CENTER FOR
MACHINE PERCEPTION

CZECH TECHNICAL
UNIVERSITY IN PRAGUE

RESEARCH REPORT

ISSN 1213-2365

Summer internship report

(Version bleeding-edge)

Oliver Porges

oliver@amset.sk

CTU–CMP–0000–00

September 26, 2011

Available at

Supervisor: Ing. Michal Reinstein Ph.D.

The NIFTi project

Research Reports of CMP, Czech Technical University in Prague, No. 0, 0000

Published by

Center for Machine Perception, Department of Cybernetics
Faculty of Electrical Engineering, Czech Technical University
Technická 2, 166 27 Prague 6, Czech Republic
fax +420 2 2435 7385, phone +420 2 2435 7637, www: http://cmp.felk.cvut.cz

Summer internship report

Oliver Porges

September 26, 2011

Contents

Abstract

This work was done as a part of the Natural human-robot cooperation in dynamic environments (NIFTi) project [1]. Our main goal was to create a data acquision module to ease the future research and debugging. The module was implemented as a Robot Operating System (ROS [2]) package. We will cover the program's structure, usage and limits. We were also debugging other parts of the software and hardware alongside to the main goal. All of the supporting information is covered in section 4.1. Our attempt for autonomous demo is covered in section 5

1 Robot Operating System - a brief description

Robot Operating System (ROS)[2] is a middleware environment created by the Willow Garage [4]. The platform aims all mobile robotic systems, a middleware running on *NIX systems with a very clear modular architecture. Every program running under ROS is a *node* (even ROS itself). Every data connection between nodes is a *topic*. Topis have a particular data-type that which has to be known to every node connected. A *Publisher* is the transmitting node and a *Subscriber* is the receiving node. ROS brings a great standarization of topic data types which is a first step necessary to build a re-usable code. There is no hierarchy among topics which can cause problems in high badwidth systems.

2 Our Solution

The very basic idea behind the package is to create a callback function for every topic (data type) we want to save. There is also a possibility to overload the callback function to take in many different datatypes and create it dynamically. Unfortunately we don't save the data into a standarized format, therefore, the open-source community would not benefit from it anyway.

The package (default name: *ctu_data_logger*) consists of two binaries, *data_logger* and *trigger_node*. *data_logger* subscibes to different topics and listens to the sensor's data. It is connected to the *trigger_node* through a *log_trigger* topic. The trigger node publishes data on this topic when a ciertain event occures. This enables the switches in the *data_logger* which then saves one data sample of every senosor enabled. All the configuration is done using the ROS's build-in parameter server.

2.1 Functions and classes

All of the program's body is stored in the *SharedObjects* class. An instance of this class is created in the *main*() function. The constructor *SharedObjects* :: *SharedObjects*() initializes all the variabiles and creates the connections to other nodes. It also reads in the parameters necesary for the program to function properly. To read more about the configuration, see section 3.
The data samples are handled through the callback functions. These functions are called every time when a piece of data is published on a certain topic.

- imageCallback()
  This function receives an image from the camera node and stores it to a parameter-defined file. A parameter *log_camera* has to exist and has to contain a valid path of a file where images should be stored. It has to contain a $\%2d$ in the file name, to hold the sample numbering.

- imageXCallback()
  These functions receive an image from a siangle camera (X is a number from 0 to 5) node and store it to a parameter-defined file. A parameter *log_cameraX* has to exist and has to contain a valid path of a file where images should be stored. It has to contain a sequence $\%2d$ in the file name, to hold the sample numbering.

- rawCallback()
  This function receives a raw image from the camera node and stores it to a parameter-defined file. A parameter *log_raw* has to exist and has to contain a valid path of a file where images should be stored. It has to contain a sequence $\%2d$ in the file name, to hold the sample numbering.

- panoramaCallback()
  This function receives a raw image from the camera node and stores it to a parameter-defined file. A parameter *log_panorama* has to exist and has to contain a valid path of a file where images should be stored. It has to contain a sequence $\%2d$ in the file name, to hold the sample numbering.

- gpsCallback()
  This function receives and saves the GPS data off the *pos_nav* topic. A parameter *log_gps* has to exist and has to contain a valid path to log file.

- LaserScannerCallback()
  This function receives and saves the SICK rangefinder data off the *scan* topic. A parameter *log_laser* has to exist and has to contain a valid path to log file.

- ImuCallback()
  This function receives and saves the inertial Xsens MTi-G data off the *imu_data* topic. A parameter *log_imu* has to exist and has to contain a valid path to log file.

- EncodersCallback()
  This function receives and saves the odometry data off a *odom* topic. A parameter *log_odometry* has to exist and has to contain a valid path to log file.

- MechanizationCallback()
  This function receives and saves the inertial Xsens MTi-G data off a *mechanization_output* topic. A parameter *log_mechanization* has to exist and has to contain a valid path to a log file.

- PclCallback()
  This function receives and saves the point cloud data off a $/nifti\_point\_cloud$ topic. A parameter *log_pcl* has to exist and has to contain a valid path to a log file. It also has to contain the $\%2d$ string to place the sample index there.

- TriggerCallback()
  This function receives a trigger off the *log_trigger* topic and sets the switches. Once they are set the callback functions will be enabled and will therefore save one sample off the enabled topics.

The switches mentioned earlier are binary values. One run of a particular callback function is allowed when its switch is enabled. Every activity of the node is logged through a separate function $ToLogFile()$. The path of the log file is taken from the parameter server $log\_logfile$. It will be saved in the current working directory if $activity - log$ is not set.

2.2 Trigger node

Another part of the package is the trigger node. This node publishes "1" on the *log_trigger* topic when a given condition is fullfilled. So far it can recognize two parameters off the parameter server. *log_distance* parameter is a floating point number expressed in meters. The node calculates Euclidean distance using the odometry topic and triggers the data acquision every *log_distance* meters.

The *joy_trigger_button* specifies a manual triggering button on the joystic. Since the indexes of the buttons start at 0 the button number three would be indexed as 2. The triggering signal will be sent every time this button is pressed. It behaves independently from the distance trigger, therefore, this trigger will not reset the distance calculation in the previous case.

3 Configuration and usage

User can easily run the package as,

rosrun ctu_data_logger data_logger

To get help setting up the necessary paramterers you can run,

rosrun ctu_data_logger data_logger -h

which displays all of the parameters used with a brief description. To run the complementary triggering node run,

rosrun ctu_data_logger trigger_node

If you are running the package using *rosrun* you also have to set the appropriate paramteters. For example

rosparam set log_camera /home/robot/data/image%2d.png

However, it is recommended to use a launch fila. A complete launch file is included with the package (launch/ctu_data_logger.launch). It can be launched as,

roslaunch ctu_data_logger ctu_data_logger.launch

3.1 Example

It is recommended to use the provided launchfile, however, this example will demonstrate a simple command-line usage. We would like to collect the odometry and laser data every 1.5 meters. We run,

```
roscore &
rosparam set trig\distance 1.5
rosparam set log_odometry /home/robot/odometry.dat
rosparam set log_laser  /home/robot/laser.dat
rosrun ctu_data_logger trigger_node &
rosrun ctu_data_logger data_logger &
roslaunch nifti_teleop_joy nifti_teleop_joy.launch
```

If we would like to trigger the acquisition manually (by button number 4)and get the image from all of the cameras saved we would run,

```
roscore &
rosparam set joy_trigger_button 3
rosparam set log_camera /home/robot/image%3d.png
rosrun ctu_data_logger trigger_node &
rosrun ctu_data_logger data_logger &
roslaunch nifti_teleop_joy nifti_teleop_joy.launch &
roslaunch omnicamera omnicamera.launch
```

3.2 Output format

There is one or more values saved on every line of the output file in a text mode. The structure of the frame repeats with every sample taken. The meaning of the values is explained below. Please note that the data are sent by other nodes, therefore, the units are not known to this node. You should consult the documentation of a particular node for more information on the value's units.

3.2.1 Image format

All of the images are saved in PNG format. There is no other option at the moment.

3.2.2 GPS frame

time stamp
latitude, longitude, altitude

3.2.3 Mechanization frame

time stamp
euler X, Y, Z
quaternion W, X, Y, Z
matrix 3 x 3

3.2.4 Odometry frame

time stamp
position X, Y, Z

Orientation X, Y, Z, W
Linear twist X, Y, Z
Angular twist X, Y, Z

3.2.5 Laser scan frame

time stamp
number of samples
minimum angle, maximum angle, angle increment, time increment, scan time,
minimal range, maximum range
ranges[]
intensities[]

3.2.6 IMU frame

time stamp
orientation X, Y, Z, W
angular velocity X, Y, Z
linear acceleration X, Y, Z

3.2.7 Laser point-cloud frame

time stamp
height, width
size of fields[]
all of the fields follow are stored on one line
{
name
offset
datatype
count
}
BIG or SMALL ENDIAN
point step, row step
invalid points indication
number of samples
samples[]

4 Other supporting information

4.1 System functionality

- All of the robot's motion is controlled through a $USB-CAN$ converter. This communication is handled in the *robot_node* and is not further accessible to other parts of the system. There is and error message in the case of engine failure but the node does not take care of it and needs to be restarted.

- The robot can be controlled by rec-reating joystick messages and not running the *joy* node.

- There are network dependent devices and the robot might become un-operational if the networking settings are changed. Even a reboot might not help. It is recommended to place such a crucial configuration into an rc start-up file so the robot would always boot up to the same and working conditions.

  As of 13.9.2011 the network was configured manually as follows,
  Robot(ctu-robot) - 192.168.2.2
  Laptop(ctu-robot-laptop) - 192.168.2.1
  Bullet-laptop(no DNS) - 192.168.2.50
  Bullet-robot(no DNS) - 192.168.2.51

  The laser rangefinder network interface has to be configured after each boot up of the rover as,

  ```
  sudo ifconfig eth8 192.168.1.118
  ```

  ROS, however, is working with the domain names. Therefore, we had to set the domain names in */etc/hosts* in order to be able to distribute the system's nodes.

- The GPS reception is weak and it can take up to 50 minutes for the Xsens unit to get a FIX. Even when any mobile phone has a fix after several minutes. The FIX time was measured at urban testing ground - yard of the Czech Technical University at Karlovo Namesti. There is also a version mismatch in the robot's *xsense_mtig* node and the current manufacturer's supporting libraries.

- If the driver for the xsense unit fails to initialze it can put the unit into an unusable mode. The settings are stored in the device, read back during the configuration procedure (by the mtig_node, MTig.cpp) and then fed back to the unit. The driver should always set the highest possible sampling frequency without any concern about the unit-stored values (120 Hz). No values shuld be read from the device and configured back - it leads to unit malfunction and will never resurrect without a change in the code.

- There is a possibility of data loss (dropping messages) if too much traffic is generated on the ROS message system. This can lead to a significant error in odometry or any other time dependent processing. This could be solved by using a real-time subsystem for time critical operations such as odometry and inertial positioning system. Some buffer sizes should be reconsidered. There is a 50 frames buffer for camera images which is unsutable for teleoperation. On the other side, odometry buffer size should be increased as much as possible.

4.2 Xsens unit

The Xsens unit is a complicated and badly documented hardware. There are Linux libraries available from the manufacturer, however, they are a part of the Software Developement Kit which needs to be registered before usage. It might take some time to obtain them. These libraries implement the low level communication protocol as well as the higher level functions. The unit has two basic modes - configuration and measurement. The configuration part is crucial to its consequent successful operation. Output modes have to be set properly to obtain correct data if any. Line 129 in MTig.cpp sets the proper output modes which are,

CMT_OUTPUTMODE_ORIENT
CMT_OUTPUTMODE_POSITION
CMT_OUTPUTMODE_CALIB
CMT_OUTPUTM
ODE_VELOCITY
CMT_OUTPUTMODE_TEMP
CMT_OUTPUTMODE_STATUS

Another issue is the operation of the INS node (created at Czech Technical University as well). It requires the mtig_node to output its orientation in Euler angles. This is set on line 134 by,

settings = CMT_OUTPUTSETTINGS_ORIENTMODE_EULER;

5 Autonomous demo

We had the idea of autonomous mapping of the surrounding environment. We have managed to connect our sensors with a Simultanious localization and mapping (SLAM) [6] module. The resulting map is saved using the occupancy grid of SALM output.

Figure 1: Map of a corridor in creation

We transform this map into a distance map every time a new occupancy grid is published.

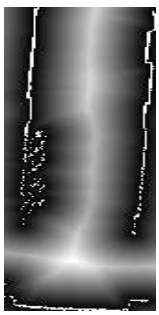

Figure 2: Distance map combined with the walls

Then we search for a nearest point of interest in the distance map (the heighest intensity pixel in a given radius). Compared with the trajectory

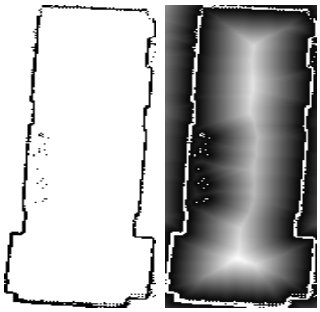

Figure 3: A resulting map and it's corresponding points of interest

already taken we get our next waypoint. This way would be able to autonomously create a map of the environment. With a small extension using *octomap* module we could extend this method to a third dimension. Alltogether with the camera data we would have a very basic and powerful tool for environment mapping and navigation.

6 Conclusion

We successfully created, tested and documented the data acquision module for the bleeding-edge version of the NIFTi project. We have managed to run the robot and dynamically switch it's nodes between machines connected to the netwrok. Demo of the autnomous mapping is not finished. The source code is saved in the /home/robot/workspace/demo directory.

7 Appendix - fully configured launch file for ctu_data_logger

```
<launch>

<!-- data acquisition configuration -->
<param name="log_camera" value="/home/robot/ctu_logger/image%2d.png" />
<param name="log_laser" value="/home/robot/ctu_logger/laser"/>
<param name="log_camera0" value="/home/robot/ctu_logger/cam0%2d.png"/>
<param name="log_camera1" value="/home/robot/ctu_logger/cam1%2d.png" />
<param name="log_camera2" value="/home/robot/ctu_logger/cam2%2d.png" />
<param name="log_camera3" value="/home/robot/ctu_logger/cam3%2d.png" />
<param name="log_camera4" value="/home/robot/ctu_logger/cam4%2d.png" />
<param name="log_camera5" value="/home/robot/ctu_logger/cam5%2d.png" />
<param name="log_raw" value="/home/robot/ctu_logger/raw%2d.png" />
<param name="log_gps" value="/home/robot/ctu_logger/gps" />
<param name="log_panorama" value="/home/robot/ctu_logger/panorama%2d.png" />
<param name="log_imu" value="/home/robot/ctu_logger/imu" />
<param name="log_irc" value="/home/robot/ctu_logger/irc" />
<param name="log_mechanization" value="/home/robot/ctu_logger/mechanization" />
<param name="log_pcl" value="/home/robot/ctu_logger/PCL" />

<!-- Trigger configuration -->
<param name="trig_distance" value="1" />
<param name="joy_trigger_button" value="2"/>
```

```
<!-- data acquisition modules -->
<node pkg="ctu_data_logger" name="data_logger" type="data_logger">
</node>

<node pkg="ctu_data_logger" name="trigger_node" type="trigger_node">
</node>

<!-- other robot nodes -->
<node pkg="joy" type="joy_node" name="joy_node" >
<param name="dev" value="/dev/input/by-id/
usb-Logitech_Logitech_Cordless_RumblePad_2-joystick" />
<param name="autorepeat_rate" value="10.0" />
</node>

<param name="/CAN_device" value="/dev/usb/cpc_usb0" />
<node pkg="nifti_robot_driver" type="nifti_robot_node"
name="nifti_robot_node" output="screen" />

<node pkg="topic_tools" type="mux" name="mux_cmd_vel"
args="/cmd_vel /nifti_teleop_joy/cmd_vel" />

<node pkg="nifti_teleop" type="nifti_teleop_joy.py"
name="nifti_teleop_joy" output="screen">
<param name="cmd_vel_topic" value="/nifti_teleop_joy/cmd_vel" />
<param name="cmd_vel_mux_topic" value="/nav/cmd_vel" />
</node>

<node pkg="LMS1xx" type="LMS100" name="LMS100">
<param name="host" value="192.168.1.72"/>
</node>

<node name="insnd" pkg="ins" type="ins"
args="-cf /home/robot/workspace/ins/mechanization_config
-o /home/robot/workspace/ins/Output -y -af avg -il"/>

<node name="omnicamera" pkg="omnicamera"
type="omnicamera" respawn="false" output="screen" />

<node name="img2pano" pkg="omnicamera" type="img2pano_lut"
args="-m 7-ih 616 -iw 808 -ph 480 -pw 960 -r 20
-path $(find omnicamera)/res/lut" respawn="false" output="screen">
<remap from="viz/image_out" to="viz/omni" />
<param name="image_transport" value="raw" type="string" />
</node>

</launch>
```

7.1 3D scan launch tutorial

1. Turn on the robot

2. Set up the network interface for the laser scanner
   sudo ifconfig eth8 192.168.1.118

3. Run the basic scanning module e.g.
   roslaunch nifti_teleop nifti_teleop_joy.launch

4. Turn on the 3D point cloud node
   roslaunch nifti_laser_assembler nifti_laser_assembler.launch

5. Make the laser head rotate (joystick button 3 + horizontal cross buttons)

6. Launch the ctu_data_logger module with the desired parameters specified

The laser assembler node sends out a point cloud on each turn of the laser. You might want to check if the laser is working properly. This can be done by checking the messages of /scan and /nifti_point_cloud. For example run

rostopic hz /nifti_point_cloud

References

[1] **The nifti project** (26. September, 2011) www.nifti.eu

[2] **Robot Operating System** (26. September, 2011) www.ros.org

[3] **OpenCV** (26. September, 2011) www.opencv.willowgarage.org

[4] **The Willow Garage** (26. September, 2011) www.willowgarage.com

[5] **Xsens unit SDK** (26. September, 2011) www.xsens.com/en/mt-sdk

[6] **SLAM** (26. September, 2011)
wikipedia.org/wiki/Simultaneous_localization_and_mapping