage can grow infinitely and is therefore not recommended.

queue_size One, Two, Three

- If your system is not overloaded you could argue that a queued message should be picked up by the dispatcher thread within a tenth of a second. So a queue size of 1 / 2 / 3 would be absolutely fine when using 10 Hz.

Setting the queue_size to 1 is a valid approach if you want to make sure that a new published value will always prevent any older not yet sent values to be dropped. This is good for, say, a sensor that only cares about the latest measurement. e.g. never send older measurements if a newer one exists.

queue_size Ten or More

- An example of when to use a large queue size, such as 10 or greater, is user interface messages (e.g. digital_io, a push button status) that would benefit from a larger queue_size to prevent missing a change in value. Another example is when you want to record all published values including the ones which would be dropped when publishing with a high rate / small queue size.

SUBSCRIBING TO A TOPIC

```python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo("I heard %s",data.data)

def listener():
    rospy.init_node('node_name')
    rospy.Subscriber("chatter", String, callback)
    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()
```

Connection Information

A subscriber can get access to a "connection header", which includes debugging information such as who sent the message, as well information like whether or not a message was latched. This data is stored as the _connection_header field of a received message.

```
print m._connection_header
{'callerid': '/talker_38321_1284999593611',
 'latching': '0',
 'md5sum': '992ce8a1687cec8c8bd883ec73ca41d1',
 'message_definition': 'string data\n\n',
 'topic': '/chatter',
 'type': 'std_msgs/String'}
```

TF

Quaternion	tf::Quaternion
Vector	tf::Vector3
Point	tf::Point
Pose	tf::Pose
Transform	tf::Transform

tf::Stamped <T>
tf::Stamped<T> is templated on the above datatypes(except tf::Transform) with elements frame_id_ and stamp_

```
template <typename T>
class Stamped : public T{
 public:
  ros::Time stamp_;
  std::string frame_id_;

  Stamped() :frame_id_ ("NO_ID_STAMPED_DEFAULT_CONSTRUCTION"){}; //Default constructor used only for preallocation

  Stamped(const T& input, const ros::Time& timestamp, const std::string & frame_id);

  void setData(const T& input);
};
```

tf::StampedTransform
tf::StampedTransform is a special case of tf::Transforms which require both frame_id and stamp as well as child_frame_id.

```
/** \brief The Stamped Transform datatype used by tf */
class StampedTransform : public tf::Transform
{
public:
  ros::Time stamp_; ///< The timestamp associated with this transform
  std::string frame_id_; ///< The frame_id of the coordinate frame  in which this transform is defined
  std::string child_frame_id_; ///< The frame_id of the coordinate frame this transform defines
  StampedTransform(const tf::Transform& input, const ros::Time& timestamp, const std::string & frame_id, const std::string & child_frame_id):
    tf::Transform (input), stamp_ ( timestamp ), frame_id_ (frame_id), child_frame_id_(child_frame_id){ };

  /** \brief Default constructor only to be used for preallocation */
  StampedTransform() { };

  /** \brief Set the inherited Traonsform data */
  void setData(const tf::Transform& input){*static_cast<tf::Transform*>(this) = input;};

};
```

Frames and Points

A frame is a coordinate system. Coordinate systems in ROS are always in 3D, and are right-handed, with X forward, Y left, and Z up.
Points within a frame are represented using tf::Point, which is equivalent to the bullet type btVector3. The coordinates of a point p in a frame W are written as
$^{W}p.$

Broadcasting Transforms
To broadcast transforms within a node, we recommend using the tf.TransformBroadcaster.
Constructor
tf::TransformBroadcaster has a no argument constructor, e.g.: tf.TransformBroadcaster();
Sending transforms
To send a transform call sendTransform() with a fully populated transform.
void sendTransform(const StampedTransform & transform);
void sendTransform(const geometry_msgs::TransformStamped & transform);

PYTHON TF
Transformer
The Transformer object is the heart of tf. It maintains an internal time-varying graph of transforms, and permits asynchronous graph modification and queries.
Transformer does not handle ROS messages directly; the only ROS type it uses is rospy.Time(). Transformer also does not mandate any particular linear algebra library. Transformations involving ROS messages are done with TransformerROS (see below).
Transformer has the following methods:

- * allFramesAsDot() -> string: Returns a string representing all frames
- * allFramesAsString() -> string: Returns a string representing all frames
- * setTransform(transform, authority): Adds a new transform to the Transformer graph. The transform argument is an object with the following structure:

```
child_frame_id          string, child frame
header
    stamp               time stamp, rospy.Time
    frame_id            string, frame
transform
    translation
        x
        y
        z
    rotation
        x
        y
        z
        w
```

These members exactly match those of a ROS geometry_msgs/TransformStamped message.

- * canTransform(target_frame, source_frame, time): Returns True if the Transformer can determine the transform from source_frame to target_frame at the time time. time is a rospy.Time instance.

- * canTransformFull(target_frame, target_time, source_frame, source_time, fixed_frame): It's an extended version of canTransform.

- * chain(target_frame, target_time, source_frame, source_time, fixed_frame) -> list: returns the chain of frames connecting source_frame to target_frame.

- * frameExists(frame_id) -> Bool: returns True if frame frame_id exists in the Transformer`.

- * getFrameStrings -> list: returns all frame names in the Transformer as a list.

- * getLatestCommonTime(source_frame, target_frame) -> time: Determines that most recent time for which Transformer can compute the transform between the two given frames, that is, between source_frame and target_frame. Returns a rospy.Time.

- * lookupTransform(target_frame, source_frame, time) -> position, quaternion: Returns the transform from source_frame to target_frame at the time time. time is a rospy.Time instance. The transform is returned as position (x, y, z) and an orientation quaternion (x, y, z, w).

- * lookupTransformFull(target_frame, target_time, source_frame, source_time, fixed_frame): It is the full version of lookupTransform.

Short example showing Transformer and setTransform:

```
import tf

t = tf.Transformer(True, rospy.Duration(10.0))
m = geometry_msgs.msg.TransformStamped()
m.header.frame_id = "THISFRAME"
m.parent_id = "PARENT"
t.setTransform(m)
```

TransformerROS and TransformListener
TransformerROS extends the base class Transformer, adding methods for handling ROS messages.

- * asMatrix(target_frame, hdr) -> matrix: Looks up the transform for ROS message header hdr to frame target_frame, and returns the transform as a NumPy 4x4 matrix.

- * fromTranslationRotation(translation, rotation) -> matrix: Returns a NumPy 4x4 matrix for a transform.

- * transformPoint(target_frame, point_msg) -> point_msg: Transforms a geometry_msgs's PointStamped message, i.e. point_msg, to the frame target_frame, returns the resulting PointStamped.

- * transformPose(target_frame, pose_msg) -> pose_msg: Transforms a geometry_msgs's PoseStamped message, i.e. point_msg, to the frame target_frame, returns the resulting PoseStamped.

- * transformQuaternion(target_frame, quat_msg) -> quat_msg: Transforms a geometry_msgs's QuaternionStamped message, i.e. quat_msg, to the frame target_frame, returns the resulting QuaternionStamped.

TransformListener
TransformListener extends TransformerROS, adding a handler for ROS topic /tf_message, so that the messages update the transformer.

This example uses a TransformListener to find the current base_link position in the map.

```
import rospy
from tf import TransformListener

class myNode:

    def __init__(self, *args):
        self.tf = TransformListener()
        rospy.Subscriber(...)
        ...

    def some_method(self):
        if self.tf.frameExists("/base_link") and self.tf.frameExists("/map"):
            t = self.tf.getLatestCommonTime("/base_link", "/map")
            position, quaternion = self.tf.lookupTransform("/base_link", "/map", t)
            print position, quaternion
```

A simple example to transform a pose from one frame to another.

```
import rospy
import tf

class myNode:
    def __init__(self, *args):
        self.tf_listener_ = TransformListener()

    def example_function(self):
        if self.tf.frameExists("/base_link") and self.tf.frameExists("/fingertip"):
            t = self.tf_listener_.getLatestCommonTime("/base_link", "/fingertip")
            p1 = geometry_msgs.msg.PoseStamped()
            p1.header.frame_id = "fingertip"
            p1.pose.orientation.w = 1.0    # Neutral orientation
            p_in_base = self.tf_listener_.transformPose("/base_link", p1)
            print "Position of the fingertip in the robot base:"
            print p_in_base
```

Exceptions used in the tf package
All exceptions in tf inherit from tf::TransformException, which inherits from std::runtime_error.

- tf::ConnectivityException: Thrown if the request cannot be completed due to the two frame ids not being in the same connected tree.
- tf::ExtrapolationException: Thrown if there is a connection between the frame ids requested but one or more of the transforms are out of date.
- tf::InvalidArgument: Thrown if an argument is invalid. The most common case is an unnormalized quaternion.
- tf::LookupException: Thrown if an unpublished frame id is referenced.

WRITING THE PUBLISHER NODE

"Node" is the ROS term for an executable that is connected to the ROS network. Here we'll create the publisher ("talker") node which will continually broadcast a message.

```python
#!/usr/bin/env python
# license removed for brevity
import rospy
from std_msgs.msg import String

def talker():
    pub = rospy.Publisher('chatter', String, queue_size=10
    rospy.init_node('talker', anonymous=True)
    rate = rospy.Rate(10) # 10hz
    while not rospy.is_shutdown():
        hello_str = "hello world %s" % rospy.get_time()
        rospy.loginfo(hello_str)
        pub.publish(hello_str)
        rate.sleep()

if __name__ == '__main__':
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

Every Python ROS Node will have this declaration at the top. The first line makes sure your script is executed as a Python script.

```python
import rospy
from std_msgs.msg import String
```

You need to import rospy if you are writing a ROS Node. The std_msgs.msg import is so that we can reuse the std_msgs/String message type (a simple string container) for publishing.

```python
    pub = rospy.Publisher('chatter', String, queue_size=10)
    rospy.init_node('talker', anonymous=True)
```

This section of code defines the talker's interface to the rest of ROS. pub = rospy.Publisher("chatter", String, queue_size=10) declares that your node is publishing to the chatter topic using the message type String. String here is actually the class std_msgs.msg.String. The queue_size argument is New in ROS hydro and limits the amount of queued messages if any subscriber is not receiving them fast enough. In older ROS distributions just omit the argument.

The next line, rospy.init_node(NAME, ...), is very important as it tells rospy the name of your node -- until rospy has this information, it cannot start communicating with the ROS Master. In this case, your node will take on the name talker. NOTE: the name must be a base name, i.e. it cannot contain any slashes "/".

anonymous = True ensures that your node has a unique name by adding random numbers to the end of NAME. Refer to Initialization and Shutdown - Initializing your ROS Node in the rospy documentation for more information about node initialization options.

rate = rospy.Rate(10) # 10hz

This line creates a Rate object rate. With the help of its method sleep(), it offers a convenient way for looping at the desired rate. With its argument of 10, we should expect to go through the loop 10 times per second (as long as our processing time does not exceed 1/10th of a second!)

```
while not rospy.is_shutdown():
    hello_str = "hello world %s" % rospy.get_time()
    rospy.loginfo(hello_str)
    pub.publish(hello_str)
    rate.sleep()
```

This loop is a fairly standard rospy construct: checking the rospy.is_shutdown() flag and then doing work. You have to check is_shutdown() to check if your program should exit (e.g. if there is a Ctrl-C or otherwise). In this case, the "work" is a call to pub.publish(hello_str) that publishes a string to our chatter topic. The loop calls rate.sleep(), which sleeps just long enough to maintain the desired rate through the loop.

(You may also run across rospy.sleep() which is similar to time.sleep() except that it works with simulated time as well (see Clock).)

This loop also calls rospy.loginfo(str), which performs triple-duty: the messages get printed to screen, it gets written to the Node's log file, and it gets written to rosout. rosout is a handy tool for debugging: you can pull up messages using rqt_console instead of having to find the console window with your Node's output.

std_msgs.msg.String is a very simple message type, so you may be wondering what it looks like to publish more complicated types. The general rule of thumb is that constructor args are in the same order as in the .msg file. You can also pass in no arguments and initialize the fields directly, e.g.

```
msg = String()
msg.data = str
```

or you can initialize some of the fields and leave the rest with default values:

```
String(data=str)
```

You may be wondering about the last little bit:

```
    try:
        talker()
    except rospy.ROSInterruptException:
        pass
```

In addition to the standard Python __main__ check, this catches a rospy.ROSInterruptException exception, which can be thrown by rospy.sleep() and rospy.Rate.sleep() methods when Ctrl-C is pressed or your Node is otherwise shutdown. The reason this exception is raised is so that you don't accidentally continue executing code after the sleep().

WRITING THE SUBSCRIBER NODE

```
#!/usr/bin/env python
import rospy
from std_msgs.msg import String

def callback(data):
    rospy.loginfo(rospy.get_caller_id() + "I heard %s", data.data)

def listener():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # name are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)

    rospy.Subscriber("chatter", String, callback)

    # spin() simply keeps python from exiting until this node is stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

The code for listener.py is similar to talker.py, except we've introduced a new callback-based mechanism for subscribing to messages.

```
rospy.init_node('listener', anonymous=True)

rospy.Subscriber("chatter", String, callback)

# spin() simply keeps python from exiting until this node is stopped
rospy.spin()
```

This declares that your node subscribes to the chatter topic which is of type *std_msgs.msgs.String*. When new messages are received, *callback* is invoked with the message as the first argument.

We also changed up the call to *rospy.init_node*() somewhat. We've added the anonymous=True keyword argument. ROS requires that each node have a unique name. If a node with the same name comes up, it bumps the previous one. This is so that malfunctioning nodes can easily be kicked off the network. The anonymous=True flag tells rospy to generate a unique name for the node so that you can have multiple listener.py nodes run easily.

The final addition, *rospy.spin()* simply keeps your node from exiting until the node has been shutdown. Unlike roscpp, rospy.spin() does not affect the subscriber callback functions, as those have their own threads.