CS8903 Special Problem : Mesh Networks for robotic teleoperation -State of the Art and Implementation for Robotics

Sylvain Chatel, Cedric Pradalier

CS8903 Special Problem : Mesh Networks for robotic teleoperation -State of the Art and Implementation for Robotics-

Sylvain Chatel under the supervision of Pr. Cedric Pradalier
Georgia Institute of Technology
Email : sylvain.chatel@gatech.edu

***Résumé*—In this small paper, we explore the different kind of Mesh network protocols as of today. The objective is to give to the reader an overview of the different existing techniques and to decide which one would be the most efficient for a robotic application. Indeed the final goal is to realize a Mesh network that would enable several robots to communicate. This paper is a state of the art of Mesh application for robotics as well as a experiment review of the use of mesh network for robotics.**

I. Introduction

Wireless Mesh Networks (WMNs) are an evolution of wireless technologies. They are made of mesh clients and mesh routers with the particularity that each node is both client and router. The idea is that even if a node is not in range, one can communicate with another node not in range by using other nodes as relays. According to Akyildiz in [1], Mesh networks offer many advantages among which reliability of the service, low cost, easy maintenance and robustness. He also states that Mesh networks can be build on various kind of computer systems including laptop/desktop PCs. Since its development, WMNs have evolved and several communication protocols have emerged. for instance OLSR, B.A.T.M.A.N. or Babel. Murray et al in [2] and Abolhasan et al in [3] both present the existing mesh protocols and their performances. Those results will be summarised in the following sections.

In [4], Li et al studied the constraint and specification a robot swarm communication network would need. And in [5], Hart et al explored and implemented mesh network for robots to be able to communicate among themselves. They also implemented the three main mesh protocols previously stated in order to point out which is the most appropriate for robots communication.

The paper is structured as follows. In Section II, we present the basics of Mesh Networks and the requirements for robotic applications. Then, in Section III, we present the different protocols and their characteristics and finally in Section IV, we will present a few implementation systems as found on the internet as of today. In section V we will present our own implementation on a turtlebot.

II. Mesh Networks

A. Quick overview

In this section, we present the main aspects of WMNs. Those results are mainly based on Akyildiz in [1].

Mesh Networks are a kind of Ad Hoc Networks but offering multi hop. In other words, each node acts like a client and a router and is able to retransmit an incoming message so that nodes can communicate without a direct line of sight. Application for WMNs are numerous from creating a LAN to a swarm robots ([1], [4]).

We now can focus on the technical aspects of a WMN. We remind the reader that the OSI model is made of seven layer : Application, Presentation, Session, Transport, Network, Link and Physical. According to [1], the physical layer doesn't really differ from other classical networks. However, the following show differences.

First, the Link layer. According to Akyildiz, the MAC protocol for WMNs have some difference with other wireless networks. Indeed, MAC for WMNs have to take into account that multi hop communication is used, that this distributed problem needs to be dealt with in a cooperative multipoint-to-multipoint communication, that mobility affects the performance of MAC and finally that Network self-organisation is a requirement. In order to meet those requirements, Akyildiz presents several solutions : improving existing MAC protocol or innovate MAC protocols. All the detail of that can be found in [1].

Second, even though WMNs will be based on IP, they have a very different routing protocol than wired networks. Because WMNs and Ad Hoc networks share similar features, protocols from the latest can be applied to WMNs. An optimal routing protocol will deal with the following : Performance metrics, Fault tolerance with link failures, Load balancing, Scalability and Adaptive Support of Both Mesh Routers and Clients. The main three routing protocols are presented in Section III.

Finally, the Transport Layer. According to Akyildiz, no transport protocol has been implemented especially for WMNs. At the time of [1], WMNs used variants of TCP or new ones such as Ad Hoc Transport Protocol (ATP) (but although presenting advantages, those kind of protocols lack compatibility with other systems).

This section was a mere summary of the WMNs technical aspects from [1].

B. Requirements

In [6], Yang et al present the requirements for routing metrics in order to satisfy good performances : route stability, good performance for minimum weight paths, efficient algorithm to calculate minimum weight path and loop-free routing.

In [5], Hart et al want to establish a mesh network connection between robots in order to proceed to robotic teleoperation. This suppose a high quality video streaming i.e. without pixelated video (appearance of artefacts), smearing video (appearance of a smeared ghosting image), choppy video. Even more this suppose no intermittent and delayed control of the robot.

III. Routing Protocols

In this section, we will present the three main kind of routing protocols for multi hop ad hoc networks. The reader must be aware that WMNs are a kind of Mobile Ad Hoc Networks (MANETs) and such networks have different kind of routing protocols categories : *reactive*, *proactive* and *hybrid*. The first one means that the routing table is set only when a message needs to be sent. The second one means that the protocol actively establishes routing tables in order to reduce the latency in sending a message. Finally the third one tends to combine the first two. All three of the following protocols are proactive routing protocols.

A. Optimized Link State Routing (OLSR)

OLSR was a first attempt to use proactive link-state algorithm to determine the most efficient path between nodes. It is currently one of the most used routing protocols for ad hoc networks.

OLSR uses the concept of Multi-Point Relays (MPRs) which are designated nodes that actually retransmit the data. These MPRs are dynamically selected and are the backbone of the OLSR protocol. The selection process is as follows : first a node *A* broadcasts a first message in order to detect its one hop neighbours with whom it shares a bidirectional path. Then *A* broadcasts a *Hello* message containing its heard neighbours, its connected neighbours (bidirectional path established) and the designated MPRs node by node *A*. When enough MPRs are selected, MPRs send Topology Control messages (TC) which allow to build a routing table of the whole topology for each node. This process is dynamically modified in order to take into account the changes in the topology. Using this method, MPRs have a complete routing table while minimising the number of TC messages.

On Figure 1, we can see an example from [2] of the MPRs repartition in a network.

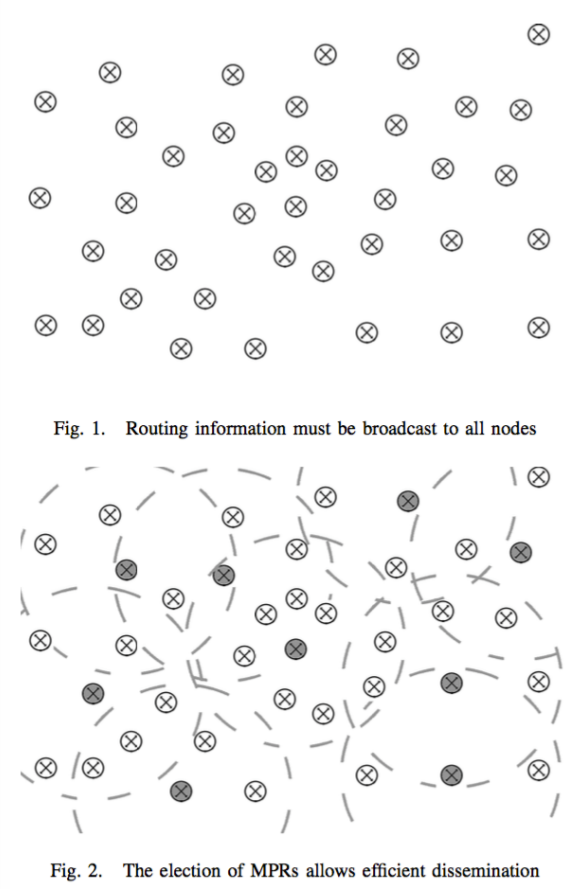

Fig. 1. Routing information must be broadcast to all nodes

Fig. 2. The election of MPRs allows efficient dissemination

FIGURE 1. Presentation of the MPRs in a network topology as seen in [2]

B. Better Approach To Mobile Ad hoc Networking (B.A.T.M.A.N.)

BATMAN is also a proactive protocol but has a very different approach on the routing problem. Indeed, its modus operandi is to discover which neighbour offers the best path to other nodes. This decentralized method enables each node to send the desired message not to a particular node but in a certain direction. In this protocol, each node broadcasts a message called *Originator Messages* (OMs) periodically. When received by another node, the latest re-broadcasts the OMs. Then, a particular route to a destination is determined based on nodes who received the more OMs from the destination. In other words, the route is determined based on the most reliable link. In order to work, this protocol lies on the packet's ability to get lost. In order to proceed that way, UDP datagram are used for the OMs : if a packet gets lost, it means that the link is not fast and reliable. This method is acccording to [3] less complex and has less hardware requirements than OLSR for instance.

C. BABEL

BABEL is another proactive protocol which lies on advanced distance vector routing algorithm. This means that routers don't have full knowledge of the network's topology. Instead, they use the direction in which the destination is and

the distance to it to calculate the best route. The technique of distance counting used is an evolution of the called *Expected Transmission Count* (ETX) algorithm. BABEL operates both with IPv4 and IPv6 networks.

D. Comparison between the different protocols

Both [2] and [3], researchers carried out a complementary study on the performance of each protocols. In this section, we aim to present those results. In both cases, a small network was implemented on hardware running on Linux. They then used the three protocols in different cases and analysed the results. To our point of view, [3] presents a more complete analysis than [2] which validates the previous study. They analysed the bandwidth, the packet delivery ratio and route convergence latency. In [3], they concluded that BATMAN offers the highest level of stability and packet delivery whereas BABEL offers the highest multi hop bandwidth and the fastest route repair time. However, in [2], they attenuate those results presenting more mixed performance results showing that OLSR is not completely outperformed by the other two.

In [5], Abraham Hart et al explored the use of the different protocols for robotic teleoperation use. They decided to focus their work on OLSR and BABEL since BATMAN wasn't developed enough at the time. They succeeded in implementing a robust mesh communication network which allows video streaming and roaming by modifying the *babeld* package. According to their report, they had to add an ETX threshold (in order to avoid route-flapping) and increase the *"hello"* packet rate.

IV. Implementation systems

In this section we aim to present the different existing implementation solution for WMNs as of today.

- *Open-Mesh* [7] : Based on the BATMAN protocol. A list of compatible hardware is available online.
- *Commotion* : Based on OLSR. Fully developed. A list of supported hardware can be found online.
- *Byzantium* : A fully working OLSR implementation.
- *BABELd* : Implementation of the BABEL routing protocol. A git depository is available. No hardware requirements are specified.
- *OLSRd* : Another OLSR implementation fully developed as well.

V. Implementation experiment

In this section, we present how we used a mesh network to proceed to robotics teleoperations. Our objective is, in a first approach, to be able to remotely control a small robot called a turtlebot. In a second approach, we will create fail-safe mode and network analysis based communications.

A. The mesh protocol

The first step is to determine the protocol we will use to create the mesh network. Thanks to the previous state of the art and the analysis of our needs, our choice was set on *Open Link State Routing* and its implementation called olsrd. OLSRd was developed as an open source module that works on any Wifi card that support Ad-Hoc mode and Ethernet devices. In our implementation, we use the latest version of olsrd : v0.9.0.3. Our turtlebot runs on a linux 3.7.8 porteus i86 platform.

First, we logged as root and installed olsrd. We downloaded it in a source folder.

```
> cd /home/turtlebot/source
> tar jxvf olsrd-0.9.0.3.tar.bz2
> cd olsrd-0.9.0.3/
```

Console 1. Get OLSRd

Next, we installed the necessary packages to compile :

```
> sudo apt-get install bison
> sudo apt-get install flux
> sudo apt-get install checkinstall
```

Console 2. Get useful modules

In order to compile, we executed the following commands :

```
> make
> sudo checkinstall
> cd /home/turtlebot/source/olsrd-0.9.0.3 ↙
↳ /lib/httpinfo/
> make
> sudo make install
> cd /home/turtlebot/source/olsrd-0.9.0.3 ↙
↳ /lib/jsinfo/
> make
> sudo make install
```

Console 3. Install OLSRd

Now we have an installed olsrd module on the machines. In order to create a mesh network, we implement this module on several laptops. In our experiment we used five identical laptops. Then the mesh network can be set by configuring in a right way each node. First, we have to determine the interface the wireless module uses.

```
> ifconfig
> iwconfig
```

Console 4. Get the initial netork status

The first output gives us the gives us all the information of the networking configuration on the node. The second informs us about the wireless interface. Let us assume that the wireless interface in use is *wlan1*. We still have to disable the network manager and configure the parameters.

```
> sudo stop network-manager
> sudo iwconfig wlan1 mode Ad-Hoc
> sudo iwconfig wlan1 essid "NetworkName"
```

```
> sudo ifconfig wlan1 <IPaddress> netmask ↙
↳ 255.255.255.0 up
```

Console 5. Configure the network parameters

Note : Please note that at reboot all those parameters will be reset to default. To avoid that we could suggest one to modify the /etc/network/interfaces file.

In our experiment, we used addresses IPv4 from the subnet 10.17.6.0/24. In other words, our IP addresses were for instance 10.17.6.102 or 10.17.6.110 depending on the serial number of the turtlebot (here 2 and 10). The *"NetworkName"* is just the name of the network, we used *"mesh22"*.
Once this configuration is done, we can create/join the network by using :

```
> sudo olsrd -i wlan2
```

Console 6. Launch OLSRd on interface wlan2

We then get the output on Figure 2.

```
turtlebot@turtlebot01: ~
IP address       hyst         LQ       ETX
10.17.6.108      0.000  1.000/1.000    1.000
10.17.6.105      0.000  1.000/1.000    1.000
10.17.6.107      0.000  1.000/1.000    1.000
10.17.6.110      0.000  1.000/1.000    1.000

--- 12:06:51.243980 ------------------------------------------------ NEIGHBORS

     IP address Hyst    LQ      ETX     SYM   MPR   MPRS  will
192.168.32.5    0.000   1.000/1.000     1.000   YES   NO    YES   3
10.17.6.107     0.000   1.000/1.000     1.000   YES   NO    NO    3
10.17.6.108     0.000   1.000/1.000     1.000   YES   NO    YES   3
10.17.6.110     0.000   1.000/1.000     1.000   YES   NO    YES   3

--- 12:06:51.244018 ---------------------- TWO-HOP NEIGHBORS

IP addr (2-hop)  IP addr (1-hop)  Total cost
192.168.32.5     10.17.6.108      2.000
                 10.17.6.107      2.000
10.17.6.107      10.17.6.110      2.000
                 192.168.32.5     2.000
                 10.17.6.108      2.000
10.17.6.108      10.17.6.107      2.000
                 192.168.32.5     2.000
10.17.6.110      10.17.6.107      2.000
```

FIGURE 2. olsrd output

Now we have a properly working mesh network running on a OLSR routing protocol. On the output, we can see all nodes in the network, the next hop, the second hops and the metrics. In order to get a more comprehensive view of the network, we use the following :

```
> watch 'ifconfig wlan2, iwconfig wlan2'
> ping 10.17.6.110
> route -n
> traceroute 10.17.6.110
```

Console 7. control the network status

Note : Please note that you might need to install traceroute since it is not a default module.

B. The first experiment

Once this was done, we conducted a series of test in order to get a idea of the network performances. We designed a experiment involving five nodes we called the *pyramidal experiment*. We set the nodes as showed on Figure 4.

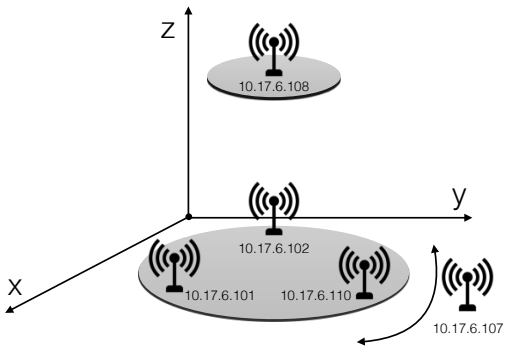

FIGURE 3. Presentation of the pyramidal experiment

We set four static nodes in a pyramidal way (10.17.6.101/102/108/110) and we keep one mobile node 10.17.6.107. Using the upper node 10.17.6.108, we create a video streaming from this node to the mobile node. The distance between consecutive static nodes was about 10 to 25 meters but in a non-line of sight environment and with other Wifi interferences.
The objective of the experiment is to try the performances of the network for high QoS applications. In order to generate the streaming, we used a video software called VLC.

Then moving around with the mobile node, we were able to witness the dynamic reconfiguration induced by the routing protocol and for low speed, the QoS of the video streaming is unimpacted by the network reconfiguration.

In the protocol, the *hello messages* were set to be sent every two seconds. In order to get better performances with higher speed, we could reduce the validity time and the frequency of those *hello messages* to have a more reactive network.

C. The use of ROS

After those successful tests, we wanted to try the teleoperation of the robots using the mesh network. In order to do so, we used ROS, an operating system design for robotic applications. The platform we used is a turtlebot controlled by the laptops previously mentioned. Our version of the turtlebot was *indigo*.
Using ROS, it was straightforward to create a teleoperation using the keyboard of the control station or a joystick. Tutorials to achieve this can be found on wiki.ros.org/turtlebot/Tutorials/indigo/.

Using the mesh network previously created and ROS, we were able to remotely control the turtlebot in an Ad-Hoc mode.

The protocol to create a teleoperation is as follows :

- On the turtlebot 10.17.6.108 :

FIGURE 4. A turtlebot

```
> roscore
> echo export ROS_MASTER_URI = ↙
↳ http://localhost:11311>> ~/.bashrc
> echo export ROS_HOSTNAME = ↙
↳ 10.17.6.108 >> ~/.bashrc
# Turn on the robot
> gnome-terminal &
1> roslaunch turtlebot_bringup ↙
↳ minimal.launch
2> roslaunch turtlebot_dashboard ↙
↳ turtlebot_dashboard.launch
3> rqt_remocon
4> roslaunch turtlebot_rviz_launcher ↙
↳ view_robot.launch
5> roslaunch turtlebot_bringup ↙
↳ 3dsensor.launch
```

Console 8. teleoperation setup form the robot 10.17.6.108

- On the Control Station 10.17.6.107 :

```
> echo export ROS_MASTER_URI = ↙
↳ http://10.17.6.108:11311>> ~/.bashrc
> echo export ROS_HOSTNAME = ↙
↳ 10.17.6.107 >> ~/.bashrc
> roslaunch turtlebot_teleop ↙
↳ logitech.launch
```

Console 9. teleoperation setup form the control station 10.17.6.107

Those command lines enable the user form the control station (10.17.6.107) to control the robot (10.17.6.108) via a LOGITECH joystick. The rviz and 3dsensor launches enable the video streaming form the robot to the station.

To get the network information, we can use the http interface available by OLSRd module jsoninfo . Several information get be fetch : neighbors, links, routes, hna, mid, topology, gateways, interfaces, status. In our experiment, we decided to only use *links* and *route* since they offer all the necessary information we want.

To access this information we use the following command line :

```
> curl http://localhost:9090/routes/links
```

Console 10. fetch the network json information

D. Towards better control

Now that the system was working properly, the objective was to create ROS packages to use the network data and take the adequate decision. Indeed, the mesh network might sometimes be weak and do not offer a sufficient connection for teleoperation. In that case, we want the turtlebot to switch to a safe mode and wait for the connection to be re-established with a good QoS o prevent any accident.

In order to achieve this goal, we built three ROS packages called nodes.

The first one is called OLSR_MONITOR. Its goal is to fetch the OLSR json information from the http interface. This node was developed in python and publishes a ROS message containing the source IP (the mobile robot), the destination IP (the control station), the gateway (next hop neighbour to the station), the link quality to the destination, the link quality of the destination, the expected transmission count (etx) and the time.

The second node is called MESH_SAFETY. Its goal is to decide based on the ROS message sent by OLSR_MONITOR whether or not to forward the /joy_input sent by the control station to the teleoperation node in the mobile robot. Based on several threshold we determined experimentally, the node forwards the command if it judges that the network offers sufficient QoS to ensure a proper teleoperation. This structure is illustrated on Figure 5.

To create those nodes and use ROS, we followed the lines stated below :

```
# ROS
> source /opt/ros/indigo/setup.bash
> source /home/turtlebot

# Create a catkin workspace
> mkdir -p ~/catkin_ws/src
> cd ~/catkin_ws/src
> catkin_init_workspace
> cd ~/catkin_ws/
> catkin_make
> source devel/steup.bash
> echp $ROS_PACKAGE_PATH

# Creation of MESH_SAFETY
> cd catkin_ws/src
> catkin_create_pkg mesh_safety
> cd ~/catkin_ws/src/mesh_safety
> mkdir src
> vim mesh_safety_node.cpp

# Creation of OLSR_MONITOR and message
> cd catkin_ws/src
```

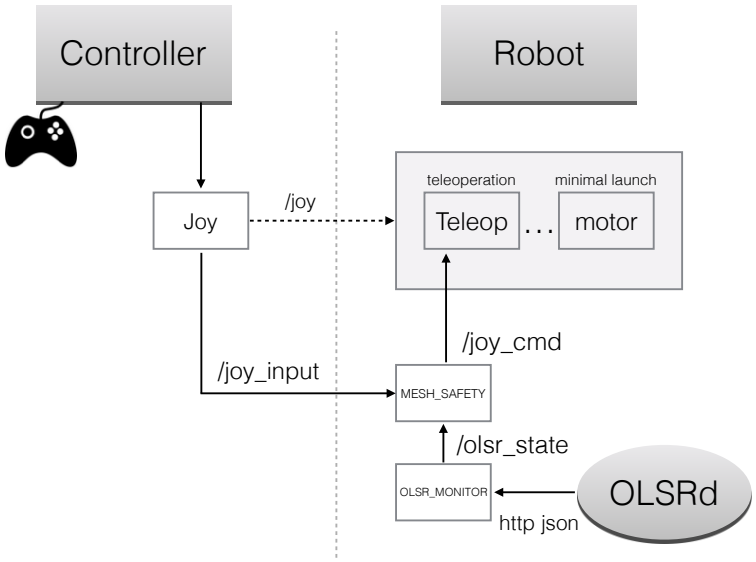

FIGURE 5. ROS hierarchical architecture

```
> catkin_create_pkg olsr_monitor
> cd ~/catkin_ws/src/olsr_monitor
> mkdir script
> vim olsrd_monitor_node.py
> cd ~/catkin_ws/src/olsr_monitor
> mkdir msg
> vim OLSR_state_msg.msg

# compile
> cd ~/catkin_ws
> catkin make
```

Console 11. ROS setup

Now that we have created empty nodes, we still have to build them accordingly with our goals.

The following code is the script code of the OLSR_MONITOR python node. As previously stated, its objective is to fetch the network information and send a olsr_state_msg.

```
#!/usr/bin/env python
# license removed for brevity
import rospy
from olsr_monitor.msg import OLSR_state_msg
import urllib, json

def talker():
  pub = rospy.Publisher('olsr_state', ↙
↳ OLSR_state_msg, queue_size=10)
  rospy.init_node('talker', anonymous=True)
  rate = rospy.Rate(2) # 10hz

  url_links = "http://localhost:9090/links"
  url_routes = "http://localhost:9090/routes"

  while not rospy.is_shutdown():
    rsp_links = urllib.urlopen(url_links)
    rsp_routes = urllib.urlopen(url_routes)

    data_links = json.loads(rsp_links.read())
    data_routes = json.loads(rsp_routes.read())

    links = data_links['links']
    routes = data_routes['routes']

    dst_ip          = '10.17.6.107'
    src_ip          = '10.17.6.101'
    msg_link_q      = links[0]['linkQuality']
    msg_nbr_link_q = ↙
↳ links[0]['neighborLinkQuality']
    msg_route_gtw   = routes[0]['gateway']
    msg_route_etx   = routes[0]['rtpMetricCost']

    for i in range(len(links)):
      if links[i]['remoteIP'].encode('latin-1') ↙
↳ == dst_ip:
        msg_link_q = links[i]['linkQuality']
        msg_nbr_link_q = ↙
↳ links[i]['neighborLinkQuality']
        break

    for i in range(len(links)):
      if ↙
↳ routes[i]['destination'].encode('latin-1') == ↙
↳ dst_ip:
        msg_route_gtw = ↙
↳ routes[i]['gateway'].encode('latin-1')
        msg_route_etx = routes[i]['rtpMetricCost']
        break

    msg_time = data_links['systemTime']

    msg = OLSR_state_msg()
    msg.header.frame_id = ""
    msg.header.stamp = rospy.Time.now()
    msg.num = msg_time
    msg.src_ip = src_ip
    msg.dst_ip = dst_ip
    msg.gateway = msg_route_gtw
    msg.link_q = msg_link_q
    msg.nbr_link_q = msg_nbr_link_q
    msg.etx = msg_route_etx


   # hello_str = "hello world %s" % ↙
↳ rospy.get_time()
    rospy.loginfo(msg)
    pub.publish(msg)
```

```
    rate.sleep()

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

Console 12. olsr_monitor_node.py

Let us now explain this code. First, we import all the necessary packages such as urllib and json. Then, we define the "talker" which is responsible for sending the message. Using the urllib library we fetch the json data from olsrd. We then extract the information we need (regarding the mobile node to the control station). Finally, we assign the variables in the ros message with the corresponding value and we publish.

Now, we have to build the decision package : the node that will take the decision to forward the /joy message (*i.e. forward the teleoperation command from the control station to the mobile turtlebot*) whether or not depending on the state of the network. We already said that the network state was determined by several metrics and parameters such as the next hop to the destination (referred as gateway), the estimated transmission count and the link quality (between 0 and 1, 1 being ideal). If the metrics reach some thresholds, the node puts a hold on the forwarding and the robot is put into a *safe mode* until the metrics get better. The code is as follows :

```
#include "ros/ros.h"
#include "olsr_monitor/OLSR_state_msg.h"
#include "sensor_msgs/Joy.h"
#include "geometry_msgs/Twist.h"

#include <sstream>

class OLSR {
 public:
  OLSR() :
      check(true), n("~")
  {
      // code constructeur
    joy_pub = ↙
↳ n.advertise<geometry_msgs::Twist>("/joy_cmd", ↙
↳ 1000);
    joy_sub = n.subscribe("/joy_input", 1000, ↙
↳ &OLSR::joy_sub_callback, this);
    olsr_state_sub = n.subscribe("/olsr_state", ↙
↳ 1000, &OLSR::olsr_state_sub_callback, this);
  }

  void olsr_state_sub_callback(const ↙
↳ olsr_monitor::OLSR_state_msgConstPtr& ↙
↳ olsr_msg) {
    // here goes the code to decide if the /joy ↙
↳ message should be forwarded
    double threshold = 0.100;

    int num = olsr_msg->num;
    std::string src_ip = olsr_msg->src_ip;
    std::string dst_ip = olsr_msg->dst_ip;
    std::string gateway = olsr_msg->gateway;
    float link_q = olsr_msg->link_q;
    float nbr_link_q = olsr_msg->nbr_link_q;
    std::string etx = olsr_msg->etx;

    if ((link_q < threshold) || (etx == ↙
↳ "INFINITE")){
      check = false;
    } else {
      check =  true;
    }
  }

  void joy_sub_callback(const sensor_msgs::Joy& ↙
↳ joy_sub_msg) {
    // here we receive /joy_input and if bool == ↙
↳ True
    joy_msg = joy_sub_msg;
    last_joy = ros::Time::now();
    if (check == true){
      run();
    }
  }

  void run() {
    ros::Rate rate(10);
    while (ros::ok()) {
      ros::spinOnce();
      if ((ros::Time::now() - last_joy).toSec() ↙
↳ > 1.0) {
      }

      joy_pub.publish(joy_msg);
      rate.sleep();
    }
  }


 protected:
  bool state;
  bool check;
  ros::NodeHandle n;
  ros::Publisher  joy_pub;
  ros::Subscriber joy_sub;
  ros::Subscriber olsr_state_sub;
  ros::Time last_joy;
  sensor_msgs::Joy joy_msg;
};

int main(int argc, char **argv) {
  // init ROS
  ros::init(argc, argv, "mesh_safety_node");

  OLSR olsr;

  //
  // enter loop
  ros::spin();

  return 0;
}
```

Console 13. mesh_safety_node.cpp

Again, allow us to explain the previous code. First, we subscribe to the /joy_input, published by the control station, and to the olsr state message sent by the OLSR_MONITOR node. Then, we check if the values in the state message reach the thresholds. If not then the node forwards the /joy_input as /joy_cmd.

E. Results

Using the protool stated below, we managed to use the mesh network and proceed to the teleoperation of a turtlebot up to three hops away from the ontrol station. However, even though commands were still reaching the robot, the latency and the QoS of the network wasn't sufficient enough

to get a proper real time teleoperation. Our analysis would suggest that by modifying the update time and hello message frequency in OLSRd we could ensure a better teleoperation. To conclude, the mesh networks offers a decent solution to teleoperation for the turtlebot but would benefit from some modifications in order to make it better than a managed network with well positioned Wifi routers.

VI. Conclusion

In this paper, we presented some distinctive features of Wireless Mesh Networks. We presented as well the three main routing protocols in order to make a choice before implementing a Mesh Network for robotic teleoperations. Through our research, we came by several hardware requirements that we must check with the Turtle bots before going any further.

In a second approach, we decided to experiment the use of mesh networks for robotics teleoperations. After choosing the OLSR routing protocol, we used one of its several implementations OLSRd. Then using ROS we managed to operate teleoperations via the mesh network with decent performances.

Références

[1] I. F. Akyildiz, X. Wang, and W. Wang, "Wireless mesh networks : A survey," *Comput. Netw. ISDN Syst.*, vol. 47, pp. 445–487, Mar. 2005.

[2] D. Murray, M. Dixon, and T. Koziniec, "An experimental comparison of routing protocols in multi hop ad hoc networks," in *in Telecommunication Networks and Applications Conference (ATNAC), 2010 Australasian*, pp. 159–164.

[3] *Real-world performance of current proactive multi-hop mesh protocols*, Oct. 2009.

[4] M. Li, H. Zhu, S. Mao, K. Lu, and M. Chen, "Robot swarm communication networks : Architectures, protocols, and applications," 2008.

[5] A. Hart, N. Pezeshkian, and H. Nguyen, "Mesh networking optimized for robotic teleoperation," 2012.

[6] Y. Yang, "Designing routing metrics for mesh networks," in *In WiMesh*, 2005.

[7] "Open-Mesh download." https://www.open-mesh.org/projects/open-mesh/wiki/Download. Accessed : 2016-09-08.