Rosxmpp : Bridging ROS Networks over XMPP

Pierre-Luc Bacon

December 18, 2011

1 Introduction

`rosxmpp` is a java-based program that allows to bridge a ROS subscriber to a remote ROS publisher attached to a different ROS master over Internet. The XMPP protocol is used to create an overlay network of ROS masters exposing their interface with a combination of the Jabber-RPC extension and Jingle ICE-UDP transport method. No code modification is needed for a local subscriber to reach a remote publisher. Through the `rosxmpp` command, topics at a specified remote master can be *imported* to the local ros master. From the point of view of the local subscribers, the set of imported topics does not differ from the others since a dedicated `rosxmppbridge` process handles the interaction with the local subscribers and hide the communication with the remote `rosxmppbridge` instance.

Participating ROS masters can be defined within a XMPP roster and rosxmpp instances can be made aware of their availability using the XMPP presence mechanism.

Because of performance considerations, Jingle ICE-UDP was chosen as the transport mechanism for the notification channel. Furthermore, this extension allows for NAT traversal so that the participating `rosxmpp` instances can communicate in a peer-to-peer fashion. In the current implementation, the TCPROS protocol is considered for tunnelling over Jingle ICE-UDP and relevant parts of the Master and Slave XMP-RPC ROS API are implemented. Note that `rosxmpp` is not meant to be a replacement for the standard `roscore` but rather exposes it transparently in a rosxmpp network.

2 Features

Most of the functionalities presented above were implemented. However, due to time constraint, the *loop* has not been closed yet. This deliverable offers the ability to : connect/disconnect to a XMPP server, query the list of available ROS masters in the overlay network, query the list of remote topics available at a given ROS master, *expose* the local topics to the other participants and *proxy* the remote topics to the local subscribers. This proxying functionality is the one that is not completed yet.

These functionalities will be explained and demoed below.

3 Installation

The `rosxmpp` project has been developed using a `ejabberd` server and deployed over two different hosts.

3.1 XMMP Server

`Ejabberd` is available from the standard Ubuntu repositories :

```
sudo apt-get install ejabberd
```

The configuration file must be edited :

```
sudo vi /etc/ejabberd/ejabberd.cfg
```

The following sections were modified as follow :

```
%% Admin user
{acl, admin, {user, "pierre-luc", "merlin"}}.

%% Hostname
{hosts, ["localhost", "merlin"]}.
```

where `pierre-luc` is the administration name for ejabberd and `merlin` is the hostname of the machine on which the server is installed. Additional users can be registered on the ejabberd as follow :

```
$ sudo /etc/init.d/ejabberd start
$ sudo ejabberdctl register rodney merlin test
$ sudo ejabberdctl register pierre-luc merlin test2
```

3.2 ROS

For Ubuntu 10.04 :

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu lucid main" >
 /etc/apt/sources.list.d/ros-latest.list'
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install ros-electric-desktop-full
$ echo "source /opt/ros/electric/setup.bash" >> ~/.bashrc
. ~/.bashrc
```

4 rosxmpp

Since only one `roscore` can be executed per system, the following instructions apply to a setup with two differents machines that can reach the `ejabberd` server previously installed.

On each system, it is necessary to create the following directory :

```
$ sudo mkdir /var/run/rosxmpp
$ sudo chown pierre-luc.users /var/run/rosxmpp
```

This directory is used to keep track of the `rosxmppbridge` instances running on the system. It contains files holding the URI to the `rosxmppbridge` XML-RPC interface serving a given identity (`pierre-luc@merlin`) in this case. The rationale for this would be support multiple *bridges* to different remote ROS masters.

4.1 Starting ROS

The following command must be executed on both machines to start ROS :

```
$ roscore
```

4.2 Connecting

```
$ roscore
$ ./rosxmppbridge merlin pierre-luc test2 rosxmpp
```

The `rosxmppbridge` process will stay in the background and the `rosxmpp` is merely a front-end to it. In this example, `merlin` is the XMPP server hostname, `pierre-luc` is the user name, `test2` the password and finally `rosxmpp` is the Jabber resource : in future versions, this parameter should be hidden to the user.

The XML-RPC protocol is used for communicating between these two. The Apache `ws-xmlrpc`[1] library was used for both client and server functionalities of the XML-RPC protocol. Since ROS uses XML-RPC for signalling, the `rosxmpp` also implements some of its protocol.

[1] http://ws.apache.org/xmlrpc/

We can verify that the rosxmpp instance is indeed connected to the XMPP server by typing the following in a separate terminal :

```
$ ./rosxmpp status pierre-luc@merlin
pierre-luc@merlin : connected.
```

4.3 Querying the list of available ros masters

On host B:

```
$ rosxmppbridge merlin pierre-luc test2 rosxmpp
```

On host A:

```
$ rosxmpp node list rodney@merlin
pierre-luc@merlin
```

The last line printed in the terminal above shows the only node available in the network : `pierre-luc@merlin`.

4.4 Querying the list of remote topics

On host B :

```
$ rosxmppbridge merlin pierre-luc test2 rosxmpp
$ rosxmpp expose http://localhost:11311 pierre-luc@merlin
```

On host A :

```
$ rosxmppbridge merlin rodney test rosxmpp
$ rosxmpp topic list pierre-luc@merlin rodney@merlin
Status 1 "current topics"
Topic /rosout_agg type rosgraph_msgs/Log
```

The last line printed on terminal shows the remote topics available on the remote ROS master B : the default `/rosout_agg` topic of type `rosgraph_msgs/Log`.

4.5 Proxying a remote topic

On host B :

```
$ roscore
$ rosxmppbridge merlin pierre-luc test2 rosxmpp
$ python rostest.py
$ rosxmpp expose http://localhost:11311 pierre-luc@merlin
$ rostopic list
/chatter
/chatter2
/rosout
/rosout_agg
```

In the `expose` command, the URI of the local ROS master is specified : `http://localhost:11311`. The `rostopic list` command shows the list of topics available locally. The `/chatter` and `/chatter2` topics are published from the test python program `rostest.py`.
On host A:

```
$ roscore
$ rosxmppbridge merlin rodney test rosxmpp
$ rosxmpp topic proxy pierre-luc@merlin rodney@merlin
Remote topics for pierre-luc@merlin are now available locally.
$ rostopic list
/rosout
```

```
/rosout_agg
/rosout_agg_xmppremote
/chatter_xmppremote
/chatter2_xmppremote
$ python rosremotesub.py
```

`rostopic list` above shows that the remote topics at B were registered to the local ROS master at A. The `_xmppremote` suffix is used to distinguish local from remote topics but this convention is subject to change in future versions.

Finally the `rosremotesub.py` program is used to initiate a subscription request to the `/chatter_xmppremote` topic. The steps involved in this operation will be outlined below.

5 Implementation details

The following files can be found under `src/` :

```
src
└── org
    ├── apache
    │   └── mina
    │       └── util
    │           └── AvailablePortFinder.java
    └── rosxmpp
        ├── cli
        │   ├── CommandHandler.java
        │   └── RosXMPP.java
        ├── connection
        │   ├── client
        │   └── server
        │       ├── Master.java
        │       ├── RosXMPPBridgeConnection.java
        │       ├── RosXMPPBridge.java
        │       ├── SlaveApiServer.java
        │       ├── TcpRosServerException.java
        │       ├── RosXMPPBridgeConnectionImpl.java
        │       ├── RosXMPPBridgeConnectionManager.java
        │       ├── RosXMPPJabberServer.java
        │       ├── Slave.java
        │       ├── TcpRosServer.java
        │       └── XmlRpcServerProxy.java
        └── transport
            ├── JingleCommunication.java
            ├── UDTMediaManager.java
            └── UDTMediaSession.java
```

`RosXMPPBridge.java` implements all the core functionality of the system and `RosXMPP.java` is merely a CLI frontend to it. All the features offered by `RosXMPPBridge.java` are exposed to the frontend via XML-RPC.

When `RosXMPPBridge.java` connects to the XMPP server, it writes the URI of its XML-RPC interface under `/var/run/rosxmpp`. When `rosxmpp` is called from the command line, this file is then retreived to get the URI at which the bridge process can be accessed.

5.1 Implementation of the proxy command

Although the functions from the *slave* and *master* API are listed on the ROS wiki, a considerable amount of reverse engineering on the ROS protocol had to be done in order to achieve this goal. Wireshark turned

out to be an indispensable tool to this account. The main steps involved will be outlined here.
From the `rosxmpp` program, `proxyRemoteTopics` is called via XML-RPC on the `rosxmppbridge` instance. This request is then received in `RosXMPPBridgeConnectionManager` (a singleton) in the `proxyRemoteTopics` method.
The list of remote topics is first obtained by calling `master.getPublishedTopics` over Jabber-RPC. This call is then handled by the remote ROS master in `RosXMPPJabberServe.java` and triggers another call to the `getPublishedTopics` to the ROS master on the remote peer. This last IPC call is executed over XML-RPC from the remote `rosxmppbridge` instance to the remote ROS master.
Once the list of remote topics is obtained, they must be registered to the local ROS master. This is achieved through the `registerPublisher` XML-RPC function on the ROS master. Note that at this point, ROS expects from the publishers to implement the *slave api*. Therefore, another XML-RPC server has to be instantiated at this point and the aforementioned *slave* server must at least provide an implementation for the `requestTopic` function.
This function is called from the subscribers after they obtained the URI of the slave API provided by the publishers they want to subscribe to. Note that the slave URI is obtained from the ROS master via the `registerSubscriber` method.
The purpose of the `requestTopic` function (implemented in `SlaveAPIHandler.java`) is to provide to the subscribers the internet address of the TCPROS channel offered by the publisher. Note that event notifications in ROS do not take place at the XML-RPC level but rather via a custom TCP protocol : TCPROS. Thus, not only the `rosxmppbridge` provides the slave API, but it must also listen for incoming TCP connection requests from subscribers.
When a TCPROS connection is accepted in `TcpRosServer.java`, a Jingle ICE-UDP session is initiated to the remote ROS master providing the topic for which the `rosxmppbridge` instance is asked to bridge for. In order for the remote `rosxmppbridge` to determine to which topic the initiated Jingle session should be used for, another `requestTopic` method was defined this time over Jabber-RPC. The purpose of this call is to *bind* a jingle session id with a topic name.
A few more steps are needed before being able to write the topic content into the jingle channel. Indeed, at the remote `rosxmppbridge`, we still have to call `registerSubscriber` on the remote ROS master via XML-RPC. This call then returns the URI to the slave API of the publisher offering this topic. Once again, we have to call `requestTopic` on the slave XML-RPC server to obtain the address of the TCPROS channel to the publisher. Once the socket has been opened to the publisher, we are finally able to read the topic updates.
Note that some steps are missing to get the functionality to work completely. We still have to read the data from the socket and encapsulate it in a udp-based protocol that offers reliability (since Jingle ICE-UDP is only defined for UDP and TCPROS assumes relability). However, the Jingle ICE-UDP transport was implemented and tested successfully with dummy data sent over UDP. A project such as Java-UDT was expected to be used to provide this encapsulation. On the initiator side, it remains to read the data from the Jingle session, and push it to the subscriber over TCPROS.

5.2 Jabber-RPC

The `smack`[2] library was used to implement most of the XMPP functionalities and extensions. However, it does not offer the Jabber-RPC [3] extension. Therefore, we had to implement it ourselves within smack. We based our implementation on *xmppxmlrpcapi*, an orphaned project from the ActiveQuant project (trading system), from Ulrich Staudinger and fixed numerous bugs.

[2] http://www.igniterealtime.org/projects/smack/
[3] http://xmpp.org/extensions/xep-0009.html