AI XPrize Architecture

Remy Bubulka, Coleman Gibson, Kieran Groble, and Lewis Kelley

November 5, 2017

The Goal

RHIT is participating in the IBM AI XPrize Competition in 2020, and the proposed project is to make a robot that can understand and react to natural human interaction. This robot would work specifically in the domain of manipulating blocks. Our part of this multi-year project is to architect a system that can be easily used and extended by future teams.

Core Requirements

- The main hardware work will be done in ROS, so we **must** interface effectively with it.
- Due to this being a research project with constantly varying ideas, plans, and needs, it should be easy to slot new components in and out in different combinations to facilitate rapid experimentation and development.
- Many different people will be working on many different types of components for this project, and that requires the ability to use different programming languages to code some of the functionality.

Networked UIMA

We decided to use the Unstructured Information Management Architecture (UIMA) as the foundation of our architecture due to it allowing for the easy restructuring, addition, replacement, and removal of different components in the pipeline independent of all the rest.

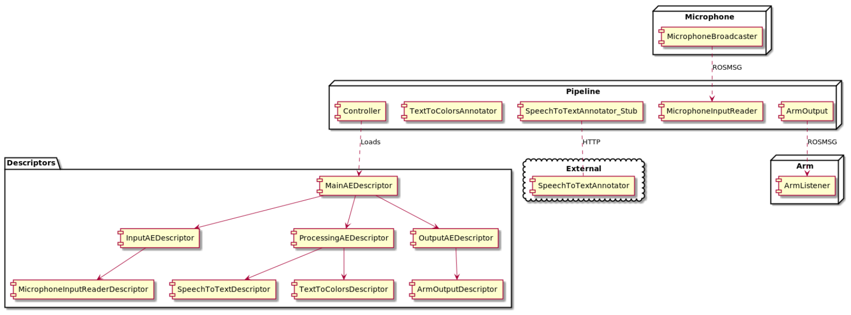
There is however a problem: while versions of UIMA do exist in other languages such as C++, they appear to have been abandoned, leaving only the Java version under active development. That leaves us the task of figuring out how to tie the Java UIMA into other languages.

We came up with a design that allows for various communication protocols to be used, but currently that protocol is only implemented using HTTP.

HTTP UIMA Protocol Description

1. Serialize CAS object into JSON using built-in UIMA functions.
2. Send Multipart HTTP request to an external server with the JSON-CAS and any necessary binary data.
3. External annotator (acting as a server) receives the request, processes it, and returns a list of annotations in JSON.

Resulting Architecture Diagram



This diagram illustrates how the various components of our architecture connect together. Starting off in the bottom left corner is our `Descriptors` folder. This is a UIMA feature that allows the use of XML Descriptor files to define the structure of the data pipeline. This separation of structure from the functions of the code allows for less coupling between code units.

Next, we have the various ROS nodes. ROS is essentially a message queuing service, and the various publishers and subscribers are referred to as ROS nodes. In this example, we have three: a publishing `Microphone`, a subscribing `Arm`, and our `Pipeline` which serves as both. These components communicate using ROS messages that can cross language barriers.

The pipeline itself is made up of a variety of `Annotator`'s, classes that take in a blob of data and add annotations to it for future annotators to evaluate. In our architecture, some of these annotators will be ROS subscribers in themselves and listen for input from whatever ROS publishers they're looking for. Likewise, some annotators near the end of the pipeline will be ROS publishers to push out the final results.

Any one of these annotators can be an "external annotator," one which simply uses our protocol to communicate with another program in another language that will do the actual work. This allows for arbitrary amounts of work to be pushed fairly effectively and efficiently onto these external processes.

Example Use Case

We're going to add an example external annotator written in Python. We'll assume there is an existing HTTP Annotator superclass in Python, so we don't need to worry about implementing the protocol. This annotator's job will be to find bits of text which refer to colors in a given string.

1. Write an XML Descriptor File using the Eclipse plugin's GUI.
2. Add the Descriptor into the pipeline by editing one or two existing Descriptor files in the pipeline.
3. Write a small piece of Java code to create a stub class that will be referred to by the Descriptor File. This class will use the existing protocol interface in Java, allowing the class to basically become a stub.
4. Write the Python annotation type which will be applied onto the CAS.
5. Write the Python annotator that will use the Python implementation of our protocol to receive the CAS, process it, and return annotations onto it.
6. Add a new item in the configuration file specifying where the remote Python server can be found.
7. Run the program like so:

```
python ColorAnnotator.py &

roscore &
rosrun edu_rosehulman_aixprize pipeline \
       edu.rosehulman.aixprize.pipeline.core.Controller
```