DESIGN AND IMPLEMENTATION NOTEBOOK

Florida Tech IGVC

APRIL 7, 2017

SENIOR DESIGN SHOWCASE

Florida Institute of Technology

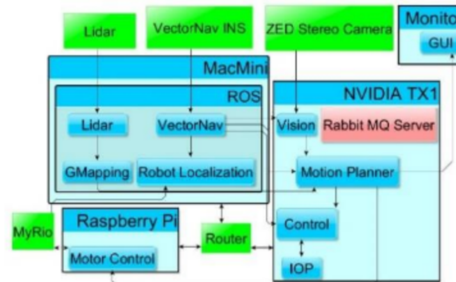
Project Name — Intelligent Ground Vehicle

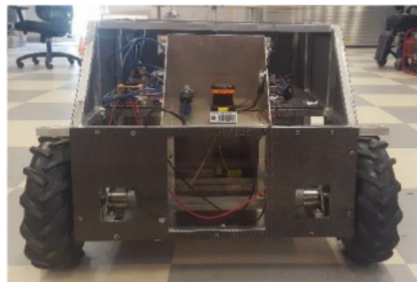
Team Member(s): Brent Allard, Adam Hill, Chris Kocsis, Rohit Kumar, Matthew Salfer-Hobbs, Kartikeya Sharma

Team Leader: William Nyffenegger
Department of Computer Science
Florida Institute of Technology

Faculty Advisor(s): Dr. Matthew Jensen, Dept. of Mechanical & Aerospace Engineering,
Dr. Marius Silaghi, Dept. of Computer Science

The Intelligent Ground Vehicle Competition (IGVC) challenges students and researchers to develop innovative autonomous vehicles capable of comprehending and completing complex, random courses. In conjunction with Florida State University (FSU), Florida Institute of Technology IGVC (FIT IGVC) has developed a robot for this year's competition, as well as a long-term baseline for further development. FIT IGVC's contributions include GPU based lane detection extendable to obstacle detection, a light-weight motion planning and mapping tool, remote control seamlessly interfaced with programmatic control, and a flexible communication framework supporting software in multiple languages across multiple devices. FSU focused on fabrication, motor control, position estimation, and obstacle detection. Both schools collaborated on mechanical design and power requirements. Hardware and software selection for the robot focused on novel hardware and techniques, including the NVIDIA Jetson TX1, the ZED Stereoscopic camera, Sampling Based Model Predictive Optimization (SBMPO), and RabbitMQ. Other technologies used include OpenCV, ROS, D* Lite, and the VectorNav 200 INS. Integration of various components will continue until the competition in June. In summary, we have developed software components compatible with a unique hardware configuration, which is innovative, extendable, and capable of competing.





Contents

Project Goals & Conception 3
Team Structure & Administration 4
Mechanical Design, Operating Modes, & Safety 6
Software Strategy 8
Line & Obstacle Detection 12
Motion Planning & Course Mapping 14
Interoperability Standards 14
Communication Framework 17
Electronics 20

Project Goals & Conception

The project was conceived with several major goals:

1. Increase student involvement in research concerning autonomous vehicles
2. Produce a long-term project, with an extendable system that is adaptable to advances in the field
3. Expose students to industry conditions
4. Develop an innovative platform using similar technologies and ideas as those currently being used in industry
5. Develop a hardware and software baseline for subsequent teams

The hardware and software developed for this project reflect the project's goals. The embedded processors and sensors selected are on the forefront of modern autonomous vehicle design, which leads the algorithms we employ to also be innovative. The software architecture is modular with language independent communication and an interoperable interface. Our overall design focuses on optimizing performance by leveraging these innovative tools to think about the autonomous problems associated with the challenge in new ways.

Concerning goal remote work, Florida Institute of Technology and Florida State University collaborated to produce the vehicle. This partnership provided students with experience designing, fabricating, and programming a robot using resources in multiple distant locations, as commonly occurs in industry. Strategies were developed to leverage the resources provided by both schools to maintain communication, and to apportion tasks.

Already, the connections and experiences gained through this project have increased student and professor involvement, and have also generated notable results. The baseline created is more than sufficient for forthcoming teams to not just compete; but, to innovate with relevant tools.

Team Structure & Administration

Competition Summary

The Intelligent Ground Vehicle Competition exists to further the development of autonomous robots and vehicles by providing the opportunity for students and researchers to improve their subject knowledge and expand the area of autonomous robotics.

The competition is focused on developing a robot capable of completing a complex course correctly within a time limit. The competition itself occurs outdoors in all conditions, except thunderstorms. Lanes made with spray painted lines, random obstacles, and GPS waypoints all define the course. Course traversal must be completely autonomous and without error. Crossing over a lane line ends a run. Hitting an obstacle ends a run. Strong penalties on scores are assessed for infractions.

Concerning the undergraduate competition, speed limits, size limits, and performance benchmarks are set to even out the competition.

The course mandates a set of technical challenges including:

- Mechanical design & fabrication
- Power systems & microcontrollers
- Motor control & Steering
- Course mapping
- Motion planning
- Lane detection & obstacle detection
- Software-hardware integration
- Position Estimation
- Waterproofing
- Communication Frameworks
- Synchronization

Team Structure

Florida Institute of Technology (FIT) and Florida State University (FSU) have spent the past two years researching and developing a robot for the competition. The first year was spent researching the problem space, defining tasks, and acquiring hardware.

This year the goal is to produce a competition worthy robot. The teams split tasks based on areas of expertise and resources. Florida Tech's team is composed of mainly computer scientists and electrical engineers, while Florida State's team is composed of mechanical and industrial engineers. Florida State also possesses more facilities and equipment for fabricating and designing the robot. FSU primarily worked on position estimation, obstacle detection, fabrication, steering, and waterproofing. FIT focused on line detection, course mapping, motion planning, software engineering, software defined communication protocols, and software interoperability. FSU and FIT collaborated on the mechanical design and power system design.

Communication

Communication between two remote entities is a critical task. Regular phone calls, online chats, and document sharing all commonly occur. Slack, a commonly used professional business messaging app, provides capabilities for business chats, private messaging, file sharing, and document sharing. Many of the plugins for Slack augment other tools. For example, the GitHub plugin allows automatic reporting of

changes to a repository in a Slack channel. Channels focused on specific topics have been created as necessary. Several cloud services are also used to share important data. Microsoft OneDrive is used to house the documentation and forms for the team. GitHub is used to share code among all the devices in use. For the design of the vehicle, GrabCAD was used to send models back and forth.

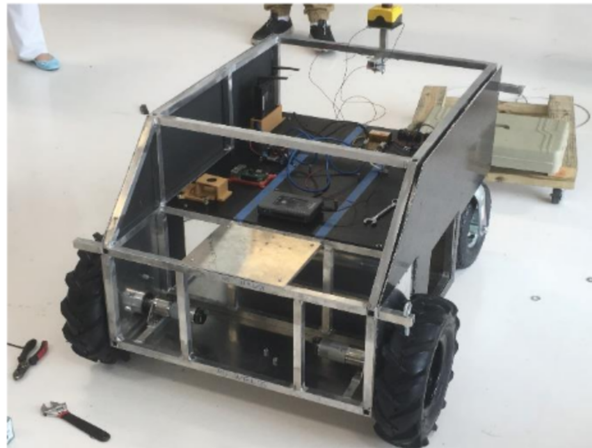
The teams conducted weekly video calls and sometimes daily chats concerning milestones, challenges, and even travel plans. Google Hangouts or simple phone calls were the tool of choice for this. Several times teams traveled to their counterpart's school to conduct kickoffs for the project.

Mechanical Design, Operating Modes, & Safety

Design Decisions Overview:

Three major sources of requirements influenced the design of the structural and steering components of the robot. The steering and durability of the robot were influenced by the competition course, which occurs outside in all but the most extreme weather conditions. Supplementing the course, competition documents themselves dictated motor and steering capabilities. Lastly, choices regarding computational hardware influenced structural components as well as the chassis. Priority was given to maneuverability and survivability.

Concerning maneuverability, three steering options were discussed. These options included rack and pinion steering, differential steering, and skid based steering. Competition requirements specify that the robot must be able to maintain a tight turn radius and the turn radius itself influences software design. Rack and pinion steering was discounted because it limits said turning radius. Skid steering provides the ability to turn in place and use high traction treads; but, the complexity of the kinematics models is an obstacle the teams didn't feel ready to challenge. The complexity of the models stem from the different constants influencing steering based on the terrain (mud, grass, etc.). Differential steering is very similar to skid steering excepting that the vehicle typically does not lose traction making turns, hence decreasing the complexity of the kinematics models. From a programmatic perspective, differential steering rotates in a predictable way which the software can accommodate. Differential steering is implemented on the robot for its high turning rate, maneuverability considering the terrain, and simplicity. Two castor wheels were placed in the rear of the vehicle to aid in weight distribution and turning.



The vehicle itself was crafted using hollow aluminum tubes for the structure. This allowed the vehicle to have a lightweight frame that could withstand physical abuse. To cover the electronics, carbon fiber covers were manufactured to provide protection from rain and other sources of water. The panels also allowed easy access to the systems. The majority of the electronics were mounted onto a single platform that could be easily accessed by removing the top panel. Fans were added to increase airflow over the electronics.

Emergency Stops

The competition rules required there be two different emergency stops for the vehicle. An emergency stop is defined by the competition officials as a way of bringing the vehicle to a complete and sudden stop. The primary stop, is a wireless switch on the handheld controller. The secondary stop is an additional

physical button placed on the robot in an accessible location, raised an acceptable height above ground level, typically on the rear of the vehicle. Both stops are required to be hardware and not software based. The design features a small platform for a large two-way switch to sit on. The platform and support are made of aluminum square tubes to withstand hard hits from people pressing the button. The button itself is wired into the RoboClaw Motor Controller. If the circuit is completed, then the power is directly cut from the motors.

The wireless emergency stop must work up to 100 feet from the vehicle. To accomplish this, two wireless Digi XBee devices are paired so if a button is pressed on one, the other signals the RoboClaw to stop.

Remote Control vs Autonomous Mode

In order to remotely operate the vehicle, a Turnigy 9x remote control receiver is connected into the Raspberry Pi. The Raspberry Pi already handles communication between the myRIO and other subsystems, including the motion planner. Adding remote control to the Pi allows for seamless switching between modes. Additionally, fulfilling a competition requirement to indicate visibly indicate modes—RC and autonomous—is implemented simply using the Pi.

The landing gear switch on the handheld controller is used as a Boolean control to switch between autonomous mode and remote control. From this, the Raspberry Pi may decide whether to pass autonomous commands to the myRIO or let the controller take over. This setup makes use of voltage dividers to prevent damage to the Raspberry PI. The WiringPi library provides functions to read PWM signals from the receiver and send commands via UART (Universal Asynchronous Receiver/Transmitter) to the myRIO.

Software Strategy

Design Decisions Overview

The overall software design methodology revolves around complexity introduced by:

- Multiple processes running time-sensitive operations
- Time-sensitive operations dependent on sensor data and abstracted sensor data
- Multiple programming languages
- Multiple hardware devices

Other constraints involved include the abbreviated timeframe of a senior design project, the research necessary to understand the software tasks, and the in-depth knowledge required of individual sub-problems.

The complexity and constraints led to two design process outcomes:

- An iterative design in which initial designs were refined by detailed research and experimentation
- The assignment of tasks solely to one individual on a team to allow development of subject matter knowledge

Through initial research, the team defined the likely software components, divided the components between schools and individuals, and made major architecture decisions. Experience led us to the most difficult components, where most of our efforts are spent developing.

The major challenges we defined are as follows:

- Developing motion planning software to traverse an unknown map and re-plan often
- Detecting lines and obstacles accurately and fast enough to influence the motion planner
- Building a map of the course in real time
- Determine the robot's position with enough fidelity to build a course map and reach waypoints
- Executing commands through a motor controller
- Communicating between processes in standardized formats across multiple languages
- Implementing an interoperable system, dictated by international standards
- Creating a GUI and logging tools for testing

Florida Tech has taken on motion planning, line detection, course building, communication, interoperability, and the GUI.

Software Architecture

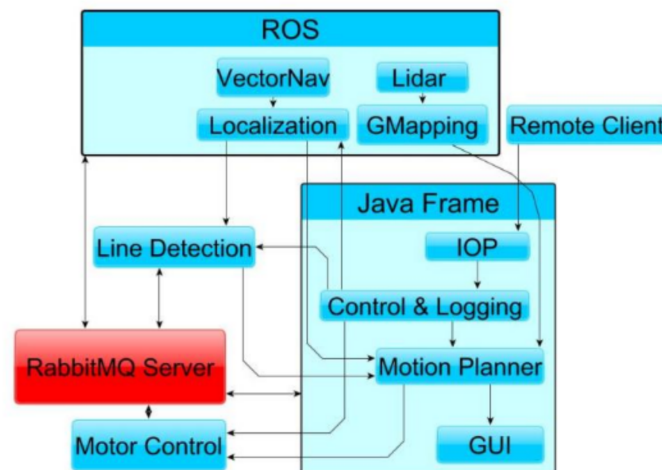
A software architecture maximizing our experience and hardware has been developed to accomplish our challenges. The architecture focuses on the idea of independent components operating asynchronously without RPC commands as dictated by a communication framework. Specifically, we wished to avoid becoming enmeshed in an approach with too much emphasis on a few massive components. Splitting the locations of components to maximize hardware was a major goal, as was maintaining a set of simple components.

The potential components defined include:

- A motor control client for interfacing with the motor controller software

- A motion planning and map building component producing motor commands
- A line detection component utilizing GPU programming; perhaps also accomplishing obstacle detection
- A localization component for producing high fidelity position estimates
- A GUI component displaying the state of the robot
- An interoperability component sufficient for the competition challenge
- An obstacle detection component if necessary using LiDAR
- An overall control component monitoring the states of every other component and changing those states as necessary

Over time we combined several components and introduced other components wrapped by ROS. Our current architecture is given below.



Languages and Libraries

Based on previous experience, the team chose Java to handle high level tasks, C / C++ 11 for low level control and sensory tasks, and CUDA to specifically handle line detection.

Java is used to accomplish the following tasks:

- Motion planning and building a course map
- Simulating courses and scenarios
- The GUI and logging services
- Abstract control of the robot

C/C++ are used to accomplish the following:

- Interfacing with motor control
- Obstacle detection
- Localization
- Sensor input
- Binding to CUDA

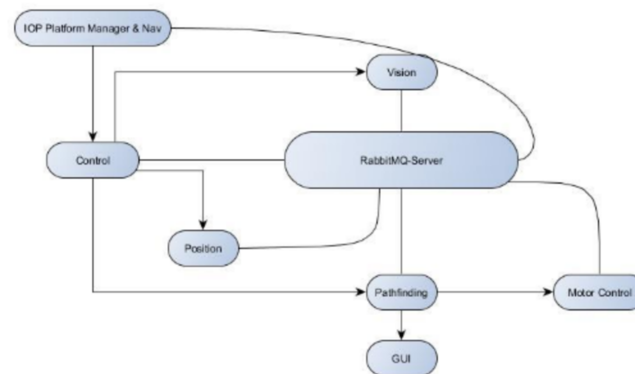
Libraries & tools used heavily include:

- WiringPi

- OpenCV
- RabbitMQ Client Libs
- GSON & RapidJSON
- JTS Virtual Lidar
- ZED Drivers
- Gradle
- Maven

Libev & Boost Control Flow

The design of the architecture maintains the independence of most of the components to limit difficulties encountered with RPC, race conditions, or even the failure of components. Few components may give commands. Those components are the interoperability component, motion planning component, and control component. The motion planning component issues actual commands for the robot to execute and control the GUI through which testing may be done. The interoperability component can relay messages from remote hosts to other components to change behavior including goals, modes, etc. Finally, control logs behaviors and can respond to situations regarding improper behaviors. The component can start, stop, and terminate components as well as change their frequency of publishes, etc.



Failure Points

The complexity of the software system guarantees the presence of many failure points. Some of these failure points we obviated through our design, others are accommodated.

The software architecture assumes that a local network connection is maintained and that RabbitMQ-Server is always running. These assumptions are safe because we are using ethernet and RabbitMQ-Server will run unless the TX1 itself fails. What is more likely is that the client libraries that RabbitMQ provides for different languages will fail on connection issues in ways that are not easy to predict.

The most notable failure points concern the communication framework and C/C++ applications. Most sensors publish on specified intervals; however, some of those sensors are synthesized into position estimations so their behavior is critical. The most vulnerable components are the interface with motor control, and the communication framework's message subscribers. Poorly formed messages are harder to detect and handle in C++ than in Java. As a result, such messages could cause the failure of a component. However, all messages are standardized with predefined classes, serialization methods, and deserialization methods. Someone somehow avoiding those predefined classes could cause significant problems.

Otherwise most of the software acts like independent publishers and subscribers, so failure of a single component will at worst cause the motion planner to give bad commands because the map it has built is inaccurate.

Future Development

This team has focused on developing a baseline for the project. The tools chosen, including hardware, are well supported and will see continuing development for years to come. However, there are areas where the software developed could see significant improvement.

The most important area is in testing. Before the project truly began, an effort was made to maintain space on the school's servers for a continuous integration server. Simultaneously the team also endeavored to develop unit tests and a comprehensive test plan. Unfortunately, the rate of development required to compete has just been too high to both write sets of tests and complete software components. Tests do exist for each individual component, but those tests are not organized. Maintaining a continuous integration server and suite of tests would go a long way to stabilizing the code base.

Another area concerns localization. At the onset of the project, the team did not fully understand all of the technical challenges associated with producing an autonomous vehicle. The hardest of those challenges, which was underestimated, is accurate position estimation. Both teams hoped for a hardware solution to localization; but, truly good localization software requires spending time understanding localization algorithms and developing a model for a robot. This will take at least a year; but, accurate position estimation is an invaluable result.

Concerning interoperability, most of the year has been spent finding and understanding documentation regarding the standards the team is required to implement. With those documents, the code written using those documents, and the documentation left by the team, the software architecture should be modified to fit within those design parameters.

Finally, a look at changing the tool used for the communication framework must occur. RabbitMQ Server works; but, there are other tools that may be better suited to the task.

Line & Obstacle Detection

Design Decisions Overview

The primary goal of the line detection system is to precisely calculate the location of lines in as little time as possible. This goal led to the choice of parallelizing the processing of image data as much as possible. The tool used for parallelization is the Graphics Processing Unit (GPU) provided on the NVIDIA TX1 board which operates as the brain of the vehicle. The GPU provided by NVIDIA provides the speed necessary to process very detailed high resolution images. After experimentation, the team decided upon using an innovative stereoscopic camera called ZED to produce images which are then processed through the GPU. The ZED itself consists of two cameras, and a powerful suite of libraries assisting data analysis. It was designed with the goal of integrating with OpenCV directly on a GPU. The ZED's direct integration with the GPU allows the algorithms to be processed almost exclusively on the GPU where the calculations can be done in a much more optimized manner.

Explanation of GPU Parallelization

Graphics Processing Units are created with the express purpose of doing very specific calculations necessary for video production. The specialized nature of the GPU and its many processors allow for the massive parallelization of calculations at very high speeds which would, on a general-purpose processor, normally take significantly longer to perform. These cards tend to have many hundreds of specialized processors which contain specific instructions to make calculations over vectors and matrices fast. The number of specialized processors enables fast computation times of expensive calculations over large matrices.

Algorithm and Implementation Details

Implementing line detection begins by converting an image from the ZED camera into a the YCrCb image. A YCrCb image consists of three channels where the Y channel denotes the intensity or grayness of an image, the Cr channel denotes the redness proportion of the image, and the Cb channel denotes the blueness of the image. The Y channel is of particular interest because grass often reflects sunlight—much stronger than spray painted white lines, which may give false positives for lines. To remove those false positives, the Y channel is passed through a homomorphic filter, which removes high intensity contrasts from the image. The filter does so by transforming an image to a log domain (basically taking the natural log of the intensity values), performing another transform (FFT) to isolate peaks and then effectively chops them off before returning to the original image. The homomorphic filter removes high intensity areas in the image and normalizes light levels across the image.

Once the image is passed through the homomorphic filter, it is then converted to gray scale format to be passed through a Gaussian blur algorithm to assist in removing remaining noise that will primarily be coming from the grass that we will be operating on. The Gaussian blur algorithm takes square sections of the image and effectively blurs the colors slightly together to produce an overall smoother image.

The image is then passed through a Canny edge detection algorithm. This algorithm looks for contrast differences in the image and then determines and creates an edge on that contrast. This step is greatly assisted by the Gaussian blur, as the blur removes many false edges that can come up due to ambient noise, such as blades of grass.

Finally, the image is then passed through a probabilistic Hough Lines detection. This algorithm produces a list of vectors that represent the found edges of the Canny Line detection. We can then use these lines, and also combine the two images from each camera, to get the best understanding of where the lines exist around us to then pass to the path subsystem to determine our course.

Future Development

Currently the algorithm is in the final phase where the data is primarily there but edge cases need to be tested and any anomalies fixed or mitigated. Currently we have identified extra testing being necessary for certain light levels which can produce false positives in the line detection algorithms.

Motion Planning & Course Mapping

Design Decisions Overview

The motion planning module is responsible for tracking where obstacles are (mapping), calculating efficient paths to waypoints on the course (pathfinding), and generating a sequence of movements which are consistent with the constraints of the robot in question to such paths (trajectory generation), while processing a stream of detected obstacles. In order ensure fast and reliable calculations, the D* Lite algorithm was paired with SBMPO to produce an adaptive pathfinding planner. It is required to dynamically calculate and recalculate the fastest way to get to the goal

SBMPO Explanation

SBMPO, or Sample Based Model Predictive Optimization, is a technique which uses a model of a vehicle to produces a graph representing the space which that vehicle can traverse. Because edges *(ways of getting between)* nodes *(locations)* are controls which can be directly fed to the vehicle, applying SBMPO has a few advantages to alternative approaches. An SBMPO graph can be flexibly applied to a large domain of pathfinding algorithms, and is generated on an as-needed basis, preventing unnecessary waste of system resources. Additionally, SBMPO decreases the size of a pathfinding problem by reducing the search space considered to areas which can actually be accessed by the vehicle, according to its model. Most importantly, because the edges of an SBMPO graph are actions the vehicle takes to get from one place to another, pathfinding and trajectory generation occur in the same step. Finding a path will always results in an appropriately generated trajectory without extra processing.

D* Lite Explanation

D* Lite is an adaptive, heuristic-guided pathfinding algorithm which allows for efficient re-planning of paths to a goal, and is optimized for cases where the start point of the search may move, the target destination is fixed, and the graph is expected to change between recalculations. This makes it ideal for robotics applications. It works by discretizing free space into small grid squares, and keeping track of the optimal method to get to each grid square in that space. In exchange for the cost of maintaining this information, D* Lite allows a path to be recalculated when new obstacles are detected more quickly than one-off algorithms such as A*.

A traditional implementation of D* Lite treats grid squares which have an X and a Y dimension as individual nodes, and considers a grid square to be connected to those adjacent to it. This algorithm has been adapted for use with SBMPO by introducing a third dimension, the direction the vehicle is facing at the time of entering the grid square, and considering a particular node to be connected only to those squares it can reach according to the model of the vehicle. These two modifications provide the benefits of SBMPO while also preventing certain circumstances which would result in a path not being found.

Future Development

Future development of the navigation module will focus on shifting from an occupancy grid representation of obstacles to a more error-tolerant probabilistic model, as well as assisting with localization by applying triangulation to detected landmarks.

Interoperability Standards

Design Decisions Overview

Interoperability (IOP) is the ability for hardware and software sets to communicate to each other in a modular setup, with no modification or understanding of the units needed. By definition, it implies open standards. An interoperable system shall communicate with all other interoperable systems regardless of

when those systems are constructed, or the environment those systems run in. Due to the competition's rules, interoperability of our system will be achieved by following a standard, issued by the Robotic Systems Joint Project Office (RS JPO), based on standards maintained by the Society of Automotive Engineers (SAE) Technical Standards Board. These standards were originally developed by the Joint Architecture for Unmanned Systems (JAUS) Committee. Interoperable standards such as JAUS typically define interfaces that software can inherit. These interfaces each define a service, and the foundation for them. JAUS services are described using the JAUS Service Interface Definition Language (JSDIL), which creates a formal Relax NG Compact schema that can be used for validation.

Documents Overview

There is currently one known required document for the competition regarding IOP. This document, however, is not enough to fully understand what IOP is and what one must do to implement an interoperable system. This led us to find six (6) other documents that provide supporting context. These seven documents, listed below, provide what we believe to be sufficient information to understand moderately what IOP is and how it is implemented in the context of this competition.

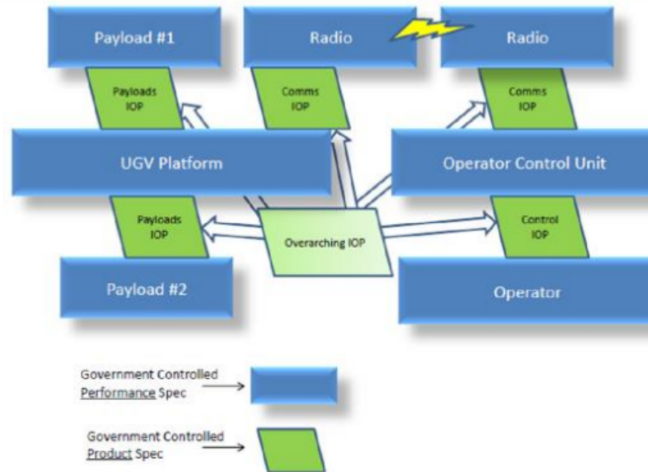
1. RS JPO Documents
   - ADA558678 – Overarching Profile (required)
   - ADA554246 – Capabilities Plan
   - ADA558838 – Control IOP
2. SAE JAUS Documents
   - AS5710A – Core Service Set
   - AS5669A – SDP Transport Specification
   - AS6009 – Mobility Service Set
   - AS5684B – JAUS Service Interface Definition Language

The SAE JAUS documents define and describe the interfaces to be used, as well as some more technical details. The RS JPO documents fill in the interfaces with more specific requirements to be followed.

JAUS Toolset

After much research, a decision was made to make use of the open source software JAUS Toolset (JTS). JTS can read the Relax NG Compact schema defined in document AS5684B (JSIDL), and provided in document AS5710A, and convert that schema into one of three different programming languages. This code is compliant with the RS JPO Interoperability Profiles.

Using the tool is simple. Import the schema, then create service sets and a component with the service definitions that were imported. After you create a component, the option to auto-generate code becomes available. After generating code, there will be certain places that require you to insert your own code to complete functionality, and they will be marked by comment stubs. When your code has been inserted, you can then build your components with the SCons build tool, which is similar to Make or GNU Autotools. Each component has its own executable, which may be run. To let all of the components communicate, you must start up one program beforehand, called the Node Manager. The Node Manager acts as the central server for your interoperable system, and each component registers with it on startup. Because of JSIDL and the Core Service Set, each component has its own unique address, so there will be no conflicts with sending data to the incorrect component service. At any time, a component can submit a request to the node manager for a list of services that are available.



High level IOP overview from RS JPO point-of-view

Issues

Currently there is one known issue with JTS that is preventing us from making significant progress constructing the framework for IOP. This issue seems to have something to do with the Node Manager, as the error only happens when the node manager is running. Investigation is still ongoing; however, it is possible the issue may lie with deprecated functions being used. The code was also auto-generated in Java, so an attempt to try auto-generating in C++ may help us to narrow down possibilities, even though the node manager itself is a prepackaged stand-alone application, and not auto-generated each time. If this issue can be resolved, then we believe that completing the IOP framework will be fast.

Future Development

The IOP framework being developed will eventually take over most of our existing project framework, including the communication framework, described in the next section. However, since the RS JPO Overarching Profile is only on version 0, and thus not complete, it does not specify interfaces for certain components, such as vision or motion planning. Thus, the communication framework we created will still be used, and heavily relied on. We can create a simple client that encodes and decodes the messages that need to be passed around between interoperable components, and non-interoperable components. Most of the predicted work in the future should take place inserting our own code into the correct methods.

Communication Framework

Design Decisions Overview

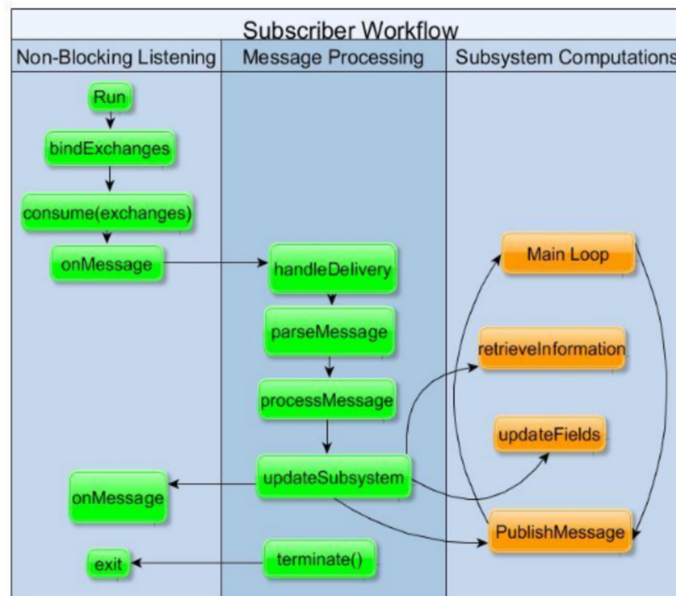
The communication framework was designed to work with multiple devices, multiple processes, and to be flexible concerning programming languages. These design choices reflect the complexity of an autonomous vehicle, the number of parallel operations, and the need for flexibility in the framework with evolving technologies.

An autonomous robot has many complex software components operating in parallel. Those components must process several different kinds of messages fast while maintaining service. Any communication framework developed cannot be based off "waiting" for instructions. Instead, communication events should be processed asynchronously—in parallel with the running process. However, asynchronous processing loops introduce issues with synchronization. Consider a simple example concerning speed. In the motor control component, there are two loops: one interacting with the motors (loop A), and a second waiting for commands from the motion planner (loop B). The motor control component only has one variable storing speed which may be accessed by both loops. If the communications loop (loop B) receives a command and tries to change the speed variable and simultaneously the interactive loop (loop A) attempts to retrieve the speed, what happens?

Furthermore, there is no guarantee that future projects will write code in the same languages or using the same tools chosen this year. Therefore, the framework itself must be generic enough to work with multiple languages.

With those considerations in mind, several tools were chosen and design decisions were made:

- RabbitMQ Server - a generic messaging server using the Advanced Message Queueing Protocol (AMQP)
- JavaScript Object Notation (JSON) as the standard messaging format
- Callback based message subscription and event handling
- Basic synchronization tools



RabbitMQ Server, AMQP, & JSON

Why not a simple implementation?

One possible implementation of this communication framework is using TCP connections between devices. A process would connect to another process using a hardware address and port number then maintain that connection. The initial setup of something like this is simple; however, extending a framework based on simple TCP connections is more difficult. What happens when there are dozens of processes running and each process wants to connect to every other process? Suddenly, many connections must be maintained, programmed to handle errors, and synchronized.

RabbitMQ Server

The concept behind RabbitMQ Server is eliminating the complexity of communication between multiple devices, by having a central entity that handles routing, ordering, and delivering messages to different processes and devices. While RabbitMQ Server is on a device visible by all devices in a network, any process on any of those devices may communicate with any other process on any of those devices. Furthermore, RabbitMQ Server offers different routing protocols, allowing messages to be routed to one, some, or all processes using the server. However, routing is slightly different with RabbitMQ Server. Instead of directly sending messages between processes, messages are posted to the server with exchange-key combinations. Processes ask the server to forward messages posted with those exchange-key combinations.

Many of the clients written for RabbitMQ also handle many errors and exceptions. For instance, if the local internet connection fails for twenty seconds, the connection itself will not fail. The connection will wait a predetermined time and attempt to reconnect until successful.

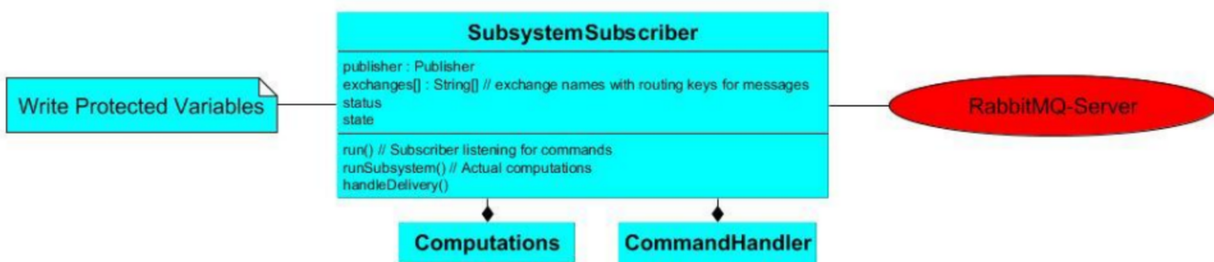
JSON

For simple applications, programmers often define their own messaging formats. The complexity of our application required using a standard, well supported, flexible format. Formats that arise include XML and JSON. JSON was selected mainly for its flexibility in Java, where messages may be directly translated into classes using a concept called reflection. Other benefits of JSON include its simplicity and the number of already existing serializing and deserializing tools available.

Implementation Details

The robot itself works around the idea of publishers and subscribers. A publisher is any program that publishes messages onto the server and a subscriber is any program that receives messages. Programs may be both publishers and subscribers; however, publishing and subscribing should be independent.

The framework was implemented in two languages with clients available via open source or officially from Pivotal, who authored RabbitMQ.



Java

The Java implementation focused on producing a framework where important data was saved in synchronized structures available to a software component. Four components are in Java – the motion planner, the GUI, the events logger, and the interoperability client. These clients are all part of one process that splits components into threads and manages them via executor services and threads. The communication aspect of all of that means that each component has its own asynchronous, callback-based subscriber, which handles an event. The callbacks increase the resilience of the framework because exceptions occurring in the framework will not propagate to the connection.

Google's GSON library for handling JSON messages was selected as the main tool.

C++

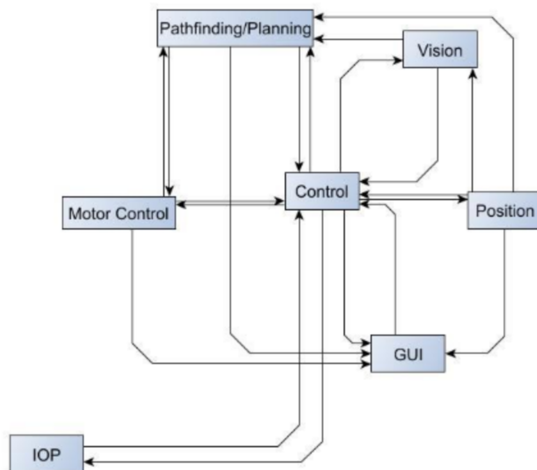
The C/C++ implementation is not as clean as the Java implementation. RabbitMQ is generally used for web or phone applications. Normally those applications use languages like Python, Java, JavaScript, Ruby, etc. Support for RabbitMQ in C++ is more limited. However, several open source libraries do exist, though none are truly complete. Several such libraries were stitched together and wrapped to produce the necessary functionality. Another open source JSON handling library called RapidJSON is also wrapped.

Library selection:

- AMQP-CPP implements a RabbitMQ client in C++ 11 with callback functionality necessary for subscribers
- RabbitMQ-C and SimpleAMQPClient implement simpler publishing methods
- Libev is a dependency necessary for AMQP-CPP that allows for the construction of event loops
- RapidJSON is one of the fastest JSON handling libraries in C++ that also provides the ability to serialize and deserialize complex JSON messages

These libraries together form the basis for the C++ frame.

Control Flow



The communication framework has been explicitly developed to not use remote procedure calls (RPC). Remote procedure calls occur when one process communicates to another process that something needs to be accomplished, and waits for that data as a reply. RPC is unreliable because unexpected errors will prevent returning information to the caller and halt components.

Instead, on arrival of a message a callback is executed. That callback changes the state of data or flags available across the program which causes a change in behavior of the program. No data is explicitly returned in direct response to a message including interoperability messages.

Future Development

Synchronization tools, in practice, are almost never the best solution to a problem. The abbreviated timeframe for development required the "quick and dirty" solution. Long term components should handle

events arriving in a thread by placing them in event queues, which are processed as fast as possible separately from communication.

RabbitMQ in C++ is not well supported. Three courses of action may be taken regarding the overall framework.

a) Continue using RabbitMQ, but significantly improve the error handling and resiliency of the current C++ implementation. RabbitMQ may be directly extended to Python if some components are written in Python in the future.
b) Switch to ZeroMQ, which is a similar library that is natively implemented in C and more common for such applications
c) Use ROS which pseudo-implements AMQP and provides simpler tools. Switching to ROS will require finding an interface for ROS with Java. Typically, ROS applications are written entirely in C++ and Python. Currently, ROS developers are talking about switching to ZeroMQ or RabbitMQ with new versions.

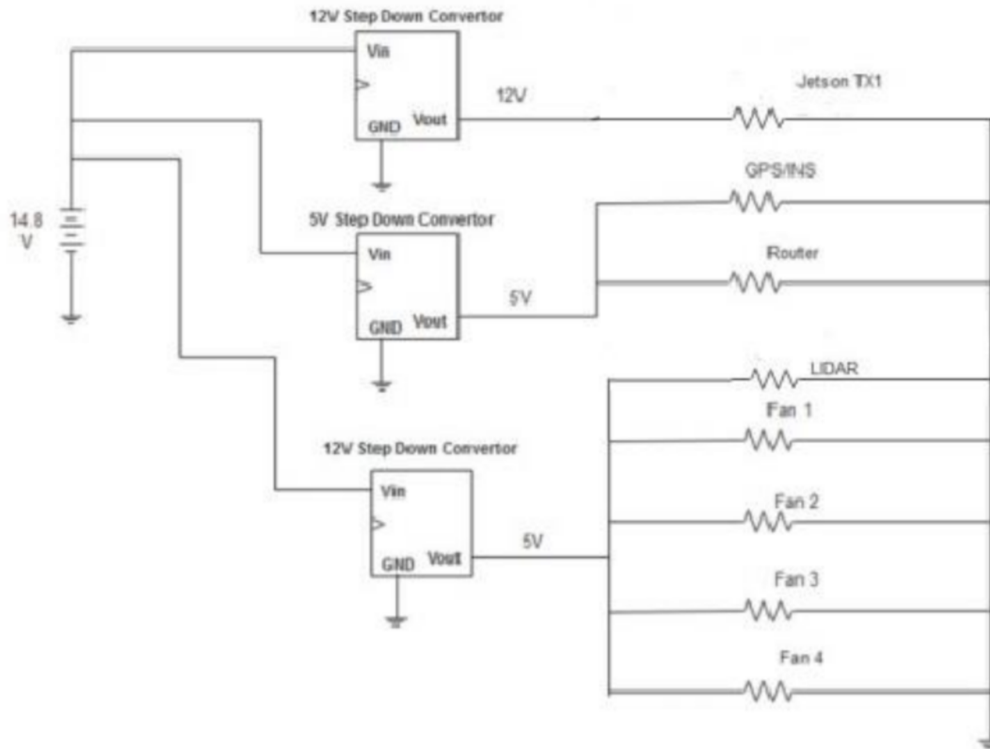
Electronics

Design Decisions Overview

Two main electrical tasks are considered in the design of the robot. The first is safely powering all of the sensors, motors, and embedded hardware. The second is enabling timely communication between each device. We decided to go with a separate voltage regulators for each component on the overall circuit. Separate regulators provide each component with sufficient current in a consistent, controllable way while accounting for the relative inefficiency of higher current output and temperature concerns. Additionally, separate regulators reduce the risk of multiple components failing due to a fault with a single regulator.

Both teams proposed ideas and collaborated on the power design; however, for the final design, the teams opted for large step down voltage regulators which output sufficient current to run multiple components. Only two different voltage buses (12V and 5V) were created and the appropriate components could be added or removed from the buses. This significantly reduced components and allowed completing the power system at the appropriate time.

Electrical System



The circuit is split into three parts and primarily powered through a 14.8 V 4S Li-Po battery. The Jetson TX1 requires 12V to operate; it exists on its own component of the circuit with a 12V regulator. A set of smaller devices, including the GPS/INS and the router, exist on a 5V bus supported by a 5V regulator. The last part exists similar to the TX1 on a 12V bus and regulator; this part consists of the heat management system (Fans) and LiDAR, both of which require 12V to operate.

Two main step down voltage regulators were chosen for this circuit, 12V and 5V regulators. We decided to go with step down regulators, as they are more efficient and stable. Our choice of a 14.8V battery made the dropout voltage became a major factor while designing the 12V bus. The dropout voltage determines how much voltage can be lost from the source until the regulator stops giving 12V constantly. The 12V voltage regulator was selected as it had a low dropout voltage, which is sufficient for our battery to not go below 13.7V as it needs to be charged before hitting that voltage. The regulators also have capacitors built into them for noise reduction.

Sensors

The sensors used on the vehicle are:

- ZED Stereoscopic Camera: This is the main vision sensor used on the vehicle. It is mainly used for line and obstacle detection. It can sense the depth on an object and its color.
- VectorNav 200 INS/GPS: A 10-Axis MEMS IMU. This is used to track the movement of the vehicle and get its location. This GPS does not use a base station, which was a requirement of the competition.

- LiDAR: The LiDAR is used as a secondary sensor to the stereoscopic camera to detect obstacles at great distances with high fidelity. Those obstacles provide additional landmarks with which to map the course and localize the robot.
- Encoders on the motor controllers

Software / Hardware Integration

The two main computers used on the vehicle are:

- NVIDIA Jetson TX1
- Mac Mini

For this project, we needed a computer that excelled in visual computation, as the computer would be in a continuous cycle of image processing while on the vehicle. The NVIDIA Jetson TX1 is built for image processing. It has a powerful GPU that can decode videos with up to 4K resolution (exceeding requirements). The board also has a USB 3.0 port, some UART ports and built-in Ethernet and Wi-Fi for communication with other devices that we will be using. We also needed a stereoscopic camera for depth perception and object/line detection. The ZED camera from Stereolabs gave us all the specifications we needed. Also, the ZED has support on the Jetson TX1, as you can download its SDK.

Apart from the TX1, we needed a computer which could handle ROS (Robotic Operating System). We use ROS to communicate with the VectorNav and LiDAR. The Mac Mini has Ubuntu 14.04 installed as its operating system for this purpose. Both the computers are connected to each other via a local router, located on the vehicle, which routes all networking packets to their appropriate devices.