Distributed Vocal Control of Robot Sets

Alan Gou and Jared Greene

E-mail:

Abstract

Using a client-server model, we construct a system that allows multiple users to control multiple robots using natural language. The robots are able to both respond to and execute on user requests in individualized manners through the use of API.AI agents. Through this architecture, vocal interaction can be incorporated into any ROS node by building a light-weight vocal interaction interpreter, interpreter specific action id, and intent on API.AI atop our system.

Introduction

The fields of Natural Language Processing (NLP) and cloud-based web infrastructure have made enormous gains in the past decade. Alongside these advances is the proliferation of Internet connections for devices and objects from microwaves to toothbrushes to even cars, such as Tesla's Model S class of vehicles.

This has spawned a new category called the Internet of Things, which encompasses connecting everyday devices such as thermostats, loudspeakers, and responsive assistants such as Amazon's Alexa and Echo. We believe that as NLP improves, the Internet of Things trend continues, and as robots become more capable, there will be an ever-greater need to build applications that can interact with these robots through the Internet.

Our Vision

Our long-term vision is a platform that unifies the locally-installed software that powers a robot with a cloud-based Application Programming Interface (API). This will allow robot-side developers to develop nodes in ROS that will expose APIs conforming to the standards specified by the platform so that developers that lack experience with ROS and robot-side development can build applications that can still interact with them in a meaningful manner.

This platform will abstract away the more involved aspects of NLP, instead providing an easy way for developers with little or no experience in implementing NLP to define queries, commands, and build up conversational flows with their robots. In the case of robots that are connected to the Internet, we can rely on the luxuries of cloud-based infrastructure to offload NLP processing to machines tailor-made for those tasks, instead of trying to implement it on a robot that must also implement grasping, movement, localization, and all manner of truly vital functionality to even have a working robot.

Related Previous Work

There has been extensive work in these areas. Voice control for robotics systems has been worked on rather extensively.[1] Natural language processing for robotic systems has moved from simplistic keyword parsing and mapping,[2] to approaches that utilize more robust natural language processing.[3] These new approaches include context understanding. citeWilliams[3] In addition, the control of multiple robot systems and the selection of individualized robots cover many of the same issues addressed in our work. These include selecting individual robots within a multi-robot system,[4] commanding multiple robots through ROS,[5] and using natural language for distributed control.[6] Important to note is that our system is not a multi-robot system in the the robots do not necessarily (though could in a future iteration) have discrete awareness of one another. Commands, in their current form, do not involve coordination between robots within the system.

Additionally, there has been work done and investigation performed on ROS systems that allow for the easy implementations of systems that include these aspects.[7]

Implementation Details

Overview

The implementation utilizes a wake-up-word paradigm for the command trigger, a client/server model for the query transmission, and the publisher/subscriber model already available through the ROS architecture. The general flow is as follows:

Wake-up phase [Client-side]

The client, which can be runnning on any device and is not a ROS node, loops in a listening state constantly attempting to locally resolve recorded sound to a dynamic set of keywords. These keywords map to an (ip address, port number) pairing which is the address of the server.
Note: Keywords work best when they are 3 syllables or more. For this reason, we used "fetch the robot" rather than "fetch" when deciding on a keyword for demoing fetch functionality.

Wake-word resolution [Client-side]

Upon a successful match, the client resolves the keyword to a subset of robots using its map of known robots. This often results in one robot, as is the case when a specific name is used, but can also be a subset of the larger group of robots or the whole group included in the list of robots that the client possesses. For example, if a client has the following robot dictionary `"'baxter':('160.43.23.16', 8080), 'fetch the robot': ('160.43.23.24', 9000)` then, in response to a match on keyword `'everyone'` the client would say `"You are commanding` `the following robots: Baxter, and Fetch the Robot"` and proceed to send the following query identically to both servers.

Query resolution [**Client-side**]

Once the robots in focus are determined, the client enters another listening phase to receive the query from the user. This phase does not timeout and will continue to listen until it no longer detects speech. Once this occurs, it sends the sound data to the cloud for sound recognition. This text is then sent to the ip/port pair for each robot in the previously determined subset.
Note: This Utilizes UDP sockets which do not guarantee transmission. No client-side validation is performed.

Query resolution [**Server-side**]

On the ROS system exists a vocal_interaction node which acts as the server, constantly listening on the designated IP/Port for client queries. Upon receipt, the server sends the plain text to API.AI using its own client-key. This is where the resolution of natural language to structured data occurs. Using an established client-key which maps to an agent on API.AI, the query is resolved to a structured JSON format. This client-key structure allows for each server to have it's own agent which it queries if this is preferred. Different agents on the separate server would mean that identical (and potentially simultaneous) queries to two different servers from a single client would be understood with respect to that specific agents previously heard requests and established intents. Returned VerbalInput message is then published to the /verbal_request topic.

Mapping the resolved query [**Server-side - Publisher**]

The returned structured JSON is then mapped to a Spoken_Interation/VerbalInput.msg. Currently, this is an extremely lightweight ROS message:

```
[spoken_interaction/Verbal Request.msg]
string timestamp
SocketInfo clientInfo
string phrase
string action_id
KeyValue[] parameters
string user_id
```

The a `VerbalRequest` message is constructed from the API.AI response and published to the `/verbal_input` ROS topic.

Query to action [Server-side - Subscriber]

Any node that wants to allow for verbal interactions simply has to

1. Create a lightweight verbal interaction interface that
   - subscribes to the `verbal_input` topic
   - listens for requests that contain an `action_id` which the given node cares about
   - parses the parameters included in the request
   - maps the requests to functions that should be triggered on receipt
   - publish any responses to the `/verbal_response` ROS topic as `VerbalResponse` message which will be sent to the original requester
2. If implementing a new action, then the developer may need to create an intent for the given agent on API.AI and any new entities involved in the requests

Responses [Server-side - Publisher]

The publisher also subscribes to the `/verbal_response` ROS topic. Whenever a new message is public, it sends that message to the client at the port/id indicated in the message

Responses [Client-side]

When started, the client establishes a listening udp socket bound to user-specified address that it includes in the sent query messages. A separate thread is run which constantly listens for responses from the server and vocalizes these responses upon receipt.

Processing Textual Queries

We use API.AI to process text queries into a JSON data format that contains detailed information on the parameters, context, and intent of the query. API.AI lets you define Intents, Entities, as well as Contexts and follow-up conversations. We will go into Entities and Intents in more detail. API.AI offers a set of RESTful API endpoints for clients to use to perform queries on text, create Intents and Entities, as well as a Webhook service to integrate query requests with third-party tools.

Intents

API.AI lets you define Intents, which are a way of training a machine learning model on a set of text queries. The user can annotate parts of each query, specifying user-defined parameters such as location, time, object, etc. An example of a query could be ”Tell me what the weather is like today.” The user could define an Intent called ”information-request”, which specifies parameters ”request-type” and ”timeframe”. The user could then annotate the word ”weather” as the value for the parameter ”request-type” and the word ”today” as the value for the parameter ”timeframe”. In the resulting JSON that API.AI sends out, it would contain the parameter names and their values in a key-value map.

Entities

Entities are the objects that Intents act on. These parameter types previously specified, such as "request-type" and "timeframe", are actually something called Entities. They can, as already seen, be any kind of abstract concept. Furthermore, API.AI lets the user define synonyms for these entities. This lets developers easily add new objects and abstractions.

Speech to Text

In our proof of concept, we have two phases in which the user interacts with the robot - Activation and Querying. In these scenarios, the user must speak to the interface, which is just our computers running a Python client. These spoken interactions must be turned into text, since API.AI only interacts natively with text.

Activation is the default phase the client is in. The client is listening for keywords. We use the Pocketsphinx library, built by Carnegie Mellon University, for keyword recognition. When the client recognizes that a keyword has been said, it moves into the Querying phase. This set of keywords is defined by the application developer - for our purposes, it is a list of the names of the robot agents that the client can interact with. For example, saying "Fetch, the robot" will be recognized by our client as an attempt to wake up and start interacting with the robot whose identifier is Fetch. If the user said "Baxter", then the client would recognize that we are trying to interact with Baxter, and adjust the context accordingly.

This context is used in the Querying phase to determine which robot to send the query to. In the Querying phase, the client listens for users to speak out complete commands. Depending on which robot was activated in the Activation phase, this query will then be parsed into text, sent to API.AI, and the result from API.AI will be sent to the vocal interface node over a Websocket. An example of a command is "Can you go to coordinates 1, 5?"

Conclusion

We were able to wake up our robot, perform voice commands to move, save a location as a landmark, and make the robot travel back and forth between various landmarks. This was done on the backbone of our vocal interface nodes and topics, as well as our API.AI account. The landmark functionality itself was contained purely within the landmark resolution node that we built. Everything else, from the client to the vocal interface nodes to the web server node, was in accordance with our vision of a platform that would enable rapid and pain-free application development.

For example, to add functionality for grasping, a developer would need to do two things.

1. Add the Intents and Entities necessary to turn a natural language text query into actionable data. If a query takes the form ”Please grab the object in front of you”, we could define three Entities. First is the `action` - ”grab”. Second could be the `target`, which, in this case, is simply ”object”. Last could be `orientation`, or `relative-position`, which would be the ”in front of you” part. After giving API.AI a few training examples for the Intents and defining synonyms for these various Entities, the developer could move on to the second part, which is building the ROS node.

2. Build to ROS node that handles the actionable data JSON that API.AI creates. This would be a ROS node that reads from the vocal request topic that our vocal interface node publishes to. If it receives an action that it is compatible with - perhaps they are identified by a key called `action-type` - then it will process that action and interact with the necessary nodes responsible for grasping and visualization.

In this way, we remove the need for the developer to learn all of NLP and handling Websockets and speech-to-text - they can concentrate solely on their specialty and area of interest. To move further towards this ideal, we would like to improve the robustness of the client right now. Ours is a Python interface - however, future applications will likely be built on smartphones or personal home devices such as Amazon’s Alexa, rather than our computers. If we could provide an API or some form of ready-made solution for building a client on the most important platforms (iOS and Android, for example), we could further simplify the lives of developers who want to build on ROS.

References

(1) Byoung-Kyun Shim, K.-w. K. *IEEE Std. 1516-2000* **2012**,

(2) Laengle, T.; Lueth, T. C.; Stopp, E.; Herzog, G.; Kamstrup, G. *Intelligent Autonomous Systems: IAS-4* **1995**,

(3) Thomason, J.; Zhang, S.; Mooney, R.; Stone, P. Learning to Interpret Natural Language Commands Through Human-robot Dialog. Proceedings of the 24th International Conference on Artificial Intelligence. 2015; pp 1923–1929.

(4) Alex Couture-Beil, R. T. V.; Mori, G. *Twenty-Ninth AAAI Conference on Artificial Intelligence*

(5) Remmersmann, T.; Tiderko, A.; Langerwisch, M. *Communications and Information Systems Conference (MCC), 2012 Military* **2012**,

(6) Remmersmann, T.; Schade, U.; Schlick, C. M. *Systems, Man and Cybernetics (SMC), 2014 IEEE International Conference* **2014**,

(7) Andre, T.; Neuhold, D.; Bettstetter, C. *Globecom Workshops (GC Wkshps), 2014* **2015**,