ABSTRACTIONS AND SECURITY CONCEPTS FOR A ROBOT OPERATING SYSTEM

BY

MURPH RICHARD MATTHEW FINNICUM

THESIS

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science in the Graduate College of the University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Associate Professor Sam King
As general purpose robots begin to find their way into the household and workplace, there will be a demand for software to run on these robots. My research group forsees the proliferation of robot apps that use a common set of abstractions to allow them to function on a variety of hardware platforms. In this thesis, I introduce a robot operating system to support these apps and detail the abstractions that it provides. I present many lessons learned from developing and debugging a number of such apps, and discuss a novel concept wherein apps and libraries are allowed to seek help from outside sources when they are unable to accomplish their goals. I show that our framework allows a robot to effectively deal with challenges, such as user authentication. I demonstrate a simple bartender app to fetch drink orders for students, and it is successfully able to deliver them in 10/10 trials in real-world conditions. I also present CLASS, a new system capable of identifying users robustly. I propose a framework for integrating a wide range of sensor values into an algorithm for identifying users, even if an attacker actively tries to impersonate a user. Our system is evaluated by using our reference robot and this platform to build a robot application that buys coffee at our local coffee shop for a user, without requiring explicit authentication. I evaluate CLASS under an adversarial model experimentally and find it to be robust and resilient to attack.
To Sam King, for sending me this email:

Hi Murph,

Are you interested at all in grad school? If so, I would like to discuss it with you. If not, I would like the chance to convince you otherwise.

Cheers, Sam
I’d like to thank my girlfriend, Renee, for all the support she has given me and for moving to Champaign to keep me company while I was a grad student. I’m sure there were more interesting places to live, sorry!

I also owe my sanity to our two dogs, Tessy and Totoro. It’s tough to have dogs while in school, but it’s worth it.

I’d also like to thank Corbin Soufrant for all his hard work as my undergraduate research assistant.

A shout-out to the ##uiuc IRC channel for keeping life entertaining, and to /r/uiuc on Reddit for keeping me connected.

Thanks to my roommate, Kevin Larson, for never minding my occasionally chaotic grad student life.

And, of course, I’d like to thank my parents for making college possible.
# TABLE OF CONTENTS

CHAPTER 1  INTRODUCTION ............................................. 1

CHAPTER 2  FRAMEWORK FOR ROBOT APPLICATIONS .............. 4  
  2.1  Design principles ............................................. 4  
  2.2  Hardware architecture ....................................... 4  
  2.3  Software architecture ........................................ 5  
  2.4  Robot capabilities ............................................. 7  

CHAPTER 3  THE HELP ABSTRACTION .............................. 10  
  3.1  Throwing and handling help exceptions ..................... 11  
  3.2  Libraries for handling help exceptions ..................... 13  

CHAPTER 4  THE OBJECT ABSTRACTION .......................... 15  

CHAPTER 5  THE HARDWARE ABSTRACTION ....................... 16  

CHAPTER 6  THE INTERACTION ABSTRACTION .................... 17  

CHAPTER 7  SECURITY CONCEPTS .................................. 20  
  7.1  Threat model and assumptions ............................... 20  
  7.2  Identifying users securely ................................... 20  
  7.3  Audit log ..................................................... 28  

CHAPTER 8  EVALUATION OF EXPERIMENTAL SYSTEMS .......... 30  
  8.1  Bartender app case study ................................... 30  
  8.2  Design iterations ............................................. 32  
  8.3  Methodology .................................................. 33  
  8.4  Purchasing coffee ............................................. 34  
  8.5  User influence on factor effectiveness ..................... 36  
  8.6  Adversarial analysis ......................................... 37  

CHAPTER 9  RELATED WORKS .......................................... 38  

CHAPTER 10  CONCLUSIONS ........................................... 40  

REFERENCES ............................................................. 42
CHAPTER 1
INTRODUCTION

Robots are an increasingly important part of society. Industrial robots have been a staple of manufacturing facilities for decades, and in recent years, military robots have seen increasing use. For example, in 2008 the U.S. Air Force had twice as many robotic planes as manned planes [1]. People use service robots for cleaning floors and carpets [2], mowing lawns [3], and driving cars [4]. Household robots are used as pets [5], and in nursing homes and for children’s rehabilitation [6]. Evidence shows that people bond with all types of robots and accept service and household robots as a part of their families [7, 8], suggesting a continued increase of robot use in the workplace and in the home.

Although robots are purpose-built typically, we view robots as general purpose computing devices that should be capable of running robot apps. By running robot apps we do not mean running Emacs and gcc, but rather robots should include a simple and general-purpose operating system (OS) for controlling the robot itself. Our goal is to make robots as easy to program as mobile phones, support multiple robot apps at the same time, and to free robot app developers from having to know intimate details about the hardware and software configuration of the robot. Although many of the same systems techniques we use on more traditional computing environments apply to robots as well (e.g., multiplexing devices), robots are fundamentally different than traditional computer systems in interesting and novel ways.

Robot apps have to interact with the world and people around them without the benefit of a well-defined interface. Mobile robot algorithms are fundamentally probabilistic [9, 10, 11], complicating tasks like identifying and authenticating the people that interact with robots. Application logic that could be simple instead becomes cluttered and full of control flow to deal with this uncertainty. Variations in robot hardware or the capability of platforms also complicates applications. Robot apps operate at a higher level of
abstraction than general operating systems. Traditional applications operate on processes, files, sockets, and users, whereas robots apps will operate on robots, people, walls, maps, locations, paths, and so on. Choosing the correct abstractions will help robot apps interact with the real world.

In this paper, we describe Isaac, a robot we built, and Frak\textsuperscript{1}, an OS we built for simplifying app development on robots and for managing robot hardware. Our key insight is that some tasks are too hard for apps to handle without exceedingly complex software, so we introduce a “get help” abstraction to handle unanticipated situations. The main mechanism in the “get help” abstraction is a help exception. The Frak system throws help exceptions on failed API calls, and the app writer can throw help exceptions explicitly when it enters an unanticipated state. Frak tries to handle these exceptions in a unreliable, but accurate, library first. If none of the libraries can handle the exception, Frak notifies the user, who can take control of the robot and help it proceed manually. By first using unreliable, but accurate, libraries and then falling back to humans to solve complex tasks, we simplify app software while still being able to do interesting and sophisticated tasks.

Our Frak OS includes abstractions for enabling app writers to write programs that operate on robot abstractions \textit{without} having to know the exact software and hardware configuration of the underlying robot system. We have a general object abstraction for identifying people, places, and things, and we built user and location abstractions on top of our object abstraction. Frak also has abstractions for generating user interfaces that enable apps to interact with users. These abstractions are independent of the underlying hardware configuration and the Frak system adapts the software automatically using whatever resources it has available.

To test Frak, we built 14 apps that we use on Isaac, and we deployed Isaac in our office for real use. Our most comprehensive app is a bartender app, which is an app for serving drinks during social events. In the bartender app, Isaac travels to the student lounge, takes drink orders, brings the drink orders back to the bartender, and then delivers the drinks to the person who ordered them originally. Other apps include a “give our advisor a message” app that runs in the background and tells a person something if the robot sees that person in the hallway, and an app that delivers travel receipts to a

\textsuperscript{1}Framework for Robot Applications, K?
secretary and asks him or her (nicely) to submit them for reimbursement.

Our contributions are:

- We introduce the novel help abstraction for dealing with uncertainty in robot applications.
- We designed and implemented a novel architecture for running robot apps, including multiple apps concurrently.
- We built Frak and the associated abstractions, and a robot, Isaac, for running experiments. We also built CLASS which provides engineers and developers with secure, easy-to-use mechanisms for designing robot applications that interact with people.
- We wrote a number of robot apps using CLASS and evaluated them in real world conditions. This includes our main test app where our robot buys and fetches a cup of coffee from our local coffee shop. We evaluate these test apps experimentally.
- We define a novel algorithm for enabling robots to identify people, and we introduce the first design principles and trade offs for choosing features that robots can use to identify people, even in an adversarial setting.
- We design and implement an audit logging system for our robot that enables system builders to evaluate past security related decisions to assess and debug any wrong moves.
CHAPTER 2
FRAMEWORK FOR ROBOT APPLICATIONS

This paper presents Frak, a system that provides writers of robot applications the ability to implement apps. Our primary goals are to make it easy to write robot apps without exposing the underlying robot hardware or libraries that drive the platform, while enabling multiple apps to run concurrently.

In this section, we discuss the design principles that guide our design, our hardware architecture, and our software architecture.

2.1 Design principles

Our design is guided by the following principles:

1. *Simple apps.* Robot apps should be easy to write.

2. *Hardware independence.* App logic should not be concerned with implementation details and hardware specifics.

3. *Future proofing.* Apps should automatically benefit from advances in robot technology that occur.

2.2 Hardware architecture

When developing our hardware architecture our goal is to use as many commodity components as possible. By using well-supported commodity hardware, we hope to keep the cost low and to make programming easier.

To build Isaac, we use an iRobot Create, which is like a Roomba without the vacuum and with a programmable interface, a netbook running Linux to
handle most of our computation, and a Kinect for video and depth sensing (Figure 2.1).

2.3 Software architecture

For our software architecture, we use a robot runtime system called ROS [12, 13]. The basic ROS architecture resembles a microkernel architecture where there is a thin software layer that is responsible for passing messages between different processes (or ROS Nodes) that implement key functionality (Figure 2.2). The ROS kernel exports a publish/subscribe interface to facilitate communication between nodes. ROS also includes functionality for basic robot abstractions, such as mapping, localization (where the robot is
located currently on the map), path planning to determine how to navigate the robot in the map, and robot locomotion.

Frak runs on top of ROS and we split Frak into two different layers: the library layer and the application layer. In the library layer, libraries use ROS functionality to implement higher-level functionality. Frak includes libraries for managing applications, users, locations, objects, and user interfaces (UIs). Frak also includes extensibility APIs for enabling application developers to write their own libraries. At the application layer, applications use the abstractions exported by the libraries.

In addition to the abstraction layers Frak implements, we also included comprehensive audit logging capabilities throughout ROS and Frak. This
audit log infrastructure records enough information at runtime to enable system builders to recreate past states and events to help debug the system. Our audit logging infrastructure takes advantage of the modular nature of our architecture.

2.4 Robot capabilities

In this section, we will detail the capabilities of our robot as implemented. We aimed to implement and use well tested algorithms. We have the understanding that other research will improve the capabilities of our robot with time.

Isaac can navigate around the halls of our building with relative ease. Most of the furniture inside of rooms is currently not on our maps. Isaac can avoid such obstacles, but they do hinder it from accurately keeping track of its location. Due to the inaccurate wheel odometry in our robot, we had to add a gyro to aid in measuring rotation. Passerbys in the hallway can cause the robot to move more cautiously (it spins around a lot to verify its position), causing slowdown. We did not attempt SLAM (Simultaneous Location and Mapping) as it was an unnecessary complication.

Using a relatively standard Haar classifier from OpenCV, Isaac is very good at detecting faces. We discovered this algorithm was prone to false positives, and were able to significantly reduce them by using the 3D information provided by our Kinect. We rule out false positives that are too large or small. We also check the geometry of the detected face to help rule out flat objects like posters and photographs.

Isaac can speak in a suitably robotic generated voice, and is surprisingly good at listening to spoken commands thanks to Google’s Automatic Speech Recognition technology. Unfortunately, due to limitations in this technology, it is unable to hear profanity.

2.4.1 Software

We implemented a GUI in QT that allows a user sitting at a PC to provide responses to help exceptions. It has a dialog for identifying people where the user is presented with a snapshot of the person and presented with options
about who they are. It also has a dialog for finding the robot that allows for full remote control, has a video feed, and displays a map so the user can locate the robot. This GUI integrates with ROS and communicates with the robot via ROS messaging and service calls.

To allow users with heterogeneous devices to interact with the robot, we developed a web application interface. Frak provides a number of widgets that applications can directly use to interact with the user - such as a Dialog for prompting questions, or AlertBoxes for notifying users. Application developers are also given an API for defining plugins that can be registered with the Frak web server. Plugins are written in Python, and can either use existing HTML templates, or define custom ones. The remote user callbacks are delivered to the running applications on the robot by the underlying communication layer provided by ROS.

In addition to the apps discussed, we have implemented a number of useful apps and test apps. We have a voice-based launcher that listens for the names of our other apps and launches them, a ‘copycat’ audio mimic app to test speech input/output (Though it usually ends up copying itself in a loop after a while), and a number of simple navigation / robot control apps (drive in circles, attempt to “parallel park” the robot, etc.

In addition to our improved face detection heuristics, we have a number of nodes for identifying users based on other characteristics. We can identify and remember shirt color and user height (when standing).

We use a MySQL database to store all of our objects and allow for persistence. If we update user information during one run (noting a new recent location, for example), or record a location, that information is saved and made available to all the apps. This way, Isaac has learned many people/locations in our building without us having to manually create a list.

We have a very thorough logging infrastructure built in to our system. We log all messages published to all topics, all ROS service calls that are made, and all results returned by them. This required a few changes to the underlying ROS architecture, but gives us a great deal of flexibility in debugging our software.
2.4.2 Performance

Isaac uses an ASUS 1215N netbook with an 1.8ghz Intel Atom D525 CPU. This is a dual core model, and during operation it is 90% utilized. The largest consumers of processing power are the Kinect video processing node (using 52%) and the localization and navigation routines (using 31%).

We ran the Kinect off of the iRobot Create’s power supply. As a whole, Isaac could operate for 68 minutes without requiring recharging (longer if it wasn’t driving around much).

It took the robot on average 133 seconds to travel from the lounge to the bar, a distance of about 112 feet (about .25 m/s).

Due largely to all the processing we did with the video stream, we averaged 23.4MB/s of bandwidth usage between ROS nodes during execution. These nodes were all on the same machine, however. In order to pipe the video feed to the remote GUI, we had to encode it with Ogg Theora and downsample the resolution significantly which reduced it to 70KB/s of bandwidth usage.
CHAPTER 3

THE HELP ABSTRACTION

One difficulty in programming robots is that they operate in unknown environments. Some algorithms are designed specifically to deal with this type of noise and work quite well, like localization algorithms that enable a robot to determine where it is located within a known map. Other algorithms become overwhelmed by the noise and have difficulty accomplishing their specified task, like identifying a person in a crowded room. In addition, robots can be limited by their hardware and might be unable to accomplish common tasks. For example, a short robot will be unable to press the down button on an elevator or to open a closed door.

One way to cope with these types of unknown environments is to write software that is more complex. However, enumerating through all possible scenarios a robot might encounter is difficult, and even if the app writer is able to anticipate all possible changes to the environment, some robots will still be unable to carry out some tasks.

In this section, we introduce the help abstraction to make dealing with uncertainty in robot apps easier by falling back to unreliable, but accurate, libraries, and falling back to users in difficult situations. The help abstraction includes exceptions that Frak throws when it encounters a failed API call, and exceptions that the app writer can throw explicitly when the app enters an unanticipated state. Exceptions that Frak throws can be caught by a library or by the app. Uncaught Frak exceptions and exceptions that the app writer throws explicitly are passed to a person who can assist the robot and help it make progress manually.

In our current implementation, Frak has four types of help exceptions. First, Frak can throw a `user_not_found` exception that signifies that the user manager was unable to locate a user. Second, Frak can throw a `robot_lost` exception that signifies that the robot is uncertain about where it is located currently. Third, Frak can throw a `robot_stuck` exception that signifies that
the robot cannot continue to follow its current path. Fourth, the app writer can throw an app help exception that is specific to an app and goes directly to a human user. This app-specific exception includes information specified by the programmer to convey to the user what type of help the app needs.

3.1 Throwing and handling help exceptions

When Frak throws a help exception, libraries have the first opportunity to handle the exception. In Frak, all libraries can register for notification of help exceptions and they can try to handle them before notifying the app. The libraries that try to handle the exception pass back information about the specific exception. This return value is a specific person in an image for the user not found exception, a location on the map for the robot lost exception, or a new path plan for a robot stuck exception. Library replies also include a confidence metric, which indicates to Frak how likely the library thinks its answer is. Frak uses this confidence metric to decide which solution to use.

If no libraries handle the exception, then Frak passes it along to the app and then the user. App developers can handle exceptions using traditional exception handling mechanisms and can react appropriately for the specific application. If the app does not handle the exception, Frak notifies the user and gives them manual control of the robot. The user can then navigate the
robot using their keyboard, pick a user, set a new path, or do whatever they want using the manual control mechanisms available in our system. Figure 3.1 shows the UI screen that the robot displays in response to robot_lost exceptions.

Figure 3.2 shows an example of how Frak handles a user_not_found exception. In this example, (1) the app issues a drive_to command to tell Isaac to drive to Bob. Then, (2) Isaac cannot find Bob and throws a user_not_found exception. Frak (3) broadcasts the user_not_found exception, and when none of the libraries handles it, (4) sends the exception to the user. The user is given a picture of Bob and a live video stream from Isaac, and the user is asked to identify Bob manually. At this point, the user has full control of Isaac and can move him around, turn on the speakers to talk, or do anything else he or she wants to try to find Bob.
3.2 Libraries for handling help exceptions

In general, we use the most reliable libraries for handling our basic functionality, and unreliable, but accurate, libraries for handling exceptions. During normal operation, we identify users using a facial recognition algorithm from the OpenCV vision library, and for localization, we use the state-of-the-art Monte Carlo localization algorithm [11] with hand-tuned parameters specific to our particular hardware and configuration. These algorithms are always available when the robot is running and are the primary way that Isaac identifies users and determines where he is located at any given time.

Figure 3.3 shows the libraries that we built to handle help exceptions. Our user_not_found library tries to detect people based off their height and the color of their shirt. For above average height men wearing gray, this library will not work well, but if we are looking for an abnormally tall person who is wearing pink, it might work. Our robot_lost library tries to determine location based on performing optical character recognition on room numbers, which only works when the robot can see a room number. Our robot_stuck library tries to knock on a closed door, which is great if someone is in a closed office, but is of little help for entering an elevator. None of these libraries would work well as a first-class library because there are many scenarios when they are unable to provide meaningful information. However, there are situations where these libraries could provide accurate information, enabling Frak to avoid passing help exceptions up to users.

When a library handles a help exception, Frak gives the library full control of the robot. This level of control gives the library the ability to move or actuate the robot to try to gain more information or accomplish the task, or

<table>
<thead>
<tr>
<th>Help topic</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_not_found</td>
<td>Identify people based on height and shirt color</td>
</tr>
<tr>
<td>robot_lost</td>
<td>Look for known landmarks (e.g., room numbers)</td>
</tr>
<tr>
<td>robot_stuck</td>
<td>Knock on door by running into it a few times</td>
</tr>
</tbody>
</table>

Figure 3.3: Frak libraries for handling help exceptions.
to gather additional data from sensors. Although this level of control provides a rich execution environment for libraries, it does present a potential safety liability if a library becomes malicious.
CHAPTER 4

THE OBJECT ABSTRACTION

In Frak, we have a general notion of an object, which Frak uses to represent nouns (people, places, and things). Apps tag objects with application-specific information and in our current implementation we focus on two key objects: users and locations. Users are the people who interact with the robot and locations are the places that it might visit. These abstractions are used both by apps and by libraries that implement the services that apps use. As we continue to develop more apps, we expect to introduce a wider range of objects.

Objects in Frak are global to all apps, and apps share labels and semantic information for objects. For example, if one app labels a location, then other apps can refer to that location using the same label. This type of sharing should help users build up descriptions of objects quickly.

Users and locations. Users represent the people that the robot interacts with. Users can be ephemeral or permanent, and they can be named or anonymous. Locations are defined by coordinates in a map. Each time an app encounters a user, Frak keeps track of the location of the user. This information enables apps to do things like identify a user, fetch a drink for that user, and bring the drink back to them. This information also enables apps to identify a user opportunistically and to deliver them a message. By associating locations with users, apps can navigate the robot based on this information by specifying commands like “navigate to Bob based on his most recent location” or “navigate to Bob based on where you usually see him at this time of day.”
In Frak, we model robot hardware in two different categories: sensors and actuators. Sensors include odometry readings, audio and video streams, gyroscopes, and any other devices the robot might use to observe the environment. Handling sensors in Frak is straightforward because they are read-only effectively, thus any libraries and apps that wish to access sensor data are allowed to.

Actuators are devices that the robot uses to move or otherwise manipulate the environment. Because apps might have vastly different uses for actuators, fine-grained multiplexing does not make sense for a robot. Instead, Frak implements a basic cooperative scheduling policy for these resources. Libraries and apps can grab control of an actuator using an interruptible lock that allows other libraries and apps to control the resource if they request it, or using an exclusive lock that gives the library or app uninterruptible control over the resource. Requests made to a locked actuator are queued until the node that holds the lock puts it back into the interruptible mode or releases it.
CHAPTER 6

THE INTERACTION ABSTRACTION

The human-machine interaction in contemporary computer systems – such as PCs and smartphones, has been well studied with programming interface and abstractions clearly defined at this layer. Unfortunately, in the robotics domain the human-robot interaction has not been modeled as well. The primary challenge in designing a UI for Frak is that the I/O capabilities are dictated by the underlying robot hardware and non-uniform across different robot systems. Robot apps can interact with people in any number of ways, including through a network-connected computer, audio I/O, video I/O, infrared light, or any other I/O technology, depending on the hardware present on the robot.

To compensate for diverse hardware configurations, Frak encapsulates the underlying hardware specifics of the robot by providing a high-level API for interacting with the user. Frak supports `speak`, `listen`, `ask`, `user.speak`, `user.listen` and `user.ask` where `speak` calls output text, and `listen` calls wait for a user to input text, and `ask` calls ask a question and wait for a reply. The `ask` function also takes an optional array of choices if the app asks a multiple-choice question. The `user` versions of these calls direct the UI to a specific user and the anonymous calls act independent of users. Frak maps these calls to the appropriate hardware for the robot it is running on. Frak also includes a `user.robot.status` call that displays a map, marking where the robot currently is on the map, and an application-specific status message to let a user know the status of the app. For example, after a user orders a drink, the bartender app uses the `user.robot.status` UI to let users know where the robot is and the status of their drink order.

On Isaac, Frak uses three different types of user interfaces. First, for apps that interact with users via a network-connected computer, like a mobile phone, Frak uses an embedded web server and a web app to interact with the user. Isaac has a QR code (2D bar code) attached to him that users
can use to get a URL for the web interface. Frak’s web interface utilizes a micro-framework named Flask and the jQuery Mobile-based UI to interact with the user. Each user has a mailbox that corresponds to her communication channel with Isaac. The user is presented with a clean interface that prompts them for input, or lets them know of the status of any outstanding tasks. Second, Frak uses the speakers and a text-to-speech library to communicate with users over conventional audio interface. To convert spoken words to utterances we have developed a library that utilizes Google speech-to-text online API. Third, Frak includes a command-line interface with additional GUI programs for power users that prefer to ssh into Isaac.
Figure 6.1: Our web-UI, prompting the user to select a drink.
CHAPTER 7
SECURITY CONCEPTS

7.1 Threat model and assumptions

This paper addresses the issue of providing security mechanisms for applications that run on mobile robots. We focus on developing mechanisms that enable robots to identify people robustly, and providing an audit log for forensic analysis.

We trust the layers upon which the CLASS software is built. This trust includes the OS and libraries used to control the robot, and the robot hardware itself. Fortifying the hardware and software layers upon which the robot system is built is an important and interesting topic, but it is a complementary issue to designing algorithms and systems for providing security services for robot applications.

CLASS is designed to operate under malicious influence. We consider attacks that originate from an attacker interacting with the robot in a way that tries to impersonate another user.

7.2 Identifying users securely

Overall, CLASS uses sensor readings from as many different sensors as possible to provide a more broad and redundant view of the physical properties of the environment. Then based on these sensor readings, CLASS derives a wide range of features that describe the people that it detects, such as their location, shirt color, or height. Using these features CLASS develops metrics for positively identifying users (i.e., is this a particular user) and metrics for negatively identifying users (i.e., is this not a particular user). With these basic metrics CLASS combines them in a reasonable way to assess its
confidence in user identification given the current set of sensor readings.

The fundamental challenges in this style of user identification are picking meaningful features, combining them in a way that can withstand some level of attack, and deciding when to fall back to a more explicit authentication technique when needed.

In this section we discuss the CLASS system for identifying users. We first discuss the CLASS API (Section 7.2.1), and the trade offs one can make when selecting features to use for identification in general (Section 7.2.2). Next we describe the specific features we use for our robot (Section 7.2.3). Then we introduce our algorithm for using these features to identify users automatically (Section 7.2.4).

7.2.1 CLASS API

From a high level, the CLASS API enables applications to compare a detected person to an earlier reading or to determine if there is a known user who matches the person it has detected. The CLASS node reads in robot specific data to extract features of people when they are detected and notifies the application when the robot encounters a person. The application can then query the CLASS identification node to determine if this person is the same person that the robot encountered at some point in the past.

The key of our API is that the application does not need to know any of the platform specific attributes of the robot and it does not need to know which features the CLASS node uses to match people. The application does have to remember key people (e.g., a user who orders a cup of coffee) so it can use this later to determine if the robot encounters the same person in the future.

When the application queries the CLASS node about identification, it can ask the CLASS node to return the most likely person and an associated confidence level for this match, or it can ask the CLASS node to compare two people and return a confidence level of the match. The application can interpret this confidence level however it sees fit for the particular application.

In general, the application controls the movements and hardware on the robot, but if the application wants to try to gain higher confidence about two people matching, it can delegate some of its control capabilities to the
Figure 7.1: Overview of how CLASS interacts with applications and other modules on the system. Oval nodes represent event handling nodes and square nodes represent service nodes that answer queries.

CLASS node. The CLASS node can then interact with the robot or the user to try to gain higher confidence. For example, to gain higher confidence the application can delegate control of the speakers to the CLASS node and the CLASS node can ask the person to turn on their cell phone to try to obtain a Bluetooth signal that is known to be associated with a specific person.

Figure 7.1 show an overview how the CLASS node interacts with the system and with applications. As the system runs, (1) the Kinect driver will pass images and depth sensor readings to a people detection algorithm, (2) which will notify the CLASS node each time it detects a person. When this happens, it will pass the image, depth sensor readings, and face location information to the CLASS node. When the CLASS node receives this notification, (3) it queries a set of features to try to detect information about the person that it found, (4) and then notifies the application that it has detected a person. The CLASS node will include all of the features in this message, but the application can treat these as an opaque data structure without having to understand the details of individual features or even what
features the CLASS node uses. After the application receives notification of a person, (5) it can query the CLASS node to see if it has a match or (6) ask the CLASS node to try to gain more confidence if needed.

As the robot runs it maintains a history of features seen in a particular environment. Using this history, the robot updates its calculations for comparing features based on the types of features that it observes. History affects each feature differently and it is up to the discretion of the robot designer to determine how to include the history in the calculations.

7.2.2 Selecting features

Our goal is to select a set of features that are suitable for identifying users automatically. CLASS also must be able to know how confident it is about a particular measurement so it can fall back to explicit authentication if it is unable to identify a person with high confidence.

When selecting features, we consider the following properties of the features when deciding whether to include them in our identification algorithm:

- Ability to identify someone positively.
- Ability to identify someone negatively.
- Difficulty of spoofing by an adversary.

We took some inspiration from the way that people might identify each other. For example, if we see someone sitting at a desk, we might assume (with low confidence) that it is the desk’s owner until we have some better indication. Likewise, we have had success using location as one of the features that CLASS considers.

In general having more features makes the identification algorithm more robust against attacks on individual features, and system builders should strive to have multiple features that perform well for each of the properties. For example, having only features that can identify people positively and negatively but are spoofed easily will be problematic in an adversarial setting. Likewise, having only features that are robust against spoofing but are unable to identify people positively will be hard to attack but unable to identify people, which is the whole point of this mechanism.
Identify positively | Identify negatively | Hard to spoof
--- | --- | ---
Location | ✓ | ✓ | +
Shirt color | ✓ | + | −
Height | − | + | ✓−
Bluetooth | + | ✓ | ✓

Figure 7.2: Properties of the features we use for identifying people. This figure shows how well each of the features we use performs for the properties related to identifying users. Higher ratings mean that a feature performs better for a given property.

7.2.3 Our current features

The features that system builders use will depend heavily on the platform they are using and the hardware and software capabilities available to them. This section discusses the features we use in our system. These features are not meant to be a definitive list of features for identifying users, but rather to serve as an example of the features we found useful for our platform and application.

The sensors available on our robot platform include wheel velocity commands (which roboticists model as sensors), robot odometry measurements (based on optical encoders that measure wheel rotation), and proximity sensors. We use a Kinect on our robot, so our platform also senses video streams and can associate distances from the robot with each pixel in the video using an infrared depth sensor array.

As a base feature our system must detect people. To detect people we use the face detection algorithm available in the OpenCV library [14]. By using the depth information from the Kinect, we significantly reduce the likelihood of false positives. This algorithm does not detect individual faces, but rather finds faces present in an image.

Using our sensors, we calculate four key features that we use for identifying users: location, shirt color, height, and Bluetooth signal strength. Figure 7.2 summarizes how well these features perform on each of the key properties we describe in Section 7.2.2.
Location

The first feature we discuss is the person’s location. The key aspect of determining a person’s location is determining the robot’s location and measuring where a person is relative to the robot using the depth sensor.

To determine our robot’s location we use the Monte Carlo Localization (MCL) algorithm [11]. In robotics, localization refers to the calculation of where a robot is located currently. Localization combines wheel velocity commands with odometry measurements to develop a probabilistic hypothesis of how far the robot has moved since the last known location. However, this estimation tends to be noisy for mobile robots, so the MCL algorithm uses depth sensor readings to gain more confidence as to where it currently is. As the depth sensor detects objects (usually walls) the MCL algorithm can gain more confidence as to where a robot is located within a building, assuming that the map of the building is known ahead of time. The MCL algorithm combines these sensor readings based on Bayesian calculations, and develops high confidence about the location. This basic algorithm is used widely in mobile robots and has been used successfully to perform localization on a tour-guide robot in the Smithsonian museum [10].

The reason we use the MCL algorithm for localization is because it maintains thousands of distinct hypotheses about the robot’s location concurrently. If the some of the hypotheses are incorrect, the algorithm will reject these hypotheses dynamically and weight more heavily higher confidence hypotheses, making this algorithm robust in noisy environments.

Location can be a suitable feature for positively and negatively identifying a person for applications where a user should be in a well-known place. For example, if a user orders a cup of coffee, they are likely to be in the same location when the robot returns with the coffee. However, for crowded spaces or an application where a user is moving around often location may not perform as well. Location is difficult to spoof because the MCL algorithm we use was designed specifically to adapt even in an adversarial setting (see the “Kidnapped Robot Problem”) where the robot can recover from large and undetected changes in location.
Shirt color

The second feature we discuss is the color of a person’s shirt. To calculate a person’s shirt color we calculate a 2D histogram of the hues and saturations of colors we find in a 10cm x 10cm square 30 cm below the center of their face.

If the robot has already seen a person and they are trying to find them again, their shirt color is a moderately accurate feature for positively identifying people. Of course many people wear the same color shirts, so this metric is not perfect, but it can help identify people to a limited degree. However, if a robot is trying to find the same person again and they have a different color shirt, this can be a fairly good indication that this is not the person they are looking for. Unfortunately, shirt color is straightforward for attackers to spoof.

Height

The third feature we discuss is a person’s height. To calculate a person’s height we can measure the distance from the top of their face to the floor using the video stream and depth sensors.

Because people are average height on average, height is usually not distinctive. However, if a person is unusually tall (like both of the authors) or if their height differs drastically from previous readings, it can be used to provide some confidence about positive or negative identification. Height can be spoofed by a small amount fairly easily, but large changes in height can be more difficult to spoof inconspicuously (e.g., an attacker is unlikely to walk around with stilts and navigate through a building unnoticed).

Bluetooth signal strength

The fourth feature we discuss is the strength of a person’s cell phone Bluetooth signal. We assume that a user would be willing to pair their phone to the robot once so that the robot can detect the signal strength of the Bluetooth signal to determine if the person (or at least their cell phone) is near by.

Bluetooth signal strength can be a good indicator for positive identification
1 \( F = \text{getFeatureProabilities(PersonA, PersonB);} \)
2
3 // calculate aggregate positive identification prob.
4 total_p = 0;
5 foreach pos in F
6 \quad total_p += (pos*(1-total_p));
7
8 // calculate aggregate negative identification prob.
9 total_n = 0;
10 foreach neg in F
11 \quad total_n += (neg*(1-total_n));
12
13 // combine two metrics
14 if(total_n > total_p)
15 \quad return 0;
16 return total_p - total_n;

Figure 7.3: Algorithm for determining the confidence level of two people being the same person. Lower confidence levels indicate that two people are unlikely to be the same person and higher confidence levels indicate that two people are likely to be the same person.

because it shows that the person is likely to be close by. There are many reasons a person might not have a strong Bluetooth signal even if they are close by, such as turning off Bluetooth on their phone or leaving their phone behind, so it can be a poor feature for negative identification, but could still be suitable for some users (i.e., someone who leaves Bluetooth on and always has their phone on them). Because we pair Bluetooth devices, spoofing the Bluetooth ID is difficult, but an attacker could potentially steal one’s phone or use a signal repeater to boost the strength of a Bluetooth signal when the phone is far away.

7.2.4 The CLASS identification algorithm

Figure 7.3 shows the algorithm we use for calculating our confidence that two people are the same person. The CLASS identification algorithm is designed around a number of desired properties. It is capable of combining many different features together through a common formula, and generally the more well-written features that you have, the better your results will be.
Even with only features that return low-confidence results, a sufficient number of them can be combined to form a reasonably confident identification. Each factor returns both a positive identification confidence level and a negative identification confidence level. Again, it is important to remember that negative identification means that the compared people are not the same - it is not the same as being unsure. The algorithm is heavily biased towards non-identification. If there is sufficient indicators of negative identification, then it will never return high confidence of identification. This helps make it very resistant to attack: even if an adversary successfully fools all of our other factors, if he is inexplicably four inches too short we will not return a confident positive identification.

7.3 Audit log

Due to the inherently probabilistic nature of many algorithms in robotics, mistakes are inevitable. Even when CLASS thinks it has a 95% confidence in its results, it is not perfect. Thus, it is important that CLASS be able to both learn from its mistakes and store enough information to understand what happened.

7.3.1 Design

The majority of the features that we have implemented or considered rely heavily on the historical probability of events occurring. They need to ask questions like “how often does this person change their clothing in the middle of the day?” or “how often is the height of a person reported incorrectly by more than two inches?” If we restrict ourselves to storing the sensor information that was used by the various factors (for example, an image of every time a person was detected - but not the full video stream), a full year’s data would take up less than 2 TB of space. For security, the audit log should be stored remotely anyways, and this much space is not going to be a problem.

Additionally, it is important to store as much data as possible about recent events. To this end, we record sufficient information to recreate the entire state of our system for the past 48 hours. This includes all the sen-
sensor messages broadcast by the sensors, the actual inputs (including full video streams) into the sensors, the decisions that the application made, etc. Based on some simple experiments, we determine that this detailed audit log would occupy less than 1 TB of space. Again, we do not see storing this data to be a significant hardship.

7.3.2 Implementation

The implementation of this audit log was very easy. The robot runtime system that we choose, ROS, contains full record/replay functionality for its inter-node communication. We simply setup ‘rosbag’ to store all published messages for the last 48 hours, allowing us to replay and examine the state of the system at any time.

Additionally, we choose to create a database of sensor readings and results that would be used by the factors, such that we could easily aggregate important information and store it for a longer period of time. This database provides the data to properly calibrate and weight the many inputs.

7.3.3 Retrospection

Using the audit log, earlier decisions can be automatically replayed using the information that the robot did not know until afterward. This allows for an application to double check the decisions that it made earlier in the day and alert the user if any potential mistakes have occurred.
8.1 Bartender app case study

Our bartender app breaks the task of serving drinks down into four main steps: taking orders, returning to the bar, acquiring the correct drinks, and giving the orders back to the correct people. Figure 8.1 shows the source code for our bartender app. To evaluate this app, we ran it 10 times where we had one of the author’s group members in a lounge and another member serving as the bartender. We ran these experiments during normal business hours with people walking around Isaac as he ran the app.

To take drink orders, Isaac looks up the location of the lounge in its database of locations and drives there. In 9/10 runs, it drives there on its own without issue, but during one it got lost and had to issue a get_help call.

Once it reaches the lounge, it uses face detection to find a new user. Seven out of ten times it successfully finds a person, but the rest are false positives in the face detection module. get_help is again used, this time to verify the identity of the user. If the user is known to have a smartphone, Frak can use the web user interface to ask them their drink order. Otherwise, it has to communicate with the speaker and speech recognition nodes. These steps are very reliable.

Returning to the bar was accomplished on every run without issue. Isaac then reads the drink orders to the bartender, waits, and drives back to the users. In our trials, we had the users remain in the same area as they were when previously found, so Isaac was able to be locate them easily.
class Bartender(frak.App):
    orders = {}
    users = None

    def on_start(self):
        # Let's assume the robot starts lost
        get_help(robot_lost)
        self.bartender()

    def bartender(self):
        while not rospy.is_shutdown():
            take_order()
            return_to_bar()
            acquire_drinks()
            give_order()

    def take_order():
        # Drive to the student lounge
        navigation.get_location_by_name("lounge").drive_to()

        bob = user_manager.find_new_user()
        order = bob.ask("What would you like to drink?",
                        ["Sprite", "Coke"])
        orders[bob.name] = order.result()
        bob.robot_status()

    def return_to_bar():
        bar = navigation.get_location_by_name("bar")
        bar.drive_to()

    def acquire_drinks(orders):
        speak("Hello, bartender. I need these drinks.")
        for order in orders:
            speak(order)
        listen()
        speak("Thanks.")

    def give_order():
        for username, order in orders.iteritems():
            user = user_manager.get_user_by_name(username)
            user.drive_to()

            user.speak("Hello %s, here is your %s" %
                        (username, order))
            user.listen()
            del orders[username]

if __name__ == '__main__':
    try:
        app = Bartender()
        app.run()
    except rospy.ROSInterruptException: pass

Figure 8.1: Code listing for our bartender app.
8.2 Design iterations

During evaluation, we had to debug and change a few things in our design that we had not considered originally.

We initially found it troublesome initializing the localization module of our robot. While it can perform global localization (locating itself in the building without knowing where it started at), doing so takes a long time and involves a significant amount of driving. For a while, we would use the ROS Visualization package to inform Frak where it was, but this was cumbersome. We found it much simpler to just assume that the robot is lost when it powers on and issue a `get_help('find_robot')` call.

Due to the short stature of our robot, we had trouble with identifying users. When a user was standing in front of Isaac, it would only be able to see their shins. We had to adjust the `find_new_user` API to tilt the Kinect upwards using its motors.

We also found that if the Kinect wasn’t pointed slightly upwards during travel it would occasionally register the ground in front of the robot as an obstacle and then attempt to avoid it, slowing down Isaac significantly.

In a related issue, reflections would occasionally cause the Kinect to return noisy data, which caused our robot to slow down until it was able to have high confidence about its localization information. One change we plan to make in future versions is to set up preferred paths for our robot to avoid these slow hallways.

This section describes our evaluation of the CLASS system. In our evaluation we measure: (1) a realistic application, (2) how well CLASS can identify users, and (3) how well CLASS can withstand an attacker who tries to impersonate a legitimate user.

<table>
<thead>
<tr>
<th></th>
<th>Height</th>
<th>Bluetooth</th>
<th>Shirt</th>
<th>Location</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>10</td>
<td>80</td>
<td>70</td>
<td>50</td>
<td>97.3%</td>
</tr>
<tr>
<td>Sitting</td>
<td>0</td>
<td>80</td>
<td>70</td>
<td>50</td>
<td>97.0%</td>
</tr>
<tr>
<td>Cell phone off</td>
<td>10</td>
<td>0</td>
<td>70</td>
<td>50</td>
<td>86.5%</td>
</tr>
<tr>
<td>Different shirt</td>
<td>10</td>
<td>80</td>
<td>0</td>
<td>50</td>
<td>91.0%</td>
</tr>
<tr>
<td>In hallway</td>
<td>10</td>
<td>80</td>
<td>70</td>
<td>0</td>
<td>94.6%</td>
</tr>
</tbody>
</table>

Figure 8.2: Results of experiment for testing CLASS’s ability to identify Alice in various situations.
<table>
<thead>
<tr>
<th></th>
<th>Height</th>
<th>Bluetooth</th>
<th>Shirt</th>
<th>Location</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>55</td>
<td>40</td>
<td>12</td>
<td>20</td>
<td>81.0%</td>
</tr>
<tr>
<td>Sitting</td>
<td>0</td>
<td>40</td>
<td>12</td>
<td>20</td>
<td>57.8%</td>
</tr>
<tr>
<td>Cell phone off</td>
<td>55</td>
<td>0</td>
<td>12</td>
<td>20</td>
<td>63.3%</td>
</tr>
<tr>
<td>Different shirt</td>
<td>55</td>
<td>40</td>
<td>0</td>
<td>20</td>
<td>78.4%</td>
</tr>
<tr>
<td>In hallway</td>
<td>55</td>
<td>40</td>
<td>12</td>
<td>0</td>
<td>76.2%</td>
</tr>
</tbody>
</table>

Figure 8.3: Results of experiment for testing CLASS’s ability to identify Bob in various situations.

8.3 Methodology

To evaluate CLASS’s ability to identify users we implemented the CLASS system on our robot and ran a series of experiments. In our first set of experiments we wrote an application for purchasing coffee on behalf of a user. In our second set of experiments focus on the specific task of identifying a user where we start with a baseline measurement that includes all features, then we intentionally perturb each of the four features and we compare the two measurements to see if they still match. To perturb height, we sat down, preventing our robot from calculating the user’s height. To perturb Bluetooth, we turned off our cell phone. To perturb shirt color, we changed shirts. To perturb location, we moved out of the office and into the hallway. After each of these perturbations we ran the identification algorithm and report how confident the CLASS algorithm was that the user was a match. In our third set of experiments we tested the robot with an adversarial user. This adversary was able to match all of the features of the real user except for one.

For our hardware, we used an iRobot Create as the base, an ASUS Eee PC 1015PEM netbook to run the software, and a Kinect. We had to assemble a power regulator to connect the Kinect to the iRobot’s power supply, and we machined a mounting plate to hold it all together\(^1\). For the software, we used the latest distribution of ROS, ‘diamondback’. We wrote our own iRobot driver for ROS, ‘irobot_create_illinois’, which should be available online soon. We used the OpenNI Kinect drivers, and Willow Garage’s ‘pointcloud to laserscan’ node to generate the depth sensor readings for ROS’s amcl navigation modules. We wrote a simple map server that provided an

\(^1\)Construction documented at http://murph.cc/isaac
occupancy map of our building, and used the ‘rviz’ tool to provide initial pos estimates. Our face detection code comes directly from OpenCV and we wrote our own ‘shirt_color_detector’ service that did the necessary processing. Our audit logging node used a MySQL database to store the long-term history of the robot’s observations, and the ‘rosbag’ utility to store the in detail full system state for the short term. The CLASS algorithm itself was implemented in python as a ROS Service, and a second ROS node was created for the Coffee application. That node was responsible for providing the overall direction to the rest of the robot (drive to location A, locate person B, etc). We used eSpeak 1.44.04 to provide text-to-speech capability to the robot, and implemented a text-based command protocol to issue commands to the Coffee application via ROS messages. Bluetooth support was a simple wrapper around ‘hcitool’ that checked for paired devices and read their signal to noise ratio to estimate distance.

8.4 Purchasing coffee

To evaluate the CLASS system, we designed an example application that required the robot to purchase coffee from the nearby coffee shop and then bring it back to the person that ordered it. This required it to identify the person who originally ordered the coffee, and to then identify the same person when it returned. Additionally, it had to avoid misidentifying anyone else that it found as that person. We will skip some details of the implementation (how someone expresses the order, path planning to the coffee shop and back, interactions with the shop) and focus on the parts related to CLASS.

We're going to step through the steps that our robot, Isaac, had to take to buy coffee for the user, Murph.

8.4.1 The user

Murph (the author of this paper) is a graduate student that has a messy desk in an office shared with other students. Isaac has had several days to observe Murph’s habits: how frequently he kept his cell phone on him, how tall he is, how likely he was to be sitting at his desk at any given time, etc. Since we knew the user so well, we supplemented Isaac’s knowledge a bit to
reduce the learning period (e.g., we informed him what color shirts Murph wears). You could say he knew the user reasonably well. However, he had yet to see Murph this day.

8.4.2 Initial order

When the user placed his order, the robot needed to successfully identify him such that it could recognize him again in the future. In this case, Isaac took a good look at the user and decided that it recognized him to 79% confidence. This was a bit low because the robot had not yet encountered the user on that day and was unaware of what he was wearing. Additionally, Isaac was unable to get a good height measurement since the user was sitting down. Much of the confidence came from the proximity of the user’s cellphone (via Bluetooth) and the low likelihood that anyone else would be sitting in that desk. However, 79% is still reasonably high and the application was satisfied that it knew who the user was.

8.4.3 The return

After purchasing the coffee, Isaac headed back towards the user’s office. Along the way, he saw a person in the hallway that, through coincidence, was wearing the same color of shirt as the user, and was equally tall. Even though those two factors return a high likelihood of identifying the user, there were negative indications due to the lack of the user’s Bluetooth signal (recently observed to be near him), and the unusual location. Overall, CLASS returned a 29.5% confidence that this was the user. This confidence percentage was much too low, so the robot continued on.

8.4.4 The return, part 2

Returning to the user’s desk, Isaac had no trouble recognizing him. He had seen the user at this location a few minutes earlier, so it was likely that he would find the user here again. He was able to find the user’s phone via Bluetooth, and this time Isaac was able to gain additional confidence in his identification by confirming that the user’s shirt matched what he’d seen
earlier. CLASS returned a 88.7% confidence, and the application happily handed over the coffee.

8.5 User influence on factor effectiveness

The effectiveness of each factor varies from one user to another. The better a factor correlated to something outstanding about the user, the more confidently it could identify him. Unfortunately, recording the habits of a bunch of CS grad students does not demonstrate this well, so let’s imagine two very different users and compare how well CLASS can identify them. We give them hypothetical histories (locations, clothing worn, etc.) that demonstrate possible users.

To test how well CLASS can identify these users, we load these hypothetical histories into CLASS and use the robot to measure how well it can identify a real user with these hypothetical histories.

Our first user, Alice, has her own office and keeps to herself in it. She is of average height, never lets her cell phone out of her sight, and always wears bright pink outfits (which would really stand out in the computer science department). As you can see in Figure 8.2, CLASS can confidently identify Alice even in situations where a particular factor might be unreliable. If someone is in her office, CLASS can be somewhat confident that it is her. If someone is wearing bright pink, it’s bound to be Alice.

Now, let’s look at our second hypothetical user, Bob. Bob tends to blend in a bit more than Alice. He does most of his work in a community area, and tends to wear grey. He’s unusually short, and loses his phone at least once a week. As you can see in Figure 8.3, it’s much harder to identify Bob. With a perfect measurement, CLASS still can only achieve 81% confidence that it recognizes him. His unusual height helps out a lot – but it’s not at all unusual for him to be sitting down and thus preventing his height from being measured – and then the confidence drops to 57.8% which is quite low.


<table>
<thead>
<tr>
<th>Experiment</th>
<th>Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Too short</td>
<td>46%</td>
</tr>
<tr>
<td>Wrong phone</td>
<td>35%</td>
</tr>
<tr>
<td>Wrong shirt</td>
<td>29%</td>
</tr>
<tr>
<td>Wrong location</td>
<td>92%</td>
</tr>
</tbody>
</table>

Figure 8.4: Results of experiment for testing CLASS’s ability to withstand an adversary who is spoofing all of the features of a user except for one.

8.6 Adversarial analysis

CLASS is designed to resist attacks by allowing a single strong negative identification to reduce the confidence level of it’s results. For our adversarial analysis, we return to our original user – the overworked grad student who just wants his coffee but is too busy to get it himself. We adopt a very strong threat model – our adversary can perfectly mimic the user except for a single one of the factors.

Figure 8.4 shows the confidence measurements that were returned when our adversary attempted to claim the coffee in each possible situation. As you can see, a single reasonably strong negative identification was enough to prevent him from stealing the user’s coffee for three of the four features. The only exception to this was location – our robot, on its own, cannot completely call out the adversary based on location. If it were able to communicate with another trusted robot or source, it might be able to (it could determine that the real user was last seen far away recently, which would be a strong negative indication).
CHAPTER 9
RELATED WORKS

We divide our discussion of related work into four main categories: robot operating systems, robot apps, robot programming, and operating systems.

The most closely related work is ROS [12, 13], which is an operating system for robots that runs on top of Linux. ROS uses a microkernel architecture and focuses on message-passing mechanisms for combining different robot subsystems together. ROS defines a set of low-level robot abstractions, such as point clouds and paths, that one can use to program a robot. Frak is built on top of ROS and reuses the basic architecture and message passing facilities (Frak libraries and apps are ROS nodes), but in Frak we focus on abstractions for robot apps and abstractions for combining libraries together when the robot encounters an unanticipated situation. Frak also focuses on mechanisms and policies for running multiple apps concurrently, whereas ROS is designed to work for a single running app. The latest version of ROS does include some primitive support for access controls on ROS nodes to limit the types of services nodes can use. We plan to use these mechanisms as a part of Frak in future versions of our software.

Additional robot frameworks and operating systems include CARMEN used for tour guide robots [15] and the CRAM framework for mobile manipulation and control programs [16]. A recent position paper by Finnicum and King [17] argued for applying OS principles to robot apps, and a position paper by Gunter [18] argued for enabling embedded systems (e.g., microwaves) to run Java programs, but the focus of these works is on security, whereas we focus on abstractions for apps.

Recently, an online marketplace, called Robot App Store, opened to promote robot apps [19]. The existence of Robot App Store and the scores of robot applications in the marketplace suggests that developers are becoming interested in programming apps for robots.

In the Robot App Store model, developers upload complete system ROM
images and *ad-hoc* instructions for uploading the app from a computer to a robot. Each app must be ported to a new robot, and has a different and complete system image. When apps run, they have exclusive and complete control over the system. Our Frak system is a stark contrast to the model being used by Robot App Store. In the Frak model, apps run on top of high-level robot abstractions, enabling apps to run on any robot hardware that Frak supports, and Frak supports multiple apps running at the same time, providing a more rich and portable execution environment for robot apps. We believe that the combination of a marketplace, like Robot App Store, and the runtime support in Frak will be a practical way to distribute robot apps in the future.

A survey by Biggs and MacDonald [20] enumerates many different programming languages and techniques one can use to program robots. More recent work looks at making pancakes based on directions downloaded from the internet and translating these directions into a prolog-like plan for a robot to execute [21]. In contrast, our focus is on abstractions for enabling app developers to write robot apps.

Our work on Frak is inspired by work on microkernel OS architectures [22, 23, 24]. Our basic architecture is a microkernel architecture, but our work focuses on unique issues inherent to programming apps for robots.
CHAPTER 10

CONCLUSIONS

In this paper we presented Frak, an operating system for robot apps. To support robot apps, we introduced a novel “get help” abstraction, that enabled app developers and the Frak framework to deal with uncertainty without complicating app software. We also built an object abstraction for managing users and locations, and a UI that enabled robot apps to interact with users without having to know the details of the underlying robot hardware. Frak also included mechanisms for multiplexing robot hardware. Although many of these techniques were guided by principles established in more traditional computer systems, the fundamentally probabilistic and uncertain environment in which our system ran forced us to make some novel design decisions.

To showcase Frak, we built a robot, Isaac, and 14 apps that we evaluated in our office in a realistic deployment. All of our runs succeeded often with the aid of the “get help” abstraction that we built to deal with unexpected states and events. Operating with just the localization and identification algorithms that we implemented, the apps would have been unreliable – however, we found that the robot was quite reliable when allowed to ask for help.

In addition, we have demonstrated that a robot built from commodity parts can use its sensors and a system like CLASS to accurately identify and locate the people that it needs to interact with. By recording sensor readings over time, the robot adapted to the people in its environment and learned how to weight each factor’s inputs appropriately. CLASS showed itself to be resilient to attack by a reasonably skilled attacker, and will become increasingly resilient and accurate as more factors are added. The inclusion of an audit log allowed for CLASS to re-evaluate its past decisions and potentially catch earlier mistakes.

Lastly, we have shown that a robot application can be made secure without the developer having to understand the details of the specific robot platform
that it is running on. The “coffee” application required no knowledge of what factors CLASS was using to compare people. As robotics continue to become more mainstream, it is important that developers be given tools such as CLASS so that they can create more secure applications.
REFERENCES


