GENERALIZING ROBOT APPLICATION DEVELOPMENT OPERATING SYSTEM AND FRAMEWORK ABSTRACTIONS

BY

ALEKSANDAR KRAVCHENKO

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, 2012

Urbana, Illinois

Adviser:

Assistant Professor Samuel T. King

Abstract

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. We foresee 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 paper, we introduce a robot operating system to support these apps and we detail the framework abstractions that it provides. We present many lessons learned from developing and debugging a number of such apps, and discuss a novel user-interface concept wherein we abstract the interaction model to allow apps to interact with users via a remote or voice interface in order to accomplish their goals. We present our audit logging infrastructure and the utility it provides to both application developers and users. We show that our framework allows a robot to effectively deal with challenges, such as user authentication and interaction. We demonstrate a simple bartender app to fetch drink orders for students, and it is successfully able to deliver them in 9/10 trials in real-world conditions.

To my Mother and Father, for their love and support.

Acknowledgments

This project would not have been possible without the support of many people. First, I would like to thank my adviser, professor Samuel T. King, for the inspiration, motivation and invaluable input that directed me throughout my graduate study. In addition, many thanks to him for providing me with an office space, connecting me with great people to work with, and the delicious pizza we shared during the long nights. This thesis is a derivative of an earlier project inspired by the works of my mentor, Murph Finnicum, to whom I thank for his dedication, ideas, and help in getting everything to work. I would also like thank Joseph Leong and Corbin Souffrant for helping with the project. Big thanks to Jump Trading for awarding me with a fellowship that enabled me to focus on my studies and drive the project to completion. Finally, I would also like to thank the amazing professors that I interacted with and my colleagues and friends that helped me stay in focus and supported me throughout my college experience.

Table of Contents

List of Figures

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 [18]. People use service robots for cleaning floors and carpets [10], mowing lawns [16], and driving cars [23]. Household robots are used as pets [19], and in nursing homes and for children's rehabilitation [14]. Evidence shows that people bond with all types of robots and accept service and household robots as a part of their families [20, 5], 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 from traditional computer systems in interesting and novel ways.

Robots interact with the world and people around them without the benefit of a well-defined interface like traditional computer systems have long enjoyed – this unrestrained interaction poses unique challenges for designers of general-purpose robotic systems. Robots have more sensors than traditional computers and they can move autonomously. Mobile robot algorithms are fundamentally probabilistic [6, 21, 22], complicating tasks like identifying and authenticating people that interact with robots. Robotic systems have huge state spaces, where the state space of a robot is the world, which encompasses far more that the files, processes, and users, that comprise modern OS abstractions. Finally, robot abstractions operate at a higher level than the general OS abstractions that traditional OSes use. OSes operate on processes, files, sockets, and users, whereas robots operate on other robots, people, walls and objects, maps, locations, paths, and so on.

In this paper, we describe Frak[1], an OS and an application framework we built for simplifying app development on robots and for managing robot hardware. Our Frak OS includes abstractions for enabling app developers to write programs that operate on robot abstractions *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 general objects, 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, independent of their location and communication device. In addition, we have implemented a robust audit logging system that almost completely captures the state of the robot in any given time and allows for investigating applications behavior. 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 and evaluate our implementation we built 14 apps that we use on Clarke that we deploy 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, Clarke travels to the room where people congregate, 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 secretary and asks him or her (nicely) to submit them for reimbursement.

Our contributions are:

- We designed and implemented a novel robot application programming framework and system for running robot apps, including multiple apps concurrently.
- We introduce a flexible user interaction model for dealing with different communication scenarios in robot applications.
- We propose and describe our audit logging infrastructure that enables applications to be more transparent to the user, and also help developers in debugging.
- We wrote a number of robot apps and evaluated them in real world conditions.

[1]Framework for Robot Applications, K?

Chapter 2

System Architecture and Design

The aim of this chapter is to provide an overview of the system and design principles that have guided Frak's architecture. The design has been influenced by our primary goal to make robot applications easier to develop by abstracting the underlying limitations of the hardware and the complexity of the software libraries, while allowing multiple applications to run concurrently. 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.

2.1 Hardware architecture

The hardware architecture has been designed 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 Clarke, we use an iRobot Create, which is like a Roomba without the vacuum and with a programmable interface that we connect to a network-connected netbook running Linux that handles most of the computation, and a Kinect camera sensor for video and depth sensing (Figure 2.1). With this selection of hardware, Clarke can speak, listen for spoken words, identify and perceive objects through the Kinect sensor, and navigate with controllable speed with the motor wheels. To allow the robot to carry things we have installed a flat surface to carry drinks and other items that can fit. Although our testing robot platform lacks any direct and controllable manipulators such as arms, it has the necessary hardware to be mobile and interactive with its surroundings.

Figure 2.1: Our robot, Clarke, includes an iRobot Create, a netbook, and a Kinect.

2.2 Software architecture

Frak builds on top of ROS (Robot Operating System) [15, 24], which is an actively developed open-source middleware specifically designed to ease the development of robot programs and platforms (Figure 2.2). ROS exposes a number of abstractions that we recognize in our system nodes, topics, messages, and services. The middleware is designed like a distributed system with nodes being the processes that perform computation and can control the execution environment. In the context of robots, one node can control the wheel motors, one node performs localization, another one can perform path planning, and so on. Each node runs in its own process recognized by the underlying Linux kernel.

Nodes interact with each other through clearly defined communication primitives. ROS provides a message-passing mechanism where nodes can subscribe and publish messages to certain topics and act upon them. A node sends out a message by publishing it to a given topic. A node that is interested in receiving certain messages will subscribe to the appropriate topic. Any node can subscribe and/or publish to multiple topics, creating a many-to-many relation between the nodes. In general, publisher and subscriber nodes are not aware of each others existence, only the context of a topic. In addition to the many-to-many communication, ROS also provides a one-to-one interaction based on a request/reply model through the notion of services. A service is similar to a remote procedure call, with one node acting as the request node, and the other returning the result to the caller.

Our systems main goal is to eliminate a lot of the boilerplate associated with writing robot apps, and encapsulate the sophisticated control logic into higher-level abstractions. Thus, we have designed Frak into two layers: the library layer and the application layer. An application in the context of Frak is the highest

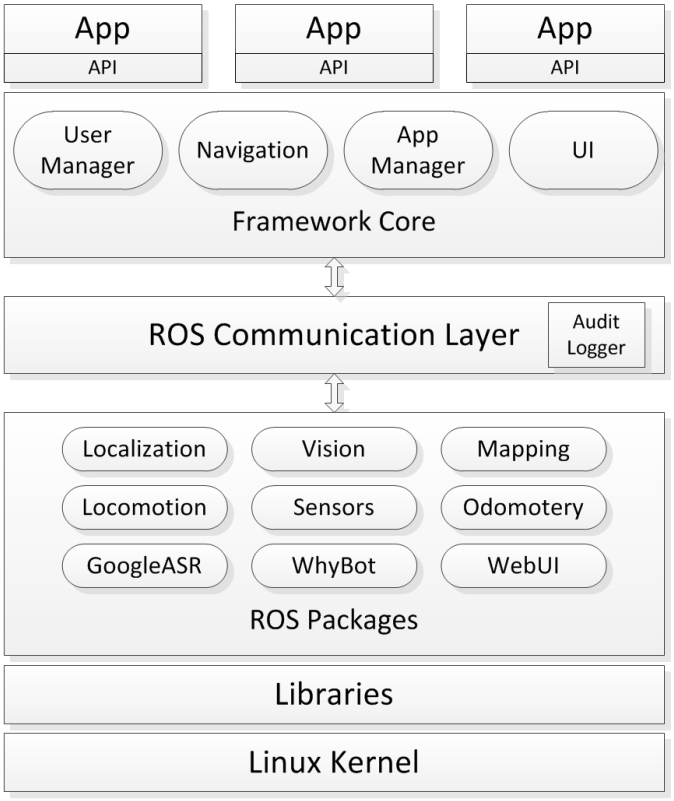

Figure 2.2: Our robot micro-kernel architecture. At the lowest layer is the Linux Kernel OS and the libraries that implement hardware-specific software that interacts with the robot and sensing hardware. At the next layer are ROS packages that implement low-level robot abstractions or expose a ROS node for a library. The ROS communication layer is used for implementing message passing between the framework core services, apps, and the ROS packages. Frak is at the top layer, which implements abstractions for apps and defines the API .

---

level of abstraction recognized in our system and is defined as a set of nodes that perform some control logic on the robot. For the application layer we have chosen Python as the main language, because of its simplicity and large third-party library support. The underlying libraries and implementation rely on ROS to provide higher-level abstractions to the applications.

2.3 Core services and objects

2.3.1 Services

Frak s micro-kernel architecture exposes a number of services that manage application state, enable applications to perform common tasks, multiplex and administer hardware access, manage environment objects such as users and locations, mediate user interactions, and help find resolutions to certain environment ex-

ceptions. Our modular design allows for the hosting of multiple services on a single process, or run different services on multiple processes, without the need to make any modifications to the existing code base. This design allows for easy scaling and fault isolation. In the current implementation, a single process is dedicated to run all core services, which minimizes the IPC overhead and duplication of global state.

The main core services are the application manager, user manager, navigation service. Applications define a configuration file that expresses what resources they require and also serve as a unique identifier throughout the system. Upon launching an application, the app manager registers the application with the system and starts to manage the application state. The app manager also audits any other service bindings an application might request.

When an application requests access to a particular service the permission list of the application is checked against the required permissions upon binding. Navigation modules serves as a proxy to the main navigation service, where applications make requests to reach a pre-determined destination recorded in our environment database.

2.3.2 Objects

In Frak, we have a general notion of an object, which we use 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. The objects and any other global environment variables live in the core service that is dedicated to managing their state and propagating the changes to the applications processes. Applications use an object proxy handle that represents the object within that context. Service calls are completely transparent to the application and are managed within Frak.

In our system, 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.

2.4 Multiplexing robot hardware

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. Upon application shutdown certain actuators, such as the Kinect sensor motor, used by the application are restored to their original state.

Chapter 3

User Interaction Abstractions

The human-machine interaction in contemporary computer systems such as PCs and smartphones, has been well studied with clearly defined interaction model and abstractions. Visual interfaces such as GUIs; programming objects such as windows, buttons, and dialogs; and peripheral devices such as mice, keyboards, and touchscreens, facilitate users virtual interactions with the machine. Unfortunately, in the robotics domain the human-robot interaction has not been as clearly modeled yet. Because of their physical and hardware properties such as being mobile and diverse in their capabilities, robots bring interesting and unique challenges to user interface design.

The primary challenge in modeling the user interaction is that the I/O capabilities are dictated by the underlying robot hardware and are non-uniform across different robot systems. Robot apps can interact with people in any number of ways, including through a network-connected computer, audio and video I/O, or any other I/O technology, depending on the hardware present on the robot. To compensate for the diverse hardware configurations Frak encapsulates the underlying hardware specifics and capabilities of the robot by providing a high-level API that encapsulates the interaction with the user. Our design aims not to limit application functionality and we enable developers to provide custom user interfaces through extensible plugin web architecture.

To compensate for diverse hardware configurations, we encapsulate the underlying hardware specifics of the robot by providing a high-level API for interacting with the user and surrounding environment when a user object is out of context. Our API exposes the following API calls: speak, listen, user.ask, user.tell, where speak calls output text, and listen calls wait for a particular phrase or sentence, 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 via user.tell calls to let a user know the status of the app. For example, after a user orders a drink, the bartender app uses the user.tell UI to let users know

where the robot is and the status of their drink order.

On Clarke, Frak uses two different types of user interfaces. First, if the user being interacted with has been entered in the system as having a network-connected computer, such as a smartphone, the interaction will be carried over through their smartphone via the remote web interface. Second, to facilitate interaction with users without a smartphone and enable human-like dialogs Frak provides a voice interface that allows users to communicate with the robot apps though natural language.

3.1 Remote interface

Since robots can change their location, one needs to be able to control and monitor the robots actions and status when the user is not physically present next to the robot. In addition, our system hosts multiple applications, and depending on the communication channel used, each application can potentially interact with multiple users at the same time. These assumptions render the possibility of having a single screen and a keyboard/mouse attached to the robot as the only communication medium obsolete. Having a single screen to display output information on, limits the capabilities of the robot and requires users to be physically close to the robot in order to carry out the interaction. In addition, some robots like Clarke are small; others like the PR2 are big and thus it can get awkward to have users directly interact with the robot. Instead, in our remote model people use their smartphones or computers as the gateway to the robots environment. This model allows for greater flexibility and is quite viable given how common connected mobile devices are becoming. We have designed the remote interface on a web platform that will allow for multiple users/multiple apps, support heterogeneous devices and abstract any physical properties about the robot from the application developer, without them worrying about its capabilities and underlying implementation. The only robot requirement is a network connection.

3.1.1 Design

The remote interface is designed on a web platform. Instead of having each application run an HTTP server and directly communicate with the user, in our design we chose to have a single and embedded HTTP server process that listens for both user and application requests/responses (Figure 3.1). The server process multiplexes and manages all user-application sessions. With this approach, we can intercept the user-application interaction and insert additional logic to enhance security, fault tolerance such as applications crashing or users exiting, and make application code less convoluted. The communication channel between the web server and the user is through an HTTP protocol, whereas applications can communicate with the

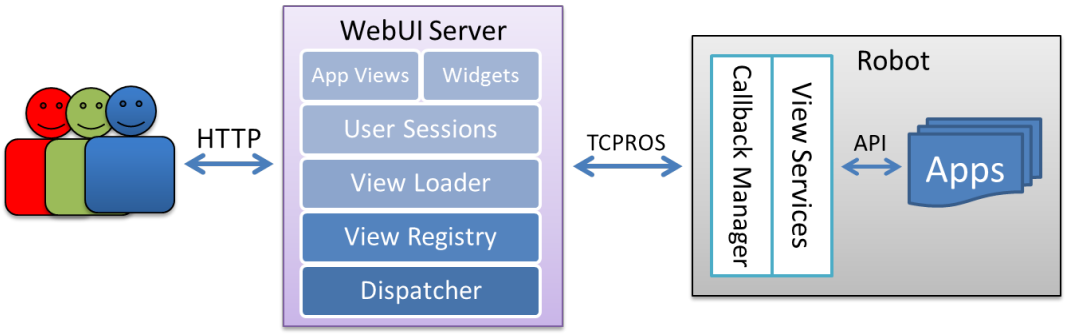

Figure 3.1: The WebUI remote interface architecture.

server over any protocol and logic. This design allows us to change the communication protocol on each end without affecting the other. In our design, we use ROS as the communication overlay for the latter. In addition, the server process can be hosted on any internet-connected machine, which makes this a modular and scalable design.

The server only hosts UI code that renders the view for the user. The server process, however, is not stateless, and in addition to keeping active user-application sessions, our design allows the view to keep additional state and data. Control logic runs within the application. When the user makes a selection, clicks a button, or completes any other action, the result is forwarded to the local application as an asynchronous callback via a feedback topic. The application can then make a decision on what to return to the view via a different result topic that the web server listens to. The view code running on the server can then display the final web page to the user. What particular messages are sent on these topics is decided by the view-control logic, as they have to agree on a common format. This is entirely left to the developer; however, our system ships with views that implement common logic such as asking the user a question, or notifying the user about applications status.

The web server uses a dispatch controller based on URL prefix to determine which applications view to load. The default view that ships with our system is indexed by the root path `\` and is the first application loaded by the web server. Applications are loaded by `/app/app_name`, and if an app with the name `app_name` has been registered with the web server, its view will be loaded on access. Applications upon startup, can register their view with the server through the UI manager service by including the location of the view to be loaded.

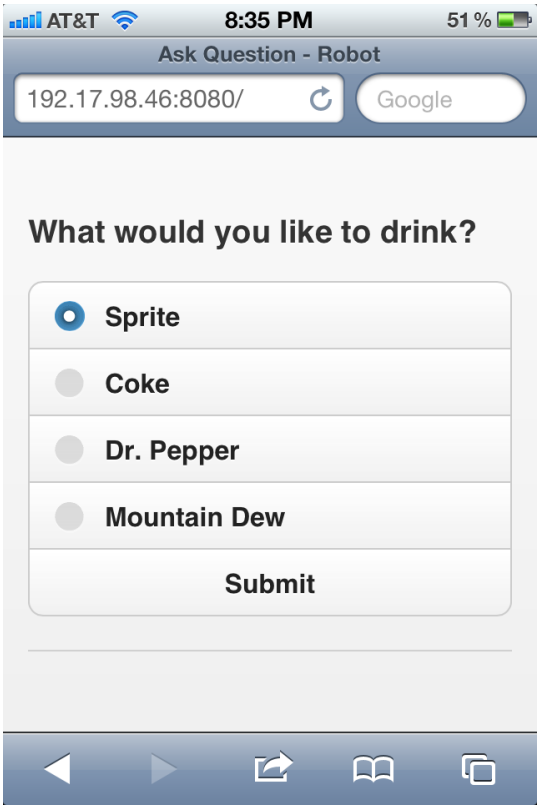

Figure 3.2: The WebUI dialog view, prompting the user to select a drink.

3.1.2 Implementation

To implement our web interface we used a lightweight micro-framework written in python called Flask. It includes a powerful templeting engine that allows us to reuse common view components written in HTML such as forms, headers, footers, etc. For the front-end, we used jQuery Mobile for its simplicity and clear looks. View logic is written as a separate Flask application, that simply defines routing rules and custom HTML/javascript pages, all of which can be accomplished in a single python file with a few lines of code. Applications can directly use ROS library calls, since the view is written in python and includes the core ROS packages. To access the web site hosted on the robot, we display a QR code (2D bar code) on the netbooks screen. Users simply scan the code with their mobile device, which initiates a private session with the robot.

We have implemented a dialog view through the user.ask(question, options) API (Figure 3.2) Since the response from the user can take a long time several seconds, or even minutes, the API call returns a future object that the application can invoke any time to block and wait for the users answer. Underneath the API call, we give each question a unique id that serves as an identifier on the server side, so when we receive the

callback from the server we would know which future object to complete. There is a DialogManager running on the server side that manages each user-application session and saves the questions and responses that need to be displayed to each user. Each user has a mailbox that corresponds to her communication channel with the robot. user.tell(answer) follows similar semantics, however the user is not given an option list, but rather a single notification message. When an application invokes the interface API, Frak encapsulates all the semantics and allows us to handle even more communication options in the future, without affecting application code or changing the high-level API.

3.2 Voice interface

When direct user interaction is not available, because either the user does not have an internet-connected device, or because a user is not known in a particular context, we have implemented a voice interaction interface through the speak and listen API calls. The listen call takes an optional array of phrases to listen for and speak takes a string to be vocalized through the speakers. This allows the robot to be more practical in situations where it needs input from an anonymous source and when external vocalization is required to either get attention or just for feedback. These high-level interaction methods hide-away many implementation details and through indirection can also invoke the user.ask or user.tell remote interface methods, for example when the voice is recognized as an existing users voice.

In order to recognize spoken utterances we have implemented a voice recognition library and framework service that leverages Googles Automatic Speech Recognition API. By using Googles cloud infrastructure, we are able to reduce the processing power required on the robot, and increase drastically the accuracy of the results. Each converted response has an accuracy measurement that we recognize in our framework and we propagate to the application layer. Applications can use that to decide if the accuracy of the recognized speech is good enough in order to make a decision. In our implementation, we use the gstreamer library with a custom voice-activity detector plugin. To make up for environment noise we sample the environment before each detected utterances beginning and end, and remove the noise through a noise reduction gstreamer plugin. Since Google requires each sample to be encoded in FLAC, we have written a python bridge for the FLAC encoder library that we have added at the end of the audio pipeline. Since each request is sent over HTTP through the network, we further minimize network latency by pipelining the voice detection, encoding, and HTTP stages. In addition, we use a python library called requests that allows us to increase responsiveness by making HTTP requests asynchronous. For vocalizing text, we use a Linux text-to-speech synthesizer called eSpeak, that allows us to specify different voice characteristics.

Chapter 4

Audit Logging

4.1 Motivation

Robots are built on top of existing computer hardware and software abstractions, and as such, inherit all of their security and privacy vulnerabilities [4]. In addition, unlike existing computer systems, robots are autonomous in their actions, execute in dynamic and unpredictable environments, are mobile, have manipulators (such as arms), and can physically interact with their surroundings. This gives applications running on the robot system great power and interactivity with the physical world. As a result, a malicious or buggy application can inflict far greater damage than any similar application running on conventional computer systems. We argue that in a robot system that hosts multiple untrustworthy applications, written by different developers, determining which application(s) could have been responsible for robots actions is required in order to provide liability and responsibility.

Research in contemporary computer systems has addressed many security and privacy issues that arise with hosting untrustworthy applications. Tools like Back-Tracker [11] have been developed, which allow system administrators to trace back intrusion attempts to their possible sources through the use of OS level objects, events, and dependency rules. While derived from conventional computers, robots use higher-level abstractions such as people, places, surrounding objects, and actions to describe events. Conventional operating systems abstractions such as files, sockets, and processes are still present; however, it would be difficult to fully explain a robots actions at that level of abstraction. Information about files and their associated reads and writes become less useful, and give way to higher-level abstractions such as a robots location through time and other sensor data. Thus, tools that apply to conventional computer operating systems are not adequate in a robot system.

Unfortunately, security in robotics systems has not been as thoroughly investigated. To the best of our knowledge, the closest work to our study that enables certain level of auditing in robots is the research by Mosenlechner et al.. Their work allows a robot to explain its reasoning - what it was doing, how, and why [13]. The system can explain control decisions and the reason for specific actions and diagnose the cause of

task failures. Although their system can answer different questions in order to explain past robots action scenarios, their robot architecture was not developed to host untrusted third-party applications and as such they do not take into consideration multiple applications running on the robot as the cause for the robots actions. In addition application use a specific domain-level control language, that restricts the application developer.

To provide application accountability in robot systems we have developed WhyBot, a system that attempts to make the results of applications running on general purpose robot systems more transparent to the user. To the best of our knowledge, this is the first system that addresses the need for audit logging at the application-robot interaction level and tries to associate robots actions back to the application executable(s). To achieve our goal we have decomposed the system into three main parts:

- A mechanism that allows users to receive answers pertaining to past robot actions and events. Such as: Was the robot in the kitchen today?, or Did the robot interact with John?
- A tool that allows us to trace specific events such as the robot being in the kitchen, back to applications that were involved in achieving this task.
- We further extend the functionality by allowing users to investigate chosen applications more closely, revealing other actions resulting from their execution.

With such a system we give users the ability to more easily understand what took place during robot programs execution. It also attempts to address and thereby help future robot users more easily understand what took place during a malfunction. It will enable us to answer questions about robots past actions, attribute those actions to applications that were running on the system, and also provide information on what else an application might have done with the system while it was running. Our approach takes into account the higher-level abstractions that robot systems work with. Not only does this allow us to answer more questions about the systems execution, but it also becomes more understandable and readable for the user.

4.2 Design and implementation

To develop a system that can reason about robot application behavior and actions, we need to be able to record all important and relevant aspects of the execution environment and provide a mechanism to interpret the recorded data. In order to achieve those goals we have implemented WhyBot on top of ROS in two components: an on-line capturing component integrated into the application platform that logs relevant

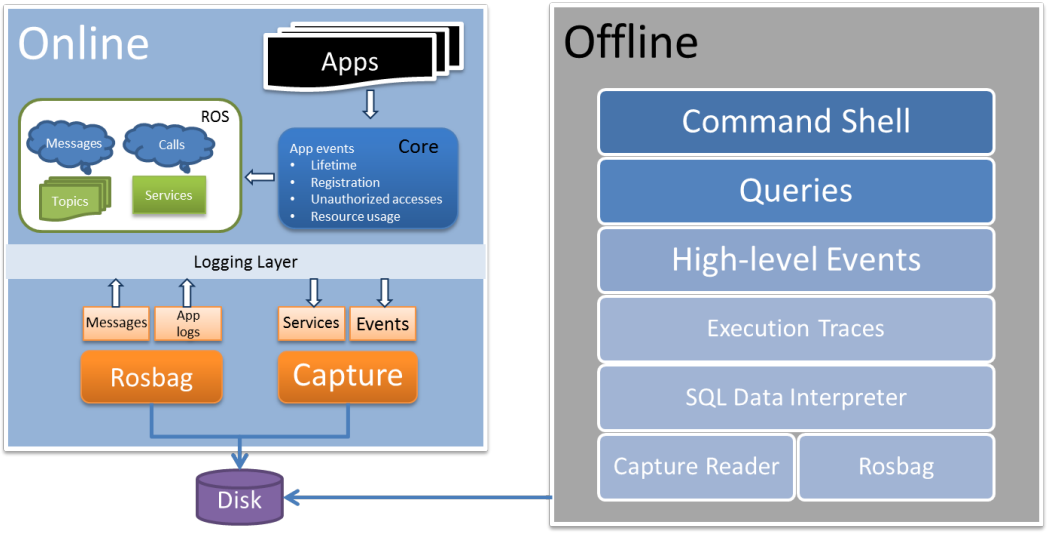

Figure 4.1: WhyBot audit logging system architecture.

events and state perceived from the robot, and an offline component that analyzes and interprets these logs to allow further introspection about robots actions. WhyBot currently records and tracks many relevant robot events, both high-level events such as entering a room, or seeing a person, and low-level events such as ROS node subscribing or receiving data from a topic or a service. We found that logging these events was sufficient to allow us to explain application behavior in several applications with acceptable time and space overhead. A general overview of our system is shown in figure 4.1.

To identify the important events to be captured, we need to understand the primitives that influence the state of the system and the behavior of the robot. In ROS, these are the control primitives that enable node interaction - messages, and service calls. WhyBot captures messages by subscribing to all active topics, and monitoring for any new ones. Our implementation allows the user to specify which topics to exclude in the capturing process, should they want to. Since we cannot control the traffic along a topic, we need to be careful in our capturing process for topics with high publishing rate such as sensors, and or messages that carry large amount of data such as video. For high-traffic topics like those, we introduce proxies that subscribe to the original topic but publish at a much lower user-specified rate. In addition, a proxy topic can perform any number of further processing such as image or video compression.

Unlike messages sent over topics, the one-to-one interaction of service calls requires extra care. ROS does not provide any mechanisms for intercepting service calls, thus in our implementation we have modified ROS clients library to capture service call events with minimal performance overhead. We asynchronously forward all events to the capture server, which are buffered and exported to disk. We capture all relevant information

associated with the call such as time of the event, the nodes involved, and the messages exchanged between them. Our capturing infrastructure enables us to serialize and deserialize messages, which can allow us to reconstruct the full state of the system later through replay.

The capturing component also records important events on the application level, such as an application registering with the frameworks AppManager, its permissions, and application debug output. By capturing registration events, we are able to map a node from ROS to an application running on the framework.

The final stage of the capture component is writing the results to disk. Currently we use SQLite to dump the results into a pre-defined database schema, which simplifies the querying for particular events and allows us to extract specific aspects of the system in the offline component.

To reason about applications behavior and its interaction with the robot system we need to reconstruct high-level events from the collected low-level semantic information recorded during the capture stage. In our current implementation, we identify important nodes, topics, messages, and service calls that we label as system resources and correspond to the frameworks core. The system resources include nodes that control the motor wheels, the speaker, the microphone, and any other hardware that an application requires permissions to access. We filter any outgoing messages from an application to these system resources to detect interesting behavior and construct an application execution trace. In our current implementation, we can infer that an application was using the motor wheels, or that an application was listening on the microphone through its execution trace. Our system also records the raw message data exchanged between the nodes and is able to deserialize it. By doing this we can further interpret the control data by looking at the message contents.

We have tried implementing an interpreter that will draw the robots movements based on the messages sent to the motor, however due to the inaccuracy of translating the message contents into actual locomotion, we are unable to fully understand the motions that an application was trying to achieve. Instead, to construct high-level events we fallback to the high-level information that our application framework provides such as the destination an application requested when it used the navigation module. Global state such as the robot entering a room is also provided by this module. Other information, such as which users an application interacted with is available through the UserManager service. We also have information that relates to things heard, words spoken by the robot, or answers received from a user through the UI service.

WhyBot comes with a command line shell that the user can use to get answers about applications events. A user can request to see the outgoing messages, the incoming messages, the services and components used, users interacted with, the places the robot has been. High-level questions such as where was the robot can be answered and the user is displayed with a list of locations such as office numbers, and timestamps corresponding to the time the robot was in there. We also provide an export command that will build a

graph in dot notation to visualize the interactions an application had with the framework. Users can also query to list applications that were running on the robot, and filter by time and even location. We have a pre-defined set of queries that we use to provide answers to those questions, and we list them with the help command.

Having captured all relevant data in the system ultimately can allow us to reconstruct the state of the robot in any given time by replaying the capture data. We have built a tool to replay the messages sent over a topic, and we have integrated replay functionality into the service components of the framework. However, we cannot provide strong ordering on the events as they really happened in the system, since we cannot infer if a message was received first or a reply from a service was processed first just by looking at the time stamps. In certain application in which the order of these events does not matter, the replay functionality works as expected, but in other applications, it can result in different outcome than the original run.

Chapter 5

Evaluation

5.1 Bartender app case study

Our bartender app breaks the task of serving drinks down in to four main steps: taking orders, returning to the bar, acquiring the correct drinks, and giving the orders back to the correct people. Figure 5.1 shows the source code for our bartender app. To evaluate the reliability of this app, we ran it 10 times where we had one of the authors in a lounge and another author serve as the bartender. We ran these experiments during normal business hours with people walking around Clarke as it ran the app. To take drink orders, Clarke 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. 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. If the user is known to have a smartphone, Frak can use the web user interface to ask them their drink order. Otherwise, it can 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. Clarke 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 it was able to be located easily.

5.2 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, which asks a user running the GUI to show where the robot is. Due to the short stature of our robot, we had trouble

```
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.tell("Drink order received")

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("Done")
    speak("Thanks")

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

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

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

Figure 5.1: Source code for our bartender app.

with identifying users. When a user was standing in front of Clarke, 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 Clarke 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.

5.3 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.

Clarke can navigate around the halls of our building with relative ease. Most of the furniture inside of rooms is currently not on our maps. Clarke 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. Bypassers 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, Clarke 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.

Clarke 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 recognize profanity.

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 s web server. Plugins are written in Python, and can either use existing HTML templates, or define custom ones.

5.4 Software

In addition to the apps discussed, we have implemented a number of useful and just-for-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, Clarke 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. We had to debug manually some applications that required us to replay some messages from certain topics, which allowed us to narrow down the cause for the problem. We often used the log to display application statistics, such as number of applications running and the resources in use, consumed and emitted messages, service calls.

5.5 Performance

Clarke 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 the iRobot Create's power supply. As a whole, Clarke could operate for 68 minutes without requiring recharging (longer if it was not 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).

We measured the amount of logging data that a three of our apps generated to get an idea as to the space and processing requirements of WhyBot. We ran the Copycat and Circles apps saving just the binary messages and again while storing both the full string expansions as well as the binary form of the recorded messages in our database. With the full strings the amount of data that needed to be saved to disk more than

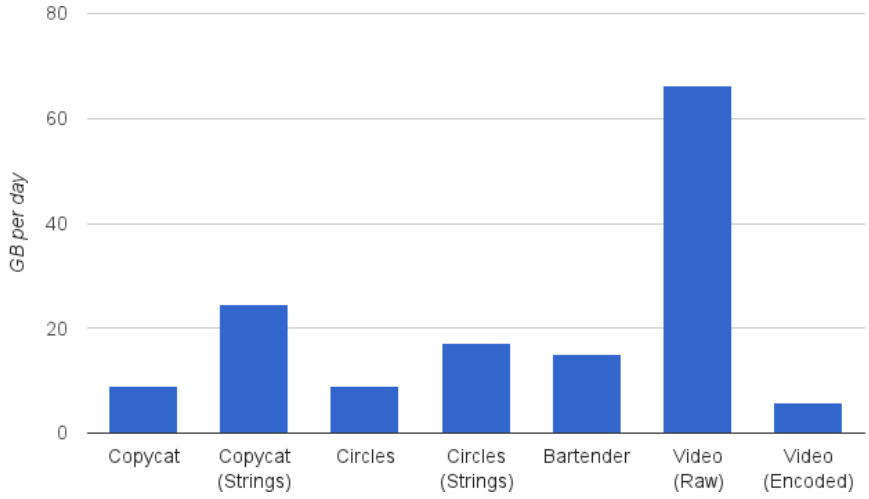

Figure 5.2: WhyBot approximate space requirements in GB/day, where a day is 24 hours.

doubled for both applications, and thus in our implementation we serialize the message data before saving to disk. Due largely to all the processing we did with the video stream, we averaged 23.4MB/s of aggregated bandwidth usage between ROS nodes during execution. These nodes were all on the same machine, however. We were able to reduce the video bandwidth to 70KB/s after encoding it with Ogg Theora and downsample the resolution significantly. We compare these space requirement results for the different apps in Figure 5.2. In our runs since we do not use them directly, high traffic topics like sensor data are heavily throttled in order to reduce space requirements. Since the bartender app was the only one using user identification, it was the only one to include video data in the capture, thus we see the increase in space requirements.

In our tests, we have found the responsiveness of our voice-recognition service to be quite usable, with average response time below 200 milliseconds per speech utterance. We have improved the accuracy of the improved results by using a noise filtration system that can isolate spoken words from background noise. In our tests, we have found that the accuracy of recognized speech is limited by the quality of microphone used. Due to the low profile of the robot, in certain occasions we had to get closer to the robot for optimal results, as the microphones on the netbook were subpar.

The robot during our tests did not have to interact with many users concurrently, however we did a small stress test to measure the responsiveness of our web server. We ran the test with one client, one application, and measured the number of requests that the server can process in both directions, that is a complete request is measured starting from the client, to the application receiving the callback, to sending the response back to the client. We only had the embedded server running, with a test client running the stress test as another process hosted on the robot. The client was sending requests to the test app, which

was responding with either yes or no. On average, we were able to achieve over 130 requests/second. Keeping the requests/second at 100, we saw CPU utilization of 68%.

Chapter 6

Related Works

We divide our discussion of related works into four categories: robot operating systems, robot apps, and reasoning about robots actions through audit logging.

6.1 Robot Operating Systems

The most closely related work is ROS [15, 24], 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 [12] and the CRAM framework for mobile manipulation and control programs [2].

Our work on Frak is inspired by work on microkernel OS architectures [7, 8, 9]. Our basic architecture is a microkernel architecture, but our work focuses on unique issues inherent to programming apps for robots.

6.2 Robot apps

Recently, an online marketplace, called Robot App Store, opened to promote robot apps [1]. 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 [3] 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 [2]. In contrast, our focus is on abstractions for enabling app developers to write robot apps.

6.3 Audit logging

Although a new point in the robot space, tracing back malfunctions and reasoning about application execution though audit logging has been previously addressed by research in contemporary systems. BackTracker [11] was written with a standard operating system in mind. It backtraces intrusions from their detection to its possible sources by building a dependency graph between processes, files and sockets, representing them as nodes. Then links these nodes by observing interactions such as system calls. For example, if a process reads a file, an edge will be drawn between the process and files corresponding nodes in the dependency graph. The key to our logging system is taking advantage of the higher-level abstractions, available in a robot system, to build similar models of applications interactions, similar to those of BackTracker, but to also overcome the additional challenges that accompany the different execution environment.

In the robot systems domain, the idea of reasoning about robots past actions has previously been addressed by systems like GRACE [17]. The system was developed as a submission for the AAAI Robot Challenge in 2003, with the intention to develop a fully automated robot system that can reason about itself and execute actions with limited human involvement. The final task in the challenge was for GRACE to give a talk and answer questions about what the system is capable of doing, by reflecting on the knowledge of its own structure and capabilities.

To the best of our knowledge, the closest related work to WhyBot is the work by Mosenlechner et.al. [13]. In their paper, the authors describe a system that facilitates the reasoning about application execution on the robot by writing control programs in the form of a plan. The plan is described by a high-level language

that involve primitives such as Achieve(Loc(Obj,Loc)) which make programs more transparent and easier to reason about. The execution of a plan generates a task tree that describes the subtasks that need to be performed in order to achieve the goal of the plan. During plan execution the belief state of the world is constantly updated by external sensors and recorded by the underlying logging system. Execution plans are saved on disk in episode knowledge, which includes the task tree and an execution trace of changes in the belief state. The authors also develop a query language that interprets the recorded data and can answer different questions about robots past actions and beliefs. In contrast to our work, the authors do not take into account the execution of multiple plans and use a specialized control language. Additionally, the questions that their system can answer are not directed to applications running on the system, but rather, towards the overall belief state and knowledge of the robot as a whole.

Chapter 7

Conclusions

In this paper we presented an operating system for robot apps and the associated application framework abstractions that we have built. To support robot apps, we introduced an application framework with high-level API that abstracted many of the underlying semantics, capabilities and hardware specifics of the robot. In addition, we proposed a novel "user.ask/tell" remote interface abstraction, that enabled app developers and the framework to deal with convoluted user interactions without complicating app software. We also built the associated object abstractions 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. The framework also included mechanisms for multiplexing robot hardware. We implemented an audit logging system that enabled us to debug our applications, and further provide more transparency to the user about application's past execution. 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 display the capabilities of the framework, we built a robot, Clarke, and 14 apps that we evaluated in our office in a realistic deployment. The most interactive app we built was a bartender, that served drinks between two locations, and with the help of the interaction abstractions was able to interact with the surrounding users in an intuitive and accurate way. The majority of our runs succeeded, often with the aid of the frameworks abstraction that we built to deal with unexpected states and events, with performance not being a limiting factor. We showcased that the information obtained from our audit logs was sufficient to provide useful information in debugging our apps, and with reasonable space requirements.

References

[1] Robot app store. http://robotappstore.com.

[2] M. Beetz, U. Klank, I. Kresse, A. Maldonado, L. Mosenlechner, D. Pangercic, T. Ruhr, and M. Tenorth. Robotic roommates making pancakes. In *Humanoid Robots (Humanoids), Proceedings of the 2011 11th IEEE-RAS International Conference on*, pages 529 –536, Oct. 2011.

[3] G. Biggs and B. MacDonald. A survey of robot programming systems. In *Proceedings of the Austrailian Conference or Robotics and Automation*, 2003.

[4] T. Denning, C. Matuszek, K. Koscher, J. R. Smith, and T. Kohno. A spotlight on security and privacy risks with future household robots: attacks and lessons. In *Proceedings of the 11th international conference on Ubiquitous computing*, Ubicomp '09, pages 105–114, New York, NY, USA, 2009. ACM.

[5] J. Forlizzi. How robotic products become social products: an ethnographic study of cleaning in the home. In *Proceedings of the ACM/IEEE international conference on Human-robot interaction*, HRI '07, pages 129–136, New York, NY, USA, 2007. ACM.

[6] D. Fox, W. Burgard, and S. Thrun. The dynamic window approach to collision avoidance. *Robotics Automation Magazine, IEEE*, 4(1):23 –33, Mar. 1997.

[7] D. Golub, R. Dean, A. Forin, and R. Rashid. Unix as an Application Program. In *Proceedings of the 1990 USENIX Summer Conference*, 1990.

[8] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of μ-kernel-based systems. In *SOSP '97: Proceedings of the sixteenth ACM Symposium on Operating Systems Principles*, pages 66–77, New York, NY, USA, 1997. ACM.

[9] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. MINIX 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89, 2006.

[10] iRobot, Inc. http://www.irobot.com/.

[11] S. T. King and C. P. M. Backtracking intrusions. pages 223–236, 2003.

[12] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (carmen) toolkit. In *Intelligent Robots and Systems, 2003. (IROS 2003). Proceedings. 2003 IEEE/RSJ International Conference on*, volume 3, pages 2436 – 2441 vol.3, oct. 2003.

[13] L. Mosenlechner, N. Demmel, and M. Beetz. Becoming action-aware through reasoning about logged plan execution traces. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 2231 –2236, oct. 2010.

[14] PARO Robots USA, Inc. Paro therapeutic robot. http://www.parorobots.com/.

[15] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[16] Robomow. http://www.robomow.com/.

[17] R. Simmons, D. Goldberg, A. Goode, M. Montemerlo, N. Roy, A. Schultz, M. Abramson, I. Horswill, D. Kortenkamp, and B. Maxwell. Grace: An autonomous robot for the aaai robot challenge. Technical report, DTIC Document, 2003.

[18] P. Singer. *Wired for war: robotics revolution and conflict in the 21st century.* Penguin Press, 2009.

[19] Sony, Inc. AIBO Entertainment robot. http://www.sony.net/SonyInfo/News/PressArchive/199905/99-046/.

[20] J.-Y. Sung, L. Guo, R. E. Grinter, and H. I. Christensen. My roomba is rambo: intimate home appliances. In *Proceedings of the 9th international conference on Ubiquitous computing*, UbiComp '07, pages 145–162, Berlin, Heidelberg, 2007. Springer-Verlag.

[21] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Probabilistic algorithms and the interactive museum tour-guide robot minerva. *International Journal of Robotics Research*, 19(11):972–999, 2000.

[22] S. Thrun, W. Burgard, and D. Fox. *Probabilistic robotics.* Intelligent robotics and autonomous agents. MIT Press, 2005.

[23] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, K. Lau, C. Oakley, M. Palatucci, V. Pratt, P. Stang, S. Strohband, C. Dupont, L.-E. Jendrossek, C. Koelen, C. Markey, C. Rummel, J. van Niekerk, E. Jensen, P. Alessandrini, G. Bradski, B. Davies, S. Ettinger, A. Kaehler, A. Nefian, and P. Mahoney. Winning the DARPA grand challenge. *Journal of Field Robotics*, 2006. accepted for publication.

[24] Willow Garage. http://www.ros.org.