ign_transport Documentation

Release 3.0.1

Open Source Robotics Foundation

Jul 14, 2017

Contents

Contents:

CHAPTER 1

What is Ignition Transport?

Ignition Transport is an open source communication library that allows sharing data between clients. In our context, a client is called a node. Nodes might be running within the same process in the same machine or in machines located in different continents. Ignition Transport is multi-platform (Linux, Mac OS X, and Windows), so all the low level details, such as data alignment or endianness are hidden for you.

Ignition Transport uses Google Protocol buffers as the data type for communicating between nodes. Users can define their own messages using the Protobuf utils, and then, exchange them between the nodes. Ignition Transport discovers, serializes and delivers messages to the destinations using a combination of custom code and ZeroMQ.

- What programming language can I use to interface Ignition Transport?

C++ is our native implementation and so far the only way to use the library. We might offer different wrappers for the most popular languages in the future.

CHAPTER 2

Installation

Instructions to install Ignition Transport on all the platforms supported: major Linux distributions, Mac OS X and Windows.

Next, you can see the major Ignition Transport versions, their availability and lifetime.

Version	Available on Ubuntu directly	Available on Ubuntu via OSRF	Available on MacOS via Homebrew tab	Since	EOL
0.y	`Ubuntu X`	`Ubuntu T`	–	February 2015	April 2021
1.y	`Ubuntu Y,Z`	`Ubuntu T,X`	`Yosemite,El Capitan`	February 2016	January 2018
2.y	–	`Ubuntu T,X,Y`	`Yosemite,El Capitan`	August 2016	TBD
3.y	–	`Ubuntu T,X,Y`	`Yosemite,El Capitan`	January 2017	TBD

Ubuntu Linux

Setup your computer to accept software from *packages.osrfoundation.org*:

```
sudo sh -c 'echo "deb http://packages.osrfoundation.org/gazebo/ubuntu-stable
`lsb_release -cs` main" > /etc/apt/sources.list.d/gazebo-stable.list'
```

Setup keys:

```
wget http://packages.osrfoundation.org/gazebo.key -O - | sudo apt-key add -
```

Install Ignition Transport:

```
sudo apt-get update
sudo apt-get install libignition-transport2-dev
```

Mac OS X

Ignition Transport and several of its dependencies can be compiled on OS X with Homebrew using the osrf/simulation tap. Ignition Transport is straightforward to install on Mac OS X 10.9 (Mavericks) or higher. Installation on older versions requires changing the default standard library and rebuilding dependencies due to the use of c++11. For purposes of this documentation, I will assume OS X 10.9 or greater is in use. Here are the instructions:

Install homebrew, which should also prompt you to install the XCode command-line tools:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/
↪install)"
```

Run the following commands:

```
brew tap osrf/simulation
brew install ignition-transport2
```

Windows

At this moment, compilation has been tested on Windows 7 and 8.1 and is supported when using Visual Studio 2013. Patches for other versions are welcome.

This installation procedure uses pre-compiled binaries in a local workspace. To make things easier, use a MinGW shell for your editing work (such as the Git Bash Shell with Mercurial), and only use the Windows cmd for configuring and building. You might also need to disable the Windows firewall.

Make a directory to work in, e.g.:

```
mkdir ign-ws
cd ign-ws
```

Download the following dependencies into that directory:

- cppzmq
- Protobuf 2.6.0 (32-bit)
- Protobuf 2.6.0 (64-bit)

Choose one of these options:

- ZeroMQ 4.0.4 (32-bit)
- ZeroMQ 4.0.4 (64-bit)

Unzip each of them. The Windows unzip utility will likely create an incorrect directory structure, where a directory with the name of the zip contains the directory that has the source files. Here is an example:

```
ign-ws/cppzmq-noarch/cppzmq
```

The correct structure is

```
ign-ws/cppzmq
```

To fix this problem, manually move the nested directories up one level.

Clone and prepare the Ignition Math dependency:

```
hg clone https://bitbucket.org/ignitionrobotics/ign-math
cd ign-math
mkdir build
```

In a Windows Command Prompt, load your compiler setup, e.g.:

```
"C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\vcvarsall.bat" amd64
```

In the Windows Command Prompt, configure and build:

```
cd ign-math\build
..\configure
nmake install
```

Clone and prepare the Ignition Msgs dependency:

```
hg clone https://bitbucket.org/ignitionrobotics/ign-msgs
cd ign-msgs
mkdir build
```

In the Windows Command Prompt, configure and build:

```
cd ign-msgs\build
..\configure
nmake install
```

Clone ign-transport:

```
hg clone https://bitbucket.org/ignitionrobotics/ign-transport
cd ign-transport
```

In a Windows Command Prompt, load your compiler setup, e.g.:

```
"C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\vcvarsall.bat" amd64
```

Configure and build:

```
mkdir build
cd build
..\configure
nmake
nmake install
```

You should now have an installation of ign-transport in `ign-ws/ign-transport/build/install`.

Before running any executables, you need to modify your `PATH` to include the `bin` subdirectory of ZeroMQ to let Windows find dynamic libs (similar to `LD_LIBRARY_PATH` on Linux). Don't put quotes around the path, even if it contains spaces. E.g., if you're working in `C:\My Stuff\ign-ws`:

```
set PATH %PATH%;C:\My Stuff\ign-ws\ZeroMQ 4.0.4\bin
```

Now build the examples:

```
cd ign-ws\ign-transport\example
mkdir build
cd build
..\configure
nmake
```

Now try an example. In one Windows terminal run:

```
responser
```

In another Windows terminal run:

```
requester
```

Install from sources (Ubuntu Linux)

For compiling the latest version of Ignition Transport you will need an Ubuntu distribution equal to 14.04.2 (Trusty) or newer.

Make sure you have removed the Ubuntu pre-compiled binaries before installing from source:

```
sudo apt-get remove libignition-transport2-dev
```

Install prerequisites. A clean Ubuntu system will need:

```
sudo apt-get install mercurial cmake pkg-config python ruby-ronn libprotoc-dev␣
↪libprotobuf-dev protobuf-compiler uuid-dev libzmq3-dev libignition-msgs-dev
```

Clone the repository into a directory and go into it:

```
hg clone https://bitbucket.org/ignitionrobotics/ign-transport /tmp/ign-transport
cd /tmp/ign-transport
```

Create a build directory and go there:

```
mkdir build
cd build
```

Configure Ignition Transport (choose either method a or b below):

1. Release mode: This will generate optimized code, but will not have debug symbols. Use this mode if you don't need to use GDB.

```
cmake ../
```

Note: You can use a custom install path to make it easier to switch between source and debian installs:

```
cmake -DCMAKE_INSTALL_PREFIX=/home/$USER/local ../
```

B. Debug mode: This will generate code with debug symbols. Ignition Transport will run slower, but you'll be able to use GDB.

```
cmake -DCMAKE_BUILD_TYPE=Debug ../
```

The output from `cmake ../` may generate a number of errors and warnings about missing packages. You must install the missing packages that have errors and re-run `cmake ../`. Make sure all the build errors are resolved before continuing (they should be there from the earlier step in which you installed prerequisites).

Make note of your install path, which is output from cmake and should look something like:

```
-- Install path: /home/$USER/local
```

Build Ignition Transport:

```
make -j4
```

Install Ignition Transport:

```
sudo make install
```

If you decide to install gazebo in a local directory you'll need to modify your `LD_LIBRARY_PATH`:

```
echo "export LD_LIBRARY_PATH=<install_path>/local/lib:$LD_LIBRARY_PATH" >> ~/.bashrc
```

Uninstalling Source-based Install

If you need to uninstall Ignition Transport or switch back to a debian-based install when you currently have installed the library from source, navigate to your source code directory's build folders and run `make uninstall`:

```
cd /tmp/ign-transport/build
sudo make uninstall
```

CHAPTER 3

Understanding nodes and topics

Nodes

The communication in Ignition Transport follows a pure distributed architecture, where there is no central process, broker or similar. All the nodes in the network can act as publishers, subscribers, provide services and request services.

A publisher is a node that produces information and a subscriber is a node that consumes information. There are two categories or ways to communicate in Ignition Transport. First, we could use a publish/subscribe approach, where a node advertises a topic, and then, publishes periodic updates. On the other side, one or more nodes subscribe to the same topic registering a function that will be executed each time a new message is received. An alternative communication paradigm is based on service calls. A service call is a remote service that a node offers to the rest of the nodes. A node can request a service in a similar way a local function is executed.

Topics

A topic is just a name for grouping a specific set of messages or a particular service. Imagine that you have a camera and want to periodically publish its images. Your node could advertise a topic called `/image`, and then, publish a new message on this topic every time a new image is available. Other nodes, will subscribe to the same topic and will receive the messages containing the image. A node could also offer an echo service in the topic `/echo`. Any node interested in this service will request a service call on topic `/echo`. The service call will accept arguments and will return a result. In our echo service example, the result will be similar to the input parameter passed to the service.

There are some rules to follow when selecting a topic name. It should be any alphanumeric name followed by zero or more slashes. For example: `/image`, `head_position`, `/robot1/joints/HeadPitch` are examples of valid topic names. The next table summarizes the allowed and not allowed topic rules.

Topic name	Validity	Comment
/topicA	Valid	
/topicA/	Valid	Equivalent to */topicA*
topicA	Valid	
/a/b	Valid	
	Invalid	Empty string is invalid
my topic	Invalid	Contains white space
//image	Invalid	Contains two consecutive `/`
/	Invalid	`/` topic is not allowed
~myTopic	Invalid	Symbol ~ not allowed

Topic scope

A topic can be optionally advertised with a scope. A scope allows you to set the visibility of this topic. The available scopes are `Process`, `Host`, and `All`. A `Process` scope means that the advertised topic will only be visible in the nodes within the same process as the advertiser. A topic with a `Host` scope restricts the visibility of a topic to nodes located in the same machine as the advertiser. Finally, by specifying a scope with an `All` value, you're allowing your topic to be visible by any node.

Partition and namespaces

When you create your node you can specify some options to customize its behavior. Among those options you can set a partition name and a namespace.

A partition is used to isolate a set of topics or services within a group of nodes that share the same partition name. E.g.: Node1 advertises topic `/foo` and Node2 advertises `/foo` too. If we don't use a partition, a node subscribed to `/foo` will receive the messages published from Node1 and Node2. Alternatively, we could specify `p1` as a partition for Node1 and `p2` as a partition for Node2. When we create the node for our subscriber, if we specify `p1` as a partition name, we'll receive the messages published only by Node1. If we use `p2`, we'll only receive the messages published by Node2. If we don't set a partition name, we won't receive any messages from Node1 or Node2.

A partition name is any alphanumeric string with a few exceptions. The symbol `/` is allowed as part of a partition name but just `/` is not allowed. The symbols `@`, ~ or white spaces are not allowed as part of a partition name. Two or more consecutive slashes (`//`) are not allowed.

The default partition name is created using a combination of your hostname, followed by `:` and your username. E.g.: `bb8:caguero` . It's also possible to use the environment variable `IGN_PARTITION` for setting a custom partition name.

A namespace is considered a prefix that might be potentially applied to some of the topic/services advertised in a node.

E.g.: Node1 sets a namespace `ns1` and advertises the topics `t1`, `t2` and `/t3`. `/t3` is considered an absolute topic (starts with `/`) and it won't be affected by a namespace. However, `t1` and `t2` will be advertised as `/ns1/t1` and `/ns1/t2`.

A namespace is any alphanumeric string with a few exceptions. The symbol `/` is allowed as part of a namespace but just `/` is not allowed. The symbols `@`, ~ or white spaces are not allowed as part of a namespace. Two or more consecutive slashes (`//`) are not allowed. If topic name or namespace is invalid than fully qualified topic name is

invalid too.

Namespace	Topic name	Fully qualified topic	Validity	Comment
ns1	*/topicA*	*/topicA*	Valid	Absolute topic
	/topicA	*/topicA*	Valid	Absolute topic
ns1	*topicA*	*/ns1/topicA*	Valid	
	topicA	*/topicA*	Valid	
ns1	*topic A*		Invalid	Topic contains white space
	topic A		Invalid	Topic contains white space
my ns	*topicA*		Invalid	Namespace contains white space
//ns	*topicA*		Invalid	Namespace contains two consecutive //
/	*topicA*		Invalid	/ namespace is not allowed
~myns	*topicA*		Invalid	Symbol ~ not allowed

CHAPTER 4

Node communication via messages

In this tutorial, we are going to create two nodes that are going to communicate via messages. One node will be a publisher that generates the information, whereas the other node will be the subscriber consuming the information. Our nodes will be running on different processes within the same machine.

```
mkdir ~/ign_transport_tutorial
cd ~/ign_transport_tutorial
```

Publisher

Download the publisher.cc file within the `ign_transport_tutorial` folder and open it with your favorite editor:

```
#include <atomic>
#include <chrono>
#include <csignal>
#include <iostream>
#include <string>
#include <thread>
#include <ignition/msgs.hh>
#include <ignition/transport.hh>

/// \brief Flag used to break the publisher loop and terminate the program.
static std::atomic<bool> g_terminatePub(false);

//////////////////////////////////////////////////
/// \brief Function callback executed when a SIGINT or SIGTERM signals are
/// captured. This is used to break the infinite loop that publishes messages
/// and exit the program smoothly.
void signal_handler(int _signal)
{
  if (_signal == SIGINT || _signal == SIGTERM)
    g_terminatePub = true;
}
```

```
//////////////////////////////////////////////////
int main(int argc, char **argv)
{
  // Install a signal handler for SIGINT and SIGTERM.
  std::signal(SIGINT,  signal_handler);
  std::signal(SIGTERM, signal_handler);

  // Create a transport node and advertise a topic.
  ignition::transport::Node node;
  std::string topic = "/foo";

  auto pub = node.Advertise<ignition::msgs::StringMsg>(topic);
  if (!pub)
  {
    std::cerr << "Error advertising topic [" << topic << "]" << std::endl;
    return -1;
  }

  // Prepare the message.
  ignition::msgs::StringMsg msg;
  msg.set_data("HELLO");

  // Publish messages at 1Hz.
  while (!g_terminatePub)
  {
    if (!pub.Publish(msg))
      break;

    std::cout << "Publishing hello on topic [" << topic << "]" << std::endl;
    std::this_thread::sleep_for(std::chrono::milliseconds(1000));
  }

  return 0;
}
```

Walkthrough

```
#include <ignition/msgs.hh>
#include <ignition/transport.hh>
```

The line `#include <ignition/transport.hh>` contains all the Ignition Transport headers for using the transport library.

The next line includes the generated protobuf code that we are going to use for our messages. We are going to publish `StringMsg` type protobuf messages.

```
// Create a transport node and advertise a topic.
ignition::transport::Node node;
std::string topic = "/foo";

auto pub = node.Advertise<ignition::msgs::StringMsg>(topic);
if (!pub)
{
  std::cerr << "Error advertising topic [" << topic << "]" << std::endl;
  return -1;
```

```
}
```

First of all we declare a *Node* that will offer some of the transport functionality. In our case, we are interested on publishing topic updates, so the first step is to announce our topic name and its type. Once a topic name is advertised, we can start publishing periodic messages using the publisher object.

```
// Prepare the message.
ignition::msgs::StringMsg msg;
msg.set_data("HELLO");

// Publish messages at 1Hz.
while (!g_terminatePub)
{
  if (!pub.Publish(msg))
    break;

  std::cout << "Publishing hello on topic [" << topic << "]" << std::endl;
  std::this_thread::sleep_for(std::chrono::milliseconds(1000));
}
```

In this section of the code we create a protobuf message and fill it with content. Next, we iterate in a loop that publishes one message every second. The method *Publish()* sends a message to all the subscribers.

Subscriber

Download the subscriber.cc file within the `ign_transport_tutorial` folder and open it with your favorite editor:

```
#include <iostream>
#include <string>
#include <ignition/msgs.hh>
#include <ignition/transport.hh>

//////////////////////////////////////////////////
/// \brief Function called each time a topic update is received.
void cb(const ignition::msgs::StringMsg &_msg)
{
  std::cout << "Msg: " << _msg.data() << std::endl << std::endl;
}

//////////////////////////////////////////////////
int main(int argc, char **argv)
{
  ignition::transport::Node node;
  std::string topic = "/foo";

  // Subscribe to a topic by registering a callback.
  if (!node.Subscribe(topic, cb))
  {
    std::cerr << "Error subscribing to topic [" << topic << "]" << std::endl;
    return -1;
  }

  // Zzzzzz.
  ignition::transport::waitForShutdown();
```

```
  return 0;
}
```

Walkthrough

```
//////////////////////////////////////////////////
/// \brief Function called each time a topic update is received.
void cb(const ignition::msgs::StringMsg &_msg)
{
  std::cout << "Msg: " << _msg.data() << std::endl << std::endl;
}
```

We need to register a function callback that will execute every time we receive a new topic update. The signature of the callback is always similar to the one shown in this example with the only exception of the protobuf message type. You should create a function callback with the appropriate protobuf type depending on the type of the topic advertised. In our case, we know that topic `/foo` will contain a Protobuf `StringMsg` type.

```
ignition::transport::Node node;
std::string topic = "/foo";

// Subscribe to a topic by registering a callback.
if (!node.Subscribe(topic, cb))
{
  std::cerr << "Error subscribing to topic [" << topic << "]" << std::endl;
  return -1;
}
```

After the node creation, the method `Subscribe()` allows you to subscribe to a given topic name by specifying your subscription callback function.

```
// Zzzzzz.
ignition::transport::waitForShutdown();
```

If you don't have any other tasks to do besides waiting for incoming messages, you can use the call *waitForShutdown()* that will block your current thread until you hit *CTRL-C*. Note that this function captures the *SIGINT* and *SIGTERM* signals.

Building the code

Download the CMakeLists.txt file within the `ign_transport_tutorial` folder.

Once you have all your files, go ahead and create a `build/` directory within the `ign_transport_tutorial` directory.

```
mkdir build
cd build
```

Run `cmake` and build the code.

```
cmake ..
make publisher subscriber
```

Running the examples

Open two new terminals and from your `build/` directory run the executables.

From terminal 1:

```
./publisher
```

From terminal 2:

```
./subscriber
```

In your subscriber terminal, you should expect an output similar to this one, showing that your subscriber is receiving the topic updates:

```
caguero@turtlebot:~/ign_transport_tutorial/build$ ./subscriber
Data: [helloWorld]
Data: [helloWorld]
Data: [helloWorld]
Data: [helloWorld]
Data: [helloWorld]
Data: [helloWorld]
```

Advertise Options

We can specify some options before we publish the messages. One such option is to specify the number of messages published per topic per second. It is optional to use but it can be handy in situations like where we want to control the rate of messages published per topic.

We can declare the throttling option using the following code :

```
// Create a transport node and advertise a topic with throttling enabled.
ignition::transport::Node node;
std::string topic = "/foo";

// Setting the throttling option
ignition::transport::AdvertiseMessageOptions opts;
opts.SetMsgsPerSec(1u);

auto pub = node.Advertise<ignition::msgs::StringMsg>(topic, opts);
if (!pub)
{
  std::cerr << "Error advertising topic [" << topic << "]" << std::endl;
  return -1;
}
```

Walkthrough

```
ignition::transport::AdvertiseMessageOptions opts;
opts.SetMsgsPerSec(1u);
```

In this section of code, we declare an *AdvertiseMessageOptions* object and use it to pass message rate as argument to *SetMsgsPerSec()* method. In our case, the object name is opts and message rate specified is 1 msg/sec.

```
auto pub = node.Advertise<ignition::msgs::StringMsg>(topic, opts);
```

Next, we advertise the topic with message throttling enabled. To do it, we pass opts as argument to *Advertise()* method.

Subscribe Options

A similar option has also been provided to the Subscriber node which enables it to control the rate of incoming messages from a specific topic. While subscribing to a topic, we can use this option to control the number of messages received per second from that particular topic.

We can declare the throttling option using the following code :

```
// Create a transport node and subscribe to a topic with throttling enabled.
ignition::transport::Node node;
ignition::transport::SubscribeOptions opts;
opts.SetMsgsPerSec(1u);
node.Subscribe(topic, cb, opts);
```

Walkthrough

```
ignition::transport::SubscribeOptions opts;
opts.SetMsgsPerSec(1u);
node.Subscribe(topic, cb, opts);
```

In this section of code, we declare a *SubscribeOptions* object and use it to pass message rate as argument to *SetMsgsPerSec()* method. In our case, the object name is opts and message rate specified is 1 msg/sec. Then, we subscribe to the topic using *Subscribe()* method with opts passed as arguments to it.

CHAPTER 5

Node communication via services

In this tutorial, we are going to create two nodes that are going to communicate via services. You can see a service as a function that is going to be executed in a different node. Services have two main components: a service provider and a service consumer. A service provider is the node that offers the service to the rest of the world. The service consumers are the nodes that request the function offered by the provider. Note that in Ignition Transport the location of the service is hidden. The discovery layer of the library is in charge of discovering and keeping and updated list of services available.

In the next tutorial, one node will be the service provider that offers an *echo* service, whereas the other node will be the service consumer requesting an *echo* call.

```
mkdir ~/ign_transport_tutorial
cd ~/ign_transport_tutorial
```

Responser

Download the responser.cc file within the `ign_transport_tutorial` folder and open it with your favorite editor:

```
#include <iostream>
#include <string>
#include <ignition/msgs.hh>
#include <ignition/transport.hh>

//////////////////////////////////////////////////
/// \brief Provide an "echo" service.
void srvEcho(const ignition::msgs::StringMsg &_req,
  ignition::msgs::StringMsg &_rep, bool &_result)
{
  // Set the response's content.
  _rep.set_data(_req.data());

  // The response succeed.
  _result = true;
```

```
}

//////////////////////////////////////////////////
int main(int argc, char **argv)
{
  // Let's print the list of our network interfaces.
  std::cout << "List of network interfaces in this machine:" << std::endl;
  for (const auto &netIface : ignition::transport::determineInterfaces())
    std::cout << "\t" << netIface << std::endl;

  // Create a transport node.
  ignition::transport::Node node;
  std::string service = "/echo";

  // Advertise a service call.
  if (!node.Advertise(service, srvEcho))
  {
    std::cerr << "Error advertising service [" << service << "]" << std::endl;
    return -1;
  }

  // Zzzzzz.
  ignition::transport::waitForShutdown();
}
```

Walkthrough

```
#include <ignition/msgs.hh>
#include <ignition/transport.hh>
```

The line `#include <ignition/transport.hh>` contains the Ignition Transport header for using the transport library.

The next line includes the generated Protobuf code that we are going to use for our messages. We are going to use `StringMsg` type Protobuf messages for our services.

```
//////////////////////////////////////////////
/// \brief Provide an "echo" service.
void srvEcho(const ignition::msgs::StringMsg &_req,
  ignition::msgs::StringMsg &_rep, bool &_result)
{
  // Set the response's content.
  _rep.set_data(_req.data());

  // The response succeed.
  _result = true;
}
```

As a service provider, our node needs to register a function callback that will execute every time a new service request is received. The signature of the callback is always similar to the one shown in this example with the exception of the Protobuf messages types for the `_req` (request) and `_rep` (response). The request parameter contains the input parameters of the request. The response message contains any resulting data from the service call. The `_result` parameter denotes if the overall service call was considered successful or not. In our example, as a simple *echo* service, we just fill the response with the same data contained in the request.

```
// Create a transport node.
ignition::transport::Node node;
std::string service = "/echo";

// Advertise a service call.
if (!node.Advertise(service, srvEcho))
{
  std::cerr << "Error advertising service [" << service << "]" << std::endl;
  return -1;
}

// Zzzzzz.
ignition::transport::waitForShutdown();
```

We declare a *Node* that will offer all the transport functionality. In our case, we are interested in offering a service, so the first step is to announce our service name. Once a service name is advertised, we can accept service requests.

If you don't have any other tasks to do besides waiting for service requests, you can use the call *waitForShutdown()* that will block your current thread until you hit *CTRL-C*. Note that this function captures the *SIGINT* and *SIGTERM* signals.

Synchronous requester

Download the requester.cc file within the `ign_transport_tutorial` folder and open it with your favorite editor:

```
#include <iostream>
#include <ignition/msgs.hh>
#include <ignition/transport.hh>

//////////////////////////////////////////////////
int main(int argc, char **argv)
{
  // Create a transport node.
  ignition::transport::Node node;

  // Prepare the input parameters.
  ignition::msgs::StringMsg req;
  req.set_data("HELLO");

  ignition::msgs::StringMsg rep;
  bool result;
  unsigned int timeout = 5000;

  // Request the "/echo" service.
  bool executed = node.Request("/echo", req, timeout, rep, result);

  if (executed)
  {
    if (result)
      std::cout << "Response: [" << rep.data() << "]" << std::endl;
    else
      std::cout << "Service call failed" << std::endl;
  }
  else
    std::cerr << "Service call timed out" << std::endl;
}
```

Walkthrough

```
// Create a transport node.
ignition::transport::Node node;

// Prepare the input parameters.
ignition::msgs::StringMsg req;
req.set_data("HELLO");

ignition::msgs::StringMsg rep;
bool result;
unsigned int timeout = 5000;
```

We declare the *Node* that allows us to request a service. Next, we declare and fill the message used as an input parameter for our *echo* request. Then, we declare the Protobuf message that will contain the response and the variable that will tell us if the service request succeed or failed. In this example, we will use a synchronous request, meaning that our code will block until the response is received or a timeout expires. The value of the timeout is expressed in milliseconds.

```
// Request the "/echo" service.
bool executed = node.Request("/echo", req, timeout, rep, result);

if (executed)
{
  if (result)
    std::cout << "Response: [" << rep.data() << "]" << std::endl;
  else
    std::cout << "Service call failed" << std::endl;
}
else
  std::cerr << "Service call timed out" << std::endl;
```

In this section of the code we use the method `Request()` for forwarding the service call to any service provider of the service `/echo`. Ignition Transport will find a node, communicate the input data, capture the response and pass it to your output parameter. The return value will tell you if the request expired or the response was received. The `result` value will tell you if the service provider considered the operation valid.

Imagine for example that we are using a division service, where our input message contains the numerator and denominator. If there are no nodes offering this service, our request will timeout (return value `false`). On the other hand, if there's at least one node providing the service, the request will return `true` signaling that the request was received. However, if we set our denominator to `0` in the input message, `result` will be `false` reporting that something went wrong in the request. If the input parameters are valid, we'll receive a result value of `true` and we can use our response message.

Asynchronous requester

Download the requester_async.cc file within the `ign_transport_tutorial` folder and open it with your favorite editor:

```
#include <iostream>
#include <ignition/msgs.hh>
#include <ignition/transport.hh>

//////////////////////////////////////////////////
/// \brief Service response callback.
```

```
void responseCb(const ignition::msgs::StringMsg &_rep, const bool _result)
{
  if (_result)
    std::cout << "Response: [" << _rep.data() << "]" << std::endl;
  else
    std::cerr << "Service call failed" << std::endl;
}

//////////////////////////////////////////////////
int main(int argc, char **argv)
{
  // Create a transport node.
  ignition::transport::Node node;

  // Prepare the input parameters.
  ignition::msgs::StringMsg req;
  req.set_data("HELLO");

  std::cout << "Press <CTRL-C> to exit" << std::endl;

  // Request the "/echo" service.
  node.Request("/echo", req, responseCb);

  // Zzzzzz.
  ignition::transport::waitForShutdown();
}
```

Walkthrough

```
//////////////////////////////////////////////////
/// \brief Service response callback.
void responseCb(const ignition::msgs::StringMsg &_rep, const bool _result)
{
  if (_result)
    std::cout << "Response: [" << _rep.data() << "]" << std::endl;
  else
    std::cerr << "Service call failed" << std::endl;
}
```

We need to register a function callback that will execute when we receive our service response. The signature of the callback is always similar to the one shown in this example with the only exception of the Protobuf message type used in the response. You should create a function callback with the appropriate Protobuf type depending on the response type of the service requested. In our case, we know that the service `/echo` will answer with a Protobuf *StringMsg*' type.

```
// Create a transport node.
ignition::transport::Node node;

// Prepare the input parameters.
ignition::msgs::StringMsg req;
req.set_data("HELLO");

// Request the "/echo" service.
node.Request("/echo", req, responseCb);
```

In this section of the code we declare a node and a Protobuf message that is filled with the input parameters for

our request. Next, we just use the asynchronous variant of the `Request()` method that forwards a service call to any service provider of the service `/echo`. Ignition Transport will find a node, communicate the data, capture the response and pass it to your callback, in addition of the service call result. Note that this variant of `Request()` is asynchronous, so your code will not block while your service request is handled.

Oneway responser

Not all the service requests require a response. In these cases we can use a oneway service to process service requests without sending back responses. Oneway services don't accept any output parameters nor the requests have to wait for the response.

Download the responser_oneway.cc file within the `ign_transport_tutorial` folder and open it with your favorite editor:

```
#include <iostream>
#include <string>
#include <ignition/transport.hh>
#include <ignition/msgs.hh>

//////////////////////////////////////////////////
void srvOneway(const ignition::msgs::StringMsg &_req)
{
  std::cout << "Request received: [" << _req.data() << "]" << std::endl;
}

//////////////////////////////////////////////////
int main(int argc, char **argv)
{
  // Create a transport node.
  ignition::transport::Node node;
  std::string service = "/oneway";

  // Advertise a oneway service.
  if (!node.Advertise(service, srvOneway))
  {
    std::cerr << "Error advertising service [" << service << "]" << std::endl;
    return -1;
  }

  // Zzzzzz.
  ignition::transport::waitForShutdown();
}
```

Walkthrough

```
//////////////////////////////////////////////////
void srvOneway(const ignition::msgs::StringMsg &_req)
{
  std::cout << "Request received: [" << _req.data() << "]" << std::endl;
}
```

As a oneway service provider, our node needs to advertise a service that doesn't send a response back. The signature of the callback contains only one parameter that is the input parameter, `_req` (request). We don't need `_rep` (response)

or `_result` as there is no response expected. In our example, the value of the input parameter is printed on the screen.

```
// Create a transport node.
ignition::transport::Node node;
std::string service = "/oneway";

// Advertise a oneway service.
if (!node.Advertise(service, srvOneway))
{
  std::cerr << "Error advertising service [" << service << "]" << std::endl;
  return -1;
}
```

We declare a *Node* that will offer all the transport functionality. In our case, we are interested in offering a oneway service, so the first step is to announce our service name. Once a service name is advertised, we can accept service requests.

Oneway requester

This case is similar to the oneway service provider. This code can be used for requesting a service that does not need a response back. We don't need any output parameters in this case nor we have to wait for the response.

Download the requester_oneway.cc file within the `ign_transport_tutorial` folder and open it with your favorite editor:

```
#include <iostream>
#include <ignition/transport.hh>
#include <ignition/msgs.hh>

//////////////////////////////////////////////////
int main(int argc, char **argv)
{
  // Create a transport node.
  ignition::transport::Node node;

  // Prepare the input parameters.
  ignition::msgs::StringMsg req;
  req.set_data("HELLO");

  // Request the "/oneway" service.
  bool executed = node.Request("/oneway", req);

  if (!executed)
    std::cerr << "Service call failed" << std::endl;
}
```

Walkthrough

```
// Create a transport node.
ignition::transport::Node node;

// Prepare the input parameters.
ignition::msgs::StringMsg req;
```

```
req.set_data("HELLO");

// Request the "/oneway" service.
bool executed = node.Request("/oneway", req);

if (!executed)
std::cerr << "Service call failed" << std::endl;
```

First of all we declare a node and a Protobuf message that is filled with the input parameters for our `/oneway` service. Next, we just use the oneway variant of the `Request()` method that forwards a service call to any service provider of the service `/oneway`. Ignition Transport will find a node and communicate the data without waiting for the response. The return value of `Request()` indicates if the request was successfully queued. Note that this variant of `Request()` is also asynchronous, so your code will not block while your service request is handled.

Service without input parameter

Sometimes we want to receive some result but don't have any input parameter to send.

Download the responser_no_input.cc file within the `ign_transport_tutorial` folder and open it with your favorite editor:

```
#include <iostream>
#include <string>
#include <ignition/msgs.hh>
#include <ignition/transport.hh>

//////////////////////////////////////////////////
/// \brief Provide a "quote" service.
/// Well OK, it's just single-quote service but do you really need more?
void srvQuote(ignition::msgs::StringMsg &_rep, bool &_result)
{
  std::string awesomeQuote = "This is it! This is the answer. It says here..."
    "that a bolt of lightning is going to strike the clock tower at precisely "
    "10:04pm, next Saturday night! If...If we could somehow...harness this "
    "lightning...channel it...into the flux capacitor...it just might work. "
    "Next Saturday night, we're sending you back to the future!";

  // Set the response's content.
  _rep.set_data(awesomeQuote);

  // The response succeed.
  _result = true;
}

//////////////////////////////////////////////////
int main(int argc, char **argv)
{
  // Create a transport node.
  ignition::transport::Node node;
  std::string service = "/quote";

  // Advertise a service call.
  if (!node.Advertise(service, srvQuote))
  {
    std::cerr << "Error advertising service [" << service << "]" << std::endl;
```

```
    return -1;
  }

  // Zzzzzz.
  ignition::transport::waitForShutdown();
}
```

Walkthrough

```
void srvQuote(ignition::msgs::StringMsg &_rep, bool &_result)
```

Service doesn't receive anything. The signature of the callback contains two parameters `_rep` (response) and `_result`. In our example, we return the quote.

```
// Create a transport node.
ignition::transport::Node node;
std::string service = "/quote";

// Advertise a service call.
if (!node.Advertise(service, srvQuote))
{
  std::cerr << "Error advertising service [" << service << "]" << std::endl;
  return -1;
}

// Zzzzzz.
ignition::transport::waitForShutdown();
```

We declare a *Node* that will offer all the transport functionality. In our case, we are interested in offering service without input, so the first step is to announce the service name. Once a service name is advertised, we can accept service requests.

Empty requester sync and async

This case is similar to the service without input parameter. We don't send any request.

Download the requester_no_input.cc file within the `ign_transport_tutorial` folder and open it with your favorite editor:

```
#include <iostream>
#include <ignition/msgs.hh>
#include <ignition/transport.hh>

//////////////////////////////////////////////////
int main(int argc, char **argv)
{
  // Create a transport node.
  ignition::transport::Node node;

  ignition::msgs::StringMsg rep;
  bool result;
  unsigned int timeout = 5000;

  // Request the "/quote" service.
```

```
  bool executed = node.Request("/quote", timeout, rep, result);

  if (executed)
  {
    if (result)
      std::cout << "Response: [" << rep.data() << "]" << std::endl;
    else
      std::cout << "Service call failed" << std::endl;
  }
  else
    std::cerr << "Service call timed out" << std::endl;
}
```

Walkthrough

First of all we declare a node and a message that will contain the response from `/quote` service. Next, we use the variant without input parameter of the `Request()` method. The return value of `Request()` indicates whether the request timed out or reached the service provider and `result` shows if the service was successfully executed.

We also have the async version for service request without input. You should download requester_no_input.cc file within the `ign_transport_tutorial` folder.

Building the code

Download the CMakeLists.txt file within the `ign_transport_tutorial` folder. Then, download CMakeLists.txt and stringmsg.proto inside the `msgs` directory.

Once you have all your files, go ahead and create a `build/` folder within the `ign_transport_tutorial` directory.

```
mkdir build
cd build
```

Run `cmake` and build the code.

```
cmake ..
make responser responser_oneway requester requester_async requester_oneway
make responser_no_input requester_no_input requester_async_no_input
```

Running the examples

Open three new terminals and from your `build/` directory run the executables.

From terminal 1:

```
./responser
```

From terminal 2:

```
./requester
```

From terminal 3:

```
./requester_async
```

In your requester terminals, you should expect an output similar to this one, showing that your requesters have received their responses:

```
caguero@turtlebot:~/ign_transport_tutorial/build$ ./requester
Response: [Hello World!]
```

```
caguero@turtlebot:~/ign_transport_tutorial/build$ ./requester_async
Response: [Hello World!]
```

For running the oneway examples, open two terminals and from your `build/` directory run the executables.

From terminal 1:

```
./responser_oneway
```

From terminal 2:

```
./requester_oneway
```

In your responser terminal, you should expect an output similar to this one, showing that your service provider has received a request:

```
caguero@turtlebot:~/ign_transport_tutorial/build$ ./responser_oneway
Request received: [HELLO]
```

For running the examples without input, open three terminals and from your `build/` directory run the executables.

From terminal 1:

```
./responser_no_input
```

From terminal 2:

```
./requester_no_input
```

From terminal 3:

```
./requester_async_no_input
```

In your requesters' terminals, you should expect an output similar to this one, showing that you have received a response:

```
caguero@turtlebot:~/ign_transport_tutorial/build$ ./requester_no_input
Response: [This is it! This is the answer. It says here...that a bolt of
lightning is going to strike the clock tower at precisely 10:04pm, next
Saturday night! If...If we could somehow...harness this lightning...channel
it...into the flux capacitor...it just might work. Next Saturday night,
we're sending you back to the future!]
```

CHAPTER 6

Configuration via environment variables

In a similar way you can programatically customize the behavior of your nodes or specify some options when you advertise a topic, it is possible to use an environment variable to tweak the behavior of Ignition Transport. Next you can see a description of the available environment variables:

Environment variable	Value allowed	Description
IGN_PARTITION	Any partition value	Specifies a partition name for all the nodes declared inside this process. Note that an alternative partition name declared programatically and passed to the constructor of a Node class will take priority over *IGN_PARTITION*.
IGN_IP	Any local IP address	This setting is needed in situations where you have multiple IP addresses for a computer and need to force Ignition Transport to use a particular one. This setting is only required if you advertise a topic or a service. If you are only subscribed to topics or requesting services you don't need to use this option because the discovery service will try all the available network interfaces during the search of the topic/service.
IGN_VERBOSE	1	Show debug information.

CHAPTER 7

How to contribute

Ignition Transport is an open source project based on the Apache License Version 2.0, and is maintained by hard-working developers for everyone's benefit. If you would like to contribute software patches, read on to find out how. You'll probably want to check out the Development Section for learning about the internal design of the library when planning your contribution.

Development process

We follow a development process designed to reduce errors, encourage collaboration, and make high quality code. The process may seem rigid and tedious, but every step is worth the effort (especially if you like applications that work).

Steps to follow

1. Are you sure? Has your idea already been done, or maybe someone is already working on it?

   Check the issue tracker.

2. Fork Ignition Transport. This will create your own personal copy of the project. All of your development should take place in your fork.

3. Work out of a branch:

```
hg branch my_new_branch_name
```

Always work out of a new branch, never off of *default*. This is a good habit to get in, and will make your life easier. If you're solving an issue, make the branch name `issue_` followed by the issue number. E.g.: `issue_23`.

4. Write your code.

   This is the fun part.

5. Write tests.

A pull request will only be accepted if it has tests. See the *Test coverage* section below for more information.

6. Compiler warnings.

   Code must have zero compile warnings. This currently only applies to Linux.

7. Style.

   A tool is provided to check for correct style. Your code must have no errors after running the following command from the root of the source tree:

```
sh tools/code_check.sh
```

The tool does not catch all style errors. See the *Style* section below for more information.

8. Tests pass.

   There must be no failing tests. You can check by running `make test` in your build directory.

9. Documentation.

   Document all your code. Every class, function, member variable must have doxygen comments. All code in source files must have documentation that describes the functionality. This will help reviewers, and future developers.

10. Review your code.

    Before submitting your code through a pull request, take some time to review everything line-by-line. The review process will go much faster if you make sure everything is perfect before other people look at your code. There is a bit of the human-condition involved here. Folks are less likely to spend time reviewing your code if it's bad.

11. Small pull requests.

    A large pull request is hard to review, and will take a long time. It is worth your time to split a large pull request into multiple smaller pull requests. For reference, here are a few examples:

    - Small, very nice
    - Medium, still okay
    - Too large

12. Pull request.

    Submit a pull request when you ready.

13. Review.

    At least two other people have to approve your pull request before it can be merged. Please be responsive to any questions and comments.

14. Done, phew.

    Once you have met all the requirements, you're code will be merged. Thanks for improving Ignition Transport!

Internal Developers

This section is targeted mostly for people who have commit access to the main repositories.

In addition to the general development process, please follow these steps before submitting a pull request. Each step is pass/fail, where the test or check must pass before continuing to the next step.

1. Run the style checker on your personal computer.
2. Run all tests on your personal computer.
3. Run your branch through a jenkins trusty build.
4. Run your branch through a jenkins homebrew build.
5. Run your branch through a jenkins windows7 build.
6. Submit the pull request, and include the following:

1. Link to a passing trusty build.
2. Link to a passing homebrew build.
3. Link to a passing windows7 build.

7. A set of jenkins jobs will run automatically once the pull request is created. Reviewers can reference these automatic jobs and the jenkins jobs listed in your pull request.

Style

In general, we follow Google's style guide. However, we add in some extras.

''this'' pointer All class attributes and member functions must be accessed using the `this->` pointer. Here is an example.

Underscore function parameters All function parameters must start with an underscore. Here is an example.

Do not cuddle braces All braces must be on their own line. Here is an example.

Multi-line code blocks If a block of code spans multiple lines and is part of a flow control statement, such as an `if`, then it must be wrapped in braces. Here is an example

++ operator This occurs mostly in `for` loops. Prefix the ++ operator, which is slightly more efficient than postfix in some cases.

PIMPL/Opaque pointer If you are writing a new class, it must use a private data pointer. Here is an example, and you can read more here.

const functions Any class function that does not change a member variable should be marked as `const`. Here is an example.

const parameters All parameters that are not modified by a function should be marked as `const`. This applies to parameters that are passed by reference, pointer, and value. Here is an example.

Pointer and reference variables Place the `*` and `&` next to the variable name, not next to the type. For example: `int &variable` is good, but `int& variable` is not. Here is an example.

Camel case In general, everything should use camel case. Exceptions include protobuf variable names.

Class function names Class functions must start with a capital letter, and capitalize every word.

`void MyFunction();` : Good

`void myFunction();` : Bad

`void my_function();` : Bad

Variable names Variables must start with a lower case letter, and capitalize every word thereafter.

`int myVariable;` : Good

`int myvariable;` : Bad

`int my_variable;` : Bad

Reduce Code Duplication

Check to make sure someone else is not currently working on the same feature, before embarking on a project to add something to Ignition Transport. Check the issue tracker looking for issues with similar ideas.

Write Tests

All code should have a corresponding unit test. Ignition Transport uses GTest for unit testing.

Test coverage

The goal is to achieve 100% line and branch coverage. However, this is not always possible due to complexity issues, analysis tools misreporting coverage, and time constraints. Try to write as complete of a test suite as possible, and use the coverage analysis tools as guide. If you have trouble writing a test please ask for help in your pull request.

Ignition Transport has a build target called `make coverage` that will produce a code coverage report. You'll need lcov installed.

1. In your `build` folder, compile Ignition Transport with `-DCMAKE_BUILD_TYPE=Coverage`:

```
cmake -DCMAKE_BUILD_TYPE=Coverage ..\
make
```

2. Run a single test, or all the tests:

```
make test
```

3. Make the coverage report:

```
make coverage
```

4. View the coverage report:

```
firefox coverage/index.html
```

Debugging Ignition Transport

Meaningful backtraces

In order to provide meaningful backtraces when using a debugger, such as GDB, Ignition Transport should be compiled with debugging support enabled. When using the ubuntu packages, specially the `-dbg` package, this support is limited but could be enough in most of the situations. This are the three level of traces which can be obtained:

Maximum level of debugging support This only can be obtained compiling Ignition Transport from source and setting the `CMAKE_BUILD_TYPE` to `DEBUG`. This will set up no optimizations and debugging symbols. It can be required by developers in situations specially difficult to reproduce.

Medium level of debugging support This can be obtained installing the `libignition-transport1-dbg` package or compiling Ignition Transport from source using the `RELWITHDEBINFO` `CMAKE_BUILD_TYPE` mode (which is the default if no mode is provided). This will set up `-O2` optimization level but provide debugging symbols. This should be the default when firing up gdb to explore errors and submit traces.

Minimum level of debugging support This one is present in package versions (no `-dbg` package present) or compiling Ignition Transport from source using the `RELEASE CMAKE_BUILD_TYPE` option. This will set up the maximum level of optimizations and does not provide any debugging symbol information. This traces are particularly difficult to follow.

Code Check

Code pushed into the repository should pass a few simple tests. It is also helpful if patches submitted through bitbucket pass these tests. Passing these tests is defined as generating no error or warning messages for each of the following tests.

Static Code Check

Static code checking analyzes your code for bugs, such as potential memory leaks, and style. The Ignition Transport static code checker uses cppcheck, and a modified cpplint. You'll need to install cppcheck on your system. Ubuntu users can install via:

```
sudo apt-get install cppcheck
```

To check your code, run the following script from the root of the Ignition Transport sources:

```
sh tools/code_check.sh
```

It takes a few minutes to run. Fix all errors and warnings until the output looks like:

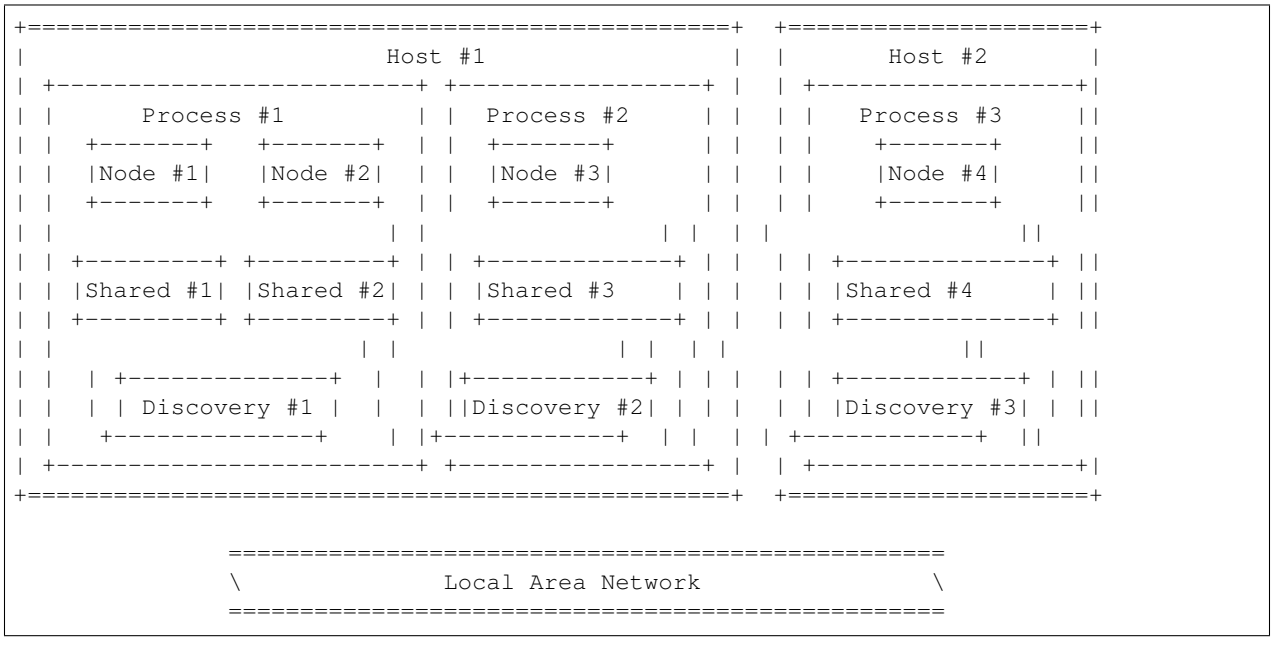
```
Total errors found: 0
```

CHAPTER 8

Internal architecture

The purpose of this section is to describe the internal design of Ignition Transport. You don't need to read this section if you just want to use the library in your code. This section will help you to understand our source code if you're interested in making code contributions.

Ignition Transport's internal architecture can be illustrated with the following diagram:



Next, are the most important components of the library:

1. Node.

   This class is the main interface with the users. The `Node` class contains all the functions that allow users to advertise, subscribe and publish topics, as well as advertise and request services. This is the only class that a user should directly use.

2. NodeShared (shown as `Shared` in the diagram for space purposes).

   A single instance of a `NodeShared` class is shared between all the `Node` objects running inside the same process. The `NodeShared` instance contains all the ZMQ sockets used for sending and receiving data for topic and service communication. The goal of this class is to share resources between a group of nodes.

3. Discovery.

   A discovery layer is required in each process to learn about the location of topics and services. Our topics and services don't have any location information, they are just plain strings, so we need a way to learn where are they located (similar to a DNS service). `Discovery` uses a custom protocol and UDP multicast for communicating with other `Discovery` instances. These instances can be located on the same or different machines over the same LAN. At this point is not possible to discover a `Node` outside of the LAN, this is a future request that will eventually be added to the library.

Discovery service

Communication occurs between nodes via named data streams, called topics. Each node has a universally unique id (UUID) and may run on any machine in a local network. A mechanism, called discovery, is needed to help nodes find each other and the topics that they manage.

The Discovery class implements the protocol for distributed node discovery. The topics are plain strings (`/echo`, `/my_robot/camera`) and this layer learns about the meta information associated to each topic. The topic location, the unique identifier of the node providing a service or its process are some examples of the information that the discovery component learns for each topic. The main responsibility of the discovery is to keep an updated list of active topics ready to be queried by other entities.

In Ignition Transport we use two discovery objects, each one operating on a different UDP port. One object is dedicated to topics and the other is dedicated to services.

API

The first thing to do before using a discovery object is to create it. The `Discovery` class constructor requires a parameter for specifying the UDP port to be used by the discovery sockets and the UUID of the process in which the discovery is running. This UUID will be used when announcing a local topic.

Once a `Discovery` object is created it won't discover anything. You'll need to call the `Start()` function for enabling the discovery.

Besides discovering topics from the outside world, the discovery will announce the topics that are offered in the same process that the discovery is running. The `Advertise()` function will register a local topic and announce it over the network. The symmetric `Unadvertise()` will notify that a topic won't be offered anymore.

`Discover()` is used to learn about a given topic as soon as possible. It's important to remark about the "as soon as possible" because discovery will eventually learn about all the topics but this might take some time (depending on your configuration). If a client needs to know about a particular topic, `Discover()` will trigger a discovery request that will reduce the time needed to discover the information about a topic.

As you can imagine, exchanging messages over the network can be slow and we cannot block the users waiting for discovery information. We don't even know how many nodes are on the network so it would be hard and really slow to block and return all the information to our users when available. The way we tackle the notification inside `Discovery` is through callbacks. A discovery user needs to register two callbacks: one for receiving notifications when new topics are available and another for notifying when a topic is no longer active. The functions `ConnectionsCb()` and `DisconnectionsCb()` allow the discovery user to set these two notification callbacks. For example, a user will invoke the `Discover()` call and, after some time, its `ConnectionCb` will be executed

with the information about the requested topic. In the meantime, other callback invocations could be triggered because `Discovery` will pro-actively learn about all the available topics and generate notifications.

You can check the complete API details *here*.

[Un]Announce a local topic

This feature registers a new topic in the internal data structure that keeps all the discovery information. Local and remote topics are stored in the same way, the only difference is that the local topics will share the process UUID with the discovery service. We store what we call a `Publisher`, which contains the topic name and all the associated meta-data.

Each publisher advertises the topic with a specific scope as described here. If the topic's scope is `PROCESS`, the discovery won't announce it over the network. Otherwise, it will send to the multicast group an `ADVERTISE` message with the following format:

```
HEADER
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|            Version            |     Process UUID Length       |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
\                          Process UUID                         \
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Message Type |              Flags            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The value of the `Message Type` field in the header is `[UN]ADVERTISE`.

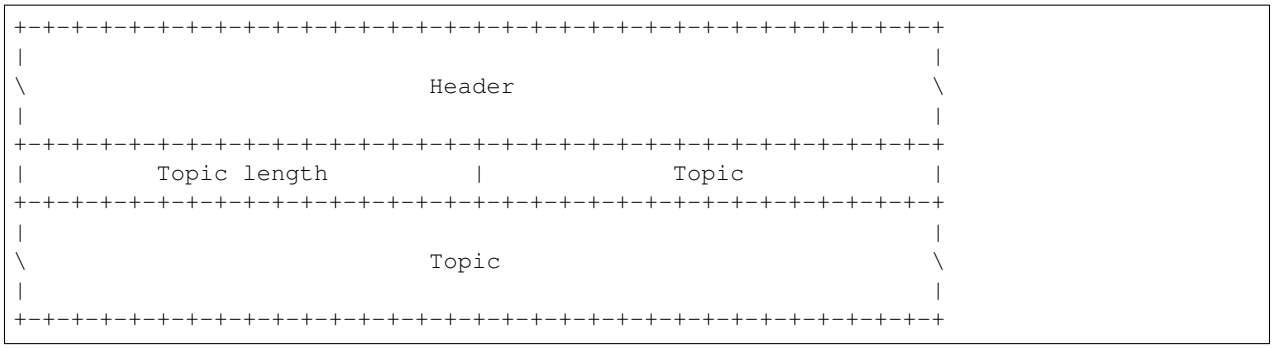
```
[UN]ADVERTISE
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
\                            Header                             \
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
\                     Serialized Publisher                      \
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

All discovery nodes will receive this request and should update its discovery information and notify its user via the notification callbacks if they didn't have previous information about the topic received. An `ADVERTISE` message should trigger the connection callback, while an `UNADVERTISE` message should fire the disconnection callback.

Trigger a topic discovery

A user can call `Discover()` for triggering the immediate discovery of a topic. Over the wire, this call will generate a `SUBSCRIBE` message with the following format:

```
SUBSCRIBE
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```



The value of the `Message Type` field in the header is `SUBSCRIBE`.

All discovery instances listening on the same port where the `SUBSCRIBE` message was sent will receive the message. Each discovery instance with a local topic registered should answer with an `ADVERTISE` message. The answer is a multicast message too that should be received by all discovery instances.

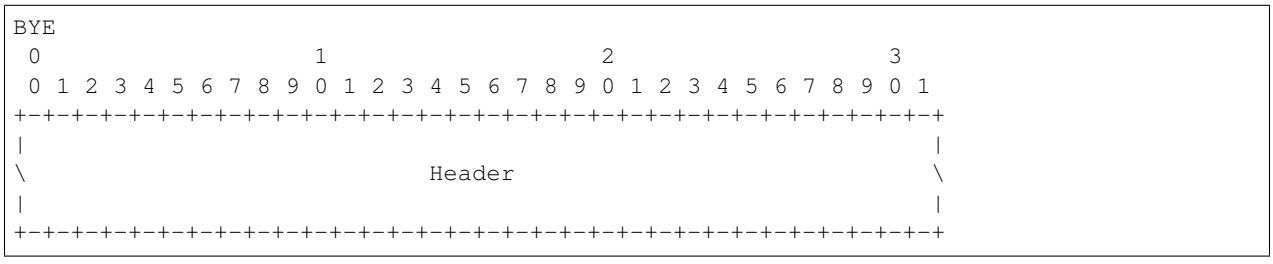
Topic update

Each discovery instance should periodically send an `ADVERTISE` message per local topic announced over the multicast channel to notify that all information already announced is still valid. The frequency of sending these topic update messages can be changed with the function `SetHeartbeatInterval()`. By default, the topic update frequency is set to one second.

Alternatively, we could replace the send of all `ADVERTISE` messages with one `HEARTBEAT` message that contains the process UUID of the discovery instance. Upon reception, all other discovery instances should update all their entries associated with the received process UUID. Although this approach is more efficient and saves some messages sent over the network, it prevents a discovery instance to learn about topics available without explicitly asking for them. We think this is a good feature to have. For example, an introspection tool that shows all the topics available can take advantage of this feature without any prior knowledge.

It is the responsibility of each discovery instance to cancel any topic that hasn't been updated for a while. The function `SilenceInterval()` sets the maximum time that an entry should be stored in memory without hearing an `ADVERTISE` message. Every `ADVERTISE` message received should refresh the topic timestamp associated with it.

When a discovery instance terminates, it should notify through the discovery channel that all its topics need to invalidated. This is performed by sending a `BYE` message with the following format:



The value of the `Message Type` field in the header is `BYE`.

When this message is received, a discovery instance should invalidate all entries associated with the process UUID contained in the header. Note that this is the expected behavior when a discovery instance gently terminates. In the case of an abrupt termination, the lack of topic updates will cause the same result, although it'll take a bit more time.

Threading model

A discovery instance will create an additional internal thread when the user calls `Start()`. This thread takes care of the topic update tasks. This involves the reception of other discovery messages and the update of the discovery information. Also, it's among its responsibilities to answer with an `ADVERTISE` message when a `SUBSCRIBE` message is received and there are local topics available.

The first time announcement of a local topic and the explicit discovery request of a topic happen on the user thread. So, in a regular scenario where the user doesn't share discovery among other threads, all the discovery operations will run in two threads, the user thread and the internal discovery thread spawned after calling `Start()`. All the functions in the discovery are thread safe.

Multiple network interfaces

The goal of the discovery service is to discover all topics available. It's not uncommon these days that a machine has multiple network interfaces for its wired and wireless connections, a virtual machine, or a localhost device, among others. By selecting one network interface and listening only on this one, we would miss the discovery messages that are sent by instances sitting on other subnets.

Our discovery service handles this problem in several steps. First, it learns about the network interfaces that are available locally. The `determineInterfaces()` function (contained in `NetUtils` file) returns a list of all the network interfaces found on the machine. When we know all the available network interfaces we create a container of sockets, one per local IP address. These sockets are used for sending discovery data over the network, flooding all the subnets and reaching other potential discovery instances.

We use one of the sockets contained in the vector for receiving data via the multicast channel. We have to join the multicast group for each local network interface but we can reuse the same socket. This will guarantee that our socket will receive the multicast traffic coming from any of our local network interfaces. This is the reason for having a single `bind()` function in our call even if we can receive data from multiple interfaces. Our receiving socket is the one we register in the `zmq::poll()` function for processing incoming discovery data.

When it's time to send outbound data, we iterate through the list of sockets and send the message over each one, flooding all the subnets with our discovery requests.

Note that the result of `determineInterfaces()` can be manually set by using the `IGN_IP` environment variable, as described *here*. This will essentially ignore other network interfaces, isolating all discovery traffic through the specified interface.

CHAPTER 9

API

Please, visit this link for version 3.x.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search