Description of InMoov's Control System

Casimir Sowinski

Introduction

The system devised for InMoov is implemented in the Robot Operating System (ROS), a meta operating system and framework that operates between the Unix kernel and everything else. It runs as a network of independent, distributed node that are nothing more than executables sending and receiving messages with other ROS nodes. The main advantages of using the ROS framework are threefold. First, the nodes communicate with TCP/IP and can reside anywhere making it easy to coordinate on board computers, micro-controllers such as Raspberry Pis or Arduinos, and external computers connected on a LAN or WAN. Second, due to the nodes' independent nature, ROS as a whole is very fault tolerant. If the node that was sending a camera stream to elsewhere in the robot experiences an exception, everything else simply loses video feed. The third great feature of ROS is the supporting community and wealth of existing packages that have been written and are maintained by programmers around the world. If you need to integrate a Simultaneous Location And Mapping (SLAM) system in your robot, there are many choices that have already been figured out that you can integrate into your system and modify to meet your needs.

When you start a node, it registers itself with something called the *ROS master*. This is a lookup service that is not unlike the DNS system. When a node starts, you need to tell the master what information that node sends and receives, the names of that information, and the formats expected. Specific message channels are called *topics*. They have a format, and a name. Instances of communication through a topic are called *messages*. Many message types are provided that are tailored to specific tasks, but you can define your own, as well. A typical message type may take the form of a simple struct: containing a string that contains the message type, some integer arrays that contain positions and orientations in space, a time-stamp field, and whatever else deemed necessary.

The illustration below, Illustration 1, shows a ROS graph that was taken during some typical running condition. This figure is auto-generated by a package name 'rqt_graph', so it is a little messy. In the graph you can see different nodes that are running which are depicted as ovals, the topics are depicted as boxes, and the arrows between them show connections and the direction of the messages. As an example, on the left there is a topic called '/joint_states'. This contains the positions, accelerations, velocities of joints in a robot as well as some other information. This topic has two nodes that may publish to it: 'n__joint_state_publisher' and '/arbotix'. This topic has many subscribers which include '/translate', '/move_group', '/robot_state_publisher', 'left_gripper_controller', and 'right_gripper_controller'. For an example of a node, look the box labeled 'translate' that is connected to '/joint_states'. The node subscribes to '/joint_states' and parses relevant information from it and publishes that information on multiple smaller topics that can be sent to the Arduino over serial.

These nodes may be written in C++, Python, or Lisp. Our system uses C++ source files and Python scripts. Other file types such as XML and YAML are used extensively to define anything from how nodes are launched and what parameters are set to how the robot is laid out physically. Since ROS operates over the Linux kernel the *bash* shell language is used extensively too. InMoov's system is divided into around 15 different packages. Packages are the smalled unit in a ROS system. Each package is made for a specific purpose or function. One package called 'robo_realsense' runs a couple

of nodes and handles communication between the RealSense and the rest of the system. Another package called 'serial_node' facilitates communication between and Arduino and everything else through USB. In this report I will detail the various packages that are used in the project.
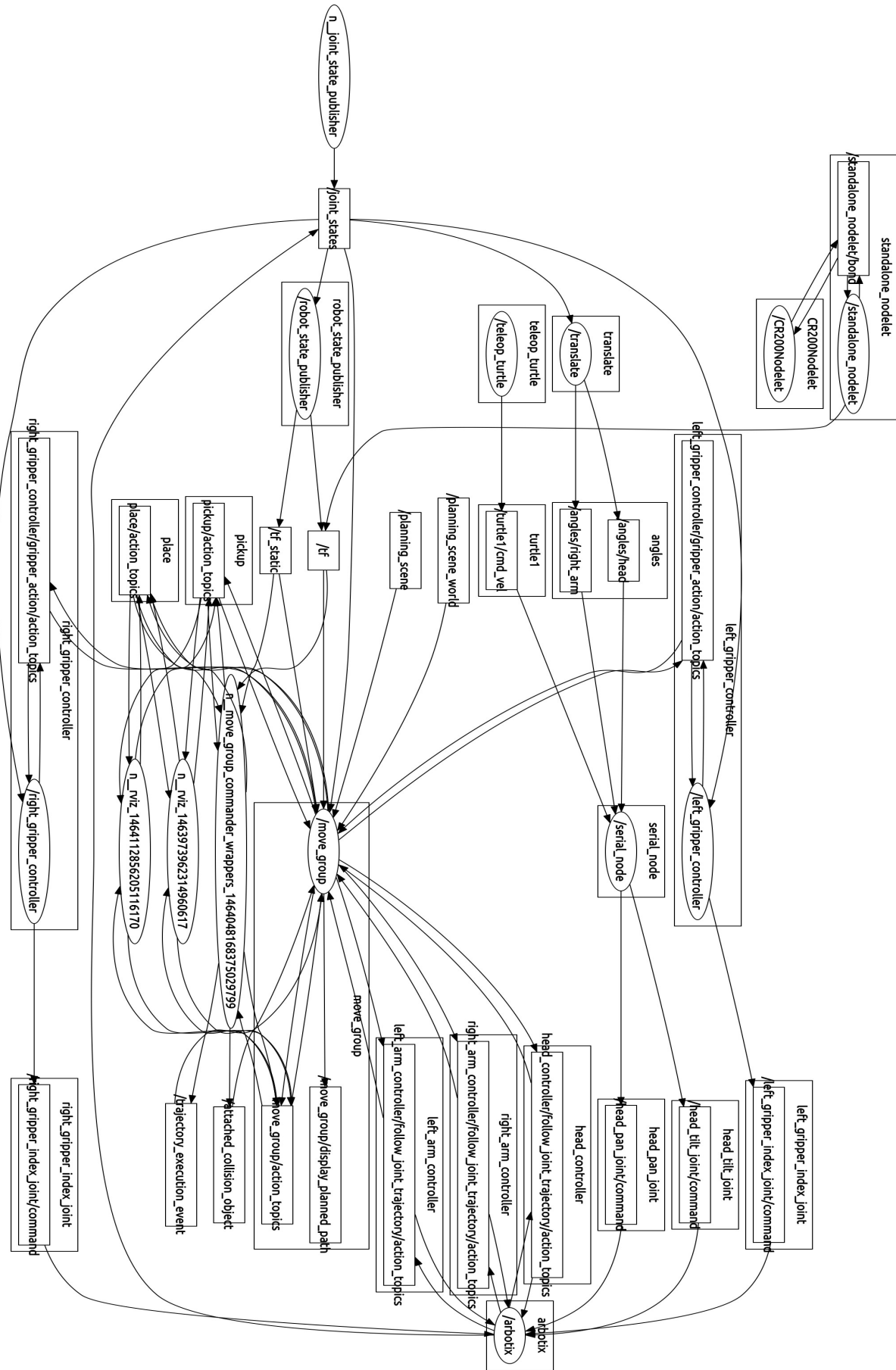

Illustration 1: Typical ROS Runtime Graph

Topics are one way that nodes can communicate in ROS, but there are few other ways that have different applications. Topic communications is used when broadcasting and anonymously receiving messages is whats needed. Its fast and reliable, but not well suited for every application. In some instances you may want to issue a command and get confirmation that the command was followed or at least you would like some confirmation that it was attempted and what went wrong. In this case you would use a ROS service call. If the command is complicated and involves many steps, a ROS action would be more appropriate. This ties into prioritizing tasks and executing higher priority tasks first. An action server would take care of this. An example of an situation requiring ROS actions and an action server would be a Roomba robot vacuum. If the robot is in the middle of a complicated process such as navigating and vacuuming and a higher priority task such as charging low batteries kicks in, it can pause the current vacuuming task, go through the navigating, docking, and charging tasks, the resume the vacuuming task. Another way nodes can communicate is the parameter system.

```xml
<!-- Make sure we are not using simulated time -->
<param name="/use_sim_time" value="false"/>


<!-- Load the URDF/Xacro model of our robot -->
<param name="robot_description" command="$(find xacro)/xacro.py '$(find
robo_description)/urdf/robo_two_arms_realsense.xacro'"/>


<rosparam command="load" file="$(find robo_moveit_config)/config/kinematics.yaml"/>
```

Text 1: Loading Parameters From a Launch File

Parameters are similar to global variables that can be set whenever and can be accessed by nodes and services. You can set parameters directly with launch files, directly in source code and scripts, or in YAML files that can be called from launch files. A simple example of explicitly setting a parameter from a launch file is shown in Text 1. In the first line of code, the parameter `/use_sim_time` is set to `false`. In the second line of code a script called `xacro.py` is used to load the URDF/Xacro files of our robot description (more on this below) onto the parameter server so that other nodes have access to the arrangement and state of the robot. The third line loads a YAML file `kinematics.yaml` that has information about the kinematic solver configuration used to process arm navigation. A snippet of code is taken from a the robo.yaml file in Text 2 that shows how to set parameters in YAML files.

```yaml
port: /dev/ttyACM0
baud: 1000000
rate: 100
sync_write: True
sync_read: False
read_rate: 20
write_rate: 20
joints: {
  # head joints
  torso_head_tele_joint:{id: 1, neutral: 512},
  head_pan_joint:        {id: 2, neutral: 512, min_angle: -90, max_angle: 90},
  head_tilt_joint:       {id: 3, neutral: 512, min_angle: -45, max_angle: 45},
  ...
```
Text 2: Parameters set in a YAML file

robo_description

The `robo_description` package contains files that describe the robot, allow for quick re-configuration, development, and modification, and files that handle viewing and testing of the robot in a visualization program. The `/meshes` folder contains the mesh files (.stl and .dae) that define the physical form of its function groups (links) and the mesh files that define its form for collision calculations. The `/urdf` folder contains a group of files that define topology/physical layout of InMoov, how the joints relate to each other. The `/launch` folder contains a collection of launch files that can load the robot in RViz in any desired way. You can load the robot with only one arm, with only the head, with or without the base, or any other way that would be convenient for testing a particular feature of the model. The package also contains a configuration file for the visualization program RViz, and the usual package meta files that describe the package itself: `package.xml` and `CmakeLists.txt`.

InMoov's physical form is defined in a series of Unified Robot Description Format (URDF) files, Xacro macro files, and a Semantic Robot Description Format (SRDF) file. These are all coded in XML. To start, .stl files for printing InMoov's parts were taken from inmoov.fr and manually assembled into functional groups in a variety of software including Autodesk 3DSMax, Inventor, and MeshLab. Low poly convex hulls were generated from the high definition mesh files for collision calculations. The URDF and Xacro files define the properties of the links and joints. Properties such as length of a joint, its moments of inertia in different axes and the like can be defined for links. Joint properties such as type (fixed, revolute, prismatic, floating), orientation and offset of the axis, and limits on angle, velocity, etc. are also defined in these files. In 6, a snippet is taken to demonstrate how this is done. Here the upper arm link, forearm link, and elbow joint is defined. The first XML tag is `<link>`, and is split into two subdivisions: visual and collision. The link is named `"${side}_arm_biceps_link"`. The `${side}` is a Xacro variable that can be filled in when the Xacro is given arguments. The purpose of this here is to allow an arm to be defined once, and then it can be implemented multiple times without having to write code for each one. In this case, I defined an arm, and have set up arguments that can be passed to it to determine whether it is a left arm or a right arm. The `${side}` will be filled in with "left" or "right" from the calling file. A snippet from the calling file is shown in Text 3. Some other arguments are passed here as well, including whether to reflect the arm's joints (left vs right), what link it should attach to, "torso" here, and the color that should be used when visualizing it.

```
<!-- Right arm -->
<arm side="right" reflect="-1" parent="torso" color="DarkGrey">
  <origin xyz="${arm_offset_x} ${arm_offset_y} ${arm_offset_z}" rpy="0 0 ${pi}"/>
</arm>
```

Text 3: Main URDF File Calling Sub-file with Arguments

In Text 4 we see that the origin of the the joint, what mesh file to use, and what color to use (which is passed from the calling file) is defined in the `<visual>` tag. Collision data is defined similarly, but will use the simpler mesh file for faster calculations. The forearm link is defined similarly. The joint between them is, of course, defined in the `<joint>` tag. This tag names the parent, child, origin, and axis of rotation as well as some limits for use by other nodes.

```xml
<!-- Upper Arm Link -->
<link name="${side}_arm_biceps_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://robo_description/meshes/arm/vis/vis_$
{side}_biceps_03.STL" scale="${scale} ${scale} ${scale}"/>
    </geometry>
    <material name="${color}"/>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://robo_description/meshes/arm/col/col_$
{side}_biceps_03.STL" scale="${scale} ${scale} ${scale}"/>
    </geometry>
  </collision>
</link>
<!-- Elbow Joint -->
<joint name="${side}_elbow_joint" type="revolute">
  <axis xyz="-1 0 0"/>
  <origin xyz="0 0.029 -0.225" rpy="-0.3 0 0"/>
  <limit effort="1000.0" lower="0.0" upper="0.8727" velocity="0.5"/>
  <parent link="${side}_arm_biceps_link"/>
  <child link="${side}_arm_forearm_link"/>
</joint>
<!-- Forearm Link -->
<link name="${side}_arm_forearm_link">
  <visual>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://robo_description/meshes/arm/vis/vis_$
{side}_forearm_03.STL" scale="${scale} ${scale} ${scale}"/>
    </geometry>
    <material name="${color}"/>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <geometry>
      <mesh filename="package://robo_description/meshes/arm/col/col_$
{side}_forearm_03.STL" scale="${scale} ${scale} ${scale}"/>
    </geometry>
  </collision>
</link>
```

Text 4: Sample of Robots Physical Description

The whole URDF/Xacro collection comprises seven files. The top file, usually `robo_two_arms_realsense.xacro`, ties everything together. It includes all of the files that define individual groups like arms, grippers, cameras, bases, etc. and defines their relationships with each other and the robot as a whole. The code in Text 3 is from this top level file. The code in Text 4 is from the general arm file `robo_arm.urdf.xacro`. Also included in the top level file is a reference to a Xacro file that contains definitions for different colors used in the robot, so that the can be put into code via Xacro variables, instead of being hard-coded every time.

robo_moveit_config

To fully integrate a robot model into the MoveIt! Framework more than a set of URDF files is needed. To take advantage of the vast calculation abilities of MoveIt! more information is needed. This is where the SRDF file comes in. The SRDF file contains more information about the semantics of the robot.

Instead of simple having a list of links and joints, certain sets of links and joints can be defined as a single functional unit, a kinematic chain.

The `robo_moveit_config` package is generated by using a MoveIt! utility called the Setup Assistant. Before starting the utility, we used the same script from before, `xacro.py`, to flatten the collection of dynamic Xacro macro files into a single rasterized static URDF file. InMoov's full static URDF file weighs in at about 900 lines of XML. The setup utility uses this static file and user input to define all of the kinimatic chains, what solvers compute what, and much else. You can also generate a default collision matrix which records which collision calculations never need to be performed due to physical constraints of the robot i.e. it won't bother to make sure that a gripper and wheel aren't in collision because the arm is not long enough to have that ever happen. After running the utility and inputing information about your robot, it generates a the SRDF file and a large number of configuration and launch files. Most of these need to be manually edited to reflect the controller configuration of you are using.

robo_arm_nav
This package is based on a package for the PR2 robot and contains a large number of Python scripts for operating the arms. There are scripts that can perform forward kinematics, inverse kinematics, grasphing operations, pick and place operations, and many more. This is mostly where scenes are stored that contain virtual environments that the robot can operate in during testing. There are a large number of RViz configuration files here as well. A collection of launch files also resides here that have specific functions such as head tracking based on point cloud analysis.
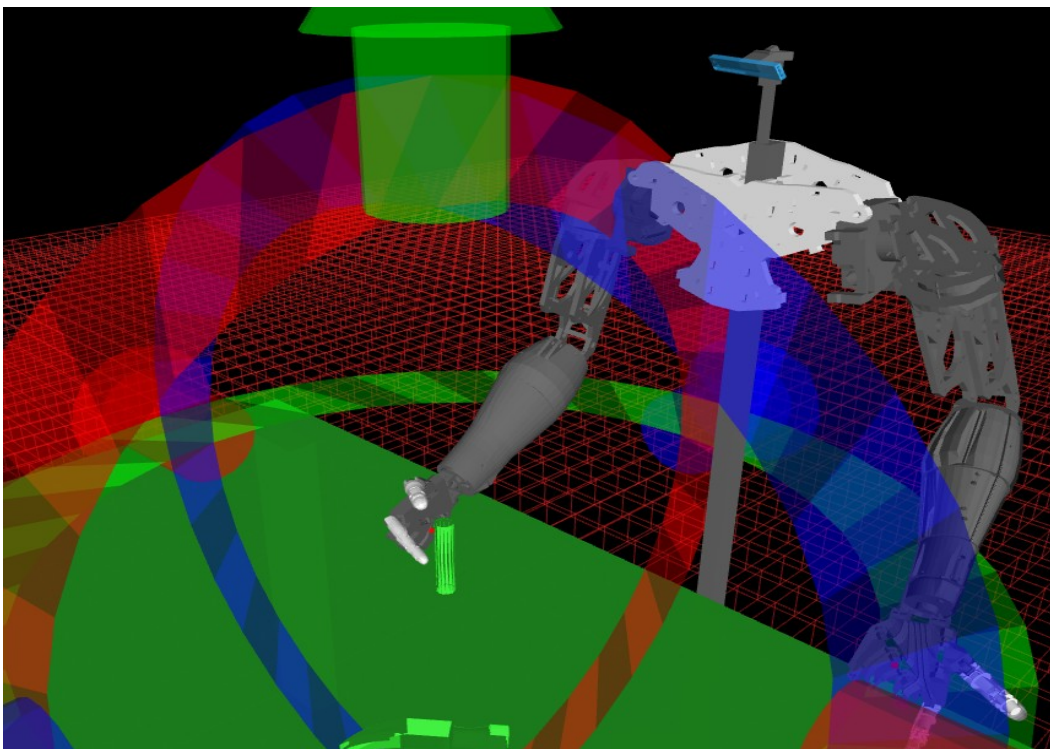

Illustration 2: Pick and Place

A simple example of using the MoveIt! API is shown in Illustration 3. On the right image the arm's starting position is shown in green and the goal position is shown in orange. The left image shows the path that the arm took to get from the start position to the end goal. This is the package that I am most actively developing currently. Some of the scripts are currently fully operating, but not all are yet. Active research is going into configuring gripper controllers to work with InMoov's humanoid styled hands. This package is based on a package used by the PR2 robot.

Another script in `robo_arm_nav` is the pick and place script, shown in Illustration 2. This is one of the most complicated scripts and is still a work in progress. There are some bugs with integrating the gripper controller I wrote with MoveIt!. When this script is fully working it will be able to interact with a perception pipeline to pick out targets from a point cloud, navigate the gripper to the desired object, grasp the object, pick it up, place it in a desired location, and let go. Throughout the process it will be running collision avoidance (with itself and the environment) and using constraint considerations such as "keep this class of water upright to avoid spilling".
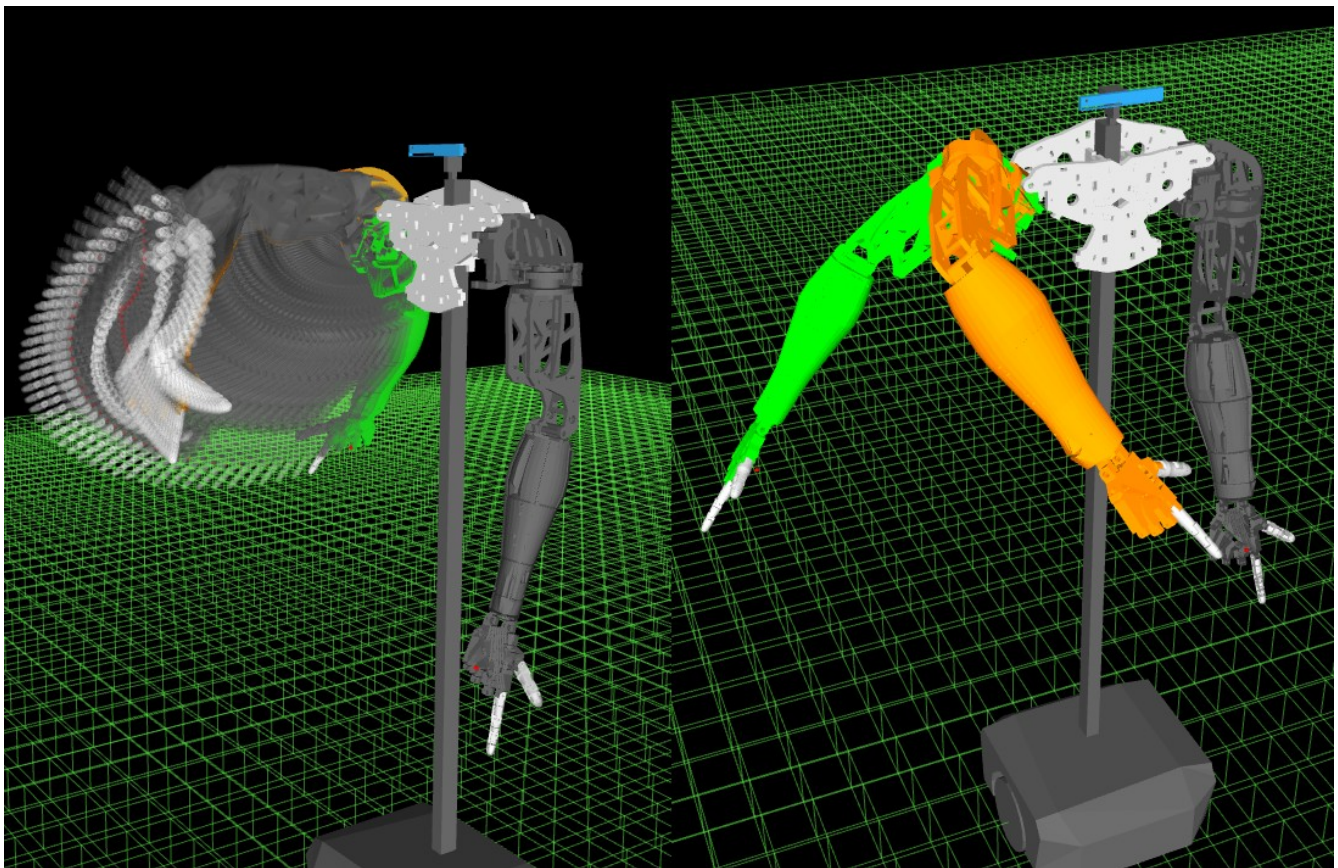
Illustration 3: Arm Navigation Example

robo_bash
The *bash* package contains a collection of *bash* scripts that can be used similarly to the launch scripts used in ROS. I developed them to also give me control over terminal windows and tabs. They act like a super launch file that calls launch files and can optionally give them command line arguments. This is intended to be used during demonstrations and would launch different collections of launch files with different arguments depending on whether you wanted to show a live pick and place demo, a perception pipeline driven demo for object segmentation, or anything else. Text 5 Shows one configuration of the master *bash* script that brings up InMoov for pick and place operations.

```
#!/bin/bash
######################################################################
# Master Bash script to bring up Inmoov in various configs
# Casimir Sowinski 2016
######################################################################
tab="—tab"
foo=""
# Build commands *need to automate file path determination
cmd1="bash -c '~/catkin_ws/src/robo_hand_01/robo_bash/arbotix.bash';bash"
cmd2="bash -c '~/catkin_ws/src/robo_hand_01/robo_bash/move_group.bash';bash"
cmd3="bash -c '~/catkin_ws/src/robo_hand_01/robo_bash/rviz.bash';bash"
cmd4="bash -c '~/catkin_ws/src/robo_hand_01/robo_bash/pickandplace.bash';bash"
cmd5="bash -c '~/catkin_ws/src/robo_hand_01/robo_bash/realsense.bash';bash"
# Which launch files to be loaded in what order
foo+=($tab -e "$cmd1")
foo+=($tab -e "$cmd2")
foo+=($tab -e "$cmd3")
foo+=($tab -e "$cmd4")
#foo+=($tab -e "$cmd5")
# Run commands
gnome-terminal "${foo[@]}"
```

Text 5: Bash master script

robo bringup

This package is similar to the `robo_bash` package except that it uses purely ROS functionality to bring the robot up in the desired configuration. This can be used for testing or for demonstration purposes. This package is based on a package used by the PR2 robot.

robo arduino

The `robo_arduino` package contains everything needed to run all of the code associated with the Arduino. All of the dependancies and libraries associated with the Arduino source code are stored in the `library` and `include` folders. A parameter file, `robo.yaml`, is also currently stored here, but will likely be moved to a different package in the future. The `src` directory holds all of the C++ source files for the Arduino. This directory contains calibration software, full controller software, source for just operating a camera gimbal, and much more.

The development flow for C++ is a bit more time consuming than when scripting in Python. Setting up the packages in ROS is similar but as Python is an interpreted language you don't need to recompile and rebuild packages in ROS's version of Cmake, catkin, every time you change the code. This doesn't usually take a lot of time to recompile/rebuild packages, but it adds up. On the other hand, C++ source is converted into machine code during compilation, so it runs faster than Python.

robo misc

The miscellaneous package has scripts and functions that don't currently have another home. There is a different version of the head tracking script here. Some other examples are scripts to publish markers in 3D space for testing purposes and a battery charge tracking simulator that will be used to develop battery management processes later. There are some other utilities like scripts that enable/disable servos. Many files that were eventually moved to other packages started here. The first version of the dummy marker node that was placeholder code for the other team's data was developed here.

robo realsense

The `robo_realsense` package contains a launch file to set parameters and start two nodes associated with the Realsense. This package is a modifed version of one that Amit Moran at Intel developed.

trans_joint
This package handles the communication between MoveIt! and the Arduino. As described in the introduction the single node that runs from this package is that translate node. This C++ node subscribes to the `/joint_states` topic that is published by the Arbotix controller node. It parses the joint angles and sends them out as published topics to be feed to the Arduino through the rosserial node. To look at one of the topics published by translate use the ROSBash command `rostopic echo {topic name}`. As an example I'll show the joint angles for the right arm, which is published under the `/angles` namespace. The ROS command is `rostopic echo /angles/right_arm`. The output is shown in Illustration 4. This formatted output is defined in translate.cpp. Each kinematic chain gets its own array and the angles are output in an array of 64-bit floating point numbers starting from the first link in the chain and ending with the last. The numbers are angles of the servos expressed in radians.

Illustration 4: Right Arm Angle Message

A better view of the data in a time series format is shown in Illustration 5. A ROS program called rqt_plot from the package of the same name was used to record and view data from `/angles/right_arm` over the course of a transition.
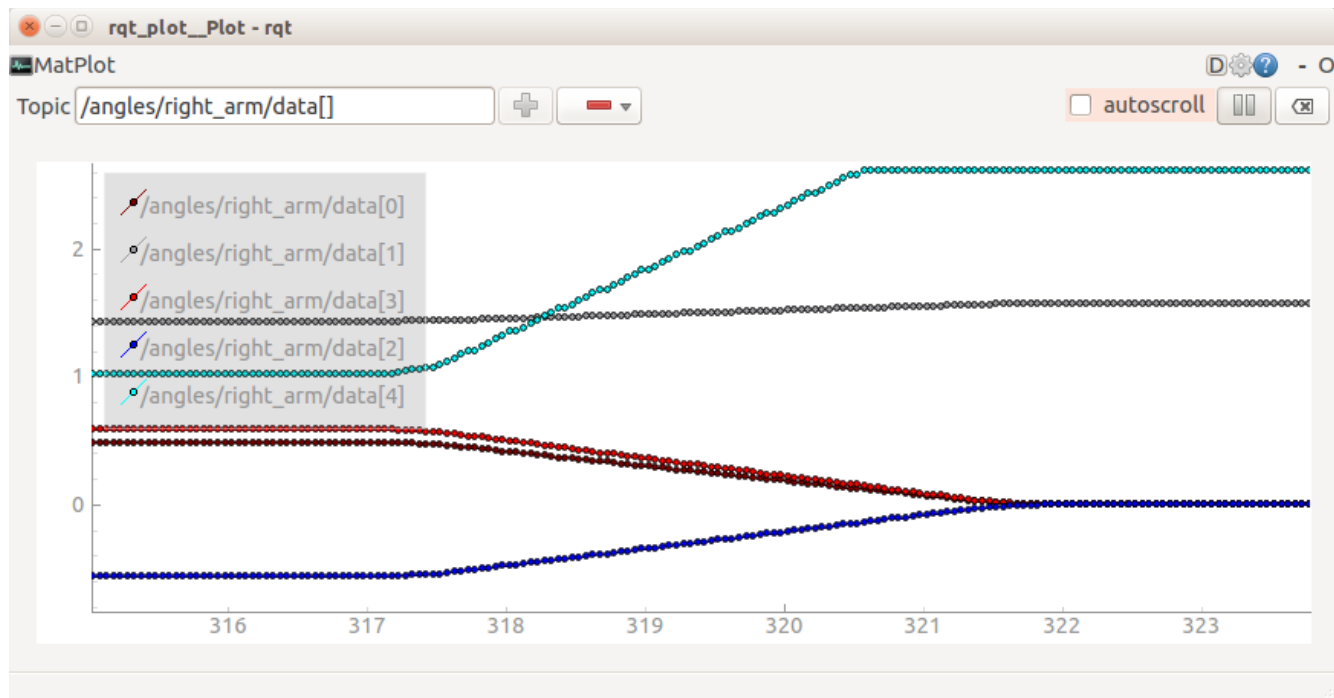

Illustration 5: Plot of /angles/right_arm from rqt_plot

rosserial
This is a cloned git repository for handling data transmission over serial to devices like the Arduino.

dummy markers
The `dummy_markers` package was a placeholder for the other perception group's work. It was meant to publish a 3D marker for use as a target for our pick and place operations. This was unused because they never finished their software enough to send our group any data. While waiting on the segmentation team's code I worked with some point cloud analysis starting with the Kinect and then transitioned to the RealSense. I used a Python script that was developed for the PR2 robot that does simple point cloud analysis and extracts the closet 'blob' in a point cloud and publishes that point as a PoseStamped topic. A PoseStamped message type contains the position tuple, the orientation tuple, and a time stamp.
In the `robo_misc` package there's a Python script called head_tracker.py that takes this published position and runs the calculations to keep the point in the center of the camera's field of vision.

robo setup tf

This package is not currently in use and was briefly used to perform transforms between different parts of InMoov and the environment. The functions originally sought in this package are now performed by the `robo_description` package. Illustration9 below shows InMoov's tf tree that is used to perform transform calculations between frames in the tree.

Difficulties

One difficulty faced in this project stemmed from the same robot being used for two different projects, Melih's and our capstone project. We interfered with each other on a few occasions and butted heads a few times. A couple of times there was conflict between us working on InMoov. Some of the times it didn't wast much time, some times it did. One time I needed the arm, but Melih was using it so I wasted time making a small mock-up of the arm for testing of my controller code. Although this particular task itself didn't take that long, it was one of many.
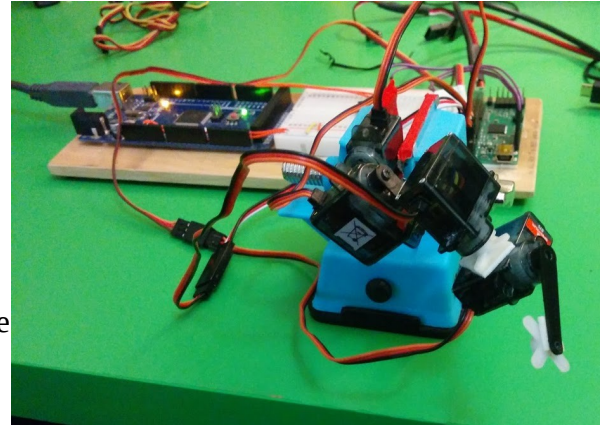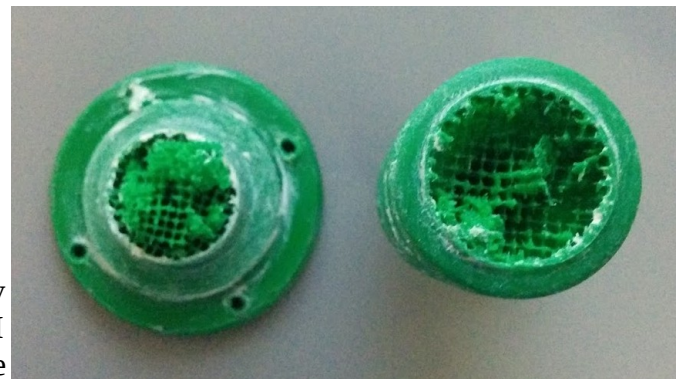

Illustration 6: Mock-up Arm

Another difficulty was getting anything purchased. I ended up combining parts of broken servo motors to make just enough working servos to get the arm working. Piecing together broken motors, buying my own or finding random hardware, wiring, glue, etc. was fairly time-consuming.

The last few weeks I spent much more time than I would've liked working on the actual physical robot as nobody had touched it in months. Other than the pan and tilt joints in the head, and the fingers on one arm, nothing had ever been made ready to actually move. The process of modifying servo motors, mounting feedback potentiometers, tweaking joints, installing hardware took a lot of time. Calibrating the joints and finding the linear mapping equations to translate between the simulated and physical robots was a non-trivial task as well. During the calibration process many pieces broke and there was no time to re-print them so I attempted repairs with epoxy and anchor screws. Future

Illustration 7: Worm Gear with Insufficient Infill

power train parts and parts that see significant torque should be printed with the maximum infill percentage possible. Having such a sparsely filled interior resulted in many pieces breaking and a lot of wasted time disassembling, reassembling, and recalibrating joints.

My biggest complaint about this project is Naser. He did basically no work the entire six months. The ONLY thing he ever did was making the rough draft of the capstone poster. During the meetings that he bothered
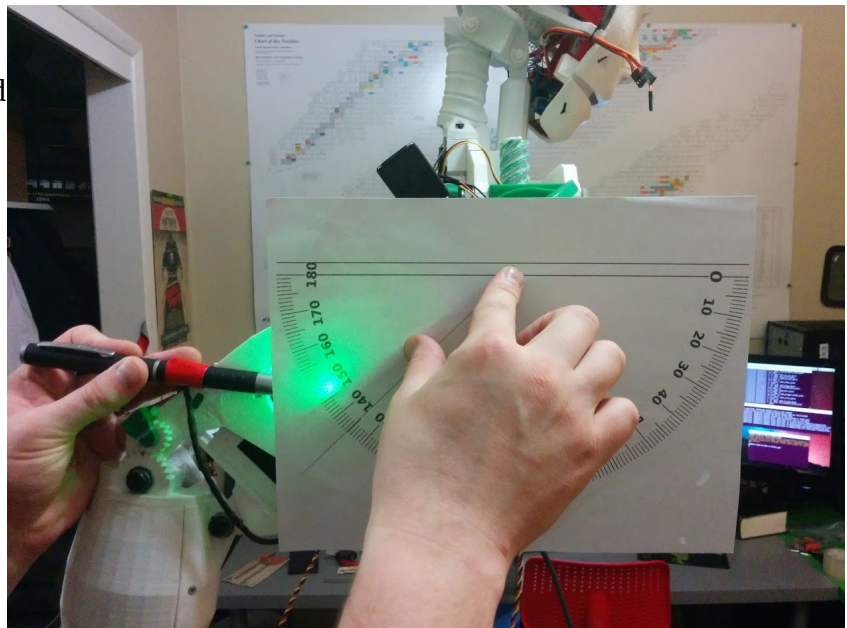
Illustration 8: Part of the Calibration Process

to show up for he would just sit there silently. He refused to contribute anything the entire time. He didn't write a single line of code. When I would ask him to complete a task he would agree, then simply not do it. For the first three months it didn't worry me because we were struggling to learn ROS and I assumed that he was doing the same. I eventually discovered that he hadn't been actually keeping up. During a few meetings I found out that he didn't even have ROS fully installed. Near the end I was desperate for help and asked him to complete a few simple tasks such as modify a couple of servos per the instructions on inmoov.fr. He told me that he completed it, but I later found out that he had gotten Melih to do it instead. He simply refused to help with anything about this project and was absolutely not on our team. I understand that I should have brought this up earlier but I didn't think that it would solve anything, and would only make the working relationship worse.
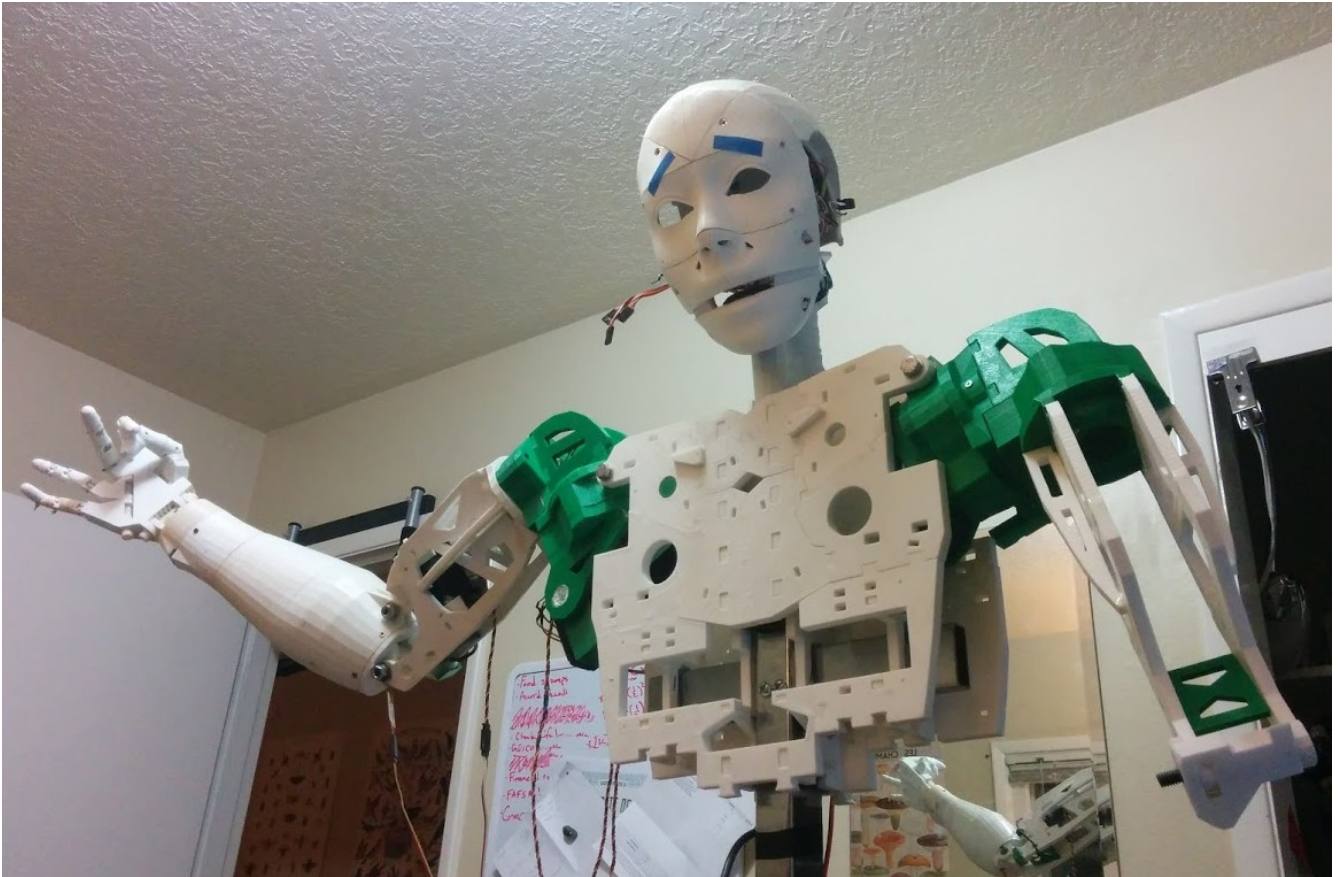
Future Work
There are a lot of loose ends that need to be tied up in this project. Given the time constraints, and only having two contributing members on the team, we had to abandon many features to save time to get a few actually working for the demonstration.

The following is a short list of features that need to be worked on:

- Finish the gripper controllers
- Make a custom kinematic solver plugin for MoveIt!
- Finish the head tracking script
- Finish the pick and place script
- Set up dynamic reconfigure for better testing
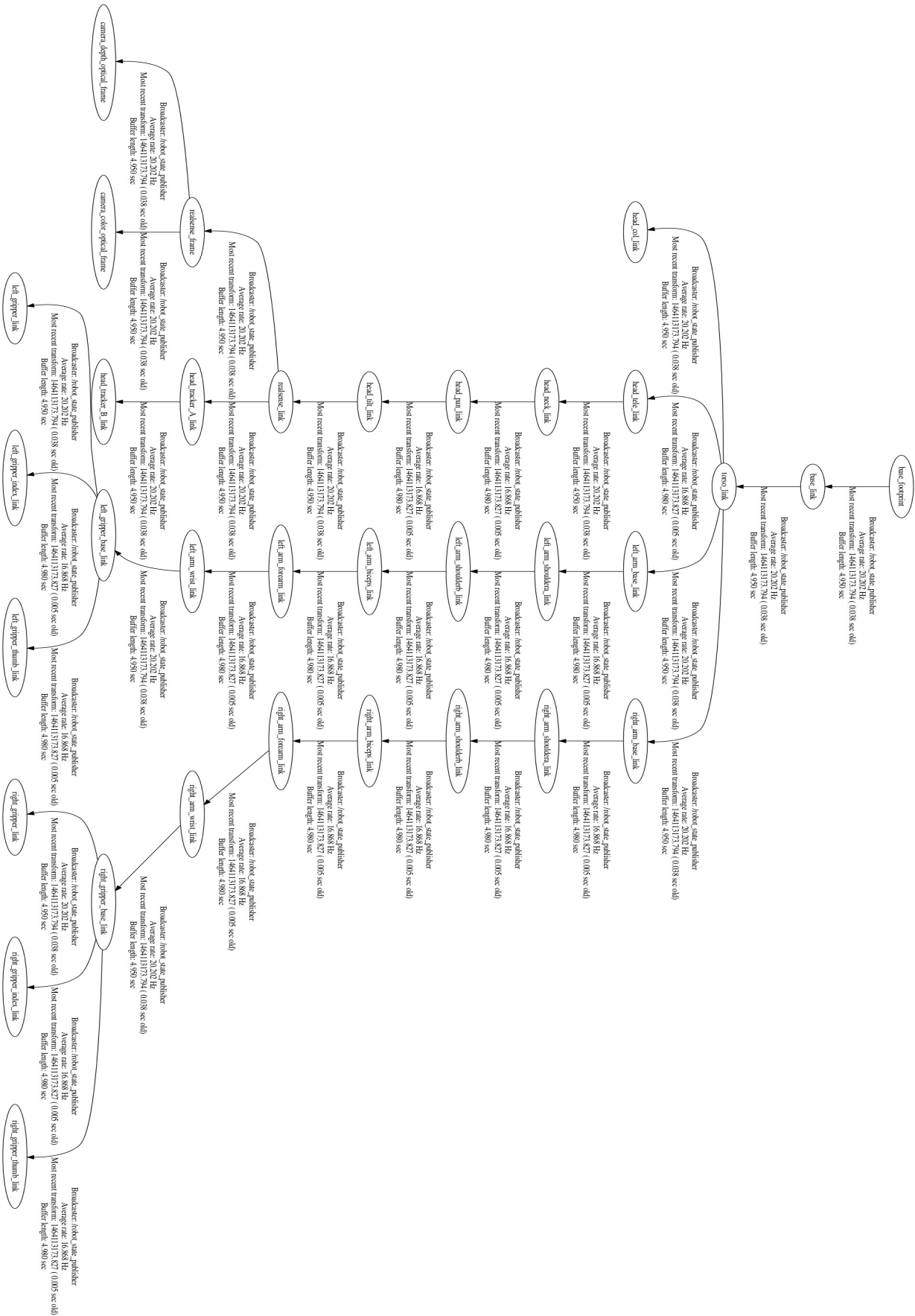- Implement LAN networking to offload CPU intensive tasks to multiple computers
- Integrate segmentation team's code

Illustration9: InMoov's tf Tree