Title
Reliability Engineering for Long-term Deployment of Autonomous Service Robots

Author
Wang, Shengye

Publication Date
2020

UNIVERSITY OF CALIFORNIA SAN DIEGO

Reliability Engineering for Long-term Deployment of Autonomous Service Robots

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Shengye Wang

Committee in charge:

Professor Henrik I. Christensen, Chair
Professor Thomas R. Bewley
Professor Ryan Kastner
Professor Scott R. Klemmer
Professor Jishen Zhao

2020

The dissertation of Shengye Wang is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2020

DEDICATION

To my parents who have always supported me.

EPIGRAPH

Failure is instructive.

The person who really thinks learns quite as
much from his failures as from his successes.
— John Dewey

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

The journey towards a Ph.D. in computer science is an adventure. I am honored to have many amazing people accompanying me through the expedition that eventually led me to this milestone in my life.

I would like to acknowledge my advisor Professor Henrik I. Christensen for his kind support. I was once lost in my Ph.D. career and almost gave up. Professor Christensen encouraged me, took me as a student, and introduced me to this fantastic robotics field. His mentorship in research is invaluable to me.

I would like to extend my gratitude to Professor Thomas R. Bewley, Professor Ryan Kastner, Professor Scott R. Klemmer, and Professor Jishen Zhao for their valuable comments and feedback as my committee members. They pointed out the imperfections in my work and helped me to improve this dissertation.

I want to thank my labmates: Dr. Vikas Dhiman, Dr. Carlos Nieto-Granda, Akanimoh Adeleye, Sumit Binnani, Andi Frank, Jiaming Hu, Francis Joseph, Po-Jung Lai, Shixin Li, Vasudharini Mannar, Anwesan Pal, Priyam Parashar, David Paz, Yiding Qiu, Michelle Sit, James Smith, Jaskaran Virdi, Quan Vuong, Ruffin White, and Danbing Zhu. I also would like to acknowledge my collaborators outside my group: Xiao Liu, Maximilian Apodaca, Zixuan Wang, Chris Taylor, Devon Merrill, Prof. Michael Taylor, and Prof. Steven Swanson. I am grateful to have productive (and challenging) discussions with these amazing scholars throughout my Ph.D. life.

I would also like to take this chance to express my gratitude to my managers and mentors during the summer internships: Dr. Yusuo Hu, Yang Zheng, Sarah Zhang, Vishnu Emmadi, Dr. Jennifer Bevan, Dr. Moe Tabar, Dr. Ken Nagata, Dr. Li Wan, and Dr. Yang Yu. They offered me exciting intern projects, taught me the best engineering practices in the industry, and helped me with any questions I had.

Chapter 3 contains material from “TritonBot: First Lessons Learned from Deployment

of A Long-term Autonomy Tour Guide Robot," by Shengye Wang and Henrik I. Christensen, which appears in the proceedings of 2018 IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN 2018). The dissertation author is the primary investigator and first author of this paper. The material in this chapter is © 2018 Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE").

Chapter 4 contains material from "Rorg: Service Robot Software Management with Linux Containers," by Shengye Wang, Xiao Liu, Jishen Zhao, and Henrik I. Christensen, which appears in the proceedings of 2019 IEEE International Conference on Robotics and Automation (ICRA 2019). The dissertation author is the primary investigator and first author of this paper. The material in this chapter is © 2019 Institute of Electrical and Electronics Engineers, Incorporated (the "IEEE").

Chapter 5 contains material from "RoboVac: Fault Injection for Long-term Autonomous Service Robots," by Shengye Wang, Xiao Liu, Shixin Li, Jishen Zhao, and Henrik I. Christensen, which is currently being prepared for submission for publication. The dissertation author is the primary investigator and first author of this paper.

Chapter 6 contains material from "Robotic Reliability Engineering: Experience from Long-term TritonBot Development," by Shengye Wang, Xiao Liu, Jishen Zhao, and Henrik I. Christensen, which appears in "Field and Service Robotics: Results of the 12th International Conference (FSR 2019)." The dissertation author is the primary investigator and first author of this paper. The material in this chapter is © 2020 Springer Nature Singapore Pte Ltd.

VITA

2014	Bachelor of Science, Fudan University
2014 - 2020	Graduate Student Researcher, University of California San Diego
2016, 2017, 2019	Internships, Google LLC.
2020	Master of Science, University of California San Diego
2020	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Shengye Wang, Henrik I. Christensen, "TritonBot: First Lessons Learned from Deployment of A Long-term Autonomy Tour Guide Robot", in *Proceedings of 27th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN 2018)*, Aug. 2018.

Shengye Wang, Xiao Liu, Jishen Zhao, Henrik I. Christensen, "Rorg: Service Robot Software Management with Linux Containers", in *Proceedings of 2019 IEEE International Conference on Robotics and Automation (ICRA 2019)*, May. 2019.

Shengye Wang, Xiao Liu, Jishen Zhao, Henrik I. Christensen, "Robotic Reliability Engineering: Experience from Long-term TritonBot Development", in *12th Conference on Field and Service Robotics (FSR 2019)*, Aug. 2019.

ABSTRACT OF THE DISSERTATION

Reliability Engineering for Long-term Deployment of Autonomous Service Robots

by

Shengye Wang

Doctor of Philosophy in Computer Science

University of California San Diego, 2020

Professor Henrik I. Christensen, Chair

Current service robots perform flawless demos under close human supervision, but they often fail when working autonomously in long-term deployments. This dissertation studies the failures in long-term autonomous service robots and proposes methods to improve their reliability.

We built TritonBot, a receptionist and tour-guide robot, as a realistic example to discover the failure modes of a long-term autonomous service robot. TritonBot recognizes people's faces, talks to people, and shows people the labs and facilities in a university building. We deployed TritonBot for hundreds of hours to identify failure modes and common issues on service robots.

Following the experience from TritonBot, we designed two reliability engineering methods to improve the robustness of service robots. First, we found software encapsulation and dynamic

orchestration streamline development workflows and avoid resource contention in service robots. Software encapsulation allows the developers to pack software into self-contained containers and simplify development workflows, and dynamic orchestration schedules the components on demand to avoid CPU/memory resource contention. We developed Rorg, a Linux container-based scheme to manage software components on service robots. Second, we found simulating a broad spectrum of rare failures at system level exposes design flaws and improves the robustness of service robots. Design errors in robotics are challenging to discover due to the need for extensive and resource-demanding testing. Broad-spectrum system-level failure injection exposes both software- and hardware-related design flaws and assists developers in reproducing rare failures, verifying the fixes, and testing the robustness of a robot system. We implemented RoboVac, an extensible and convenient fault injection framework that works at the system level and covers many failure patterns seen in long-term autonomous service robot deployments.

After working with TritonBot for two years and implementing reliability engineering methods, we concluded a few design principles for a long-term autonomous service robot at different levels in the system hierarchy. These design principles guide robust and reliable long-term autonomous service robot designs.

We use a set of automated tools, engineering methods, and design principles to build service robots that are available 24×7, and we call it “Reliability Engineering for Long-term Deployment of Autonomous Service Robots.”

Chapter 1

Introduction

Robotic research and application are advancing at an incredible rate over the last few years. The industry has begun adopting collaborative robots that can work with humans side by side; self-driving, flying, underwater, and space robots are playing more significant roles in commercial applications. We are on the edge of seeing service robots getting ready to assist people in real homes and offices [1]. However, although current service robots perform flawless demos and win applause under controlled settings with an operator's guidance, they tend to fail in prolonged work without close supervision. The lack of long-term autonomy has become one of the major obstacles that prevent more extensive use of service robots in our daily life.

Previous research studied long-term autonomy in navigation [2], deployment [3], and human-robot interaction [4, 5], but few of them investigated the system reliability. With the enhancement of the robot system integration, failures reduce the availability of service robots and thus prevent its scale-up [3, 6]. This dissertation studies reliability engineering in service robots and applies reliability engineering methods to software infrastructures, testing approaches, and robot system design.

1.1 Motivation

While many researchers have built service robot prototypes that work perfectly under close supervision, deploying an autonomous robot in an open environment for a long time is not always trivial. In our early experience with a tour-guide robot prototype, TritonBot, even with relatively decent engineering practice, we still found many reliability issues in several aspects, including hardware, software, networking, human-robot interaction, and so on during its initial deployment [6]. These problems reduced the usability of the TritonBot system, which effectively degraded the autonomy of TritonBot. In addition to TritonBot, other service robot systems suffer from similar issues [2, 4, 3].

Without long-term autonomy, service robots are more likely to hinder than help — besides the inconvenience of service disruption to the end-user, the developer or the user would have to manually intervene in the running of a service robot and take measures to fix it. Unfortunately, the lack of resilience rooted in the research and prototype robot system development — many developers tend to find the simplest method to make the robot "just to work" — and the ad-hoc solutions usually lead to a fragile system that has difficulties in feature iterations and failure analysis. The absence of "reliability thinking" drives the robots away from long-term autonomy, in which the robots continue to work and evolve over an extended period.

Yet service robot reliability engineering did not receive much attention until recently when more service robots start to show up in airports, shopping malls, and hotels to provide services [7]. With research in reliability engineering, we hope to improve the usability of service robots and close the gap between the status quo and the expectation that they can enter homes and offices settings to assist people in their daily lives.

1.2 Thesis Statement

The design of robust robot systems for service applications poses a significant challenge. In this dissertation, the primary hypothesis tested is:

> Software encapsulation and dynamic orchestration streamline development workflow and reduce resource contention in service robots; simulating a broad spectrum of rare failures at system level exposes design flaws and assists developers in improving the robustness of service robots.

The following approach has been pursued to verify the hypothesis:

1. Identification of typical failure modes in service robots that require intervention using a tour-guide robot application.
2. Design of reliability engineering methods for long-term autonomous service robots:
    - Scalable software infrastructure with encapsulation and dynamic orchestration to streamline development workflow, to optimize resources scheduling, and to auto-recover components under failures.
    - Simulating a broad spectrum of rare failures using system-level fault injection to detect design errors and improve the robustness of service robots.
3. Summarization of design principles to improve the reliability of service robot, and evaluation of the benefits of the reliability engineering methods and tools using a tour-guide robot application.

1.3 Problem Scope

Reliability is a broad concept in robotics. Different robot systems have different characteristics and therefore different challenges in reliability. While many problems and solutions in this dissertation apply to different kinds of robots, we focus on *long-term*, *autonomous service robots.*

Service Robots

There are different kinds of robots, including industrial robots, autonomous driving vehicles, unmanned aerial vehicles, underwater vehicles, and so on. Some of these robots execute repeated and identical tasks in structured environments. In contrast, service robots perform a broad spectrum of useful services for the well-being of humans; they work in less-predictable semi-structured environments. Examples of current service robot applications include providing information, guiding tours, delivering packages/food/drink, and others [1]. The variable and diverse usages expand the possible points of failures on service robots, but the all-purpose generalization also relaxes the user's tolerance on the robots.

Long-term Autonomy

Many robots are used for a relatively short amount of time: an underwater robot in scientific research only accompanies a diver on work for a few minutes or hours. Autonomous driving cars deliver passengers a few hours before they go back to a garage daily where engineers have a chance to refuel and inspect. On the other hand, a deep spacecraft needs to work autonomously for years before it reaches its destination.

Although service robots have the chance to charge daily, they are expected to be autonomous for a few months, and should only require maintenance at an interval of years. Without long-term autonomy, a service robot brings more troubles than services. The users will frequently try to operate/fix the robot instead of enjoying its convenience. The robot can update its software and reboot at nighttime (much like the current smartphones); user-doable maintenance task may happen once per a few months, similar to household appliances such as coffee machines; but maintenance tasks requiring professionals should occur much less frequently, as a well-designed automotive only requires a yearly oil change and seldom repair.

Failures Modes

Failure may happen in any aspect of a service robot system, and an all-in-one solution is generally not feasible. Many software failures are transient, detectable, and often recoverable. Hardware failures are detectable but usually hard to be fixed by the robot itself. Other failures, such as human-robot interaction failures, represent individual sub-fields in robotics and requires special handling. This dissertation emphasizes software failures in which appropriate engineering practice can reduce, mitigate, and solve the problems. Besides, we also discuss other detectable failure modes.

1.4 Summary of Contributions

This dissertation makes four major contributions: (1) identifications of typical failure modes in service robots using TritonBot, a realistic service robot application. (2) demonstration that, through software encapsulation and dynamic orchestration, it is possible to streamline development workflow and reduce resource burden for service robots. We implemented the software encapsulation and dynamic orchestration using Rorg, a scalable software management system for service robots. (3) demonstration that, by using fault injection involving hardware, software, and other failures, it is possible to expose design flaws without extensive and resource-demanding testing in service robots. We developed RoboVac, a system-level fault injection framework that simulates a broad spectrum of rare failures to illustrate the use of fault injection in robot systems. (4) summarizing design principles that improve the reliability of service robots.

1.4.1 Identifications of Failures Modes using TritonBot

Existing research covered failure modes of long-term autonomous service robots in human-robot interaction [8, 9], localization and navigation [2], long-term deployment [3], but none of them studied system reliability for a service robot with reasonable system complexity.

We built TritonBot, a building receptionist and a tour-guide robot, to identify the failure modes in long-term autonomy with an emphasis on system failures. TritonBot recognizes people's faces, talks to people, and shows people the labs and facilities in the building.

TritonBot leverages commercial platforms designed for service robots [10] and software stack widely used by many other service robots [11]. It has long-term memory, and it performs multiple tasks (e.g., receptionist and tour guide) without reprogramming. The tour guide task in open space is slightly challenging but not too difficult, which is similar to other tasks performed by service robots today. In general, TritonBot represents modern service robots and their applications in terms of the system architecture.

After a few deployments, we qualitatively and quantitatively studied the failures in TritonBot and learned first lessons in software, hardware, and human-robot interaction [6, 12].

1.4.2 Scalable Software Management and Orchestration

Experience from TritonBot shows two challenges in scaling up robot software: First, when the number of software components increases, organizing, deploying, and monitoring them become tedious and error-prone to the operators. Second, because of the power budget and weight limit, computationally intensive software often exhausts all the computing resources in an on-robot embedded computer.

Similar platforms have similar challenges but different solutions. Although smartphone platforms such as iOS and Android are also size- and weight-constrained, they only need to keep a single independent user-facing and can easily stop inactive non-foreground tasks. Datacenter applications are composed of several components that depend on each other like service robots, but scaling-up in datacenter mainly means spawning more instances of the same program to handle requests from the users while scaling-up in service robots means adding more features to the system. Because of the multi-component nature and limited computing resources, scheduling strategies in iOS and Android from the smartphones, as well as management tools like Borg [13], Mesos [14] and Kubernetes [15] from the datacenters, do not apply to service robots.

Software encapsulation and dynamic orchestration were rarely used in the robotics community due to the lack of a systematic approach and an appropriate tool. To enable scalable software management in service robots, we implemented software encapsulation and dynamic orchestration on service robots and designed a software infrastructure “Rorg” that orchestrates, monitors, and auto-recovers software components. Rorg integrates with ROS [11], the de facto standard robotic middleware. It leverages Linux containers to encapsulate and organize software. Using dynamic orchestration, it decreases on-robot computer resource usage without introducing noticeable latency in the robot’s actions. In addition to methods and tools to automate development workflow and optimize resource scheduling, this work also provides experience and quantitative results showing the benefits of automation in the software system of service robots.

The experimental results from Rorg show that software encapsulation and dynamic orchestration in service robots reduce maintenance burden and avoid resource contention. The principles underneath in Rorg, including self-contained component encapsulation and automatic life-cycle management with dependency resolution, ease maintenance burden and save computing resources. These principles are useful not only in service robots but also in similar computer systems composed of numerous simultaneously running tasks that have limitations in computing resources.

1.4.3 Broad-spectrum System-level Failure Simulation

Failure propagation happened during the TritonBot deployment led us to a lesson that many service robots are not prepared for failures. Design errors in robotics are difficult to discover due to the need for extensive and resource-demanding testing.

Fault injection is a conventional technique to reproduce rare errors in normal operations. While simulating errors was common in software engineering, replicating a broad-spectrum of errors, including hardware failures, is a less-typical practice in robotics. Existing fault injection methods in pure software systems only consider input/output in computer media such as disk and networking, but service robots interact with the physical environment. Previous works in robotics inject failures to individual components like navigation [16, 17, 18], or they only work in simulations [19]. However, no existing effort covers a broad spectrum of failures at the system level that usually happen in a real service robot deployment.

We illustrated the advantage of broad-spectrum system-level failure simulation for service robots by implementing RoboVac, a fault injection framework with broad coverage of failure patterns seen in long-term autonomous service robots. RoboVac is a toolbox that applies various methods to inject general failures to software processes, software communication layers, and networking in addition to individual components like robot actuators and navigation stack. RoboVac generates a broad spectrum of failures, which helps developers to observe the robot's behavior under failures, to discover unexpected design errors, and to benchmark a robot system's resilience. We applied RoboVac to TritonBot; it covered all of the failure patterns we have seen in previous deployments, it exposes hidden design flaws that otherwise are hard to discover, and it helped the developers to verify the fixes and test the robot's robustness.

Through the RoboVac experience, we verified that system-level fault injection could replicate a broad spectrum of erroneous conditions in cyber-physical systems with only minor changes to an existing system. These injected failures expose design flaws and allow the developers to understand the robot's behavior under unexpected conditions, verify fixes, and prepare the robots for future failures.

1.4.4 Summarization of Design Considerations in Service Robots

Previous research concluded a few design guidelines for building a robot system using ROS [20]. From our two-year TritonBot deployment, we summarized the design experience and principles that will help developers to build service robots with the autonomy of months and/or years.

During the past two years, we regularly do maintenance, fix issues, and roll out new features on TritonBot. We improved the TritonBot system with more robust human detection, speech interaction, logging, and so on. We summarized the deployment history and the issues that occurred during the deployments. We identified reliability engineering challenges in three aspects of long-term autonomy: scalability, resilience, and learning. We formulated various techniques to confront these challenges, and we applied the above methods, including infrastructure engineering and fault injection to TritonBot. We concluded a list of practical design principles and rules that we can use as a "toolbox" to guide a robust and reliable long-term autonomous service robot design. From the case study of a tour-guide robot, TritonBot, this dissertation shows that reliability engineering reduces manual interventions and increases the availability of service robots.

1.5 Outline of the Dissertation

The dissertation is structured as follows: Chapter 2 discusses the background of reliability engineering and related work in service robots. Chapter 3 presents the design of TritonBot, a realistic example of service robots. We summarize the early lessons observed in TritonBot. Chapter 4 discusses scalable software management in service robots and presents Rorg, a software management system that leverages Linux containers to deploy and monitor software for service robots. Chapter 5 discusses using broad-spectrum system-level fault injection to test service robots and presents RoboVac, a fault injection framework, for testing and improving the reliability of service robots. Chapter 6 presents the full deployment history of TritonBot and summarizes design considerations and principles that improve the reliability of service robots. Chapter 7 discusses general reliability engineering and future works in service robots, and it concludes this dissertation.

Chapter 2

Background

There are previous research and engineering efforts in all of the reliability engineering, long-term autonomy, and various service robots. This section investigates the background and related works of these fields.

2.1 Long-term Autonomous Service Robots

A few pioneering long-term autonomous systems share similar goals with this dissertation, for example: CoBots, STRANDS, BWIBots, and the classic Sage.

CoBots [21], the collaborative robots from Carnegie Mellon University is a long-term autonomy system with multiple mobile robots that serve users in a building. They have traveled over $1,000$ km in total. Targeting at building-wide, the CoBot robots project intensively study long-term mapping, localization, and navigation under an unstructured environment. They receive tasks from a website interface and use a touchscreen to interact with people, but they actively seek help from a human when the tasks are beyond their capability, such as pressing an elevator button or making coffee in the kitchen [22]. The developers studied the robustness of the system using the number of interventions and the mean distance (in km) traversed between interventions.

The European STRANDS project [3] deployed four robots for security and health-care scenarios. They gather knowledge about a human-inhabited environment. With auto-charging capability, the robots reached a few weeks of autonomy with a single deployment, and the long-term study continued for years. The STRANDS robots have two roles: security and care. The security robots monitor an indoor office environment, and they generate alerts after detecting prohibited or unusual events. The care robots work in an elderly care facility; they guide visitors, provide information to residents, and assist in walking-based therapies.

BWIBots [4] is a custom-designed multi-robot platform for AI, robotics, and HRI that aims to be an always-on, permanent fixture in the University of Texas at Austin Computer Science building. They listen to human's speech from afar, ask questions and understand human intentions, and also execute a sequence of actions to complete user's requests.

The classic Sage [23] was a long-term autonomous tour-guide robot deployed in 1998. Sage worked in a natural history museum and promoted many smaller but equally important exhibits in the dinosaur hall. Even with many people (including many children) interacting with it, Sage achieved 135 autonomous, error-free, and totally unsupervised operation days. The developers summarized the deployment statistics, bugs and issues, and the improvements throughout the Sage deployment. Sage provided a snapshot of the early tour-guide robots. Although the sensors are quite similar to the tour-guide robots nowadays, software technologies such as localization, navigation, and voice recognition and synthesis have significantly improved since then.

These research projects studied robotic reliability from different perspectives. CoBots focused on mapping, localization, and navigation. STRANDS emphasized very-long-term deployments but with a potentially more structured task such as security patrol. BWIBots has the center of human-robot interaction. Sage was an early attempt to study robotic reliability 30 years ago, but the algorithms, tools, and best practices for robotic systems have developed significantly since then. This dissertation studies the reliability of long-term autonomous service robots from a system perspective.

2.2 Tour-guide Robots

Tour guiding had been a practical scenario to evaluate many technologies in robotics. Tour-guide robots in history provide valuable experiences in robot design and deployment. Table 2.1 listed a few highly-recognized successful tour-guide robots.

Sage [23] worked in the Carnegie Museum of Natural History for 174 days, and it achieved 135 error-free days. RHINO [24] worked at Deutsches Museum Bonn; it moved at 0.8m/s, and it proactively found people to engage them. Minerva [25] was deployed in the Smithsonian Museum of American History for 14 days, traveled 44 km, gave 620 tours, and showed 2600 exhibits. Pearl and Flo [26] provided guidance for elderly people in an elderly living facility. Robox [27] was a robot installed in the Swiss National Exhibition expo.02 with unique multi-modal interaction capabilities. Jinny [28] guided visitors from a hallway to a room in a research facility, but it targeted for museums. RoboCart [29] was a shopping assistant that helps visually impaired customers navigate a grocery store and carry goods. Urbano [30] was an interactive tour guide in exhibitions; it was also used to demonstrate and test autonomous navigation algorithms. Robotinho [31] is a humanoid robot that entertains people with intuitive, multi-modal interaction. The nine TOOMAS robots [32] traveled together 2187 kilometers in three different home improvement stores in Germany; they guided 8,600 customers to the locations of their products of choice. CATE [33] was an outdoor tour robot that moved along a series of waypoints marked by RFID tags. Konrad and Suse [34] were tour guides in a multi-floor building at Ilmenau University of Technology; they were able to track people using the leg, face, motion, and upper-body detectors. Obelix [35] was an outdoor robot that works in crowded urban areas; it was able to travel 3 km autonomously through the city center of Freiburg. SPENCER [36] was a robot deployed Schiphol airport that guided passengers from passport control point to local connections gates. A health-check robot [37] in a hospital guided users through a list of scheduled examinations; it leverages a remote, centralized schedular to optimize examinations for different

Table 2.1: List of well-known tour-guide robots.

Name	Environment	Sensors	User Interface
Sage [23]	Museum	Sonar, Camera, Tactile	Push-buttons, Display, Speech
RHINO [24]	Museum	Laser, Sonar, Camera, IR, Tactiles	Push-buttons, Display, Sound, Web UI
Minerva [25]	Museum	Laser, Sonar, Camera	Touch-screen, Speech, Robot face, Web UI
Pearl and Flo [26]	Assisted Living Facility	Laser, Sonar, Camera, Microphone	Touch-screen, Verbal, Robot face
Robox [27]	Exhibitions	Laser, Camera, Microphone, Tactiles	Push-buttons, Verbal, Robot face
Jinny [28]	Museum / Office	Laser, IR, Microphone, Tactiles	Touch-screen, Push-buttons, Verbal, Robot face, Robot arm
RoboCart [29]	Shopping Mall	Sonar, RFID	Keypad, Speech
Urbano [30]	Exhibitions	Laser, Sonar, Microphone	Touch-screen, Verbal, Robot face, Arm, Web UI
Robotinho [31]	Museum	Laser, Sonar, Camera, Microphone	Verbal, Robot face, Arm
TOOMAS [32]	Home Improvement Store	Laser, Sonar, Camera, Microphone, Tactiles	Touch-screen, Speech, Robot face
CATE [33]	University (Outdoors)	Sonar, IR, RFID	Touch-screen, Sound
Konrad and Suse [34]	University Building	Laser, Sonar, Camera, Microphone, RFID, Tactiles	Touch-screen, Speech, Robot face
Obelix [35]	City (Outdoors)	Laser, GPS, Camera	Touch-screen
SPENCER [36]	Airport	Laser, RGBD	Touch-screen, Boarding pass reader
Health-check Robot [37]	Hospital	Laser, Camera	Touch-screen, IC card reader
KeJia [38]	Shopping Mall	Laser, Camera, Microphone	Verbal, Mobile App, Robot face, Arm

users. Kejia [38] is a shopping assistant in shopping malls that is robust to challenges in dynamic environments; it was deployed for 40 days and served 530 customers.

2.3 Reliability Engineering

Reliability engineering is not a new concept in engineering; it is a critical study that keeps an engineered system reliable, available, maintainable, and safe [39]. Some companies in the industry even have created a *Site Reliability Engineer (SRE)* role in supporting growing Internet businesses [40]. Birolini summarizes theories and provides qualitative approaches to the study of the reliability, failure rate, maintainability, availability, safety, risk, quality, cost, liability, and so on [39]. O'Connor et al. further give field-specific reliability engineering examples of mechanical systems, electronic systems, software, design for reliability, manufacturing, and more [41]. Beyer et al. from Google discuss "site reliability engineering" and the principles and practices that keep Google's planet-scale production system healthy and scalable [40]; they combine automated tools and appropriate engineering and emergency response workflows. Recovery-oriented computing, or ROC, shifted the concerns from the traditional mean time between failures (MTBF) to mean time to repair (MTTR) [42]. By taking recovery under failures into primary consideration, the researchers showed ROC reduced recovery time and increased availability in five case studies. Program analysis is another approach to test the computer system and find potential issues; these analyses include static compiler instrumentation [43], symbolic execution [44], dynamic binary instrumentation [45], and so on.

Although reliability engineering is studied in many other fields, the requirements and restrictions in these fields are different from service robots. In datacenters, applications require availability as high as 99.99%, but they have access to almost infinite computing resources, and on-call engineers supervise the system [40]. In the automotive industry, the ECU performs a small fixed set of tasks during its life cycle, which makes standard practice and testing methods possible [46]. For space applications, the most significant challenge comes from radiation that affects electronics reliability [47]. As a result, few of the works mentioned above are directly applicable to service robots given the gap between traditional computer systems and cyber-physical systems; yet despite the disparity between planet-scale datacenters and mobile robot platforms, successful engineering practices in traditional computer systems have considerably inspired robotic reliability engineering.

2.4 Challenges in Long-term Autonomous Service Robots

Reliability engineering for service robots is a less explored area. It has its unique challenges and opportunities.

First, service robots need many components to perform perception, recognition, navigation, reasoning, and other tasks. Their system architectures are as complicated as some datacenter applications. Design principles and testing methods in some more straightforward but critical systems like automotive computers [46] are not directly applicable to service robots. Second, service robots carry limited computing resources, and they can not offload many critical or latency-sensitive tasks to external devices. As a result, although redundancy proved to be a helpful approach to improve the reliability of computer systems [40], it is hardly applicable to service robots. Third, service robots are changing and developing systems that meet new environment, get updates, and learn new knowledge over time, which makes full test coverage difficult. On the other hand, task-specific robot systems are less dynamic, and demo robots only need to work for a short amount of time; therefore, extensive but straightforward tests can reveal most of the issues on them.

Still, service robots have their opportunities. The recent service robots are not performing life-critical tasks. Therefore, they can tolerate more unavailability (a few minutes per day) without majorly affecting user experience with proper communication. It can sometimes seek some help from humans, although too much human intervention defeats the purpose of service robots. In general, for long-term autonomous service robots in the near future, reliability engineering includes both reducing failures and living with failures.

Chapter 3

TritonBot - A Long-term Autonomous Service Robot

While service robots usually perform perfect demos with close human supervision, they often fail when working by themselves for an extended period. To study failure modes and human-robot interaction patterns in long-term deployments, we built TritonBot, a long-term autonomous robot working as a building receptionist and a tour guide. It recognizes people's faces, talks to them, and guides people to the labs and facilities in an office building. TritonBot does not only serve the users and visitors of the building, but it also provides a portable toolbox for long-term autonomy and human-robot interaction research. This chapter presents the design of TritonBot and the lessons we learned from the first-month deployment with respect to technical and human-robot interaction aspects. In the following chapters, we will use TritonBot as a platform to study various aspects of service robot reliability engineering.

In its initial one-month deployment, TritonBot worked for 108.7 hours, actively interacted with people for 22.1 hours, greeted people 2950 times, guided 150 tours, and traveled 9.9 kilometers. Later deployments provided more experience and lessons from long-term autonomous tour-guide robots. We share the components of TritonBot using an open license to help the community to replicate the TritonBot platform and inspire long-term autonomy and human-robot interaction research.

3.1 Introduction

Robot technology has advanced significantly over recent years, and we have finally reached a milestone where we can apply these innovations in real-world applications. However, while robots often perform flawless demos and win applause under controlled settings with an operator's guidance, they frequently fail when working for an extensive length of time in unstructured environments without close supervision [1]. As discussed in the previous chapters, long-term deployment adds more uncertainties and covers many corner cases in the technical components or human-robot interaction behaviors. Long-term autonomy had become a significant challenge that prevents service robots from entering an office or home environment to assist people.

We present TritonBot, a long-term autonomous robot deployed in an unsupervised open environment. TritonBot works as a receptionist and a tour guide of a university office building. It recognizes people's faces, talks to people, and shows people the labs and facilities in the building. However, TritonBot is not only another tour-guide robot: In addition to serving the visitors to the building, the system is used to discover failure modes in a long-term deployment of a robot that actively interact with people in an open environment. Besides, TritonBot is also used in other research projects to discover the short-term and long-term interaction patterns between the robot and humans since the robot receptionist engages in many interactions every day. Figure 3.1 shows TritonBot and its variant, BoxBot. Despite the difference in the shapes, BoxBot runs the identical software and exhibits the same behavior as TritonBot.

In the first month deployment, TritonBot and BoxBot received much attention from visitors: They have been serving the guests for 108.7 hours in total, actively talked and walked

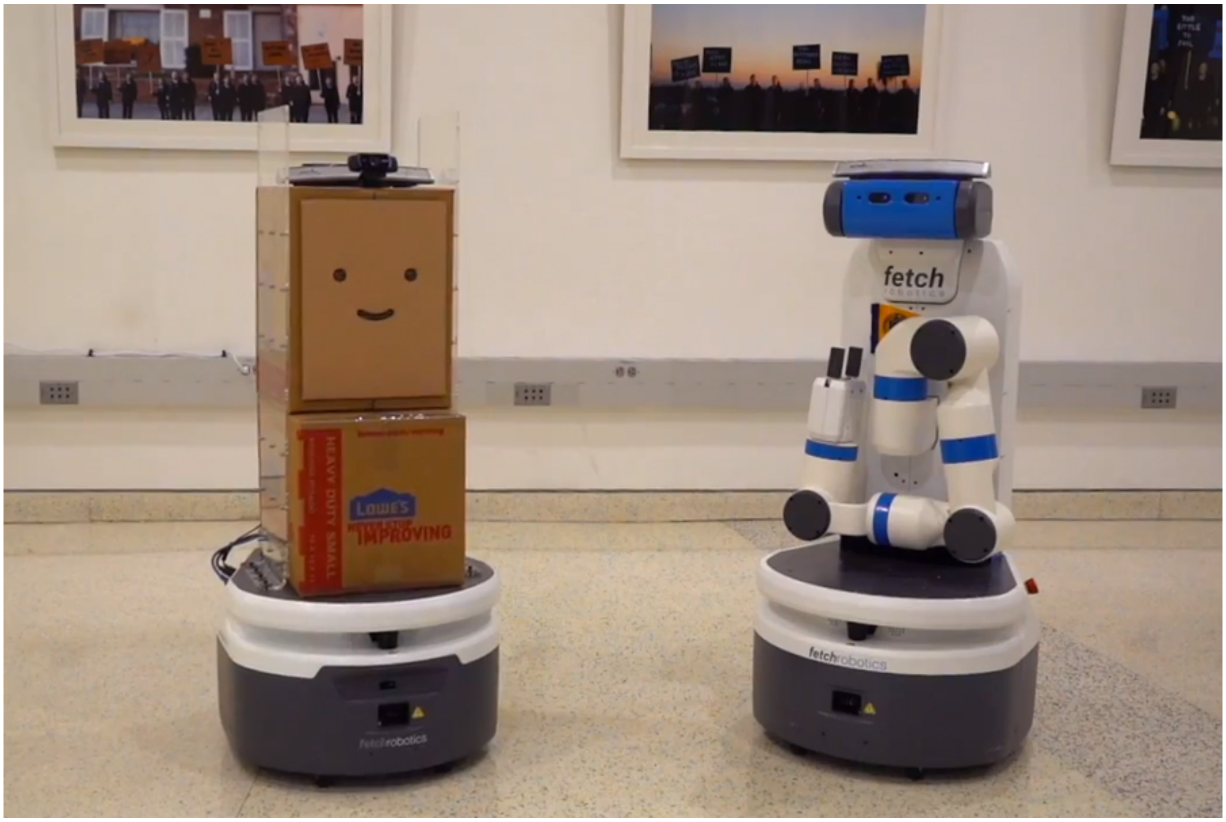


Figure 3.1: A picture of TritonBot and its variant, BoxBot. They take turns and work every weekday, chat with people, and navigate with people to introduce places of interest in an office building. TritonBot (right) is based on *Fetch Research Edition* [10], and BoxBot (left) is a *Freight* robot with an upper-body made of cardboard boxes.

with people for 22.1 hours, provided 150 tours, traveled 9901.1 meters, have been listening for 4.0 hours (2,616 counts) and speaking for 10.7 hours (14,037 counts). While this chapter focuses on the initial TritonBot design and deployment, later improvement and deployments discovered more issues and concluded design principles for long-term autonomous service robots (Chapter 6). Unlike other long-term autonomous robots, TritonBot is a standalone service robot system that actively engages people and draws people's attention. It interprets users' intent only using voice interaction and talks back to people. With the deployment, we were able to study potential issues in resilience, scalability, and learning of a robot system in a dynamic environment. We released the components in TritonBot under an open-source license to enable the community to replicate our platform for long-term autonomy and human-robot interaction (HRI) research.

This chapter is structured as follows: Section 3.2 discusses some existing long-term autonomous robots and other related work in addition to those in Chapter 2. Section 3.3 presents TritonBot's behavior, components, and its ability to be used as a toolbox for HRI studies. Section 3.4 discusses the lessons we learned in both technical and human-robot interaction aspects of TritonBot from the first month of deployment. Section 3.5 presents a quantitative analysis and statistics on TritonBot's performance based on the initial deployment, and Section 3.6 concludes this chapter.

3.2 Related Work

TritonBot shares its goals with a few pioneering long-term autonomous systems, including Sage [23], CoBots [21], STRANDS [3], and BWIBots [4]; previously Section 2.1 discussed the highlights about these long-term autonomous service robots. Early tour-guide robots like RHINO [48] and Minerva [49] provided interactive tours to visitors in museums with an emphasis on robust localization and navigation in a crowded environment; Section 2.2 listed many successful tour-guide robots in the history.

Interactive robots have appeared in many scenarios other than guiding tours. Valerie [50], the "Roboceptionist" at Carnegie Mellon University, is an early robot with a personality that exhibits social competence and remains compelling to interact with for an extended period. Kanda et al. deployed Robovie robots in a corridor of a shopping mall for five weeks to provide information to the public [51]. Komatsubara et al. use the Robovie robots in an elementary school for five weeks with the intent to increase the curiosity in the science of children [52]. The robot recognizes the face and asks questions to the students, but it requires an operator's intervention for speech recognition. Bohus et al. created a stationary directions robot to study open-world human-robot interaction [5], and they concluded several failure modes [8]. Chung et al. studied the Savioke Relay robots in hotels and created a fast-prototyping tool to program the robots [7].

Tonkin et al. studied social robots at airports and discussed design methodology for designing communication pattern for interactive robots [9].

TritonBot shares a few similarities with these pioneers, but also has its specialty. TritonBot only interacts with a human through speech; therefore, it receives many unexpected instructions, and they help the researchers to understand people's free expectations on a robot. Also, TritonBot has long-term memory, recognizes people, and interacts with people every day, so it can collect and analyze communication patterns and histories, which makes it a toolbox for future systems and HRI studies.

3.3 TritonBot Design

The TritonBot operates in the hallway of an office building, discovers passersby with face detection and interacts with them using speech. This section describes the scenario of TritonBot deployment and the robot's behavior, components, and its capabilities as a research toolbox.

3.3.1 The Tour Guide Scenario

TritonBot works as a receptionist and a tour guide in Atkinson Hall, which is a six-story, $250,000$ sq. ft. building at the University of California San Diego. The building is an office building for regular users, but sometimes hosts meetings, conferences, or events. Figure 3.4 shows a map and the surroundings of the area where the robot is deployed. TritonBot stands in the hallway of the building facing the entrance at an angle, the angle sets the camera away from facing the entrance where excessive outdoor sunlight is coming from. TritonBot continuously detects human faces in its view. When it sees a person, it actively engages the person by greeting him or her, but it gives up if the person disappears before it finishes a self-introduction monologue. Figure 3.2 shows a visitor interacting with TritonBot.

If TritonBot finds a match of a recently seen face with a record in its face database,

Figure 3.2: A visitor interacting with TritonBot.

it will greet the person by name. Otherwise, it starts asking for name and offers a few trivia questions. Figure 3.3 shows an example of the interaction between TritonBot and a visitor. The quizzes are simple, but they provide a time frame when the visitor is looking at the robot. TritonBot continuously collects face samples while talking to the person and uses the highest-quality face samples to update its face database. During the entire interaction, TritonBot repeats the questions if it does not receive an response in time, but it just gives up after three retries. Afterwards, TritonBot offers a guided tour to the user. If the user accepts the tour, the robot will show the visitor around and introduce a few places of interest in the building, including a fast-prototyping facility with 3D printers, a robot showroom (robot zoo), a smart home demo room, and a gallery (Figure 3.4). If the robot fails to move to a target, it politely asks people to step back because it “needs personal space to maneuver.”

We try to keep TritonBot autonomous as much as possible to study failure cases. TritonBot

TritonBot: Good morning. My name is TritonBot. I am a robot working here at the Contextual Robotics Institute. May I ask what is your name?
Human: My name is Daisy.
TritonBot: Nice to meet you, Daisy. I prepared some quiz for you. Would you like to give them a try?
Human: Sure.
TritonBot: Where does the president of the United States of America live?
Human: The White House.
TritonBot: You got it! The president of the United States of America lives at the White House in Washington DC. Do you want to try another question?
Human: Yeah.
TritonBot: What is the most popular sport throughout the world?
Human: I don't know.
TritonBot: No worries. Soccer is the most popular sport throughout the world. Do you want to try another question?
Human: No, thanks. I'm good.
TritonBot: Thanks for chatting with me. I feel honored. I am a tour guide. Can I show you around?
Human: Yes, please.
TritonBot: Glad to hear that. Please follow me.
(TritonBot moves towards the Prototyping Lab.)
TritonBot: We are now in the Prototyping Lab. The prototyping lab provides prototyping and engineering services to UCSD researchers, including embedded electronics, mechanical devices, robotic contraptions, real-time networking software, and 3D printed structures.
(TritonBot moves towards the Robotic Zoo.)
TritonBot: Here is the robot zoo. You can see my fellow robot, Diego-san. He can see people, understand gestures and expressions, and even learn from people. Diego-san is not the only robot here, many of my other fellow robots are coming very soon.
(TritonBot introduces a few other locations and stops at the elevators.)
TritonBot: Here is the elevator, and it is our last stop. Thanks for letting me show you around. Please enjoy your stay. Take care and have a great day.

Figure 3.3: A dialogue between TritonBot and a visitor. TritonBot starts the dialogue when it sees a face, but gives up if the face disappears during its self-introduction. When TritonBot matches a face with an acquaintance, it will greet the person with the name and skip the trivia questions.

is programmed to go to a charging dock when the battery is low, and it returns to work when it finishes charging. However, in the early deployment, we closely monitor the robot to prevent any accidents: A student always sits in the "smart home" room when TritonBot is actively working. He monitors the robot and manually overrides the robot control in case of emergency. Although the student never engaged the emergency stop in the first month, the working time of the robot was limited to the student's schedule. Figure 3.5 presents the simplified representation of the state machine that controls the behavior of TritonBot.

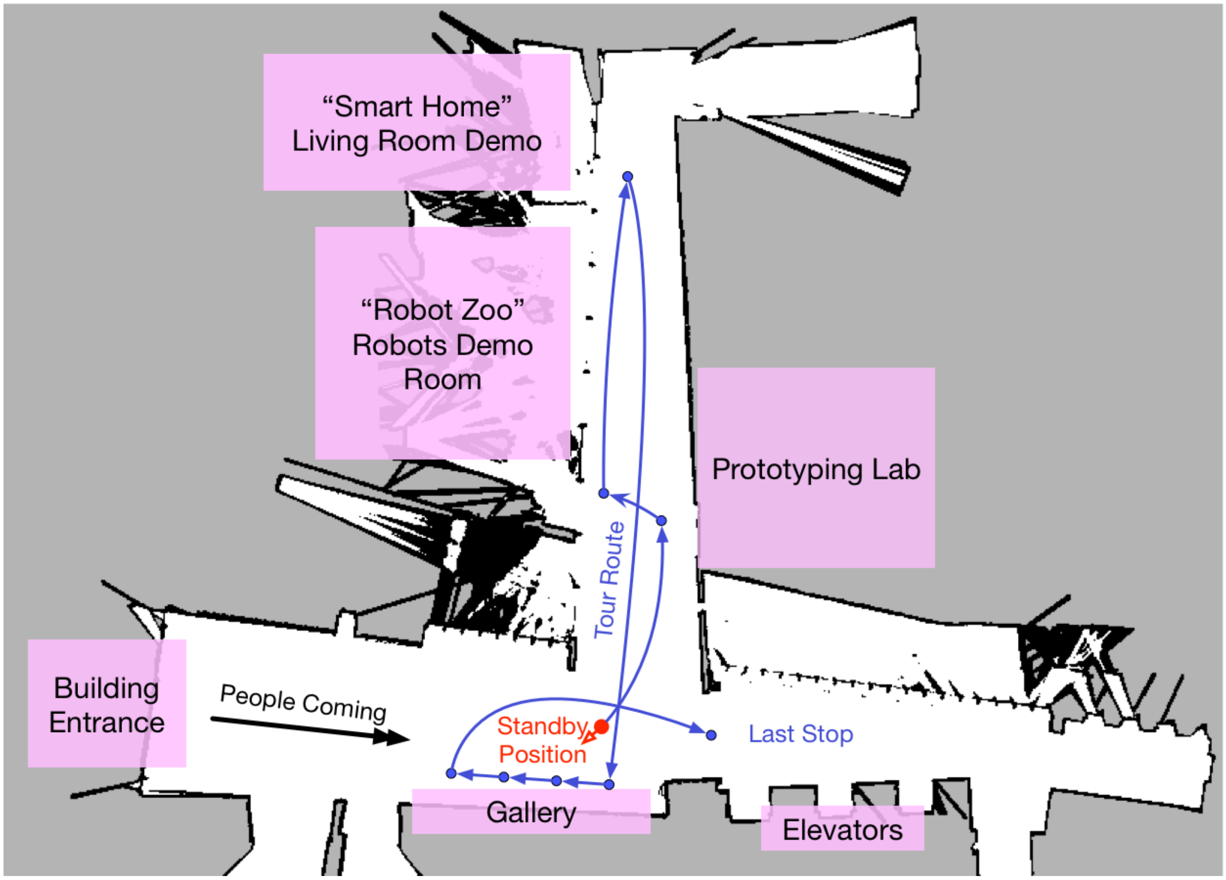


Figure 3.4: A map of the hallway in the office building where TritonBot is deployed. The robot keeps a standby pose when it is waiting for people. The blue trajectory represents the route that the robot guides people, and the blue vertex indicates the places of interest where the robot will stop and introduce.

3.3.2 TritonBot System Overview

TritonBot is based on the *Fetch Research Edition* platform from Fetch Robotics Inc. [10]. we also built a TritonBot variant, BoxBot with a more economic *Freight* platform from the same manufacturer with a custom cardboard box torso and head. The cardboard box upper body allows researchers to change the design or adding new sensors to the platform with a minimal cost. Despite the differences in shapes, the related hardware features and system architecture of the two robots are identical. These two robots take turns in the deployment, which opens time windows for hardware maintenance and software upgrade.

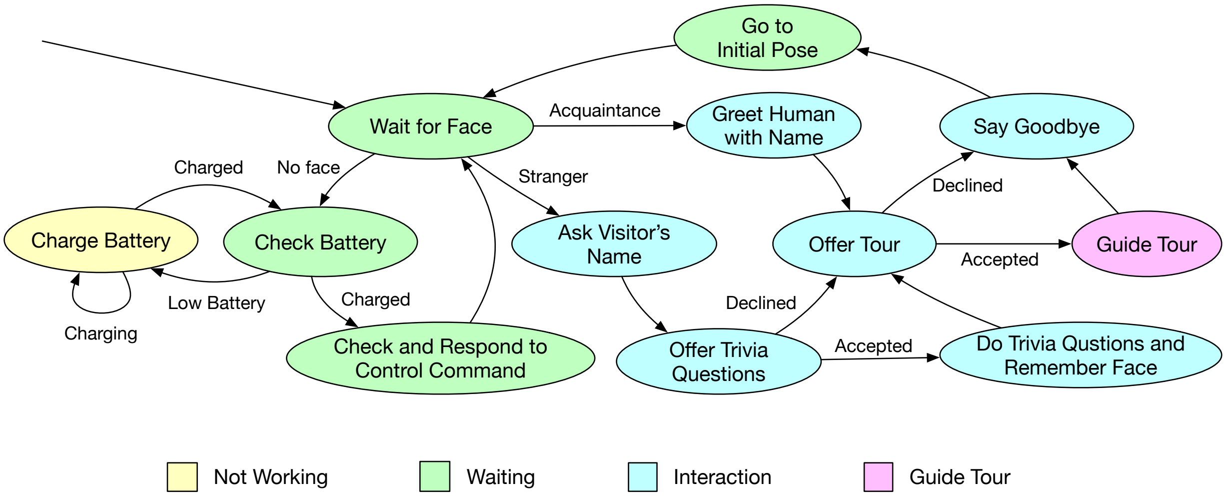


Figure 3.5: The state machine that represents the behavior of TritonBot. Different color indicates different state categories. The time spent in each category is shown in Figure 3.9.

The robot has an RGBD camera with about 70° field of view, a microphone array with sound source localization support, and a loudspeaker. We installed a back-facing laser scanner at the opposite side of the stock laser scanner at the leg level. The base of the robot contains two differential drive wheels and a consumer-grade computer (Intel i5-4570S CPU, 16 GB memory, 1 TB solid-state disk); two lead-acid batteries keep the robot running for about 6 hours with a single charge.

TritonBot utilizes many third-party components and open-source software to accomplish its task. It is running Robotic Operating System (ROS) Indigo version (released in 2014) [11]. Openface [53] pipeline detects human faces in the pictures captured by the camera and converts these face images to embeddings that are used to calculate the similarity between two faces. Cartographer [54] provides localization and mapping for the TritonBot platform, and the ROS navigation stack "movebase" [55] controls the robot to move around. Cloud service comes with excellent help when an open-source alternative is not available or does not meet our needs. Google Cloud Speech API [56] translates speech audio to text with nearly real-time feedback. TritonBot used an Android tablet for voice synthesis initially, but we switched to a commercial solution later because of its superior performance. To summarize, we integrated many existing components to build TritonBot, but we wrote about $100,000$ new lines of code (mostly C++ and Python) to build the entire TritonBot system.

3.3.3 TritonBot for Human-Robot Interaction Studies

TritonBot works closely with people, and the long-term deployment allows it to meet people and interact with them over a long timespan. Although the early deployment of TritonBot is targeting long-term autonomy research only, and is not recording any personally identifiable information (PII), we believe TritonBot is capable of distinguishing people and studying the human-robot interaction behaviors.

Working as a robot receptionist, TritonBot starts interacting with a person when it sees a face, which could provide an estimation of the age, gender, or other background information of a person [57]. Since TritonBot works in an office building, it can see a relatively fixed set of people. In a long-term deployment, the robot can observe the change in the way that the same person interacting with the robot. The robot can also observe a general group of people and study their interaction with the robot.

The tour guide scenario is a test field for social-aware navigation research. TritonBot observes the environment using a laser scanner, a microphone, and a camera, which provides an estimation of the number of the humans around and their pose [58]. While a user is following TritonBot during a tour, the robot observes the relative location between itself, the person, nearby obstacles, and incoming people, which allow the robot to learn the rules in social-navigation. Although people's curiosity about the robots dominates the interaction pattern in the early deployment, we expect to see a noticeable difference in the interaction pattern of TritonBot and BoxBot in the future and improve the robot shape and design.

TritonBot is designed as a portable and replicable HRI research platform, and we open-source TritonBot to provide a research platform for HRI and long-term autonomy researchers. Most components we built for TritonBot are open-sourced either as ROS packages or standalone components and are available at https://github.com/CogRob/TritonBot.

3.4 Lessons from Initial Deployment

In a month-long deployment of TritonBot, we have seen people enjoy chatting and walking with a robot, but we have also seen many shortcomings in the system. These shortcomings include failures and faults, improper or inefficient robot operation workflows, and imperfections in human-robot interaction behaviors. This section presents the main lessons we learned from the first-month deployment of TritonBot.

3.4.1 Hardware Failures

Just like any other engineered system, failure is unavoidable in robotics. In contrast to software errors, hardware failures are usually uncommon, but the long-term deployment extensively tests the system and reveals failure cases. During the development and deployment of TritonBot, we encountered a few hardware issues:

Battery failure. During our early deployment, the BoxBot over-discharged its batteries (two 12V lead-acid batteries), and the failed batteries could only support the robot for 10 minutes (as opposed to 12+ hours of standby time in normal condition). We replaced the faulty batteries and set up a battery monitoring program to prevent battery from over-discharging.

Electrical part failure. The charging port on TritonBot failed, and the robot attempted many times going back and forth trying to dock itself to a charging dock when the battery was low. The student monitoring the robot discovered the unusual behavior and shutdown the robot program. After that, we programmed the robot to shutdown itself when its battery is low and auto-charging fails.

Mechanical failure. The connection between a caster and the base on TritonBot became loose during long-term but normal operation. We fixed the issue and notified the manufacturer.

Device driver failure. The robot computer fails to recognize a USB camera sometimes during the deployment, and reconnect the USB connector did not solve the issue. The only known working solution was to reboot the internal computer, and we ended up replacing the camera with another model.

Since hardware errors are usually hard to recover from automatically, fail-safe is a critical design principle: the robot should detect failures and take actions to prevent failure propagation or further damage. Component-level monitoring is helpful sometimes: A battery voltage monitor will prevent the batteries from over-discharge, and a retry limit on the docking action and a vibration sensor will prevent further physical damage to the robot.

Because hardware malfunction is less usual, they are easy to ignore and not covered by most test cases. In fact, the TritonBot control software was programmed to give up and report docking failure after five docking attempts, but the higher-level control software immediately restarted docking because the battery-level was still low. The hardware failures lead us to the need for extensive testing, which requires a fault injection/simulation mechanism to produce/reproduce potential errors.

3.4.2 Network Connectivity

TritonBot accesses the Internet via a wireless connection to reach cloud services, receive commands from operators, and report status to the monitoring station. Since it roams in a large area of the building and a campus-wide wireless network already exists, setting up an ad hoc wireless network for it is not worthwhile. Using an existing wireless infrastructure with multiple access points and broad coverage leads us to a situation that (1) signal strength is not consistent in all the locations, (2) the robot sometimes loses connection when it roams between multiple access points, (3) network is not secured.

We took a few measures to confront these issues. First, we did a signal strength survey in the building and excluded blind spots from the tour path. Besides, the robot checks network connection every minute and resets its wireless interface when there is a failure. We also installed a wireless access point on the robot to provide “escape hatch” access to the robot in case of unrecoverable connection failures. Furthermore, we set up a firewall on the robot to block unexpected incoming connections from the Internet.

3.4.3 Software Failures

In a developing system, software errors are frequent when rolling out new features, improvements, or bug fixes. While most of the software flaws disappear in the testing stage, the remainders are usually hard to discover or even to reproduce. However, they are easy to recover from by restarting the program, and the loosely-coupled system architecture in ROS enables such recovery mechanism without causing a system-wide outage. Nevertheless, this mechanism requires coordination between the software components. From our experience, we have found two design principles are particularly useful to tolerate and recover from rare and transient software errors:

1. Any component must tolerate temporary unavailability of other components and should not propagate the failure. They should resume working if the depending component recovers.
2. Restarting a program shall always help it to enter a clean and steady state. If a component fails in some way and cannot recover, it is better to terminate and restart the program than to remain in an unknown and unresponsive state.

These two principles allow the system to tolerate transient errors. In a typical scenario, the erroneous programs terminate and return to a normal state; meanwhile, the counterparts will wait until they recover rather than propagating the error. Moreover, these design rules decouple the programs at the application level, which allows programs to regularly restart when the system is idle, as Google points out that “a slowly crash looping task is usually preferable to a task that hasn't been restarted at all [40]." As a bonus, these design principles also enables dynamic updating of the system at the cost of temporary and short unavailability.

These two principles helped the robot running stable for most of the time. But a slight violation of the first principle caused unavailability of our system that it could not automatically recover from: One day, the behavioral control component tried to communicate with a logging service that timed out, and the behavioral control stalled. The same symptom happened three times during the entire month deployment, and we discovered the error message in the last occurrence and fixed the control state machine.

This incident leads us to another lesson: we assumed the reliability of some software components (the logging service in this case) since they rarely fail. We are developing a mechanism to inject errors into software components, which will help discover potential design flaws.

3.4.4 Software Deployment

TritonBot's ROS system is running about 50 nodes when it is working, and more than half of them are in-house customized software. Initially, we ran these programs directly under a Linux system, but this method does not scale. As we were adding more components to the system, software dependencies started to conflict with each other, runtime configurations scattered at multiple locations in the file system, and software versions were hard to track. After testing several solutions, we ended up using Linux containers (Docker [59] specifically) to manage the robotic software in TritonBot.

Linux containers provide operating-system-level virtualization and allow building and shipping applications easily as self-contained archives. Pioneer work introduced Linux containers technology to robotics [60], but to the best of our knowledge, TritonBot is the first robotic system that uses Linux containers to manage the entire robotic software stack. Every TritonBot robotic software component is running inside a container, including the ROS master node and the robot drivers. These containers share the host networking stack to allow smooth communication. A container orchestration tool, `docker-compose` helps managing runtime parameters. All the ROS runtime configurations, including the `roslaunch` files, are collected in a single version-control system, which allows the developers to trace the complete change history of configuration change through the robot's life cycle. We arrange the container images in a hierarchy so that some images share the same base image with common libraries, and updating a base image will also update its dependents.

In short, Linux containers help to manage robotic software on a developing and scaling system, help to maintain high availability, limit fault zones, and allow easy replication and distribution of robotic software.

3.4.5 Navigation

TritonBot uses Cartographer [54] for localization and map creation and uses the ROS navigation stack, "move base" to move around. On top of the navigation stack, we created a topological map to guide the robot to predefined routes, as shown in Figure 3.6. The topological map contains waypoints and path: a waypoint represents a featured location such as an intersection of aisles or a place of interest, and a path between two waypoints is line-of-sight that the robot can easily travel. A navigation planning service determines a series of waypoints for the robot to go through before it reaches a destination. Each path has an estimated travel time and is updated when a new estimate is available.

The two major failure modes in TritonBot's navigation behavior are localization failure and execution failure. Most 2D SLAM software matches current laser scan with a pre-built map to estimate the current location of the robot, and when the robot is surrounded by many temporary obstacles (e.g., humans), it may match the observation with an incorrect location. We only observed this failure once when the robot was surrounded by about ten people, and the robot quickly corrected the issue afterward as it moved around a bit and gained more observations. A sudden change in the robot's location estimation is usually an indicator of a localization failure.

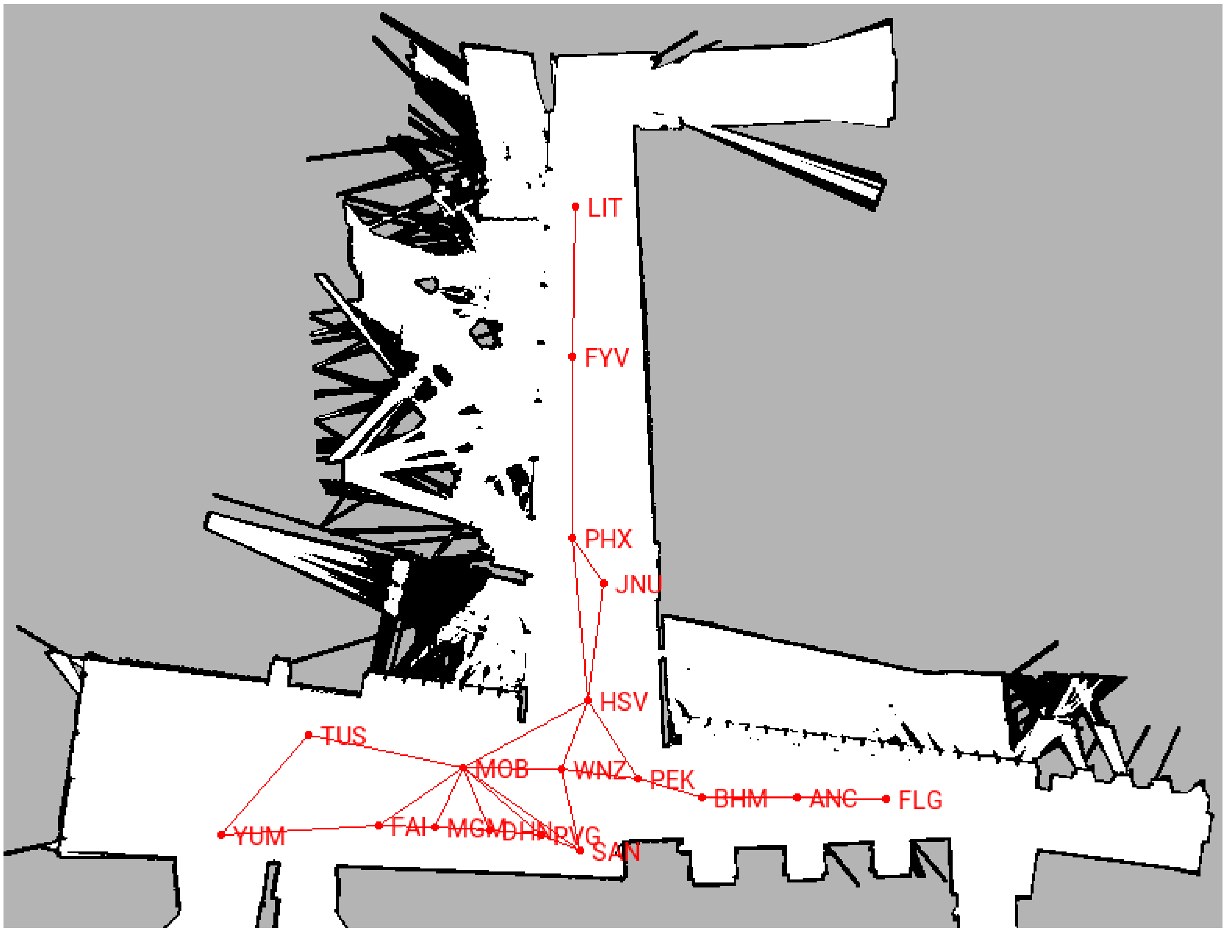


Figure 3.6: The topological map that TritonBot uses to navigate. Waypoints are marked with three-letter names, and paths connect the waypoints that the robot can move between.

On the other hand, dynamic obstacles, especially humans are the biggest challenge in robotic navigation. Some users tried to block the way of the robot to test the robustness of the robot. When TritonBot fails to move, it says “I need some personal space to maneuver, can you please step back.” In the most case, people will yield to the robot, and the robot resumes after executing some recovery behavior.

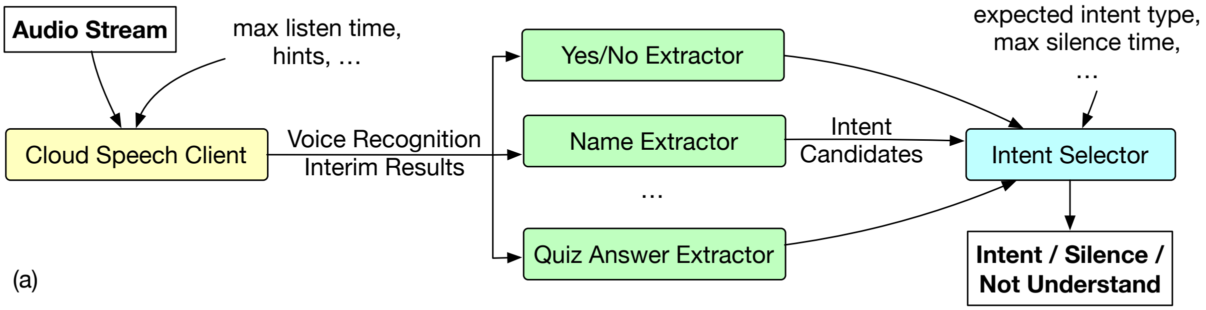
Another lesson we learned from robot guiding person is that the person sometimes leaves the robot in the middle of a tour. As a result, the robot continues to introduce the place of interests to nobody. Without full user study it is hard to conclude why the users leave the robot, but this situation hints that the robot should respond to the person leaving or joining the tour. As an improvement, we are integrating leg detection into the system and use it to confirm the person is still around during the navigation.

TritonBot moved 9901.1 meters in the first month of deployment. Among the $1,722$ attempts moving from a location to another, 97.4% attempts were successful, and 44 attempts failed. Most of them are because of timeout (32 attempts) and move failures (8 attempts) due to dynamic obstacles partially or fully blocking the robot's path. Other failures are because of cancellation command (three attempts, expected behavior) or temporary unavailability in topological map service (one attempt). Moving from a location to another requires traversal of several paths in the topological map (Figure 3.6), and TritonBot and BoxBot traversed $4,876$ paths. Traveling path `PHX-FYV` and `FYV-LIT` caused 20 failures in total. We did not collect enough data to conclude an exact reason, but we did see some people trying to block the path of the robot in this spacious aisle, and only walked away when the robot requested space to maneuver.

3.4.6 Speech and Dialogue

TritonBot has a voice recognition pipeline that converts audio stream to user intent data structure as shown in Figure 3.7 (a). TritonBot uses Google Cloud Speech API to convert voice input from the microphone [56], which returns real-time interim results and rich metadata about the speech, including a confidence score and start/end time for each of the words in the utterance. A few intent extractors convert utterance transcript to candidate intents. The intent selector either yield an intent data structure or special intents for "silence" and "not understand." The intent selector also stops the robot's listening when it can return an expected result so that the robot can respond quickly. Most of the current intent extractors are implemented with matching the words in the input sequence to regular expressions in a template sequence using longest common subsequence algorithm, and Figure 3.7 (b) shows an example.

An interesting observation is that sometimes people respond to questions, especially



(b)

```
my(10) name(10) is '\w+'<name>        name(10) is '\w+'<name>
'\w+'<name>(.1)                       Im(10) '\w+'<name>
'\w+'<name>(.1) '\w+'<name2>(.1)      "my(10) name(10) is '\w+'<name> '\w+'<name2>"
```

Figure 3.7: Speech recognition pipeline in TritonBot and an example of the templates that used to extract the intent of the user. (a): Speech recognition pipeline for TritonBot. Google Cloud Speech client converts audio stream to transcripts, and intent extractors interpret the transcript and generate intent candidates with confidence scores. The intent selector chooses an intent candidate based on expected intent type, time, and other criteria. The Intent selector also stops listening if it can return a result. (b): Templates used in "name extractor." Name extractor is a template-based intent extractor that converts the response to "what is your name?" to structural intent representation. Each word in the template is `regex<capture name>(weight)`, and the template is matched against voice recognition result using longest common subsequence algorithm to calculate a confidence score based on `weight`. Words with `capture name` can be used to fill in some fields of an intent.

yes-no questions, even before the robot finishes talking. As a result, the robot misses the response because it was not able to capture the very first part of the utterance. In late deployment, we changed synchronization mechanism to allow the voice recognition to start 0.3 seconds before the robot finishes speaking according to previous studies [61], so that the robot can capture the full response from the human but almost none of the speech from itself.

3.4.7 Face Recognition

TritonBot has a head camera (Primesense Carmine 1.09) at 1.5 meters from the ground facing 15 degrees upwards, but the camera mounted on BoxBot is at 1.2 meters height facing about 30 degrees upwards. Both TritonBot and BoxBot can see an undistorted face, but BoxBot cannot detect people standing too close (less than 0.5 meters) from the robot because of the tilt of

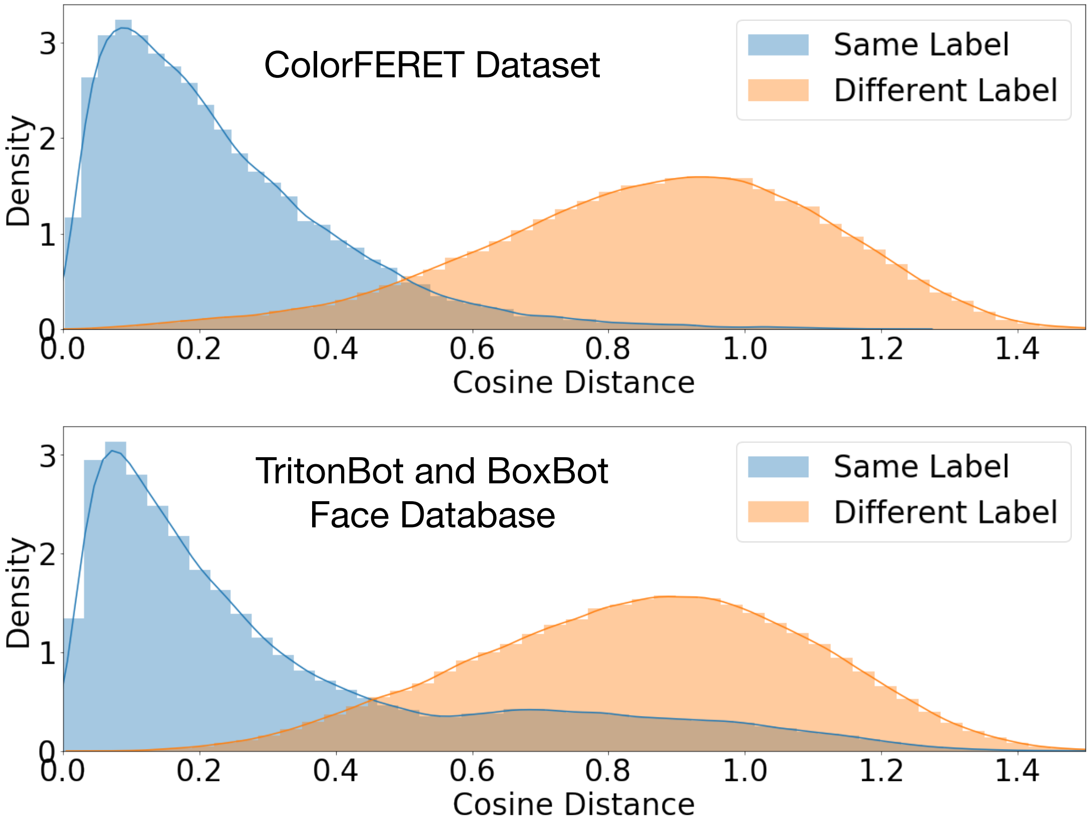


Figure 3.8: The distribution of cosine distance of the OpenFace embeddings, between the faces of the same person and the faces of the different persons in the ColorFERET dataset and TritonBot and BoxBot's face database.

the camera.

TritonBot uses deep neural networks based on Openface [53] to recognize visitors' face. As a starting point, we use the nearest neighbor to find the best match given a face. When the cosine distance ($d(\mathbf{A},\mathbf{B}) = 1 - \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$) is between the detected face and the candidate is greater than a threshold, the face recognition reports an unknown face indicating a stranger. The threshold was initially set to 0.5 by balancing false positive and false negatives on the ColorFERET dataset [62], and we will adjust to reflect the actual dataset for the next deployment. Figure 3.8 shows the distribution of cosine distance of the Openface embeddings of the same person and different persons from the ColorFERET dataset, and the actual faces TritonBot and BoxBot remember.

While TritonBot usually recalls the name of a seen person, as expected the mis-classification ratio increased as the database size is increased. One of the reasons is that sometimes people face the robot sidewards, and the robot could only see a side face, which is easy to confuse on different people.

3.4.8 Logging

Logging in long-term autonomy robots provides opportunities for data analysis, testing new software, and replaying the scene. ROS provides a logging utility, `rosbag` to log general communication during the robot operation. We create a program to record a substantial portion of ROS communication in TritonBot, including laser scanner data, localization estimation, the command to move the base, etc. Even though the robot only records `rosbag` files when it is actively interacting with a person, we have collected 367.8 GB “rosbag” files in the first month of deployment.

Although “rosbag” records many details in ROS communication, it is far from ideal for recording internal states for software components in a long-term autonomy robot because of two major flaws: (1) Changing the log record fields definition in `rosbag` invalidates all previous logs, which is inconvenient for an actively-developed long-term autonomy robot. (2) `rosbag` is designed to capture the communication between ROS components, and it is costly to send massive internal states through network sockets when they only meant to be written to a disk.

To overcome these deficiencies, we created a logging library, “Universal Logger” to save internal states of both ROS and non-ROS components. Universal Logger records structural log data in an efficient and backward-compatible Protocol Buffer [63] format. While maintaining strong-typed structure, Protocol Buffer also allows adding new fields without affecting previously serialized logs. Every robotic component using Universal Logger writes logs to the disk independently without going through any communication interface. Also, Universal Logger automatically creates new binary log files when date changes or a single file is larger than a certain size, and we arrange these files in directories with the creation date. For example, speech recognition logs of Feb 27, 2018, on BoxBot is stored in the following path: `boxbot/2018/02/27/dialogue/speech_recognition/1519752112.pb.gz`. The logs are transferred to a storage server and removed from the robots everyday to save the storage space. So far we have collected 2.8 GB data in Universal Logger format in the first-month deployment, and use them to study long-term autonomy (and conclude the result in this chapter).

3.4.9 Safety

Safety comes first when a robot is deployed in an open environment. Mobile platforms of TritonBot and BoxBot have a built-in hardware level safety mechanism that slows it down when the laser scanner detects movements nearby, but a grad student always monitors the robot physically close to the robot for the first-month deployment to avoid unexpected malfunctions. However, in the first-month deployment, the guardian never engaged the emergency stop mechanism. There was one case that a child tried to put his foot under the robot, but the robot slowed down immediately, and the child stepped back by instinct.

Requiring in-place human supervision harms the purpose of long-term autonomy experiments, since the robot working time is limited to the guardian's work time. With more confidence in robot's safety, we will slowly move to remote monitoring: we have created a private website to show the vitals of the robot and the camera image for safety monitoring purpose, and we deployed remote e-stop buttons that engage run-stop mechanism on the robot via the control network. We are also setting up bump sensors on the robot to stop its motion when it hits something.

3.5 TritonBot Initial Deployment Results

In the first-month initial deployment, TritonBot and BoxBot together worked 18 days in total, skipping holidays and special events in February 2018. TritonBot only operated two

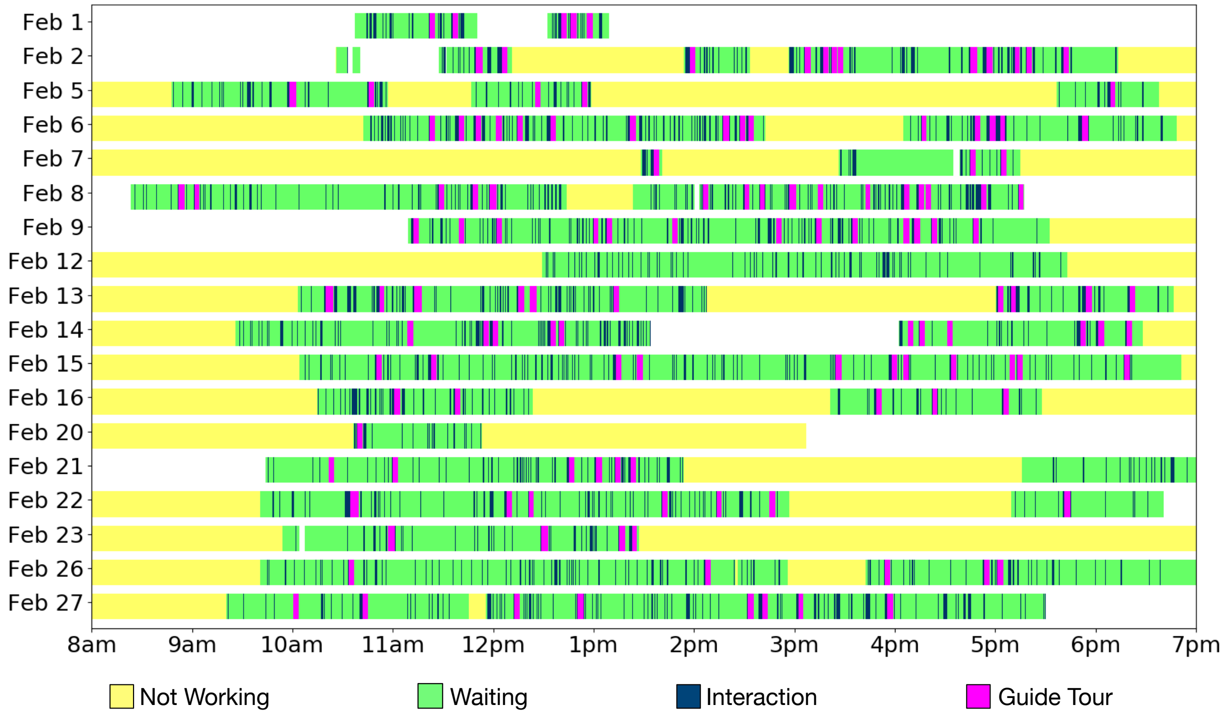


Figure 3.9: Working log of the robots during the initial deployment. The robots have worked for 108.7 hours (waiting, interaction, and guiding tour), actively interacted with people for 22.1 hours (interaction and guiding tour). Feb 1, Feb 8, and morning Feb 13 came from TritonBot, and the others came from BoxBot. Blank indicates the robot is under maintenance or taken to special events and the top-level control software is not running, and light yellow background means the robot is charging itself.

and a half days because of maintenance and other projects requiring the platform, but BoxBot worked for 15 and a half days. Figure 3.9 shows the breakdown of the activity of the robots. The two robots have been serving the guests for 108.7 hours in total, actively talked and walked with people for 22.1 hours, and spent 8.25 hour in guiding people around.

TritonBot and BoxBot remembered 167 persons with 8,681 face samples, and they did 97,284 face detection and recognitions. As for speech, they had been listening for 4.0 hours (2,616 counts) and speaking for 10.7 hours (14,037 counts). The robots greeted people 2950 times. It recognized acquaintances 416 times, although the false positives are not determinable at the current times. In the 434 attempts of asking a visitors' name, it heard a response 239 times. The robots played 133 trivia games and asked 307 trivia questions. Among the 292 answers, 224 were the correct answer, 47 were wrong answers, and the user indicated they did not know the answer in the other 21 responses.

The robot moves when they guide people to places of interest or need to go to a charging dock. The robots traveled 9.9 kilometers during the deployment, giving 150 tours in total. It attempted to move to a specific location (a waypoint in the topological map, see Figure 3.6) $1,722$ times, and $1,678$ (97.4%) attempts were successful. To accomplish these targets, it traveled $4,876$ times on the edges of the topological map, spent 6.1 hours in total.

3.6 Conclusion

Long-term deployment reliability is one of the most crucial parts of any feasible autonomous service robots. TritonBot served as a study in robot-human interaction patterns and long-term service viability before they can enter our homes and businesses. This chapter discussed TritonBot's design and summarized lessons we learned from the first-month deployment of TritonBot.

Just like any engineered system, failures are unavoidable in robotics. As a design principle, the robot design should prevent hardware failures from propagating and cause further and/or permanent damage to the robot. Software failure should be tolerated in a loosely-coupled system since most of the failures can recover by restarting a program. More importantly, testing helps discover design flaws, and fault injection/simulation in hardware, software, and networking will help to discover and test failures in robotics beforehand. System engineering helps to reduce the effort of maintaining a long-term autonomy robot development platform, and Linux container significantly decreased the stress of managing a scaling robot system.

The most significant challenge of robots is still human-beings. From our observation, people tried to block the path of the robot, try to fool the robot in talking, and ignores the robot talking to itself when they leave. Sometimes humans unintentionally face the robot side-wards, expect the robot to respond when it is working on something else, or talk to the robot in a way that it could not understand. Long-term deployment exposes all of these cases and helps us design better robot behavior.

Our future goal is to improve TritonBot to meet and exceed the expectations of its users. We aim to formalize research questions based on our first-month observations and continue the long-term deployment of TritonBot to study its autonomy for a longer timespan. In the next chapters, we will discuss efforts to improve the reliability and the long-term autonomy of TritonBot in different aspects. After all, when a robot enters our homes and businesses, it should not require frequent maintenance like automobiles.

Acknowledgements

Chapter 3 contains material from “TritonBot: First Lessons Learned from Deployment of A Long-term Autonomy Tour Guide Robot,” by Shengye Wang and Henrik I. Christensen, which appears in the proceedings of 2018 IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN 2018). The dissertation author is the primary investigator and first author of this paper. The material in this chapter is © 2018 Institute of Electrical and Electronics Engineers, Incorporated (the “IEEE”).

Chapter 4

Scalable Software Management and Orchestration in Service Robots

From the TritonBot experience, we learned that scaling up the software system on service robots increases the maintenance burden of developers and the risk of resource contention of the computer embedded in robots. As a result, developers spend much time configuring, deploying, and monitoring the robot software system; robots may utilize significant computer resources when all software processes are running. This chapter presents our method of using software encapsulation and dynamic orchestration to reduce maintenance burden and resource contention in service robots. To pave the way for software encapsulation and dynamic orchestration in service robots, we present a software management system, “Rorg,” that containerizes software components and a resource time-sharing mechanism incorporated with the Robot Operating System (ROS). Rorg leverages Linux containers for developers to pack software into self-contained images and runs them in isolated environments. It also allows the robot to perform dynamic orchestration and to turn on/off software components on demand to avoid resource contention. We evaluate Rorg with the previously presented long-term autonomous tour-guide robot, TritonBot: Rorg manages 41 software components on the robot and streamlines our workflow; it reduces

CPU load by 45.5% and memory usage by 16.5% on average. We believe software encapsulation will improve the scalability of other service robots, and switching to dynamic orchestration from static orchestration will reduce resource contention in other multi-purpose systems.

4.1 Introduction

Robotic research and application are advancing at a high pace in recent years, and service robots have started to enter the consumer market and assist humans at home and offices [1]. Whereas deploying service robots is becoming easier, the robot systems themselves are becoming more complex [64]. From our previous experience, a tour-guide robot for human-robot interaction (HRI) research in a university office building consists of about 65 software programs for face recognition, voice recognition, navigation, and various other tasks [6]; autonomous driving vehicles have even more components for localization, pedestrian detection, mission planning, motion planning, and so on [65]. As a robot system evolves and expands, the complexity of the software components is becoming a more pressing issue that, in turn, challenges scaling up.

Developing software for robot applications has two primary challenges: maintenance burdens and resource limitations. First, when the number of software components increases, organizing, deploying, and monitoring them becomes tedious and error-prone [66, 67, 68]. Second, computationally intensive programs use much of the computing resources in the on-board computer [69, 70]. They challenge scaling up since a service robot can only carry a computer with moderate processing power and memory capacity due to power and weight considerations. Efforts have been put into tackling these issues, but the infrastructure side of robot systems has only received limited attention.

Recently, software encapsulation techniques, such as the Linux containers, have become a popular tool to deploy software in datacenters and the cloud [59, 71, 15]. Linux container allows developers to pack software and dependencies into self-contained images, deploy them onto different targets with minimal configuration, and isolate them from the host system to varying degrees. On the other hand, dynamic orchestration enables time-sharing computing resources and avoids resource contention. These techniques open an opportunity to mitigate the maintenance burden and resource limitation in service robots. However, few research and commercial service robots currently benefit from software encapsulation and dynamic orchestration due to three primary reasons:

- Most of the current research and commercial service robots use certain middleware or framework [11, 72, 73] to facilitate loosely-coupled architecture design, and among them, ROS [11] is the most popular representative. While ROS-based software can run inside encapsulated environments in theory, there are limited experiences to containerize the robotic software.
- Service robots have numerous software components with complex relationships, and not all of them are in use at the same time. Although existing tools can leverage Linux containers to provide interfaces for one component to start/stop/monitor another, few take the dynamic runtime pattern of programs into account.
- Few orchestration tools are designed for environments with tight overall resource budget, as they target for datacenters and cloud platforms [15] where almost unlimited computing resources are available.

Our goal is to enable efficient software organization and scheduling on autonomous service robots in a high-performance and scalable manner. Combining software encapsulation and dynamic orchestration, we propose Rorg, a toolkit that leverages Linux containers to manage and schedule software on service robots. Rorg makes three key contributions:

- Rorg adopts a Linux container engine, Docker [59], to run robot software in individual containers. Rorg works with ROS [11], the de facto standard robotic middleware. It provides default configurations to run ROS software without modification, and we provide an example setup of a fully-functional tour-guide robot application with Rorg.

- Rorg organizes the robot software into multiple Linux containers and models the static and dynamic relationships between them. Rorg provides an interface to create, query, update, delete, start, stop, and restart these containers manually or programmatically.
- Rorg allows the software components to time-share computing resources: It pauses or shuts down inactive services to save computing resources but reactivates them when they are required for an upcoming task. Rorg monitors the system and keeps the resources utilization at a reasonable level.

We evaluate the implementation of software encapsulation and dynamic orchestration using Rorg on a long-term autonomy tour-guide robot, TritonBot (Chapter 3). Rorg and its early prototype helped us to keep the robot working for over six months. The tour-guide robot went through 126 software version updates and 71 configuration changes. Rorg now manages 41 services (containers) on the robot. Rorg dynamically orchestrates the software components, only runs necessary components, and shuts them down when they are idle to save computing resources. The robot consumes 89.4% of CPU time and 3.41 GB of memory on average without Rorg. With Rorg, the average CPU drops to 48.7%, and the memory usage is 2.85 GB, and we notice the robot responds much faster. Although web service and datacenter applications have wildly adopted container technology, Rorg addresses the specific challenge on service robots: multi-purpose systems with limited computing resources but have high performance/responsiveness requirements.

The rest of this chapter is organized as follows: Section 4.2 discusses related work and the unique challenges of service robots software management. Section 4.3 presents the implmentation of software encapsulation and dynamic orchestration, design of Rorg, and its features to streamline development workflow and avoid resource contention. Section 4.4 evaluates software encapsulation and dynamic orchestration with a real tour-guide service robot and measures its performance, and finally Section 4.5 concludes this chapter.

4.2 Background and Related Work

In some aspects, service robots are similar to smartphones, embedded systems, and datacenter applications. Like smartphones, service robots are limited by size, weight, and power constraints. But smartphones have a single, isolated user-facing task packed in a self-contained package. To save computing resources and power, smartphone system such as iOS and Android shuts down independent background tasks that are not in use. Service robots, however, are composed of a variety of different, parallel tasks. The single-foreground-task scheduling mechanism in smartphones is hardly applicable to service robots.

Many embedded systems have similar constraints. But most of these embedded systems are specialized in doing one task. All the components of that task need to be running for the system to function correctly. Yet service robots are required to offer different services and perform different tasks at different times. The multi-purpose nature of service robots invalids the scheduling mechanisms in traditional embedded systems.

Datacenter applications share the most similarities in software organizations with service robots. Both datacenter applications and service robots have a numerous number of loosely coupled components. However, scaling-up patterns in datacenter applications differ from that in service robots. Scaling-up in datacenter applications means spawning more instances of the same program to handle requests from the users. In contrast, scaling-up in service robots means running a different set of components for various tasks under resource constraints.

Because of the multi-component architecture, multi-purpose natures, and limited computing resources, scheduling strategies in iOS and Android from the smartphones, real-time designs in embedded systems, and management tools like Borg [13], Mesos [14] and Kubernetes [15] from the datacenters, do not apply to service robots.

Linux containers (or operating-system-level virtualization in general) have received much attention in datacenters and cloud applications recently [74]. Relying on operating system kernel features, Linux containers provide isolated runtime environment with minimal performance overhead. They also provide a convenient approach to build, deploy, and run applications on different machines. Unlike hypervisors or virtual machines that run fully virtualized kernels, Linux container leverages the kernel of the host and thus is much lighter weighted [75]. Popular Linux container engines include Docker [59], LXD [76], and others. Despite different branding and user interface, they all exploit the same underlying Linux kernel features such as namespaces and control groups, and thus offer similar performance.

Previous research applied Linux containers to robotics to some extent. White et al. use Linux containers to run ROS with Docker [60], but their examples do not illustrate actual challenges on building and maintaining a moderate-scale service robot system. Mabry et al. exploit Docker to deploy software on a maritime robot [77], but their approach only applies to application-specific robot and does not scale up to fit the more complex software system of service robots. Avalon [78] is a crowd-robot scheduler that distributes tasks to multiple robots; it also leverages Linux containers to encapsulate the applications, but it does not address challenges on a standalone service robot. Cognitive Interaction Toolkit (CITk) [79] adopts Linux containers to construct and run robot simulation experiments; a later work RoboBench [80] is a benchmark suite based on CITk that reproduces robotic simulation on workstations. SwarmRob [81] is a toolkit to share experimental heterogeneous robots using Linux containers. However, none of CITk, RoboBench, or SwarmRob was tested on physical robots.

Modern applications in datacenters are often built with the “microservices” pattern: individual microservices communicate and cooperate with each other to deliver greater functionality [40] — the same pattern also exists in ROS-based robot system design. Besides, other hierarchies in a robotic system can also find their counterparts in datacenter applications, as shown

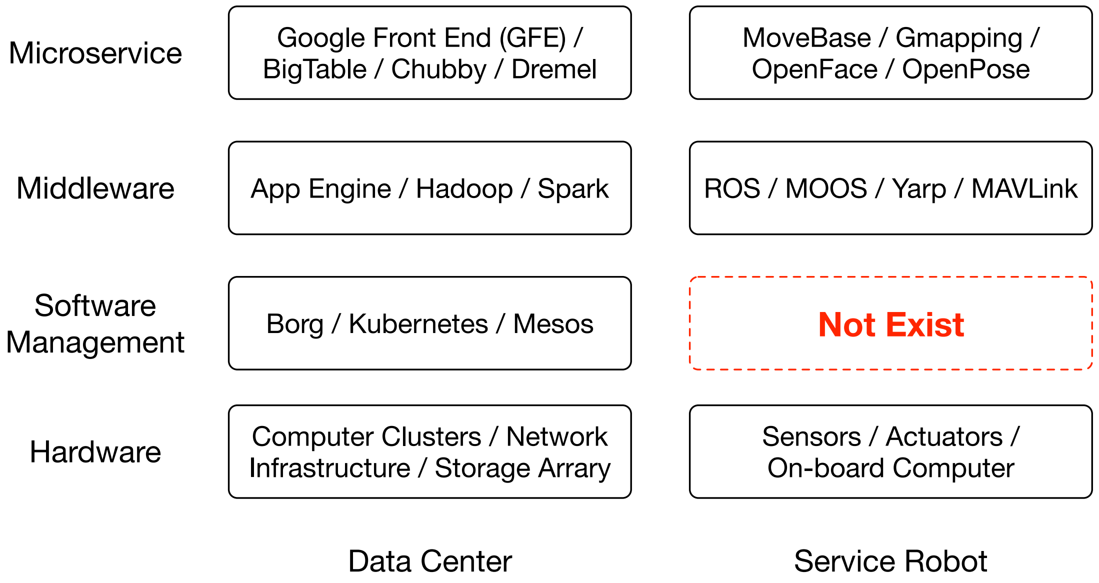


Figure 4.1: Robot systems are similar to datacenters. Each hierarchy in a service robot can find its counterparts in a datacenter, except for the “software management system.” We built Rorg to fill in the gap and address the unique challenges of software management in service robots where resources are sacred.

in Figure 4.1. However, most service robots do not have a “management system” counterpart to datacenters, and that is where Rorg comes in.

In datacenters, the main challenge for a software management system is redundancy. At Google, Borg [13] runs two extra instances of each microservice in their planet-scale computer system to tolerate the failure of one instance while updating another instance [40]; it distributes the instances across different physical locations to increase reliability. Similarly, Kubernetes [15] and Mesos [14] are popular open-source software management systems that combine Linux containers with sophisticated scheduling to provide load-balancing, to avoid single point failures, and to scale-up when needed.

In summary, no existing solutions address the unique challenges of software management on service robots — multi-purpose systems with limited computing resources yet have high performance/responsiveness requirements.

4.3 Rorg Design

To address the aforementioned challenges, we propose Rorg, a software manager toolkit for service robots. In essence, Rorg is a set of programs that receives requests from developers or programs, and creates, starts, pauses, stops, and removes software components on a service robot at an appropriate time. Rorg consists of three design principles: First, Rorg is effortless to use; it containerizes robotic applications (ROS-based in particular) with minimal configuration. Second, Rorg is scalable; it targets managing moderate and large-scale robot applications up to hundreds of programs or ROS nodes. Third, Rorg avoids the risk of computer resources contention in the robot; it eliminates unnecessary computation and improves robot responsiveness.

4.3.1 Linux Containers for Robotic Applications

Rorg leverages Docker [59] as its underlying backend to containerize robotic software, but it provides default configurations to run ROS-based software with minimal configuration since currently ROS is the most popular robotic middleware. Because ROS applications are not designed to cross network address translation (NAT) devices, Rorg by default uses the host computer's network stack for the containers to exclude default communication barriers that come with Docker and extra performance overhead in multiple networking stacks. In addition, Rorg provides a "driver" option to configure microservices that talks to sensors/actuators with higher privilege to access host peripherals. Rorg monitors these programs for the unexpected restart, and it asks for extra confirmation when restarting these programs manually. Last but not least, Rorg still opens all the lower-level Docker parameters to the users, but Rorg alerts users to potential configuration error. In short, Rorg makes Linux container easier to use for robotic applications without sacrificing its original functionality.

Rorg enables efficient deep-hierarchy container image building. Docker allows the user to pack a program along with its dependencies into a standalone "image"; these images can form a

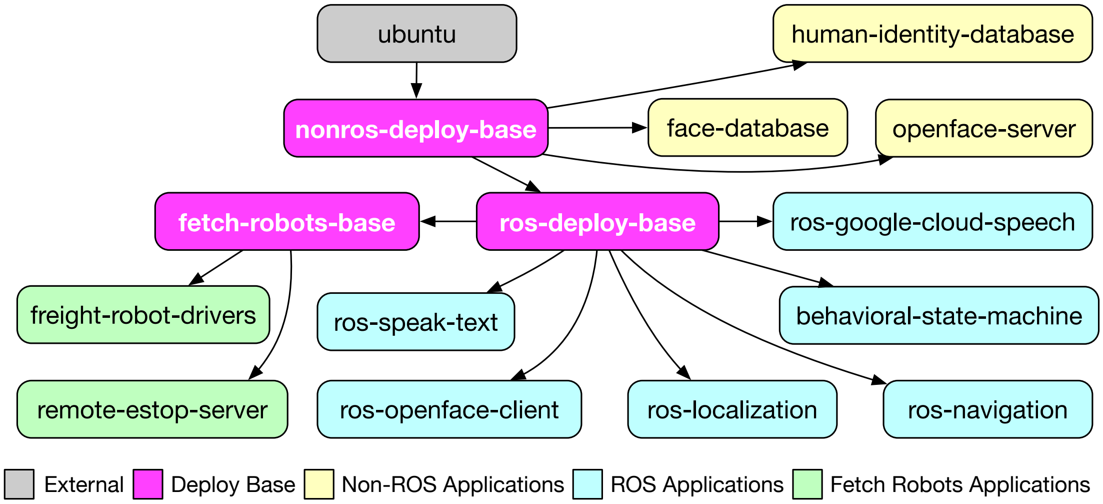


Figure 4.2: Container images (and their hierarchy) that support a robot receptionist and tour guide [6]. By carefully arranging the hierarchy, each application can use a most appropriate "base image" and save build time and disk space without creating conflicts. Rorg provides tools that help the developer to build the images with a deep hierarchy correctly.

hierarchy to save build time or disk space when they share common parts. Figure 4.2 shows part of the images hierarchy in a service robot. A deep hierarchy allows each application image to use an appropriate base image to maximize libraries reuse and minimize the chance of conflicts, but it also leads to a challenge: the developer needs to build the images in a correct order for the end image to reflect the updates in the hierarchy. Rorg provides a script to automate the task: it sorts all of the images in a topological order of the dependency graph, pulls external dependencies, and then builds dependent images before the children images. Therefore, Rorg retains Docker's optimization such as caching while building the images, but it builds the images correctly without a larger infrastructure such as a continuous integration system.

Rorg leverages Linux containers to simplify robotic software development. Rorg comes with a script that prepares a seamless development environment inside a container, which is based on the same base image for deployment. The unified environment eliminates the inconsistency between development and deployment, which helps the developer to prepare for deployment at the very beginning. Since the development environment is encapsulated inside a container, the developer can create a fresh environment with minimal effort in case the development environment is contaminated.

Rorg keeps the full history of an application for later review or postmortem analysis. It records the full log of the changes to Docker container in machine-readable “protobuf” [63] format. Also, Rorg leverages Docker’s “mount” feature to map a directory on the host machine to a container, so that the developers can store runtime configurations like “`roslaunch`” files to a version control system (VCS) and attach them to the container. The history in VCS combined with the Rorg log allows the developers to recover the state of a robot to any previous checkpoint. As an example, we provide our TritonBot tour guide system along with Rorg to demonstrate these practices.

The benefits of Rorg are not free — the developer needs to write extra code to configure Rorg. Since Rorg runs all programs inside Docker containers, deploying a robotic program with Rorg requires creating a Docker image and defining runtime parameters. Seemingly an extra effort, the image blueprint (`Dockerfile`) and the runtime configuration actually serve as a document to reproduce the execution environment, which helps organizing robotic software from another perspective; many readily available ROS images and examples further make this process easier [60].

The above Rorg features exploit most of Docker’s potentials to build, ship, and deploy ROS-based robotic software; we plan to extend Rorg support to other robotic middleware in the future. These features keep indivudal robotic software components organized and lighten developer’s burden. The next section will discuss Rorg’s effort to organize robotic software system in a whole and avoid resource contention.

4.3.2 Scalable Robotic Software Organization

The basic element in Rorg is a *service*. Usually, a service represents one container instance: for example, each of the localization, navigation, and face recognition software is a service. The “service” concept is consistent with “microservice” in datacenter applications, and it is similar to

ROS "nodes" or Linux processes in terms of granularity. Rorg does use non-regular service to simplify its semantics (for example, `developer` is a meta-service that represents a developer's actions, which will be discussed later in this section), but regular services are created with Docker images and runtime configurations. Rorg provides interfaces to create, query, update, and remove a service programmatically or through a command-line interface.

Although each service differs from each other in terms of the role in the system, Rorg sees them as identical in orchestration. A service may "request" another: a *request* is a relationship between services — the requester service will use the requested services until it releases the request. Rorg only keeps requested services alive and ceases services that are not requested by any other services. For example, when the robot needs to talk, a *behavioral* service can request the *speak-text* service before invoking voice synthesize. For an active service with implicit dependencies, Rorg automatically requests the dependent services. For example, a *navigation* service must always request *localization* service because the robot cannot navigate correctly without awareness of its position. A newly created service is not active until another service "requests" it. When terminating a service, Rorg also automatically releases all its owned requests to prevent "requests leak."

4.3.3 Time-sharing Computing Resources

Rorg avoids resources contention on the on-robot computer by time-sharing computing resources. Because the robot carries limited computing power, the resource-contented computer should allocate the appropriate amount of resources on programs' demands. With the robotic software informing Rorg of the components usage, Rorg enables the required services and ceases or pauses inactive services, and thus reduces resources usage.

Figure 4.3 demonstrates a simplified Rorg-powered tour-guide robot example that detects human faces, chats with people, and offers tours to the visitors. The robot takes five consecutive actions in its work cycle; Rorg manages services accordingly to avoid resources contention. ① In

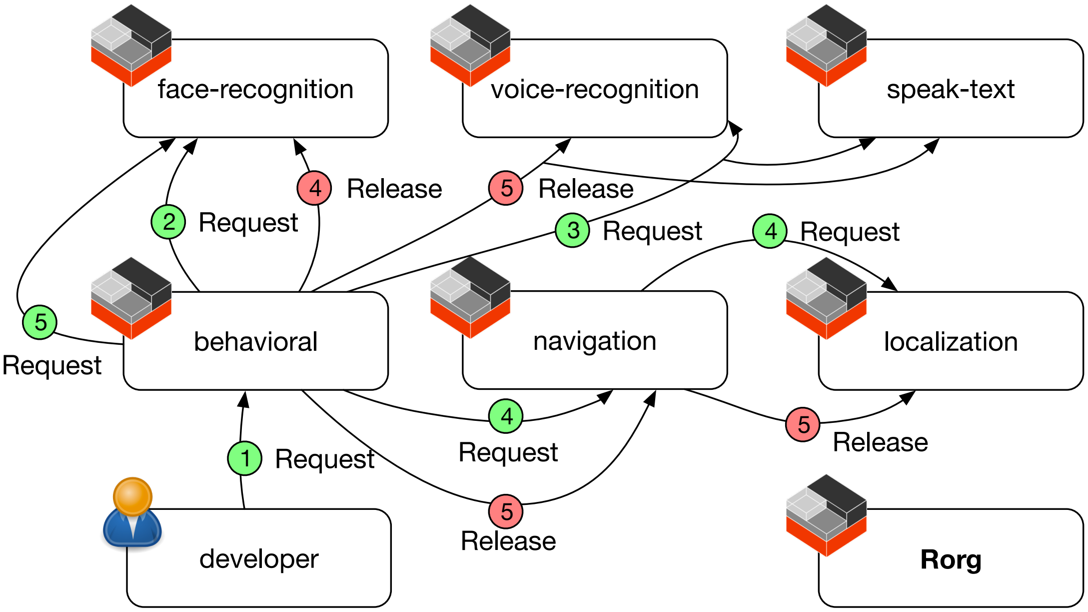


Figure 4.3: Services in a receptionist and tour-guide robot [6]. Each service is a Linux container; Rorg itself also runs inside a container, but it is not a service. The services contact Rorg to start or stop their peers: ① The developer makes the robot autonomous. ② The robot starts to detect visitors. ③ The robot chats with a visitor. ④ The robot moves around to guide a tour. ⑤ The robot returns to its standby state.

the beginning, the human developer requests the *behavioral* service that controls the overall behavior of the robot through a command-line program. The requester is set to a *developer* meta-service that is always assumed active. ② Then the robot becomes autonomous, and *behavioral* requests *face-recognition* to detect human faces. ③ When the robot sees a visitor, *behavioral* requests *voice-recognition* and *speak-text* to chat with the visitor. ④ When the visitor decides to take a tour with the robot, the *behavioral* requests *navigation*, which implicitly requests *localization* (for robot pose estimation) to move the robot around. The *behavioral* also releases its previous request to *face-recognition*, as the robot does not detect human face during the tour. ⑤ After the tour, *behavioral* releases all of the services but requests *face-recognition* to return to the initial state. When *navigation* becomes inactive, Rorg automatically releases *localization* as it was only requested by *navigation*. Note *localization*, *navigation*, and *face-recognition* are all computationally intensive. In such setup, the robot does not waste CPU time on *face-recognition* during navigation, nor does it assign any resources for *localization* during the waiting state.

Using Rorg for time-sharing computing resources is easy in general, but it is complicated for certain services. For example, the developer can specify an implicit dependency between *navigation* and *localization* when creating *navigation* service, thus *localization* will always start when *navigation* becomes active. Rorg provides several interfaces and client libraries, including ROS-service interfaces, general remote procedure call (RPC) interfaces [82], Python, and C++ libraries to help developers to write code to send/cancel Rorg requests for services with dynamic behavior (e.g., *behavioral* service). Besides, we are investigating more automated methods to make Rorg more programmer-friendly.

It is worth pointing out Rorg's time-sharing method is not the only way to reduce resource usage on service robots. A common option is to offload computation, but it is limited by latency constraints and privacy concerns on service robots. Another option is to start and stop individual Linux processes, but Linux containers are much cleaner (e.g. the developers are free from accidentally leaving orphan processes running) and provide more functionality (e.g. pausing a process without sending `SIGSTOP`). A third option is to design an event-driven system architecture; it can avoid unnecessary computation without shutting down any components, but such option usually involves a total system overhaul. Besides, by often ceasing services, Rorg provides an opportunity for them to restart refresh frequently, as a previous long-term autonomous service robot Sage experienced many memory leak issues which can be mitigated by restarting components often [23], and as Google points out "a slowly crash looping task is usually preferable to a task that hasn't been restarted at all [40]."

4.4 Evaluation

We test Rorg using our TritonBot system, a real long-term autonomous service robot [6]. This section evaluates Rorg from two aspects: First, to show Rorg's effectiveness on software maintenance, we compare our daily workflow before and after introducing Rorg to manage the software in TritonBot. Second, to show how Rorg reduces the risk of resource contention, we compare the CPU and memory usage with and without Rorg.

4.4.1 Experimental Setup

Rorg targets at service robots that perform different tasks at different times. In previous work we built TritonBot to serve as building receptionist and a tour-guide robot [6]. TritonBot stands to face the building entrance and continuously detects faces with its camera. When TritonBot sees a visitor, it will greet the person by name if it could recognize the face, or it will ask the visitor's name and associate the name with the face. TritonBot also offers trivia questions and navigates with the visitors and introduces places of interest in the building; it uses a leg tracker to make sure the visitor is following it. The behavior of TritonBot is controlled by a finite state machine.

TritonBot is built on a commercial mobile robot platform "Fetch Research Robot" [10]. As discussed in Chapter 3, it senses and interacts with the environment using a camera, a directional microphone, a loudspeaker, two laser scanners, and a mobile platform with two differential-drive wheels. The core of the platform is a computer composed by an Intel i5-4570S CPU (4 cores, 4 threads) operating at 3.20GHz, 16 GB memory, and 1 TB solid-state storage. Seemingly outdated, the computer was a medium-configuration when the two-year-old robot was manufactured. We believe TritonBot represents service robots in the middle of their lifecycles. TritonBot's software system consists of 65 ROS programs, and it heavily leverages open-source and third-party software to support its tasks. For example, Openface [53] pipelines human faces detection and the face similarity calculation; an open-source leg tracker [83] tracks people around the robot; Cartographer [54] provides simultaneous localization and mapping (SLAM) for the robot, and the ROS navigation stack "move base" [55] navigates the robot around. TritonBot also uses cloud service like Google Cloud Speech API [56] to transcribe speech audio to text. These

Table 4.1: General TritonBot maintenance workflow with and without Rorg.

Maintenance Task	Without Rorg	With Rorg
Deploy a new software component.	Copy source files to the robot; install the dependencies; compile the software; fix potential dependencies errors; write a startup script.	Write and test a `Dockerfile`; build and upload the container image to a "Docker registry"; use a Rorg command-line tool to create a new service.
Update a software component.	Remove original software and residuals but keep configuration and reusable data files; install the new version; update configuration files.	Rebuild and update the original `Dockerfile`; run a Rorg command-line tool to refresh the update.
Rollback a software component.	Remove original software and residuals; reinstall the old version; rollback configuration files.	Rollback configuration files; run a Rorg command-line tool to rollback.
Update software configuration.	Locate and update the configuration in scattered places; kill the program process tree; restart the program.	Update the configuration in a unified location; run a Rorg command-line tool to restart the service.
Develop software.	Install the same libraries and tools on the robot to the developer workstation; clean up or even reinstall the system in case of library contamination; before deployment, fix the inconsistency between deployment and development environments.	Use the same container image (with libraries and tools) to develop on workstations or robots; recreate the container with Rorg to restart from fresh; not to worry about inconsistency between deployment and development environments.

components were eventually packed as individual Rorg services.

4.4.2 Managing Software System

The complexity of TritonBot system puts a lot of maintenance burden on the developers. Many of these software components depend on different libraries (the "dependency hell"), and there are many runtime parameters, configurations, and other supporting files associated with each of these components. As the robot system scales up, keeping the system running becomes a tedious and error-prone task.

Table 4.1 compares our workflows of general maintenance tasks with and without Rorg. With Rorg, we manage our robotic software in a much cleaner and organized manner. Since the maintenance effort is a subjective concept and hard to quantify, we leave the decision of whether Rorg helps the developers in operating and evolving the system to the readers.

At this time, TritonBot is running 65 ROS nodes as 41 Rorg services. We group some tightly-coupled ROS nodes together as a single Rorg services (for example, the cartographer node and the accompanying occupancy-grid-map converter). In the past year, we pushed 126 software version updates and 71 configuration changes to TritonBot with the help of Rorg.

4.4.3 Avoiding Resources Contention

Many of the programs in TritonBot are resources intensive. When the robot is operating without Rorg, the average CPU usage is often around 90%, the memory utilization occasionally reaches 100% due to a potential memory leak. Therefore, TritonBot is on the borderline of resource contention where insufficient performance slows down the robot. As a result, the response latency of the robot is creating an unsatisfactory user experience — the robot only moves a few seconds after it says “please follow me.”

We evaluate the decrease in resource usage with Rorg using three setups: simulation, emulation, and deployment.

Simulation

In the simulation, we feed the state machine trace (state transition log) to a simulator and assign empirical CPU and memory usage (medium number collected from a month-long deployment) to each of the services. The simulation experiment is fast and the result is always consistent, but it does not reflect the dynamic resources consumption by a program. We run

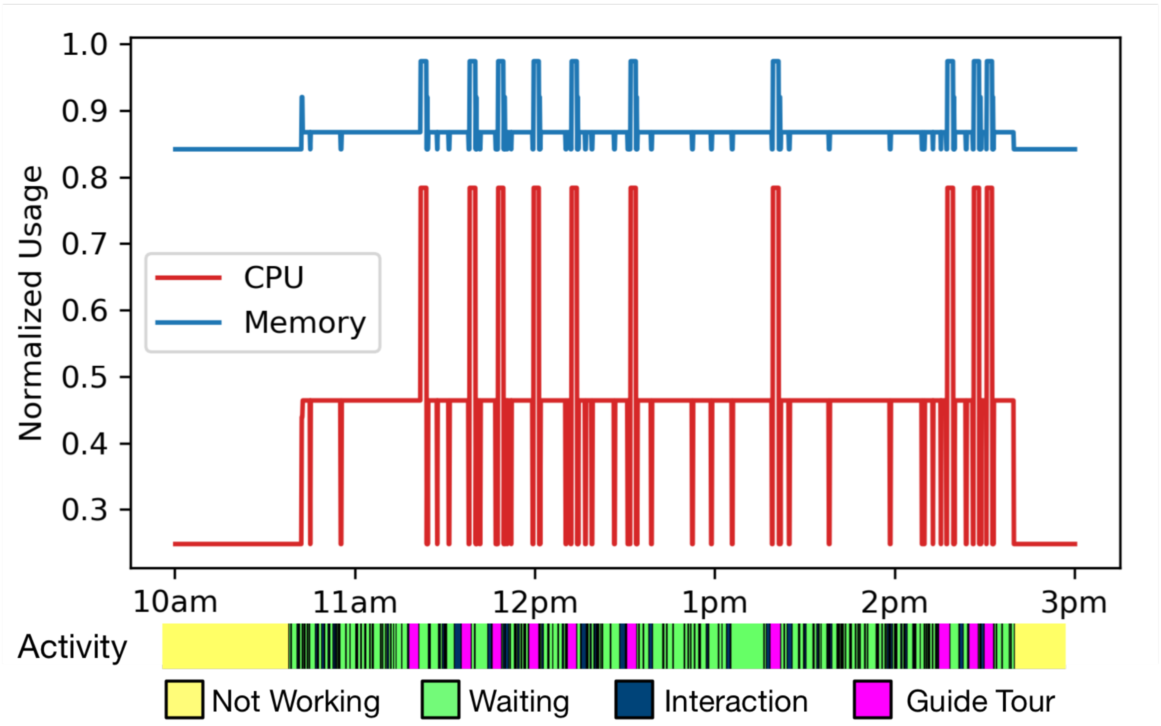


Figure 4.4: Rorg simulation result of TritonBot using a trace collected on February 6, 2018. We normalize the CPU and memory usage to a baseline where all the components are always active. Only calculating active time, Rorg reduces CPU usage by 52.6% and memory usage by 12.5%.

the state machine trace collected on TritonBot back on February 6, 2018, a typical day for TritonBot [6]. The robot was deployed from 10:40 am to 2:40 pm that day, worked for four hours in total. It greeted the visitors 188 times and guided ten tours. As shown in Figure 4.4, the simulation result indicates 52.6% reduction to CPU usage and 12.5% reduction to memory usage by introducing Rorg to TritonBot.

Emulation

In the emulation, we run the actual components on a workstation with the same specs as the robot embedded computer, but we play back sensor data and discard the control commands sent to the actuators. The emulation experiment reflects the dynamic resource allocation of the programs, and it does not require TritonBot to move when we are profiling the system. The baseline system has all of the Rorg services running, but the system with Rorg only runs the

Figure 4.5: Rorg emulation result of TritonBot. The activity of the robot at every moment is shown along with the CPU/memory usage. Only taking active time into account, Rorg brings 37.2% reduction to CPU usage and 40.2% reduction to memory usage.

services that are required at the emulated moment. Figure 4.5 shows the results from emulation, which reflects the fluctuation in CPU and memory usage when a service is starting or stopping. Rorg reduced CPU usage by 37.2% and memory usage by 40.3% in average in the emulation on average.

Deployment

We also deploy Rorg on the real TritonBot platform to observe its performance. Reduction of resource usage in deployment is the golden standard to evaluate Rorg's performance. However, due to the dynamic nature of the environment, we can only guarantee similar experimental conditions in a relatively short period. Figure 4.6 shows the variation of resources usage in a 60 minutes TritonBot deployment. We had the robot guided three tours and greeted people 30 times

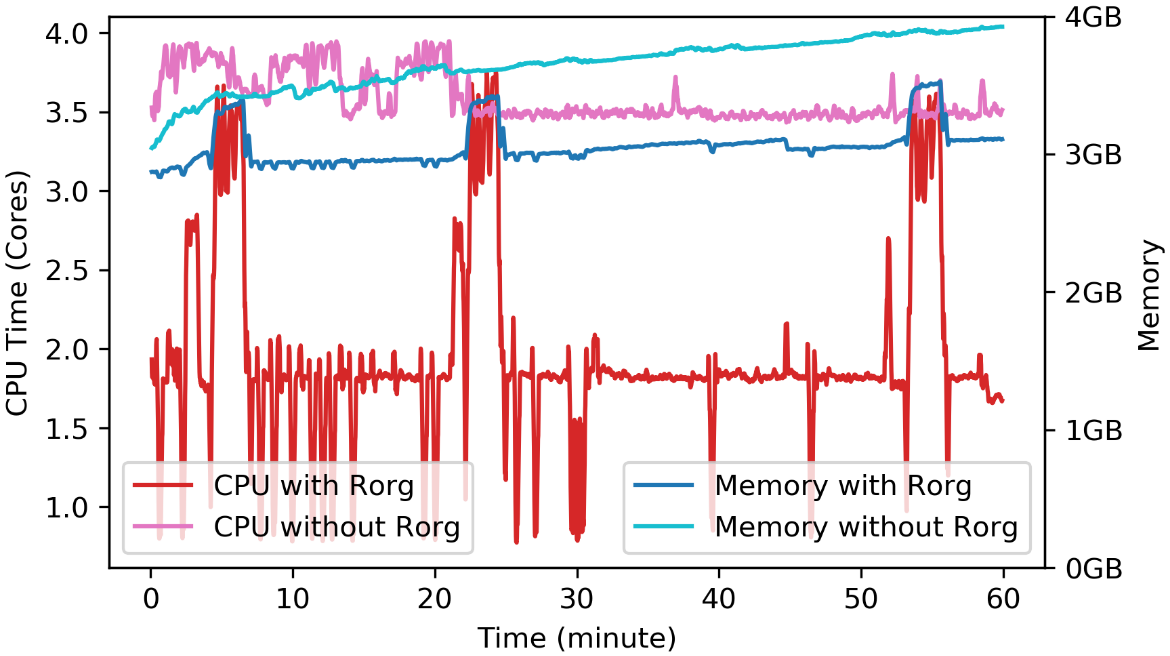


Figure 4.6: Rorg deployment result of TritonBot. We deployed the robot for one hour and collected performance data. With Rorg there is 45.5% reduction in CPU usage and 16.5% reduction in memory usage on average. Although the events do not align in the baseline and in the Rorg experiments, the robot gave three tours and greeted people 30 times in both scenarios.

during the experiment. Rorg reduced CPU usage by 45.5% and memory usage by 16.5% during the deployment on average.

The CPU usage improvements are consistent in three experiments, but the memory usage differs. We found that the simulation and emulation correctly reflect CPU usage by switching on and off CPU-intensive tasks like face recognition and navigation, but the memory usage is less accurate because of difference of sensor/actuator drivers and data playback. Nevertheless, simulation provides a fast way to evaluate the theoretical benefit of Rorg, and emulation provides a fair comparison between the baseline and the system with Rorg.

4.5 Conclusion

Service robots suffer from tedious development workflow and resource contention. We propose software encapsulation and dynamic orchestration to confront these issues. This chapter presents our implementation of software encapsulation and dynamic orchestration on service robot, Rorg, a tool that leverages Linux containers to streamline workflow and reduce resource consumption. Rorg eases developers' effort to orchestrate microservices on service robots, and it enables efficient time-sharing resources to avoid resource contention. We tested Rorg using a service robot application — TritonBot [6], a receptionist and a tour-guide robot. Experimental results by simulation, emulation, and deployment show that Rorg reduces 45.5% CPU and 16.5% memory usage. We release Rorg as open-source software (available at https://github.com/CogRob/Rorg). We also provide configurations to build, deploy, and run TritonBot as an example of using Rorg.

Linux containers have the potential to play a more critical role in service robots. The containers can run inside sandboxes to increase the security of a system. Through the use of containers, it is easy to share "applications" across robots and institutions; consequently, they can serve as a vehicle for increased sharing of research methods. In the future, an end-user can download "container" to a robot just like installing an App on a smartphone.

The underlying ideas in Rorg, including component encapsulation and lifecycle management with automatic dependency resolution, ease maintenance burden and save computing resources in cyber-physical systems. These principles are useful not only in service robots but also in similar computer systems composed of numerous simultaneously running tasks that have limitations in computing resources. Our future goal is to apply Linux containers and other system engineering methods to create a more reliable and scalable robot system infrastructure, as well as exporting our experience from robotics to other fields in engineering. We hope practical infrastructure research will bridge the gap between robots in the laboratory and robots in real life, and we look forward to seeing robots assisting us in daily life in the near future.

Acknowledgements

Chapter 4 contains material from “Rorg: Service Robot Software Management with Linux Containers,” by Shengye Wang, Xiao Liu, Jishen Zhao, and Henrik I. Christensen, which appears in the proceedings of 2019 IEEE International Conference on Robotics and Automation (ICRA 2019). The dissertation author is the primary investigator and first author of this paper. The material in this chapter is ⓒ 2019 Institute of Electrical and Electronics Engineers, Incorporated (the “IEEE”).

Chapter 5

Testing Robots using Broad-spectrum System-level Failure Simulation

Design errors in service robots are difficult to discover due to the need for extensive and resource-demanding testing. As a result, current service robots often encounter unexpected failures that require manual intervention to recover. Unfortunately, these failures are typically non-deterministic and hard to reproduce. This chapter introduces our approach of discovering design flaws and improving the robustness of service robots using broad-spectrum system-level failure simulation. Traditionally, fault injection offers developers with an effective approach to study the long-term reliability of service robots: It allows the developers to investigate the behavior of a robot under erroneous conditions; it also provides a uniformed framework to benchmark the fault resilience of a robot system. However, replicating a broad-spectrum of errors, including hardware failures, is a less-typical practice in robotics. Most of the existing model-based fault injection works in robotics do not scale well to the increasing failure patterns of modern and future service robots, and the other system-level methods only work in simulations. We present a broad-spectrum system-level failure simulation approach with RoboVac, a fault injection framework with broad coverage of failure patterns seen in long-term autonomous service robots. It is capable of injecting failures to modern service robot systems (especially those built with ROS) without specific knowledge about the robot application. Its fault injection methods can generalize to many service robots with minimal effort. We applied RoboVac to TritonBot, a real tour-guide robot application (Chapter 3); it covered all of the failure patterns we have seen in previous deployments, and it helped the developers to discover unknown design flaws and verify the fixes. We made RoboVac open source to help the developers and researchers to build more robust and reliable service robot systems.

5.1 Introduction

As robot technology matures, functioning without intervention is now a significant challenge for long-term autonomy and the wide acceptance of service robots. Previous long-term deployments [6, 2, 3, 4] revealed the failure modes of service robots in home and office settings. Since the developers tested these robots to their best effort during the development, most of the failures happening in the actual long-term deployment are rare, nondeterministic, and hard to reproduce. As a result, the developers had to rely on hidden clues in a massive amount of log to debug, and verifying a fix is non-trivial when the condition that triggers the bug is random or undetermined. Besides, the risks of some unrevealed failures may remain hidden until the robot is deployed and works in the target environment for an extended period, which could lead to catastrophic outcomes. Furthermore, the lack of a systematic method to benchmark the fault-tolerance of a robot system also prevents developers from assessing the resilience of an autonomous service robot system.

Fault injection is a conventional technique in software engineering that intentionally introduces failure to test error handling code paths that are rarely followed in normal operations. With fault injection, developers can reproduce erroneous conditions, and therefore, more easily debug and verify fix in software systems; they can also discover unhandled exceptions and assess the reliability of a system. While this technique is common in software engineering, its application in service robots is not yet well-rounded, and there are three primary challenges:

- First, few fault injection tools work with modern robotic software architecture. Most modern systems leverage middlewares like ROS [11] to form a loosely-coupled distributed system, but none of the popular middlewares [11, 72, 73] provides fault injection capabilities. On the other hand, handcrafting fault injection for each software component is inefficient, inflexible, and unscalable.
- Second, service robots have a broad spectrum of failure patterns. Previous work on mobile platforms [17], telesurgical robots [16], and autonomous vehicles [19] explored model-based fault injection. However, since many failures in service robots are derived from various design flaws in corner cases [6], model-based fault injection does not scale well on service robots.
- Third, service robots are cyber-physical systems with safety concerns. Service robots interact with surrounding environments using a variety of sensors and actuators, which makes failure patterns different from those in pure-software systems. Therefore, fault injection into such systems requires careful design and evaluation of the injection methods.

Our goal is to introduce fault injection to service robots and cover a broad spectrum of failure modes specific to service robots. To this end, we propose RoboVac, an extensible and convenient fault injection framework that covers most of the failure patterns seen in long-term autonomous service robot deployments. Unlike previous work, RoboVac works at the infrastructure layer of a service robot system; it is capable of injecting a broad spectrum of failure modes; as a framework, it is extensible to more kinds of failures. It works with standard robotic middleware, and therefore, it is compatible with many commercial/research robot platforms. RoboVac has three key contributions:

- RoboVac works with ROS [11], one of the most popular robot middlewares. By replacing the basic ROS library with a RoboVac-enhanced version, all existing ROS programs attain fault injection capability from RoboVac without recompiling.
- RoboVac leverages operating system features, middleware communication channels, and other general mechanisms to provide a broad-spectrum coverage of failure patterns. It is capable of injecting failures without specific knowledge about the robot application.
- RoboVac takes the robotics context into account and provides flexibility such as scripting and multi-failure injection. We used RoboVac on a service robot application, TritonBot, and demonstrated our practices to confront the challenges in TritonBot deployment. RoboVac helped us to reproduce previously seen but rare failures and verify the fixes; it also revealed hidden design flaws in the system.

Our experience RoboVac revealed that system-level fault injection could replicate a broad spectrum of erroneous conditions in cyber-physical systems with only minor changes to an existing system. These injected failures expose design flaws and allow the developers to understand the robot's behavior under unexpected conditions, verify fixes, and prepare the robots for future failures. We hope RoboVac will act as a "vaccine" to service robots and prepare them for potential failures, and we believe effective fault injection approaches will enable much future research in robot reliability. The rest of this chapter is organized as follows: Section 5.2 discusses related work about fault injection in robotics and other systems. Section 5.3 summarizes failure modes commonly seen in service robots. Section 5.4 presents broad-spectrum system-level fault injection, the overall RoboVac design, and its fault injection methods. Section 5.5 evaluates RoboVac within the context of TritonBot, a real tour guide and receptionist robot. Finally, Section 5.6 discusses the potential use of RoboVac and concludes this chapter.

5.2 Background and Related Work

Fault injection intentionally introduces failure to the system and therefore reproduces the system state under erroneous conditions [84]. It tests error-handling code paths and enables many other testing methods such as fuzzing [85]. Fault injection has been applied to many areas, including robotics and traditional computer engineering.

Fault Injection in Robotics

Previous works have done some fault injections onto robotics. Christensen et al. use fault injection on mobile platforms to detect failures by comparing actual feedback from sensors with models corresponding to certain failure conditions [17]. Koopman et al. use fault injection to simulate failures at the drive-by-wire middleware layer in automotive systems [86]. A later work, Sabotage [19], a simulation-based fault injection framework for autonomous driving vehicles, provides a strategy to assess automotive safety throughout the development stages. Similar to the ISO 26262 [46] standard that recommends identifying potential safety hazards in the automotive industry, ISO 14971 [87] points out a similar requirement for medical devices. Alemzadeh et al. use fault injection to validate the robustness of safety mechanisms in telesurgical robot systems [16]. Targeting embedded systems that are similar to robots, DESTECS uses fault injection to enhance the dependability of the embedded software for cyber-physical systems [88]. ASTAA uses fault injection on the inter-component level to stress-test autonomy systems to find potential bugs [89]. In addition, many other projects in robotic safety analysis and fault diagnosis also use fault injection to validate their proposals [90, 91]. However, most of these approaches only focus on a few failure models; adapting these approaches to current service robots is a time-consuming task. As a result, they do not scale well on modern service robots.

Fault Injection in Computer Systems

The concepts of software fault injection exist in computer engineering for a long time [92] and are well studied by the academia and the industry [93]. Software fault injection not only helps to discover design flaws and track software bugs, but also emulates hardware errors to evaluate robustness against them. There are three general fault injection approaches: injection of code changes, data errors, and interface errors [94]. Changing a small set of instructions achieves code-change error injections, but altering control flow often requires a large number of code changes. Duraes et al. propose a tool to change the binary code of the application to emulate specific types of fault [95]. Data errors are the most common form of errors in software, and analyze tool injects faults in memory or registers at either random or given time [96, 97, 98]. Interface errors refer to the errors appearing in the communication between different software components (e.g. functions and threads), which usually corrupts input and output [99, 100, 101]. Injecting interface errors does not require software modification or recompilation. Fault injection in glibc (FIG), is a tool that injects errors at the application/library boundary with minimal run-time overhead [102]. FIG is a part of recovery-oriented computing (ROC) [42]. Besides, a wild range of commercial tools in software testing adopt fault injection [103, 104, 105, 106, 107].

Special Challenges in Service Robots

The challenges of fault injection in service robots differ much from that in traditional computer systems or other types of robots. The computer failures like flipping bits can happen on robots, but they are not specific to robots, and the countermeasures from computer system research are equally applicable. For example, memory with ECC capability can detect and correct data bits flips, and majority voter provides reliable computing circuits. On the other hand, application-specific robots often have special "weak points" that are more prone to errors. RAVEN II, a telesurgical robot, only has three major software components: network component, control component, and console component [16]; the researchers were able to pinpoint a few common failure patterns that cause severest faults. But service robot systems are more complicated and scalable systems; they consist of many loosely-coupled components and thus have a broad spectrum of failure modes. TritonBot [6], a tour guide and receptionist prototype robot, has about 41 components for face recognition, voice recognition, voice synthesizer, localization, navigation, knowledge database, etc.; the developers have observed failure modes scattered in hardware, software, networking, and so on during its initial deployment. As a result, failures in special-purpose robots and general computer software only share a small portion of failures seen in service robots; there is little experience of applying fault injection to modern service robot systems.

5.3 Failure Modes in Service Robots

Long-term deployments of service robots reveal unexpected failure patterns in the development phase. Since the primary goal of RoboVac is to cover a broad spectrum of failure modes, failures exposed in long-term deployments hints the major failures types that RoboVac targets. A few long-term autonomous service robots have been deployed for months, and their developers have reported their failure modes: TritonBot [6] is a receptionist and tour-guide robot that integrates many machine learning and robotics technologies into a single-robot system; it is designed to discover long-term robotic reliability issues in home and office settings. The classic Sage [23], CoBots [21], STRANDS [3], and BWIBots [4] are also impactful long-term autonomous service robot systems that reported reported deployment experiences and failure statistics (Section 2.1). Besides, some previous work also studied failure in other robot systems [7, 108].

In general, failures in these robot systems fall into one of the three categories: hardware failures, software failures, and networking failures.

5.3.1 Hardware Failures

The most obvious difference between a service robot and a traditional computer system is that service robots are cyber-physical systems that interact with the environment using sensors and actuators. Although other failures such as mechanical and power failures happened on TritonBot [6], they either affect sensor readings or actuator functioning, or they are beyond the reach of software failure injection. The following of the section focuses on sensors and actuators failures.

Sensor Failures

Sensors have three failure patterns: no reading, wrong readings, and sensor noise. TritonBot once experienced no reading failure from its laser scanner: the failure happened three times during a one-month deployment, and we had to reconnect the system battery to recover the robot. Our other robot had a faulty ultrasonic sensor that kept returning maximum readings, and we had to replace the damaged sensor to solve the issue. In low-light conditions, the consumer-grade camera on TritonBot has severe luminance and chrominance noise, and it affects face detection on the robot.

Actuator Failures

Actuators are components that move and control a mechanism or system in the robot and its surroundings. TritonBot once had a slipping wheel because the gas spring that pushes the wheel-assembly to the ground was worn out. In another TritonBot incident, the robot was not able to dock into its charging stand to start charging because of a failed electrical connector. In addition, the STRANDS robots had issues of getting stuck on a carpet and causing navigation failures [3]. In all cases, the actuators did not perform the desired action precisely.

5.3.2 Software Failures

Active deployment and iteration on robot software bring many software issues into long-term autonomous service robots. There are three common software failure patterns: unexpected crash, slow response, and incorrect behavior. Each of them has different causes and symptoms.

Unexpected Crash

Unexpected software crash happens because of software bugs, assertion error, out of memory, and other causes. The STRANDS project explicitly reported crashes in their beta software [3]. In the early deployment of TritonBot, voice recognition software had memory leak issue and crashed every few hours. Unfortunately, unexpected crashes are very common on robots, especially in robot prototypes that heavily utilize open-source software.

Slow Response

Resource contention and concurrency issues lead to slow software responses. On TritonBot, the face recognition pipeline has seconds of latency when the robot navigates around, as the software compete for the limited onboard resource due to power and weight constraints. The STRANDS robots also had task time-outs in their software and even deadlocks that caused infinite delay [3].

Incorrect Behavior

Imperfect algorithms designs or corner cases lead to incorrect software behavior. In a previous TritonBot deployment, the face recognition pipeline recognized a round clock as a person; it kept talking to the clock on the wall until the developers turned it around. Similarly, STRANDS robots used to get stuck near obstacles [3] due to software design flaws.

5.3.3 Networking Failures

Many mobile robots leverage network to access to cloud services or other resources. The throughput, latency, and connectivity often vary by time, location, and Internet service providers (ISPs). Imperfect networking caused problems on both the TritonBot [6] and the STRANDS robots [3]. There are two primary modes in networking failures: availability issues and performance fluctuation.

Availability Issues

Unavailable connection affects the robot components that rely on the Internet to function. TritonBot occasionally suffers from temporary connection loss when it roams between multiple access points, as the robot works in a $6,000$sq. ft. open space. It also encountered a longer period of Internet unavailability when the network devices are under maintenance because developers do not have control over the networking infrastructure in the office building. Similar situations also apply to other service robots in the home and office settings.

Performance Fluctuation

TritonBot usually has high-bandwidth and low-latency wireless Internet access (125.91 Mbps download, 139.21 Mbps upload, 8 ms round-trip latency connecting to an off-site datacenter as of writing). However, we also observed performance fluctuation when there is a major event in the same area or when the robot moves into WiFi blind spots. The latency sometimes spiked to a few seconds, and the throughput dropped to a few kilobytes per second. TritonBot [6] heavily depends on Google Cloud Speech API [56] to perform speech recognition, but the cloud service once refused to serve TritonBot because it could not stream audio in a near-real-time rate.

5.4 RoboVac Design

RoboVac is designed to integrate into an existing robot system with minimal effort; it enables the developers to coordinate the fault injection timing and assert properties of the system during testing. It provides various fault injection methods to cover all the failure modes discussed in Section 5.3. This section presents framework architecture and fault injection details of RoboVac.

5.4.1 RoboVac Architecture

A RoboVac system has three major components: a RoboVac controller that coordinates fault injection in the system, multiple programs loaded with RoboVac fault injection client libraries that perform the actual fault injection, and test scripts that specify the injection timing and type of fault. The three components communicate via a remote procedure call (RPC) interface [82]. Figure 5.1 shows the RoboVac framework.

Cooperating with Existing Robot Systems

To realize the easiest adoption of fault injection on an existing robot system, RoboVac does not require recompilation or modification of any existing code. It takes different approaches for non-ROS and ROS programs: For non-ROS programs, a user can "preload" the RoboVac client library into a Linux process using the dynamic linker features (`LD_PRELOAD`); this approach also works on the Python programs — the RoboVac client library is loaded into the Python interpreter. For C++ and Python ROS programs, RoboVac replaces the ROS library (`libroscpp.so` or `rospy`) to load and coordinate with the RoboVac C++ or Python library. Both C++ and Python RoboVac client libraries provide the same functionalities; they spawn a looping thread to poll fault injection directives from the RoboVac controller, and they pass fault injection directives to the rest of the program. The RoboVac client library also provides some universal fault injection

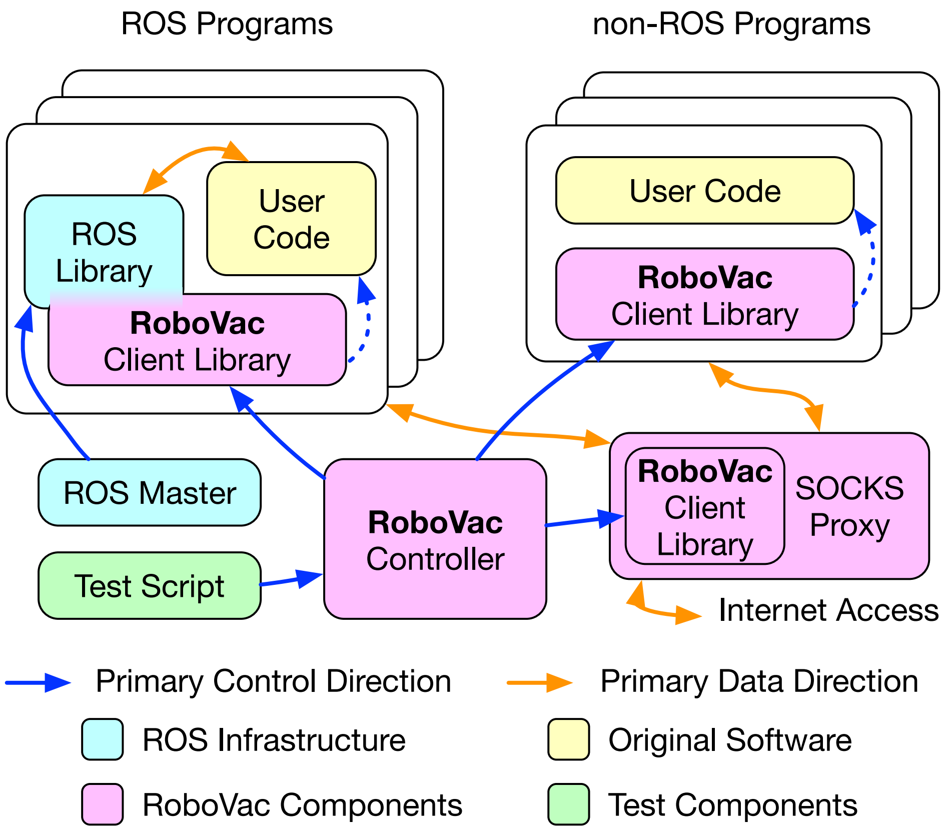


Figure 5.1: The architecture of a system with RoboVac. RoboVac has three parts: controller, clients, and the test script. The RoboVac library (`librobovac.so` or `robovac` Python package) is loaded into ROS and non-ROS programs; it passes fault injection directives to the rest of the programs, and it has some basic fault injection capabilities such as crashing the program. In ROS programs, the ROS library (`libroscpp.so` or `rospy`) works with RoboVac to decide when and what to change in the middleware layer. Besides, user code can optionally interact with RoboVac library to decide whether to inject failures (the dotted lines). The SOCKS proxy server also receives commands from RoboVac to reshape the network traffic between the robot system and the Internet.

capabilities. A previous work, FIG [102] used similar techniques such as preloading a dynamic library in traditional computer programs, but RoboVac inject errors to a service robot at a system/ infrastructure level.

RoboVac provides application programming interfaces (APIs) for the test script to introspect the system states and the client features, as well as to enable and disable fault injection. The test scripts can specify a particular RoboVac and decide when and what types of faults to inject to the system. The users may program the test script in any RPC-supported programming languages; they can even perform non-RoboVac actions to set up and tear down a test in the same script.

5.4.2 Fault Injection Methods

RoboVac leverages operating system features, middleware communication channels, and other general mechanisms to provide a broad-spectrum coverage of failure patterns. This section explains the implementation details of these mechanisms.

Injection to Middleware (ROS)

Modern service robots often leverage middleware like ROS [11] to decouple the software components: each software component exposes abstract interfaces through the middleware, but the implementation details remain hidden from other components. The exposed interfaces provide a convenient means to manipulate the input and output of a software component, and RoboVac leverages them to inject failures. While researchers have developed other middlewares for robots [11, 72, 73] and their constructions are similar, RoboVac firstly supports ROS [11]. ROS provides three channels of communication: ROS topics, ROS service calls, and ROS actions. RoboVac supports fault injection through all these three channels. Figure 5.2 demonstrates all of the fault injection methods on the middleware level.

ROS topic is a publish-subscribe framework: a ROS program can "publish" a message to a topic, and the ROS program that "subscribes" to the topic receives the message through a function callback. RoboVac supports two mechanisms to inject failure topic: message drop and message redirection (Figure 5.2 (a)). (1) Message drop: RoboVac can disable the callback upon reception of a message at a certain probability or on a per-subscriber basis (Figure 5.2 (a) ⑤); it can also stop the publisher from publishing a message to a topic (Figure 5.2 (a) ③). This mechanism simulates the drop of messages due to sensor issues, slow computation, or network errors. (2) Message redirection: ROS itself provides a "remapping" feature for the topics, but the

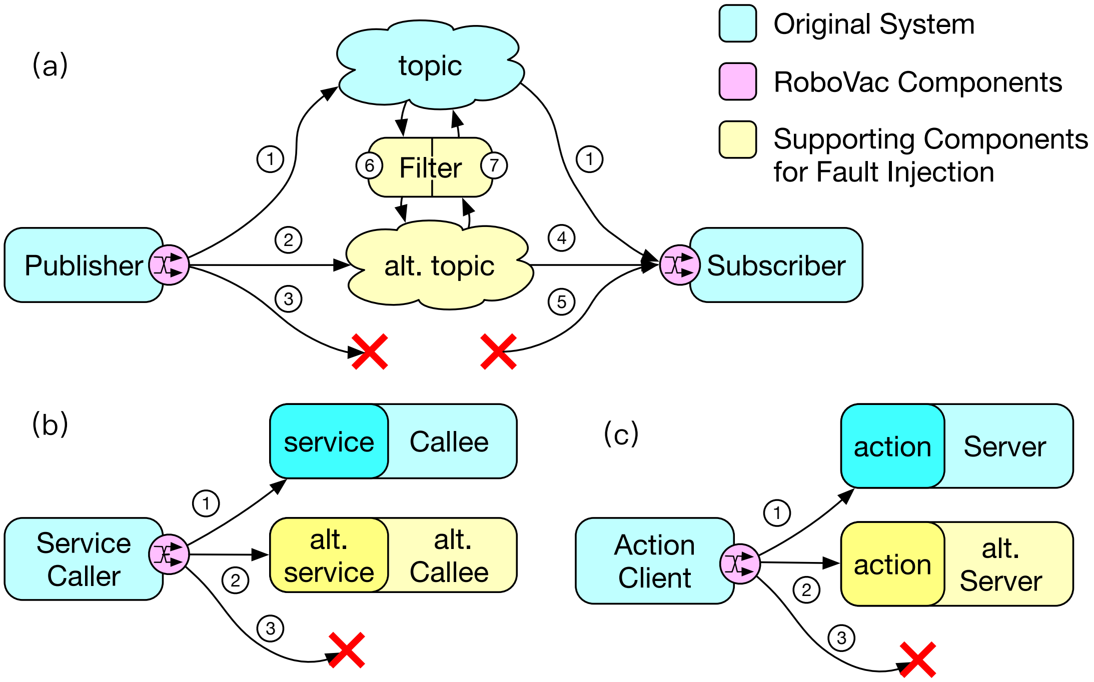


Figure 5.2: Fault injection methods at the middleware (ROS) level, including injecting to ROS topics, ROS services, and ROS actions. (a) ROS topic fault injection: ① is the message flow in without fault injection. ② and ④ redirects the message flow; they are used with an additional filter program (⑥, ⑦) that manipulates the message and injects failures in the message flow. ③ and ⑤ drops messages at a probability. (b) ROS service fault injection: ① is the normal service call. ② redirects service call: an alternative endpoint handles the service call and injects failures. ③ aborts service calls at a probability. (c) ROS action fault injection. ① is the normal action execution flow. ② redirects the action: an alternative endpoint handles the action execution. ③ aborts action at a probability in different styles such as immediate failure, timing out, etc.

mappings are static and must be set before a program starts. RoboVac allows seamless and online redirection of messages (Figure 5.2 (a) ②/④), which is useful when injecting noise to messages. With message redirection, the developer can insert a "filter" program between the publisher and the subscriber to add noise or modify other properties of the messages (Figure 5.2 (a) ⑥/⑦). As an example, to add noise to laser scan ranges, a filter (Figure 5.2 (a) ⑥) subscribes to the original scan topic (Figure 5.2 (a) ①), but RoboVac controls the subscriber to read manipulated messages from an alternative topic (Figure 5.2 (a) ④). The user code in the subscriber is not aware of the redirection and thus requires no change. To ease the developer's burden, RoboVac provides a few common filters such adding Gaussian noise to a few common ROS message types.

ROS service is a unary RPC framework; a ROS program calls on a "service" and receives the response from another ROS program. RoboVac provides the service fault injection by manipulating the service call (Figure 5.2 (b)): RoboVac either sends redirected service call to an alternative callee which implements a user-defined failure behavior (Figure 5.2 (b) ②), or drops the call to simulate temporary unavailability of the callee at a user-defined probability (Figure 5.2 (b) ③).

The last ROS communication mechanism is ROS action. It is an asynchronous task execution mechanism (in other words, asynchronous RPC with streaming response). In addition to user-defined requests and responses, the ROS action library (`actionlib`) tracks an independent and universal state such as pending, running, failed, aborted, etc. RoboVac leverages the action states to simulate failure patterns such as action failures, timeouts, or aborts without knowing the action details (Figure 5.2 (c) ③). It also supports forwarding an action call to an alternative endpoint and modifies request/response to simulate fault behaviors (Figure 5.2 (c) ②).

Injection to Linux Processes

Most service robots (especially those powered by ROS) run on the top of the Linux operating system. RoboVac leverages Linux signals to simulate many common error patterns in Linux processes. Table 5.1 shows the signals used by RoboVac and their corresponding failure patterns. With these signals, RoboVac can reproduce failure patterns such as unexpected program crash, temporary or permanent program stall, or unavailability of some programs.

RoboVac sends most of the signals to a process from the client library thread in the process itself. However, it sends `SIGCONT` from a proxy program with appropriate privilege, because a `SIGSTOP`ed process cannot send a signal. While `SIGSTOP-SIGCONT` pair can pause a program, the program can detect `SIGCONT` and exhibit different behavior from simple CPU throttling. We are working on an alternative approach to use a transparent "cgroups freezer" mechanism, which is especially convenient for the robot systems that leverage Linux containers

Table 5.1: Linux signals used to inject failures into Linux processes.

Signal	Usage	Failure Patterns
`SIGKILL`	Kills a process (cannot be caught)	Simulates a program encounters an erroneous condition and quits without cleaning up
`SIGSTOP`	Pauses a process (cannot be caught)	Simulates a program temporarily hangs; resumes on `SIGCONT`
`SIGCONT`	Continues a process	Continues a stopped process to simulate it returning to running
`SIGINT`	Exits a process gracefully	Simulates a process terminating itself gracefully

for software deployment [109].

Injection to Networking

RoboVac can introduce extra latency, limit the bandwidth, or completely cut off the connection when a robot software component accesses to the Internet. Both the latency and bandwidth are adjustable. Therefore, RoboVac simulates five types of imperfect Internet connectivity patterns: normal, low throughput, high latency, low throughput with high latency, and disconnection from the Internet. It uses a SOCKS proxy [110] to intercept and reshape the networking traffic (Figure 5.1). Robot software connects to the Internet through a SOCKS proxy [110]. The proxy server forwards the traffic in full speed during normal conditions, but it reshapes the traffic pattern under fault injection. RoboVac SOCKS proxy introduces no performance overhead for local ROS traffic as it only affects the Internet traffic. While other methods are also available to manipulate Internet connectivity (i.e. `iptables`), SOCKS proxy provides a much simpler programming interface to filter, audit, and generate statistics about Internet traffic.

Ad-hoc Injection to Specific Software

Aforementioned methods are generic and cover a significant portion of failure patterns. However, some frequent and vital failures are rarely covered by any general approaches. A common example is that imperfect algorithms do not handle corner cases well. To cover the special cases, RoboVac supports the ad-hoc fault injection to specific programs. In this approach, developers implement fault behavior inside the program and specify several injection entries. At each injection entry, the program checks with the RoboVac client library to determine whether to inject fault behavior. Therefore, RoboVac framework still holds control over the fault injection. For example, we modify `movebase`, the common ROS navigation software, to inject "ghost obstacles" in the occupancy grid to test the robot's behavior when it is stuck near obstacles. Although ad-hoc fault injection is not generic or flexible, it is effective for testing the specific functions of a program.

Summary

Table 5.2 presents the failure modes, their corresponding fault injection methods, and their usage. These fault injection methods cover all of the failure modes discussed in Section 5.3.

5.5 Evaluation

This section evaluates RoboVac in the context of real long-term autonomous service robots. First, we apply RoboVac to the people detection subsystem in our tour-guide TritonBot to study the impact of sensor noise amplitude on performance. Then, we present our experience of using RoboVac on the entire TritonBot system to help reproduce failures and verify fixes. Lastly, we evaluate the performance impact of RoboVac on a service robot system to ensure that the performance impact of RoboVac does not affect the behavior of the robot.

Table 5.2: RoboVac covered failures and injection methods.

Failure Mode	Injection Method	Fault Injection Details
Sensor no reading	Middleware topic	Cuts the subscriber off from message flow
Sensor wrong reading/noise	Middleware topic	Redirects the subscriber to a manipulated message flow
Actuator malfunction	Middleware topic	Redirects the actuator driver to a manipulated message flow
Actuator malfunction	Middleware service, action	Intercepts the actuator command and always returns failure
Software crash	Linux signal	Sends `SIGINT` or `SIGKILL` to the process
Software stall	Linux signal	Sends `SIGSTOP` then `SIGCONT` to the process
Algorithm result error	Ad-hoc fault injection	Manually modifies the software to include error
Algorithm result error	Middleware	Manipulates data or control flow in the pipeline
Network issues	Network fault injection	Manipulates Internet connectivity, latency, or bandwidth

5.5.1 Example Case: People Detection under Sensor Noise

We use RoboVac to study the influence of sensor noise on the TritonBot human detection subsystem. TritonBot uses both visual perception and the laser scan to confirm the existence of a person before starting a dialogue with a user: it incorporates the open source OpenFace [53] to recognize face and an open source leg tracker [83] to detect human legs. TritonBot only starts talking to a person when it spots a front face and detects legs within a certain range (1.7 m) in front of it. However, the camera image is noisy in low-light situations, and strong light interferes with the laser scan beams. Face detection and leg detection algorithms are sensitive to sensor noises. With more noises, the true positive rate of the algorithms becomes lower, and the algorithms need to see more samples before they can confirm a positive detection. As prior work requires manually setting up a system for emulating those failure situations [111], RoboVac can investigate the impact of noises with minimal effort.

The experiment runs the person detection pipeline on a computer with identical hardware configurations as the TritonBot [10]. Since the sensor records data in a standard format (ROS message type), RoboVac can inject noise to the corresponding data readings through middleware injection. A test script sets up the fault injection and invokes `rosbag` to replay the sensor data. We add Gaussian noises to both image and laser scan readings. Figure 5.3 shows the latency distribution according to the injected noise levels. When noises added on data from both sensors are low, TritonBot recognizes the person as soon as he approaches and looks at the robot (within 4.72s); as noises get higher on either sensor, TritonBot spends more time to react to human appearance; when noises are exceptionally high, the robot cannot recognize human at all. The observations align up well with our expectation of real failure cases.

5.5.2 RoboVac on TritonBot

In prior work we built TritonBot to serve as building receptionist and a tour-guide robot [6]. It talks to visitors, remembers the visitors' face, and navigates around to show places to the visitors. The initial one-month TritonBot deployment revealed many issues on TritonBot. We used RoboVac to recreate the failure conditions of TritonBot to debug the design flaws in TritonBot and fix the handling behavior.

Table 5.3 shows the failure patterns on TritonBot and their corresponding injection methods; RoboVac covers all of them. While some failure patterns do require specific model-based fault injections to reproduce, most of the failures are generated using generic fault injection methods. The general injection mechanisms do not require developers to have knowledge of the specific implementation. For example, RoboVac reproduces the always-fail auto-docking issue on TritonBot by injecting faults to ROS actions, and the developer only specifies the injection target (the behavioral state machine) and the action name (`/dock`). The developers can use the general

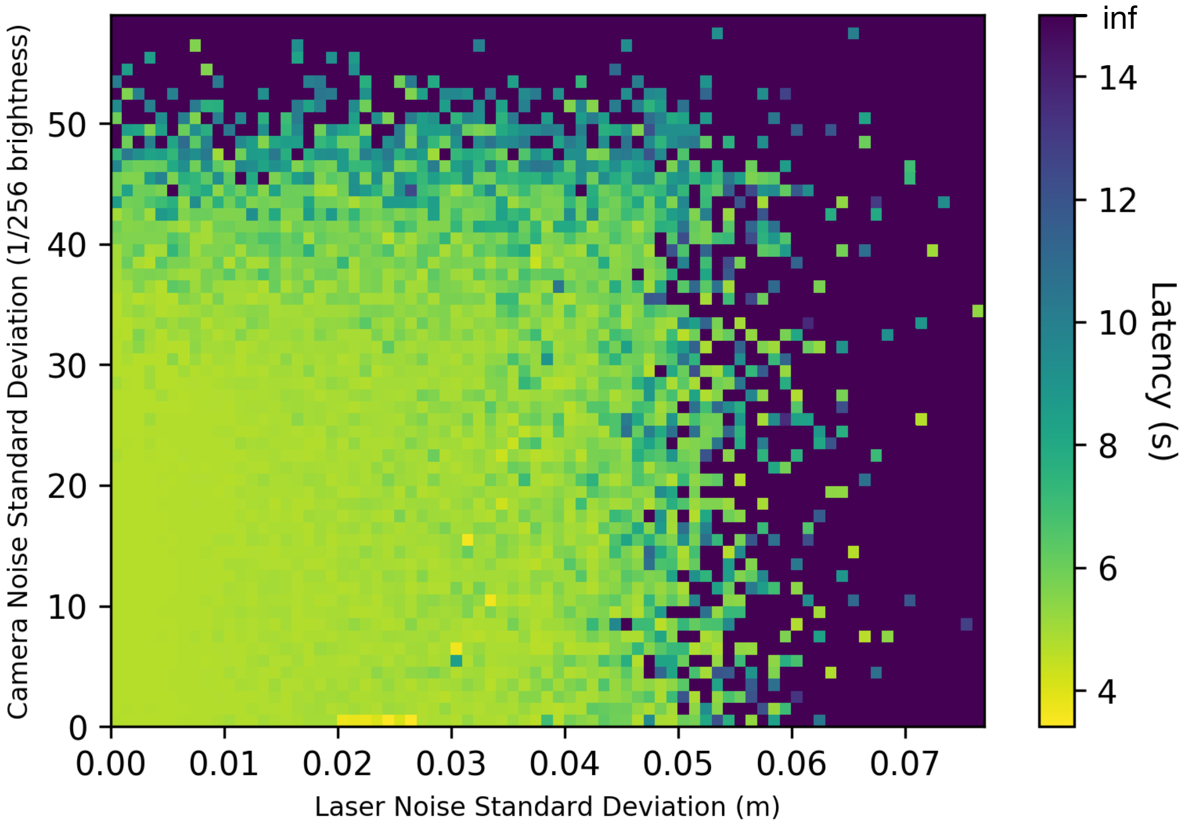


Figure 5.3: Latency to detect a person v.s. noise levels injected on laser scan ranges and image pixel values. The heatmap represents the latency between the time that a person starts walking toward the robot and the time that robot detects the person. The noise injected to both sensors are independent Gaussian noises: The laser scan noise $\varepsilon_l \sim \mathbf{N}(0, \sigma_l^2)$ is added to original laser scan ranges (σ_l is the x-axis), and the pixel noise $\varepsilon_i \sim \mathbf{N}(0, \sigma_i^2)$ is added to the RGB channels of the pixels in the image (σ_i is the y-axis). The latency grows larger as higher noise is injected to the system.

fault injection methods in RoboVac to test many aspects of the systems; furthermore, RoboVac has the potential to work with fuzz testing methods to automatically discover design flaws.

We also use RoboVac to inject sensor and actuator failures on TritonBot. For example, we simulate a free-spinning right wheel by intercepting the velocity commands and setting the right wheel velocity always to be 0, while the left wheel velocity remains untouched. However, injecting faults to actuators has a potential safety hazard — the injected failures may bypass the last safety barrier that prevents the robot from causing catastrophic damages. We engineered our

Table 5.3: Failures in TritonBot, corresponding fault injection methods and handling behaviors.

Failure Effect	Injection Method	Handling Behavior
Laser scanner does not report measurements	Cuts off message flow in the `base_scan` topic	Emails the developer if `base_scan` is idle for minutes
Charging port failure	Mark `dock` action as always fail	Gives up docking after five attempts and alerts the developers
Cloud speech API not reachable	Disconnects Internet from the speech recognition client	Goes back to the charging dock and waits for Internet recovery
Voice recognition software random crashes	Kills the program using Linux signals	Restarts the software in case of dead process
Voice recognition returns wrong result	Redirects the message subscriber to a filtered topic	Enables fuzzy matching in the recognition results
Face recognition slow response	Pauses and restarts the face recognition client.	Monitors the latency of the pipeline and alerts the developer to abnormality

emergency-stop protection mechanisms always to use original sensor data and send final actuator commands, but we consider triggering these mechanisms as an overall failure.

5.5.3 RoboVac Performance Overhead

Computing resources are limited on service robots, but fault injection requires some additional computation. We evaluate the performance overhead of RoboVac using an emulation of the TritonBot system with the robot control state machine trace collected in a previous deployment (February 6, 2018, a typical day of TritonBot deployment). Figure 5.4 compares the CPU usages of an emulated TritonBot system under two conditions: (1) with fault injection to the camera and laser scan data using RoboVac, (2) without fault injection to either mentioned sensor. According to the experimental results, fault injection with RoboVac only introduce slight performance overhead to the on-robot computer. The CPU usage with fault injection is only 1.29% higher than that without fault injection. The performance overhead is unlikely to change the performance

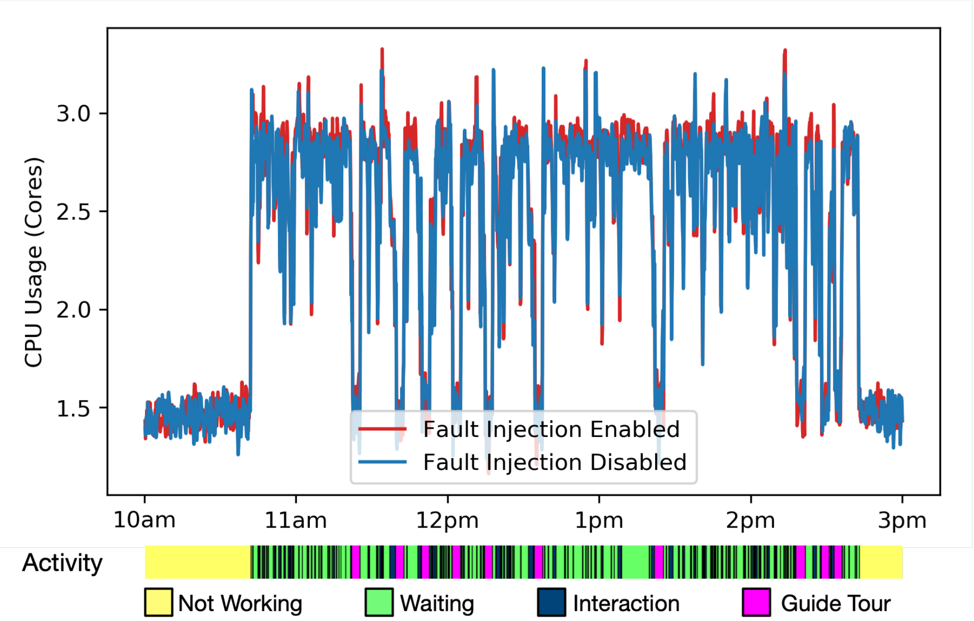


Figure 5.4: The performance impact of introducing RoboVac and fault injection into a system. We ran TritonBot emulator with and without fault injection (noise injection to laser scan ranges and image pixel values), and observed the CPU usage is nearly identical. There is only a 1.29% increase on average CPU usage after enabling fault injection.

profile of the robot, and the behavior of the robot under fault injection shall reflect the state of the robot under an actual failure.

5.5.4 Potential Applications of RoboVac

RoboVac not only supports injecting individual failures into a robot system. As an extensible framework that works at the system/infrastructure level, it has many potential usages that otherwise not achievable by other fault injection methods in robotics.

One of the possible use is fuzzing [85] at the system scope. RoboVac can discover potential fault injection points automatically. RoboVac can randomly inject a combination of its built-in failure patterns and extra user-defined failures at different severity levels. Its scripting features can monitor and system and record the undesirable behavior of the robot along with the failures being injected.

We can use RoboVac to build a benchmark suite and quantify the robustness of different robot systems using the same standard. Many robot systems are based on ROS, and the majority features of RoboVac works with ROS programs without recompiling them. Therefore, we can design a common benchmark suite that covers common abnormal conditions in a robot design. The benchmark suite will be in the form of an automated script that first introspects the system and finds out system-specific configurations (e.g., the camera topic name) with the help of the user. Then, RoboVac can inject failures in the benchmark suite into the system while monitoring the system status. The results from different robot systems represent their resilience to different failure modes, and they are comparable under the same benchmark suite.

A pitfall in reliability engineering is that some components rarely fail, and the developers tend to assume they would never fail. Google intentionally introduced downtime to one of their microservices and break the developers' false assumption so that they can prepare the dependent service for failures [40]. RoboVac can achieve the same idea in service robots. Since RoboVac has a very low resource usage overhead, it is possible to seldomly inject failures while the robot is operating. Although the robot may encounter more problems initially, as the developers harden the robot, the robot system will become more reliable in the long run. With this idea, the developers will be able to discover potential unexpected situations in a much shorter time and, therefore, prepare the robots for failures earlier.

While RoboVac is not designed to replicate security issues in service robots, it is possible to test a robot system's security flaws using the fault injection methods in RoboVac. Instead of injecting noise into the pipeline, we can inject intentionally constructed images to trick the image recognition algorithms, or we can perform a man-in-the-middle (MITM) attack on selected SSL connections. Then we can test whether the image recognition algorithm is robust or if the programs that use the Internet validates certificates correctly.

While the abovementioned are ideas that have not been implemented, they remain as our future work. We believe RoboVac opens many research opportunities in the reliability of service robots that are worth pursuing at the system level.

5.6 Conclusion

Reliability is a key factor for service robots to achieve long-term autonomy, and fault injection is a powerful tool for the developers to discover hidden design flaws and assess the reliability. This chapter proposes and implements RoboVac, a software fault injection tool that supports modern robotic software architectures and covers a broad spectrum of failure patterns previously seen in some influential long-term autonomous service robots. RoboVac can inject four types of failures as of writing: middleware-layer failures, software process failures, networking failures, and application-specific failures. Most of its fault injection methods can apply to many service robots with minimal effort. We test RoboVac with our tour guide and receptionist robot – TritonBot. The experimental results show that RoboVac is able to inject multiple failures into the system effectively; its fault injection methods cover all of the previously seen failure patterns on the robot; the performance overhead during fault injection is acceptable.

Through the RoboVac experience, we verified that system-level fault injection could replicate erroneous conditions in cyber-physical systems with only minor changes to the existing system. These injected failures can help the developers to understand the robot's behavior under unexpected circumstances, verify fixes, and guide a fail-safe robot design. We hope RoboVac can help robotic reliability researchers to find new research problems and do experiments on real robots, and we hope broad-spectrum system-level fault injection will directly or indirectly improve the reliability of other kinds of systems.

Acknowledgements

Chapter 5 contains material from “RoboVac: Fault Injection for Long-term Autonomous Service Robots,” by Shengye Wang, Xiao Liu, Shixin Li, Jishen Zhao, and Henrik I. Christensen, which is currently being prepared for submission for publication. The dissertation author is the primary investigator and first author of this paper.

Chapter 6

Design Considerations in Long-term Autonomous Service Robot

In the past two years, TritonBot was our primary research platform to experiment with new ideas. This chapter presents our continued experience and results from the TritonBot project. We regularly did maintenance, fixed issues, and rolled out new features on TritonBot; we deployed TritonBot several times to study reliability challenges in long-term autonomous service robots; we identified reliability engineering challenges in three aspects of long-term autonomy: scalability, resilience, and learning. For the TritonBot experience, we formulated techniques and summarized design principles to confront challenges in robotic reliability engineering. Our experience shows that proper engineering practices and design principles reduce manual interventions and increases general reliability in long-term autonomous service robot deployments.

6.1 Introduction

Recent robots often perform perfect demos under controlled settings and close human supervision, but they tend to be less reliable when working autonomously for an extended period in unstructured environments. To pave the way toward long-term autonomy, we built TritonBot (Chapter 3) to study reliability challenges in developing and deploying a long-term autonomous service robot that interacts with people in an open environment. TritonBot is a robot receptionist and a tour guide deployed in an office building at UC San Diego. It greets visitors, recognizes visitors' faces, remembers visitors' names, talks to visitors, and shows visitors the labs and facilities in the building. Inspired by the DevOps practices in software engineering [40], we reconfigured the robot platforms, developed infrastructure, and invented some other techniques to support our long-term development and experiments. Chapter 3 summarized lessons in the initial TritonBot development; this chapter presents our continued efforts in making TritonBot more reliable and our results from the TritonBot project.

Long-term autonomy consists of three primary factors: scalability, resilience, and learning: Scalability enables a robot system to grow and gain more features smoothly. Resilience allows a robot to adapt to environmental changes and tolerate transient errors. Learning helps a robot to benefit from experiences and become more capable over time. Our contributions in this chapter are: (1) Identification of failure modes and reliability challenges using the TritonBot deployment history. (2) Formulation of engineering practices that reduce manual interventions during long-term robot deployments. (3) Collection of design considerations that increase the reliability of long-term autonomous service robots. We tested the engineering practices and design considerations on TritonBot, but they are also applicable to other robot systems with scalability, resilience, and learning requirements.

The chapter is organized as follows: Section 6.2 presents our improvements on TritonBot since the initial deployment. Section 6.3 presents the TritonBot deployment history in the last two years and problems during the deployment. Section 6.4 describes our efforts in making the TritonBot system scalable in its long-term evolvement (Scalability). Section 6.5 shows our practices in making TritonBot resilient to failures (Resilience). Section 6.6 presents our attempts to improve TritonBot over time both autonomously or with the help from developers (Learning).

Section 6.7 summarizes a few practical engineer practices that developers can use as a “toolbox” to design reliable long-term autonomous service robots. Finally, Section 6.8 concludes this chapter.

6.2 TritonBot Improvements

We continued to use TritonBot as our primary research platform to study reliability and long-term autonomy. After the initial deployment (Chapter 3), we improved various aspects of the TritonBot system. These improvements make TritonBot a more friendly robot, which allows us to experiment with reliability engineering further.

TritonBot Components

Figure 6.1 shows the current software architecture and the dataflow in TritonBot. We use many open-source components to build the system, but we wrote more than $100,000$ lines of code to integrate the system in addition to the open-source components we use. Table 6.1 presents an estimate of lines of code in TritonBot. With this complexity, we believe TritonBot can represent a service robot in the real world.

After the initial deployment, We applied Rorg (Chapter 4) to manage the components on TritonBot. The TritonBot software splits into 41 Rorg services. The software went through 756 updates, and the Rorg configurations were updated 100 times. We used RoboVac (Chapter 5) to test not only the individual components such as face recognition and navigation but also the robot’s behavior at a larger scale.

TritonBot Behavior Control

TritonBot’s high-level behavior is defined by a state machine implemented using a standard state machine library, SMACH [112]. Figure 3.5 in Chapter 3 demonstrates the simplified

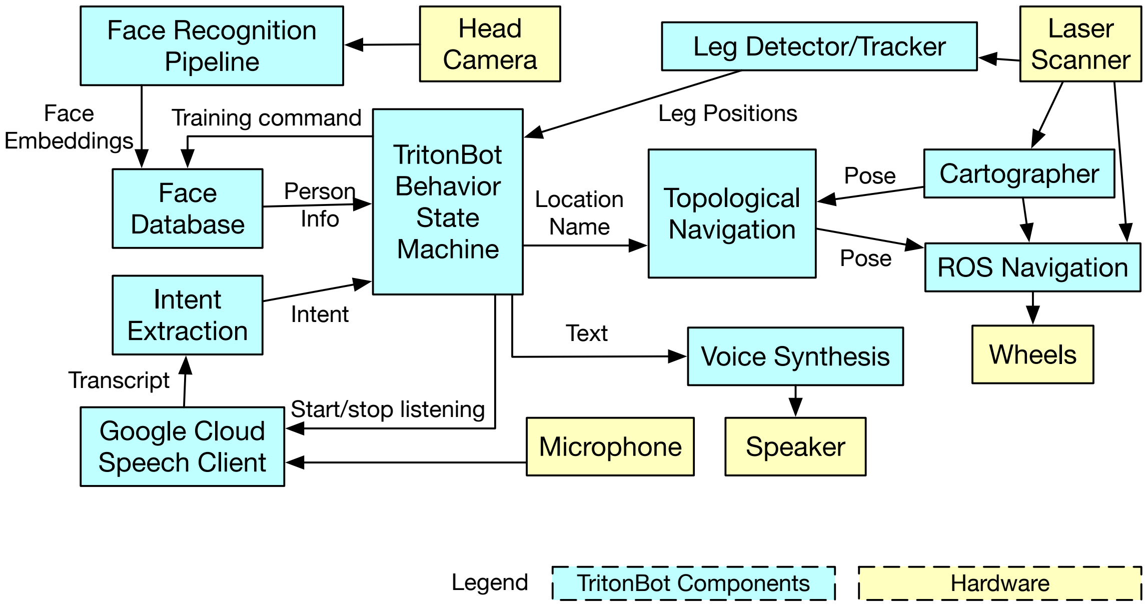


Figure 6.1: A block diagram shows the primary components in TritonBot and the dataflow between them. A state machine controls the behavior and standalone components: face recognition, voice recognition, leg detection and tracking, localization, and navigation.

state machine. Although other models such as behavior trees [113] or hierarchical task networks [114] are sometimes more powerful or flexible, we leave them as options for future TritonBot improvement.

We enhanced the partition of the state machine, and it roughly splits into four parts: idle, waiting, interaction, and tour guiding. The robot is in the idle states when it is charging; it also monitors and responds to the developer's commands such as undocking or moving to the ready position. When the robot is in the waiting states, it discovers visitors using the output from the face and leg detection; it also monitors commands so it can go back to the charging dock when instructed. The robot enters the interaction states when it discovers a visitor, and if the visitor accepts the tour, it enters the tour states. In these states, the state machine coordinates different subsystems such as face recognition, leg detection, speech recognition, speech synthesis, localization, navigation, and others to realize TritonBot's features. In some states with chances of failures (e.g., speech recognition or navigation), TritonBot usually attempts three times before it

Table 6.1: An estimation of lines of code (LOC) in TritonBot.

Language	Number of Files	Lines of Code
Python	367	38953
C/C++	374	28069
Build Rules (CMake and Bazel)	334	16437
Rorg Service Defination	188	7847
Shell Scripts (sh, bash, zsh)	340	7633
Protocol Buffers (ProtoBuf)	100	4405
XML	84	3711
YAML	41	2427
ROS Launch	90	1954
Text	149	1742
ProtoBuf Data (ASCII ProtoBuf)	45	1509
diff	15	722
Lua	9	626
ROS (msg, srv, action)	28	147
JSON	3	18
Total	2167	116200

gives up.

In this update, we also added a special stationary "trivia-only" mode that only provides trivia questions. We put the robot into this mode for demos and long-term unsupervised deployments because it does not move, and thus it imposes minimal risks on the surroundings. This mode eventually contributed to a three-month fully unattended deployment.

Leg Detection

After the initial deployment, we realized TritonBot lacked an efficient approach to check the user's proximity when it is not directly facing the user. Therefore we enhanced TritonBot with an open-source, random-forest-based leg detector to discover people nearby [83]. The input to the leg detector comes from the laser scan, and the leg detection is used in two scenarios.

First, TritonBot uses leg detection to confirm a person to greet. TritonBot primarily uses face recognition to discover users. However, because of the wide field-of-view camera, TritonBot used to talk to uninterested visitors that walk towards the side of the robot; occasionally, TritonBot talks to the air for the false-positive face detection. With leg detection, TritonBot checks if a visitor is within a small range in the front of the robot (Figure 6.2 (a)). Usually, the range is a rectangle of 2 meters length and 1.6 meters width, but the stationary trivia-only mode uses a much smaller range (1 meter length and 0.6 meter width in front of the robot).

Second, TritonBot checks if a visitor is following the robot during a tour. TritonBot verbally communicates with a visitor if it does not detect a visitor within a square of 3×3 meters (Figure 6.2 (b)), and it aborts the tour if both leg detection and verbal communication fail.

Although the leg detector incorporates the static map of the environment to filter static obstacles that look like legs, it still reports some false-positives (and usually more false-positives than false-negatives). Therefore, our use of leg detection is a supporting role such that a false positive has less severe outcomes.

Voice Recognition and Dialogue

In the initial deployment, a common human-robot interaction flaw was that the visitor gave up and left the robot when TritonBot misunderstood the visitor — it did not offer an opportunity to correct misunderstanding. Therefore, we improved the dialogue system with four major improvements: (1) TritonBot asks for confirmation for open-ended questions (e.g., "What is your name?"). (2) If the robot is not able to understand a response, it repeats the questions with

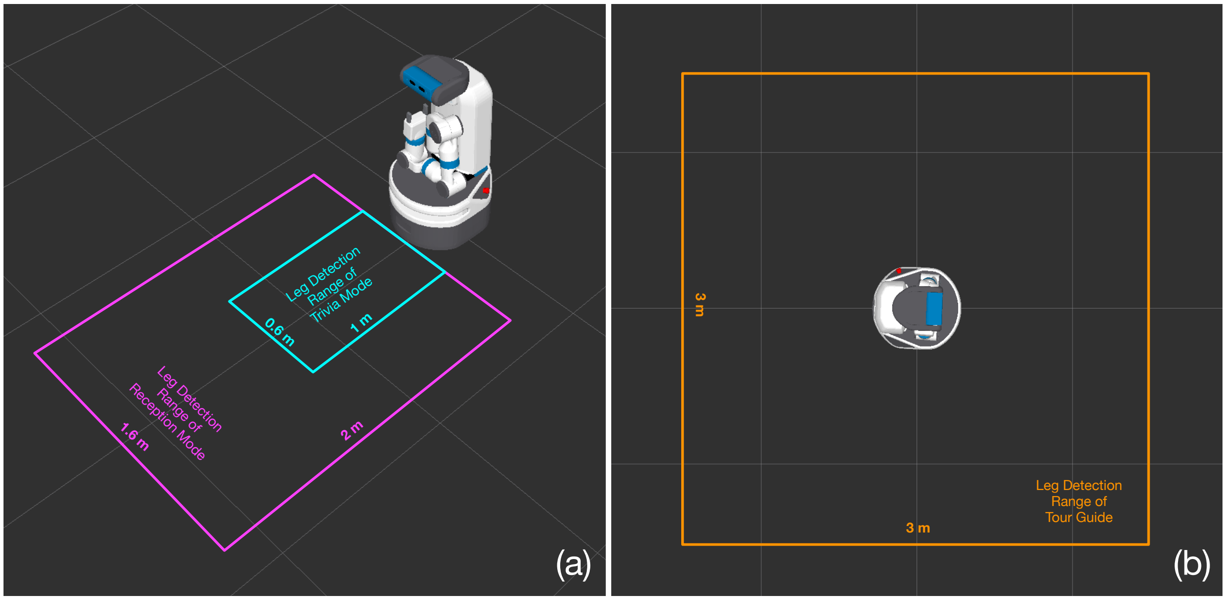


Figure 6.2: Leg detection ranges of TritonBot. (a) The leg detection range of reception and trivia mode. (b) The leg detection range when TritonBot is guiding a tour.

additional instructions. (3) TritonBot repeats the questions if it did not receive any response, but it gives up if it loses track of the visitor (when face detection and leg detection are both negatives). (4) TritonBot gives up after too many failures.

For example, Figure 6.3 shows the state machine to ask the question of "what is your name?" It reads backs the recognized name to the user, and it handles both non-recognizable results and silence gracefully. While the state machine approach does not grant open dialogue capability, it proves to be useful for robots that expects a fixed interaction pattern.

Logging

We further improved the logging of TritonBot to provide detailed information. Table 6.2 lists the items we log in TritonBot. We built a backward and forward-compatible universal logging library to manage these log items (Section 6.4.2).

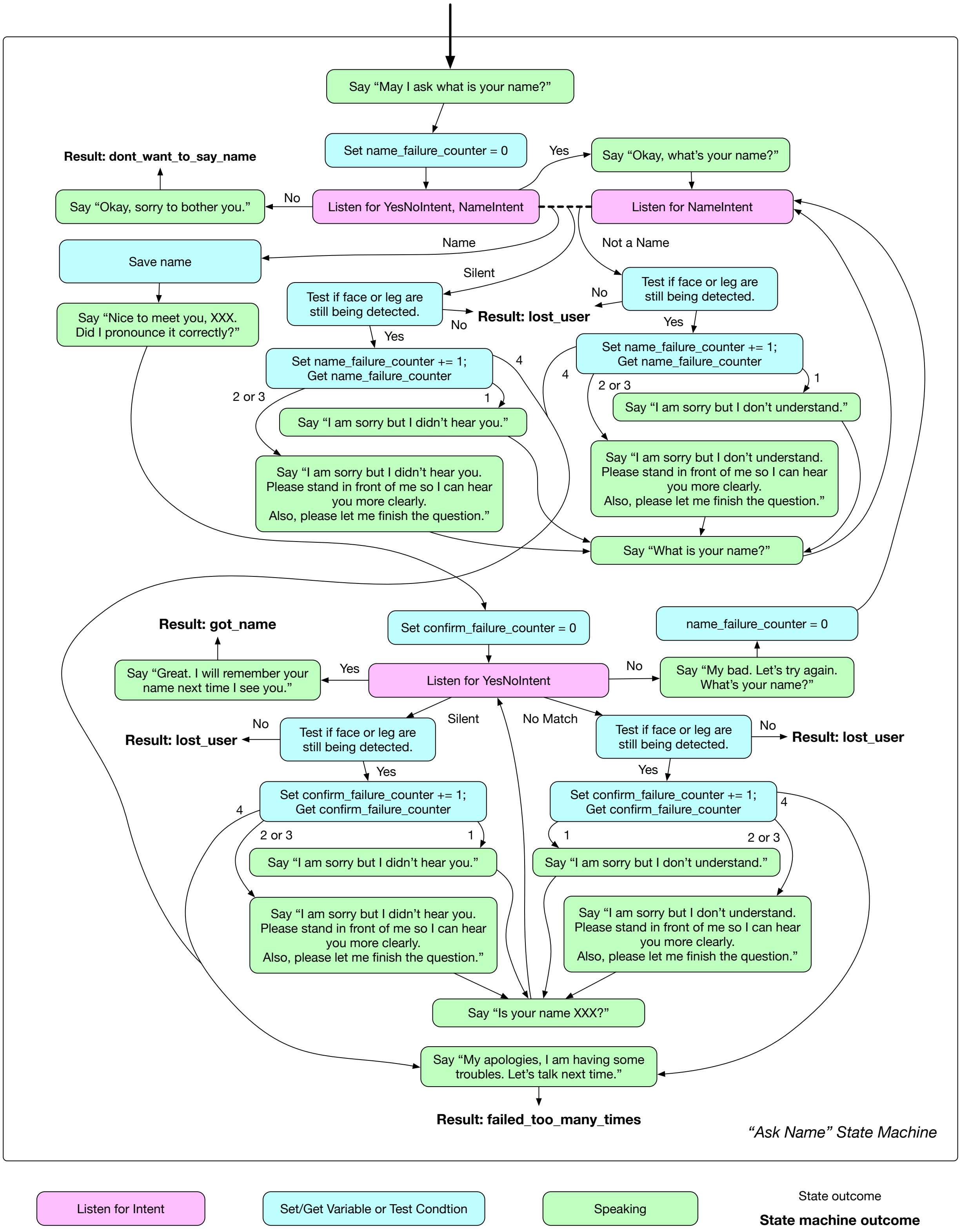


Figure 6.3: The state machine to handle question and answer on “May I ask what is your name?” to the visitor.

Table 6.2: Types of log entries collected in TritonBot.

Name	Description
`FaceImage`	Cropped and aligned face images for face recognition. Logged in debug mode only.
`VoiceRecognitionLog`	Voice recognition results from Google Cloud Speech. Including interim (partial) and final results.
`RobotSpeechLog`	Text sent to voice synthesis and spoken by the robot.
`DirectGoToLog`	Logs of navigating the robot from the current position directly to a waypoint.
`AutoDockLog`	Logs of directing the robot to its charging dock and starting charging.
`MoveToLocationLog`	Navigation logs of moving the robot from the current position to a waypoint via a few other waypoints.
`GoHomeAndDockLog`	Navigation logs of moving the robot to its charging dock via waypoints and starting to charge.
`SmachLog`	SMACH [112] state machine logs, including entering and exiting states, state outcomes, and state transitions.
`DockerContainerStats`	Periodic status snapshots of Docker containers running in the onboard computer, including resource usage, uptime, etc.
`PsUtilStats`	Periodic status snapshots of the processes running in the onboard computer, collected using psutil [115].
`EventLog`	Manual intervention events logged on the onboard computer, including manual driving, shell commands issued, etc.
`PingStats`	Periodic statistics of the latency of accessing the Internet and local developer network.
`SpeedTestStats`	Periodic statistics of the bandwidth of accessing the Internet.

6.3 TritonBot Deployment History

TritonBot went through three significant deployments. The initial deployment was in February 2018; the “trivia-only” long-term deployment was between June 2019 and September 2019; the recent TritonBot deployment was between October 2019 and December 2019. This

Table 6.3: Metrics for long-term autonomous service robots.

Name	Description
Mean Time Between Failures	The average time that the robot can operate normally.
Mean Time To Repair	The average time needed to repair a failed robot.
Mean Time Between Interventions	The average time the robot can run autonomously.
Mean Travel Distance Between Interventions	The average distance that the robot can move autonomously. Useful for robots that moving around is a primary features.
Mean User Interactions Between Interventions	The average user interactions that the robot can perform autonomously.
On-board Computer Resource Usage	Average and peak resource usage on the on-board computer. A resonable amount of slack will allow the robot to respond in a timely manner.
Internet Connection Quality	Latency, bandwidth, and uptime for the robot to connect to the Internet. Important for robots that relies on cloud services.
Task Success Rate	The success rate of different tasks performed by the robot.
Task Execution Time	Mean, maximum, and percentiles of task executation time that represent the general quality of task performed by the robot.
Average Battery Life	The time that the robot can work in a single charge.

section presents the history and problems encountered during these deployments.

6.3.1 Metrics for Long-term Autonomy

In reliability engineering and previous long-term autonomy research, the engineers and researchers use a few metrics to quantify the performance of the robot. Table 6.3 lists a few metrics that represent the performance of a long-term autonomous service robot.

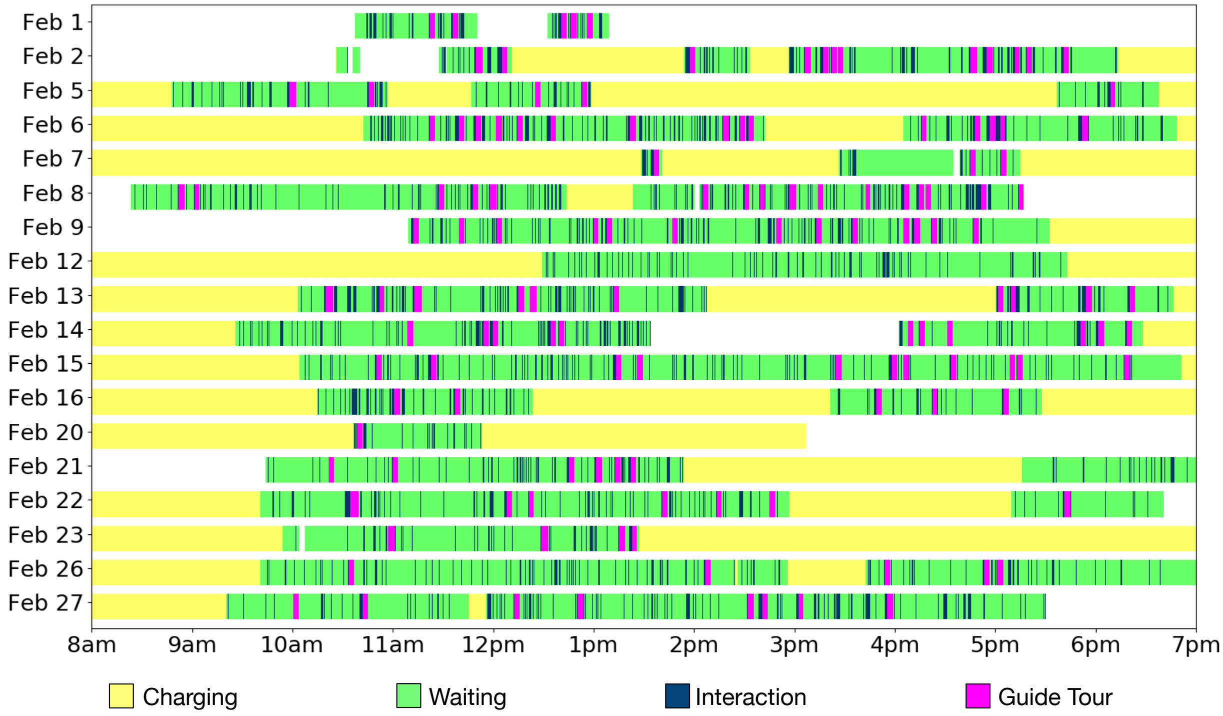


Figure 6.4: Work log for TritonBot in the February 2018 deployment.

6.3.2 The Initial Deployment

The initial TritonBot deployment was performed in February 2018. TritonBot worked for 18 days. It was working for 108.7 hours, interacting people for 22.1 hours, talked and listened for 14.7 hours, traveled 9.9 kilometers. Figure 6.4 presents the deployment log of the initial deployment.

The initial deployment focused on testing the usability of TritonBot. Some metrics listed in Table 6.4 are estimated from its logs. Qualitatively, TritonBot encountered the following problems.

Hardware Issues

- The battery on BoxBot failed and was only able to support BoxBot for 10 minutes, instead of the regular 12 hours working time. We replaced the battery.

Table 6.4: Metrics for the initial TritonBot deployment in Feburary 2018.

Metric	Value
Days Worked	18 days
Greetings	2950 times
Interacted - New	434 times
Interacted - Known	416 times
Trivia Questions Done	133 times
Tours Guided	150 times
Total Running Time	108.7 hours
Total Interaction Time	22.1 hours
Travel Distance	9.9 km
Mean Time Between Failure	56.1 hours
Mean Time Between Repair	5.9 hours

- The charging port on TritonBot failed, and TritonBot was not able to charge using the charging dock. We replaced the charging connector.
- The base of TritonBot was generating noise when it was moving. We found a caster on the bottom became loose, and we tightened the screws.
- BoxBot often failed to use the USB camera (it happened about once per three days). We initially rebooted the onboard computer when the failure occurs, but we ended up replacing

the camera.

- TritonBot often lost its wireless Internet connection (it happened about once per three days). We made a script to check the network connection every minute and reset the wireless network as needed, and the problem went away.

Software Issues

- TritonBot had localization failure and moved towards the wrong directions during some tours (it happened about twice per week). After it moved for a short distance, it was always able to re-localize and continue to function correctly.
- TritonBot once failed to move because its route planner put it too close to obstacles. We manually drove the robot away from obstacles.
- Occasionally TritonBot failed to plan a route and to move. It was using a laser scanner to detect and clear obstacles, but sometimes shadows of obstacles could not be cleared, effectively blocking the robot (it happened about once per day until fixed). We expanded the clearing range so that the laser scan always detected and cleared a circle of five-meter diameter around the robot.
- The TritonBot behavioral state machine was not able to communicate with a logging service and caused TritonBot to stall (happened three times during the entire deployment). We restarted the whole system to recover TritonBot for the first encounters, but we fixed the logging service afterward.

Robot-Human Interaction Issues

- TritonBot usually asks the question, "*May I ask what is your name?*", some visitors responded with "*Yes*," and TritonBot reply with "Nice to meet you, Yes." We improved the handling of the answer to this question after this deployment.

- When TritonBot was guiding a tour, some visitors left in the middle of the tour, leaving the robot talking to nobody. We improved the tour behavior using leg detection after this deployment.
- TritonBot misclassified a clock face afar as a human face, and it tried to talk to it, even there is no object in close proximity. We improved TritonBot; it comfirms people using leg detection before greeting the visitor after this deployment.
- TritonBot talks a lot, and a few building users describe the robot as “annoying.”

The lessons we learned from the initial deployment were reported in Chapter 3. After the initial deployment, we improved TritonBot in different aspects: We introduced Rorg to manage the software components on TritonBot. We used RoboVac to inject failures to TritonBot to make sure TritonBot behave well under unexpected situations. We added a leg detector/tracker to check people around TritonBot. We added more loggers on TritonBot to record what was going on. And we redesigned the TritonBot state machine to handle more interaction patterns.

6.3.3 The Stationary Deployment in Summer

During the initial TritonBot deployment, we always had a human “chaperone” (a student) to monitor TritonBot in case of unexpected situations. The chaperone has a wireless emergency stop button to stop the robot. However, having a student monitoring TritonBot limits its work time. To test TritonBot for an extended period, we put the TritonBot to a stationary trivia mode for an entire summer. The robot has a giant blue box in front of it and a label encouraging people to interact with it. In this deployment, TritonBot still remembered faces, always offered trivia questions, but it never moved around. Figure 6.5 shows the stationary TritonBot and BoxBot in deployment.

The stationary deployment began on June 11, 2019, and it ended on September 20, 2019, covering the summer break of 2019 at UC San Diego. TritonBot worked in the robot zoo for 81

Figure 6.5: The stationary TritonBot and BoxBot. They start to interact with visitors if they entered the blue-tape-marked zone.

Table 6.5: Metrics for TritonBot and BoxBot stationary deployment.

Metric	TritonBot	BoxBot
Days Worked	81 days	102 days
Greetings	108 times	486 times
Interacted - New Visitors	52 times	181 times
Interacted - Known Visitors	16 times	9 times
Trivia Questions Done	37 times	20 times
Total Running Time	1902 hours	2431 hours
Total Interaction Time	1.10 hours	3.67 hours
Mean Time Between Failure	952 hours	2431 hours
Mean Time Between Repair	545 hours	N/A

days during the deployment. The deployment was interrupted because the on-board computer failed on July 17, 2019, and a developer reset it on August 8. It interacted with people for 108 times. Because the robot zoo is a restricted access area, the interaction is limited. Nevertheless, the long-term deployment tested the reliability of TritonBot in a "safe" place. BoxBot worked in a lab for 102 days during the deployment. It interacted with people for 486 times. Access to the lab is limited to a group of students, so the interaction was limited. But the three-month autonomy tested the reliability of the system. Table 6.5 summarized a few metrics for this stationary deployment. The list only contains a few common metrics for long-term autonomous service robots; robots for different purposes will have their specific metrics as well.

6.3.4 The Two-month Deployment

After our two-year continued development on TritonBot, we deployed TritonBot for another two months between September 2019 and December 2019. This time TritonBot worked for 44 days. It was working for 195.1 hours, interacting people for 17.5 hours, talked and listened for 13.2 hours, traveled 4.84 kilometers. Figure 6.6 presents the deployment log of this deployment.

Compared to the initial deployment, TritonBot had less frequent interaction because we limited its range to detect people to prevent unsolicited interactions. Also, we improved its face recognition accuracy, so it was less likely to mistake strangers for acquaintances. Because TritonBot offers trivia questions to strangers but not acquaintances, many people did not have the patience to finish the session.

Table 6.6 summarized a few metrics for this deployment. A repeated issue was the “rosbag” service often fail after a ROS library update between this and the previous initial deployment. The ROS version we use became “end-of-life” and no longer maintained by the time we start this deployment. Because this repeated failure started to happen after the deployment started, and it was an example of repeated failure, we decided to keep it for studying repeated problems in TritonBot.

Comparing the mean time between failures (MTBF) with the previous deployment in February 2018, even TritonBot gained more features, MTBF slightly increased. We expected a much higher MTBF without the abovementioned “rosbag” service issue, which must be resolved by external effort. The mean time to repair (MTTR) was reduced by 39%. The changes in the metrics are a combined result of applying reliability engineering methods (e.g., scalable software management and system-level fault injection for testing) and continued TritonBot development as listed in Section 6.2.

We record all the problems and manual interventions that happened during this deployment; The interventions include operator’s notes and history from a shell command logger on the

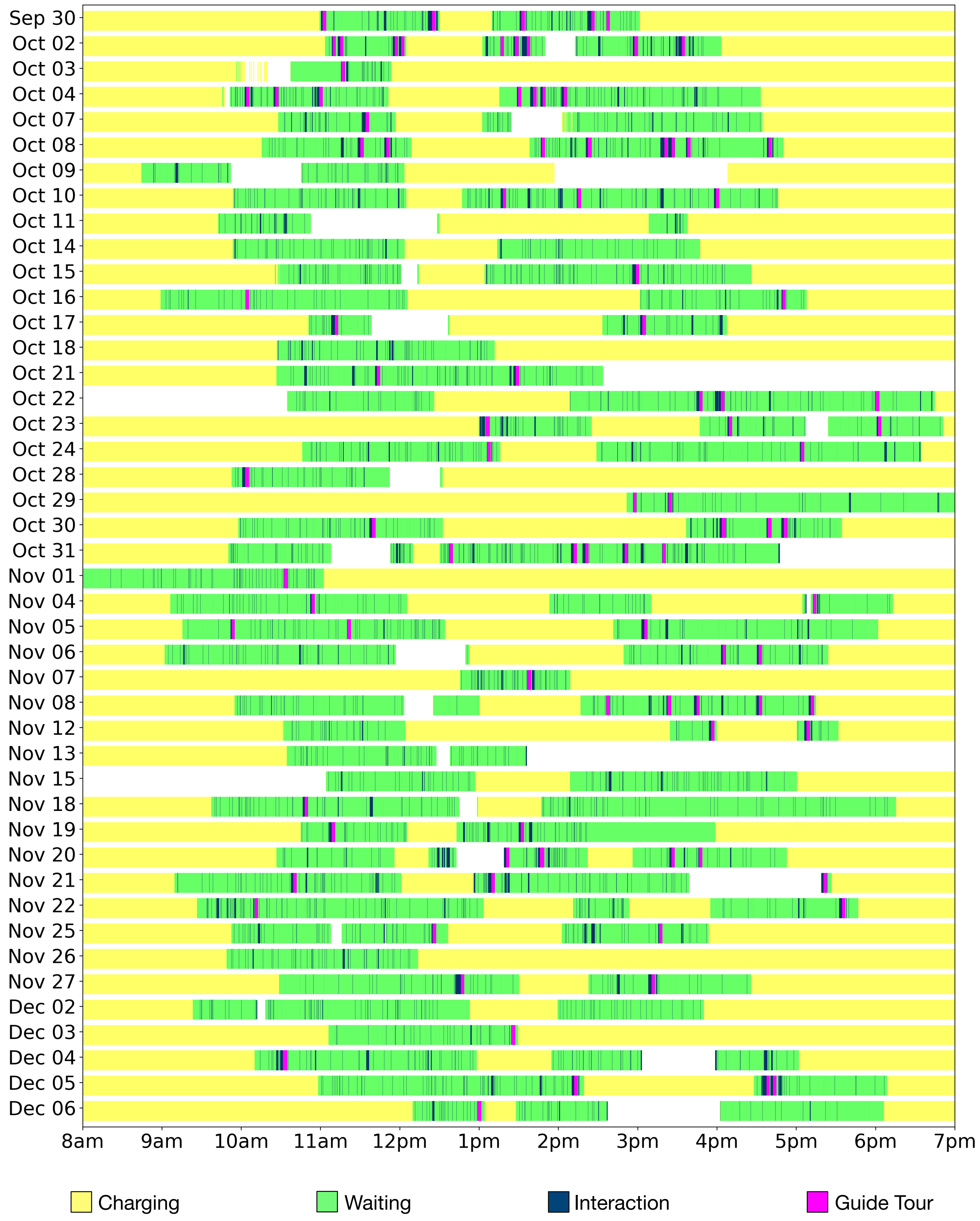


Figure 6.6: Two month deployment log from TritonBot in 2019.

onboard computer. Table 6.7 listed the problems, and manual intervention occurred during the deployment.

Table 6.6: Metrics for the two-month TritonBot deployment between September 2019 and December 2019.

Metric	Value
Days Worked	44 days
Greetings	3506 times
Interacted - New	504 times
Interacted - Known	117 times
Trivia Questions Done	305 times
Tours Offered	286 times
Tours Accepted	101 times
Total Running Time	195.0 hours
Total Interaction Time	17.5 hours
Travel Distance	4.84 km
Mean Time Between Failure	57.0 hours
Mean Time Between Repair	3.6 hours

Table 6.7: Operator's log of TritonBot from the two-month deployment between September 2019 and December 2019.

9/23/2019 3:36 PM	The laser scanner on TritonBot was not sending UDP packets. Fixed by manually powering off TritonBot entirely and powering it back on.
9/24/2019 5:33 PM	Multiple ROS action servers failed to receive a goal. Debugged and found a race condition in ROS implementation. Added extra delay in our code as a workaround while waiting for their fix.
9/30/2019 2:47 PM	TritonBot's camera was looking downwards after returning to its ready position. Fixed the issue and refreshed the configuration.
10/2/2019 2:05 PM	TritonBot's arm failed to maintain its position. The problem is not affecting TritonBot's performance. Noted the problem for future fix.
10/2/2019 2:20 PM	Logs on TritonBot took too much space. Removed all logs from TritonBot.
10/3/2019 10:01 AM	Laser Scanner failed on TritonBot. Powered off TritonBot entirely and powered it on.
10/3/2019 11:13 AM	Robot arm control software (moveit) did not start properly on TritonBot. Debugged and found misdesigned software from platform manufacture. Fixed the issue and refreshed the configuration.
10/4/2019 9:48 AM	Laser Scanner failed on TritonBot. Powered off TritonBot entirely and powered it on.

Table 6.7: Operator's log of TritonBot from the two-month deployment between September 2019 and December 2019, continued.

10/4/2019 9:53 AM	The TritonBot state machine did not start properly because of a race condition in Rorg. Debugged and fixed the issue in Rorg.
10/7/2019 2:03 PM	The robot's battery died. It turns out someone engaged the emergency stop and forced the robot back to its charging dock, but the robot was not properly plugged in. Recharged the battery.
10/7/2019 2:17 PM	The external speaker was not powered on after the previous incident. Manually turned on the speaker.
10/9/2019 10:44 AM	TritonBot's state machine failed. It was not able to communicate with a logging service because of a communication bug in the ROS library. Noted the issue and restarted affect parts.
10/9/2019 1:59 PM	Paused the TritonBot deployment because the area (hallway) is being used for an event.
10/11/2019 12:28 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
10/15/2019 9:08 AM	TritonBot did not receive any response from visitors all day yesterday. Reason under investigation.
10/15/2019 10:30 AM	TritonBot mislocalized itself. Manually gave TritonBot a pose estimation.

Table 6.7: Operator's log of TritonBot from the two-month deployment between September 2019 and December 2019, continued.

10/15/2019 12:14 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
10/15/2019 1:04 PM	Voice recognition failed. It turns out the microphone was muted (most likely by a stranger) for an entire day. Reenabled the microphone.
10/17/2019 12:36 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
10/21/2019 4:19 PM	TritonBot's battery was dead. The battery percentage reading was random. Recharged the battery.
10/23/2019 5:24 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
10/28/2019 12:30 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
10/31/2019 11:53 AM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
10/31/2019 11:59 AM	Some students interacted with the robot in a very unexpected way. They tried to move the head and the arm of the robot. Reset the robot after they leave.

Table 6.7: Operator's log of TritonBot from the two-month deployment between September 2019 and December 2019, continued.

10/31/2019 5:18 PM	TritonBot's battery was dead. The battery percentage reading was random. Fully discharged the battery and recharged it.
11/1/2019 7:17 AM	Expanded the leg detection range (from 0.6m to 1.6m) of TritonBot to solicit more people. Restarted the state machine.
11/4/2019 5:27 PM	Changed the orientation of TritonBot to solicit more people. Reloaded the map service.
11/5/2019 12:17 PM	The robot's head was moved to the side (most likely by a stranger). Had the robot to go back to its ready position.
11/6/2019 9:04 AM	TritonBot mislocalized itself. Manually gave TritonBot a pose estimation.
11/6/2019 10:28 AM	The robot's head was moved to the side (most likely by a stranger). Had the robot to go back to its ready position.
11/6/2019 12:50 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
11/6/2019 12:52 PM	The e-stop button was pressed (most likely by a stranger). Manually released the button.

Table 6.7: Operator's log of TritonBot from the two-month deployment between September 2019 and December 2019, continued.

11/8/2019 12:25 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
11/13/2019 12:38 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
11/13/2019 1:41 PM	The e-stop button was pressed (most likely by a stranger). Manually released the button.
11/13/2019 1:41 PM	A curtain was set up in the hallway for an event. Paused TritonBot deployment for the day.
11/18/2019 12:58 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
11/19/2019 12:05 PM	The robot's head was moved to the side (most likely by a stranger). Had the robot to go back to its ready position.
11/20/2019 1:19 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
11/22/2019 2:11 PM	The commander service failed. Collected logs. It turned out to be a communication issue in ROS. Noted the issue and restarted affect parts.

Table 6.7: Operator's log of TritonBot from the two-month deployment between September 2019 and December 2019, continued.

11/22/2019 5:04 PM	The microphone was muted (most likely by a stranger). Reenabled the microphone.
12/2/2019 10:19 AM	The state machine failed to receive any external response. It turns out ROS communication failed.
12/3/2019 11:06 AM	The commander service failed again. Collected logs and it turns out to be an communication issue in ROS. Noted the issue and restarted affect parts.
12/4/2019 4:01 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.
12/4/2019 4:24 PM	The speaker volume was set to minimal (most likely by a visitor). Reenabled the speaker.
12/6/2019 4:02 PM	TritonBot's state machine failed again. It failed to communicate with a logging service. Noted the issue and restarted affect parts.

6.4 Scaling up TritonBot over the Long-term Deployment

Scalability is one of the three most important characters in long-term autonomy; it allows the developers to grow and expand a robot system without overhauling the existing architecture of the system and affecting normal operations. This section discusses our effort in making TritonBot scalable.

6.4.1 Scalability Challenges in TritonBot

TritonBot faced many challenges in scalability that add difficulties in its development and evolvement:

- Backward- and forward-compatibility: Scaling up requires fast software iteration. But the lack of backward- and forward-compatibility forces the developers to complete coding and testing on all related components before rolling out a new feature; rolling back one component also affects all related parts. Long-term logging also becomes a challenge when the developers add, update, or remove fields in the log format.
- Software architecture: Tightly-coupled software limits the scalability of a computer system, but decoupling software components introduces difficulties in interfacing and coordination between the components. Besides, when TritonBot offloads computationally-intensive software components to other machines, crossing network boundaries and communicating over the Internet bring in concerns in accessibility, latency, and confidentiality.
- Software management: With numerous robotic programs with different requirements running together to form a complete system, managing software running on the on-robot computer and other hardware is challenging. The limited computing power available on-board worsens the problem when computationally-intensive programs compete for computing resources; they tend to create resource contention and exhaust all CPU or memory capabilities.

6.4.2 Forward- and Backward-compatibility

Many robot systems use Robot Operating System (ROS) [11] to split the entire software system into multiple programs. ROS programs communicate through a publish-subscribe pattern to work with each other, and ROS provides an interface description language (ROS messages) to support the pub-sub framework. But any change to the message definition, no matter how insignificant it may be, invalidates all previously serialized data and generated libraries. Such inflexibility helps ROS to become a consistent community-maintained robotic software toolkit, but it limits the long-term evolution of an autonomous robot system, and it makes serializing ROS message a bad choice for long-term logging.

In TritonBot, we leverage an open-source libraries Protocol Buffer (ProtoBuf) [63] to provide forward- and backward-compatibility. ProtoBuf is a language- and platform-neutral extensible mechanism for serializing structured data, but serialized ProtoBuf messages remain accessible even if the format description changes (in a compatible way). ProtoBuf can even work with RPC (remote procedure call) frameworks to provide backward compatibility between different programs. We use ProtoBuf to store sequential or small structured data, such as the topological map for navigation, face embedding of visitors, and long-term logs.

As an example, TritonBot leverage ProtoBuf to save its long-term log. We created a "Universal Logger" that stores a stream of ProtoBuf messages into compressed and size-capped files and distributes them into directories with a date and timestamp. Because ProtoBuf format is forward-compatible, adding extra fields to the log format does not affect previously-stored logs; being backward-compatible, analysis programs written against an old ProtoBuf format will continue to work with newly generated logs. As an example, the voice recognition program on TritonBot initially only records voice recognition results that trigger the robot's response; later, when we moved to a more capable voice recognition engine, we updated the data format to include the interim results. Thanks to the compatibility, the analysis script can still read previously-stored logs.

6.4.3 Decoupling Software Components

Dividing a large system into smaller modules is a common practice is robotic software engineering. ROS made this process easier by providing not only a publish-subscribe infrastructure but also many ready-to-use modules. With the help of ROS and a remote procedure call framework, gRPC [82], we implemented the TritonBot system as a number of standalone and independent "microservices." The "microservices" design pattern not only provides reusability, but it also isolates failures. We carefully engineered the microservices so that any microservice can tolerate transient unavailability of its dependencies, and restarting a microservice will help it to return to a clean state. Under these two design principles, most of the failures we encountered can be solved by easily restarting one or a few microservice without affecting the entire system.

ROS has two more shortcomings in addition to the lack of compatibility: the inflexible networking requirements and the lack of security support. ROS programs (nodes) assume bidirectional direct TCP connection between each other on all ports, and ROS does not provide any encryption and authentication support. Therefore, ROS programs cannot communicate over networking environments with firewalls or network address translation (NAT) devices, and it is dangerous to rely on ROS alone to carry communication over the Internet. To overcome these shortcomings, we built a part of the TritonBot system with an open-source library, gRPC [82] alongside with ROS: gRPC is an open-source remote procedure call (RPC) framework that supports both synchronous or asynchronous, unary or streaming backward- and forward-compatible ProtoBuf messages in both requests and responses; it leverages HTTP/2 channels with optional encryption and authentication that can easily pass network devices. We build and released a ROS package `grpc` [116] that helps the users to generate, compile, and link ProtoBuf libraries and gRPC stubs within the ROS build environment. Therefore, we were able to offload computationally intensive tasks from TritonBot to on-premise servers or even cloud infrastructure in a scalable manner.

The face recognition pipeline is an important sub-system in TritonBot, and it contains

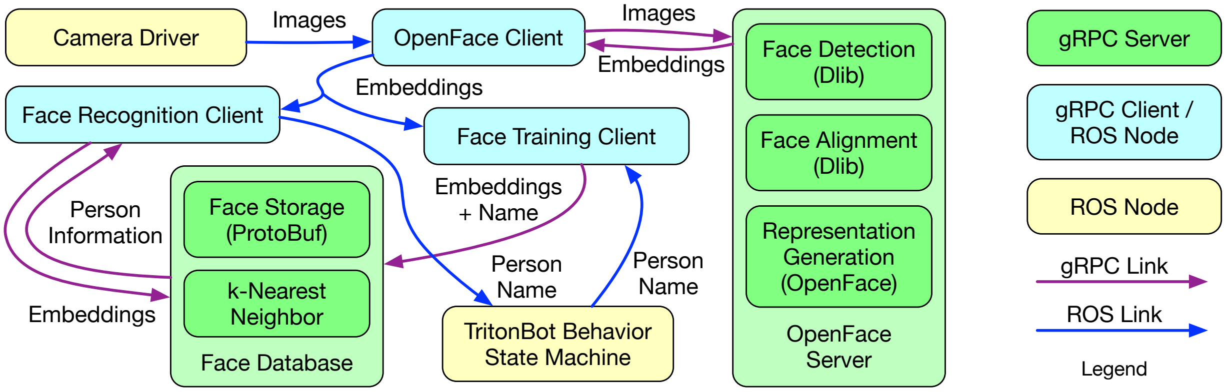


Figure 6.7: The face recognition pipeline in TritonBot that is made of a few microservices. Some ROS-based microservices bridge the ROS system with standalone gRPC servers, so that TritonBot exploits the existing open-source components in ROS and backward- and forward-compatibility from gRPC. After running TritonBot for some time, we moved the "OpenFace server" to a remote computer with powerful GPUs to save computing resources on the on-robot computer.

multiple microservices (Figure 6.7). Each of these modules perform a relatively simple task: the "OpenFace server" extracts face from a given image and generates embeddings; the "OpenFace client feeds image from the driver to the server and makes the embeddings available to other ROS parts; the "face database" stores faces with labels and looks up potential matches when given an image; adding to the face database (training) and looking up labels rely on two other modules — the face training client and the face recognition client. Using gRPC for the OpenFace server and face database maximizes TritonBot's scalability: Since the ProtoBuf format is forward-compatible, the face database file did not require any conversion when we added an option to store the face images along with the embeddings. ProtoBuf/gRPC format is also backward-compatible: when we updated the face database server, the old client continued to work, and we had a chance to test the new server before implementing a new client. In addition, gRPC helped us to offload the computationally intensive face embedding generation program from the robot: we put a backup face embedding generation server on a powerful host system behind a firewall (for network address translation) and a reverse proxy (for authentication); the robot access the service over the Internet when needed.

In general, design considerations in decoupling software speed up software iteration and increase the scalability of a robot system; backward- and forward-compatibility enable decoupling in many scenarios.

6.4.4 Managing Robotic Software with Linux Containers

In TritonBot, we use Linux containers to manage robotic software deployment and provide a unified development environment. Linux container [117] provides a convenient approach to build, deploy, and run applications on different machines. Backed by the Linux namespaces that isolate and virtualize system resources for a set of processes, Linux containers are much lighter-weighted than fully virtualized kernels in hypervisors or virtual machines. We run each microservice inside a dedicated container, so that we can enable, disable, update, roll-back, or control them independently.

Initially, we adopted Docker [59], a popular and open-source container management tool to run every software component (41 microservices in total) in separate containers. Docker isolates the execution environment for each of them. When the TritonBot system grows larger, resource contention becomes an issue in further scaling up. In datacenters, scaling up software systems means spawning more program instances and load-balancing the tasks among them; popular tools like Kubernetes [15] leverage Linux containers to provide a unified platform to manage the programs. A service robot like TritonBot only carries limited computing resources; however, it is not using all the components at the same time — for example, TritonBot does not face recognition results when it is moving, and vice versa. We built Rorg [109], an open-source container-based software management system. Rorg not only provides an accessible mechanism to use Linux containers in a robot programming environment, but it also models the component dependencies and decides what microservices to start to fulfill the robot's request. With the help of Rorg, the TritonBot uses 45% less CPU than before, which leaves the developer with plenty of room to add more features. Rorg also adds some additional benefits to service robots: because the software components have chances to stop and restart often, transient issues like memory leaks have a less significant outcome. This conclusion is consistent with the experience from Google that “a slowly crash looping task is usually preferable to a task that has not been restarted at all [40].”

In conclusion, tools like Linux containers not only benefit traditional computer systems but also improve the scalability of service robots. The different use scenario on service robots requires unique customization of the tools.

6.5 Tolerating and Coping with Failures

Resilience is the ability of a robot to adapt to environmental changes and tolerate or recover from transient errors. The TritonBot developers tried to make the robot as robust to failures as possible, and this section discusses our efforts.

6.5.1 Resilience Challenges in TritonBot

None of the engineered systems is immune from failures, and long-term deployments further expose the errors in a service robot. The TritonBot system has a few challenges to become resilient to failures:

- Transient failures: Some failures on service robots are transient and recoverable. For example, TritonBot sometimes loses its network connection when it enters and exits a WiFi blindspot, but a simple reconnecting attempt can effectively fix this issue. The two challenges in dealing with transient failures are (1) identifying the failures and (2) implementing the fix.
- Overloaded system: Almost all basic robot functionalities rely on the only programmable part of TritonBot, the on-robot computer. The computer is affording too much functionality

that a minor issue on the computer will lead to serious outcomes: when the Bluetooth stack on the computer fails, the developers lose the ability to drive the robot manually; when the computer encounters networking issues, the developers cannot connect to the computer.

- Lack of monitoring: Autonomous service robots are expected to work without close human supervision, but the developer does not have the means to understand the system characteristics and discover failures. However, too close supervision defeats the purpose of autonomy.

6.5.2 Recover from Transient Failures

Frequent self-diagnosis and self-repair is a practical approach to discover and recover from some transient failures. TritonBot periodically runs some scripts to check and fix any potential issues. After the TritonBot encountered WiFi issues multiple times, we created a script that checks the Internet connection (by pinging a commonly known website) and restarts the wireless interface in case of a failure. In another case, a Linux system service (`sssd`) sometimes gets killed and does not restart properly; we created another script to restart it in case of failure. However, the challenge in fixing transient failures it not making patches, but instead identifying the failure cases. Long-term deployments expose these transient failure patterns; occasional monitoring (Section 6.5.4) helps the developers to capture these failures.

While hand-made scripts are useful in complementary to system service management tools, another type of transient failure happens in robotic software is that some programs get killed when they encounter unhandled exceptions or have design flaws. In TritonBot, any unexpected program crash triggers restarting itself or its programs group. While crash looping programs often indicate issues in the system, following proper design principles, we observed that some infrequent programs restarts have little effect on the overall reliability: In the early TritonBot development, we had an issue that the voice recognition software crashes and restarts every a

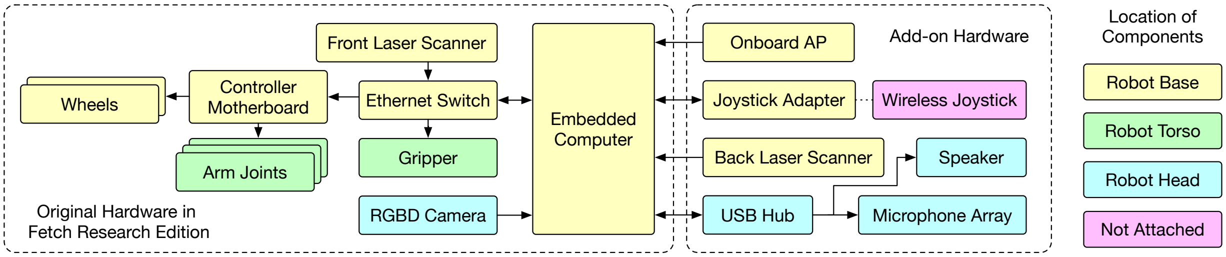


Figure 6.8: Main hardware components in the TritonBot system. We added some additional hardware to the Fetch Research Edition.

few hours. The system continued to work (possibly with a few unnoticed attempts to retry voice recognition); only later did we find the unusual restarts in the logs and fixed a memory leaking issue. We have concluded three design principles to handle transient failures in robotic software: (1) Never to hide failures fail silently; it is better to crash a program and expose issues. (2) Restarting a program should help it to enter a clean state and recover from transient failures. (3) Any software component should tolerate the unavailability of another component to avoid chain reaction on failures.

6.5.3 Relying on Separate Subsystems

Much like loosely-coupled microservices limit the scope of failures in the software system, finer-grain hardware organization improves the stability of the hardware system. Figure 6.8 shows the hardware components in TritonBot. The bare Fetch Research Edition platform only has two parts in its system architecture: an embedded computer to run user programs, and a low-level controller to control the actuators. When we built the early TritonBot prototype, we relied solely on programming and configuring the embedded computer. Soon we found that the on-robot computer becomes a frequent point of failure. In the TritonBot evolution, we added a wireless access point to the robot as an “escape hatch” in addition to the regular encrypted access channel over the Internet. We also pair the manual-override gamepad directly with a USB dongle that emulates a joystick device instead of the Bluetooth radio in the embedded computer. As a general design principle, we found that relying on dedicated devices reduces single-point failures and prepares the robot and the developer for unexpected situations.

During the TritonBot deployment, we observed a few times that the laser scanner stopped to work when its power and ethernet connection are still intact. The only way to recover from this condition was to power-cycle the laser scanner by toggling the master power switch on the robot. From this experience, we learned that many hardware failures are also transient, and they can be fixed by power-cycling the device. If there are individually programmable circuit breakers before the power supply of the peripherals, the robot will be able to restart part of its hardware and overcome transient failure without affecting a big portion of the system.

6.5.4 Monitoring the System and Logging

The challenge in fixing transient failures it not making patches, but instead identifying the failure cases; occasional monitoring and long-term logging help us to achieve such a goal. In the TritonBot project, we mainly use two tools: First, we built an Android app to see the robots' view and battery status in real-time. Second, the robot keeps detailed logs about its status and decisions; it analyzes its log and sends a summary to the developers every night.

The robot monitor (Figure 6.9) is an Android tablet. The robot captures its battery level, charging status, and camera images every a few seconds, and it sends them to a central server through an authenticated channel over the Internet. The tablet displays these state of the robot and gives the developer an overview of the robot status with a simple glance. The minimal real-time monitoring helps the developers to discover obvious but unexpected conditions: we experienced several times that a visitor turns the robot's head to the side; there was once that a visitor touched the "mute" button on the microphone; another case was that the robot keeps retrying docking the charging station infinitely because of a failed connector. Only did we discover these problems were we able to program the robot to respond correctly to these situations.

TritonBot also logs the details about its internal states and events, including CPU and memory usage, Linux containers and processes information, network access bandwidth and latency, behavioral state machine states, speech recognition details and voice synthesis history, robot localization and navigation status, robot pose and battery status, and manual intervention and command executed in terminal, etc. The robot analyzes its daily log and sends an E-mail about its daily work summary to the developers every mid-night. The daily report includes the interaction transcript and a summary of its daily events such as the number of humans engaged, trip traveled, and so on. Reading the E-mail allows the developers to understand the robot's performance in general, while more detailed logs generated by different components are available for further analysis. Currently, different components in TritonBot generate 14 different types of logs in total. The TritonBot log-format is backward- and forward-compatible (Section 6.4.2), and it is compact: TritonBot generates about 23.3 MB gzip-compressed logs daily on average when the robot is actively deployed. We collected about 9.6 GB ProtoBuf-based logs from TritonBot in the past two years, and we were able to generate many insights from the data. Since the log format is optimized for machine-reading, scanning through all of the log entries only takes a few minutes.

Both of the monitoring methods retain the robot's autonomy, but they allow the developers and the users to understand the robot's status at different levels.

6.6 Learning from the Past

Learning in long-term autonomy suggests that a system can learn from the past and improve its performance. TritonBot learns from the deployment experience and improves itself over time. In addition to the robot learning by itself, the TritonBot developers also learn from the past failures: we created a fault injection tool to recreate past failure scene on TritonBot, study rare failure conditions, and improve the robot system.

Figure 6.9: An Android tablet that displays the robot's eye view and battery status. The developer occasionally monitors the general status of the robot during the deployment.

6.6.1 Learning Challenges in TritonBot

TritonBot has two primary learning-related issues:

- Repeated failures: TritonBot moves around when it shows the visitor the places of interest in the building. However, we have observed that it gets stuck on some paths at a significantly higher rate than others. Even with much experience of similar failures, it continues to make the same mistakes because it always plans for the shortest path.
- Rare failure types: Some failures on TritonBot are triggered with a few "coincidences." These failures are difficult to reproduce, which prevents the developers from an in-depth investigation of the failures; when the developers come up with potential fixes, there is no practical method to verify the fixes.

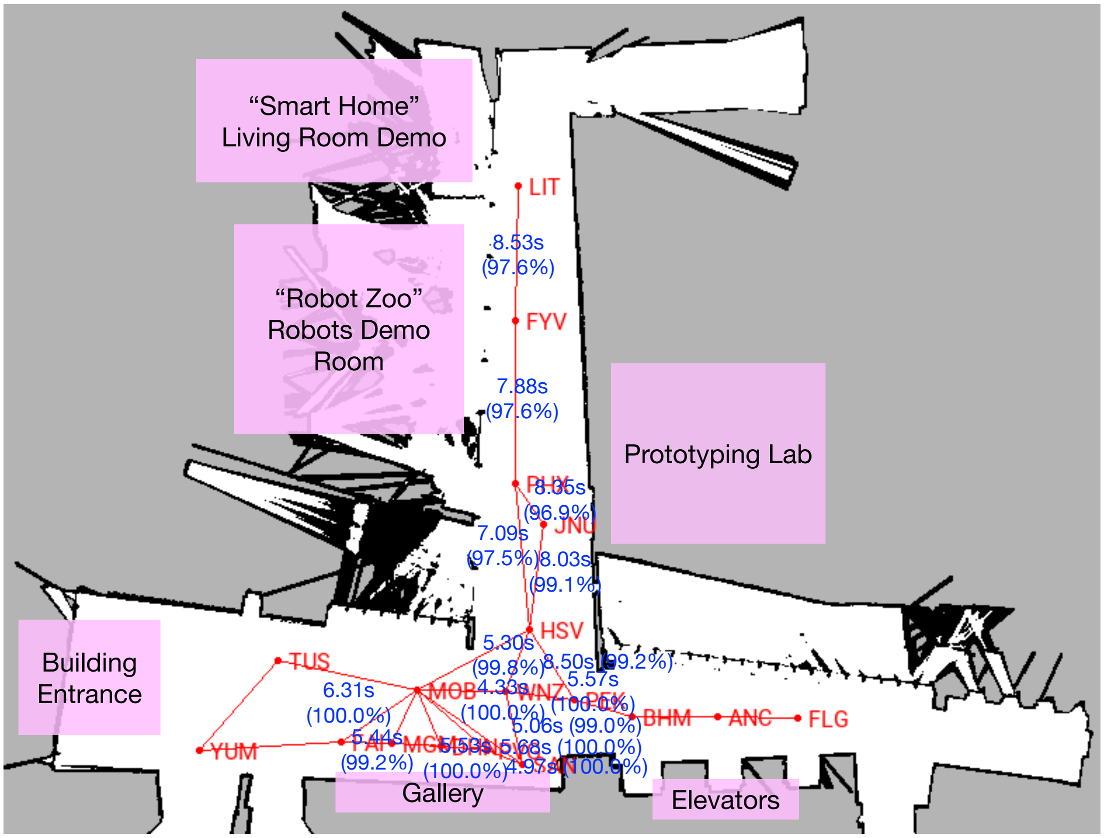


Figure 6.10: The topological navigation map of TritonBot. The map shows the average time to traverse an "airway" on the map as well as the success rate.

6.6.2 Learning from Long-term Deployment

Unlike short-term demos, long-term autonomous service robots accumulate experience over time. These experiences can turn into precious knowledge that improves the robustness of the robot. TritonBot applies this idea to navigation — when it moves around, it records the time of traveling each path on a map, and it avoids the paths that it took too long to travel when generating a move plan.

The TritonBot moves around to show the places of interest to the visitors. On top of the classic *movebase* navigation stack from ROS [55], we added a topological map layer of waypoints and paths (Figure 6.10). The core of learning is a "traffic control" service: When TritonBot decides to move to a waypoint, it requests a "traffic control" service to generate a plan — a list of waypoints connected by paths. TritonBot calls *movebase* to execute the plan with some preset time limit and error allowance. After traveling each path, TritonBot reports the travel time back to the traffic control service. The traffic control service internally adjust the cost of each path according to the feedback, and the change affects the future plans. Figure 6.10 also presents the average traversal time and success rate of each path in a previous TritonBot deployment.

In the TritonBot deployment, using past experience to improve the system is a fundamental design principle. In another example, TritonBot record all of the utterance that it can not understand during its conversations with people, and the developers use these utterances to improve its intent extraction algorithm. In conclusion, learning from past failures is a convenient and effective way to increase reliability over time in long-term robot deployments.

6.6.3 Learning from Rare Failures

Software fault injection is a conventional technique in software engineering; it intentionally introduces failure to test a system's response to failures. We created a fault injection tool, "RoboVac" to inject failures to the TritonBot system. It helps us to find unknown design flaws, to verify fixes, and to benchmark the system's resilience.

RoboVac offers a unified framework for general fault injection needs on service robots. It leverages the ROS message passing framework (topics) to simulate failures in sensors, actuators, or even between software; it also enables the developer to inject failures at the general Linux process level to simulate a program crashing or stuck; it can reshape the network traffic to simulate different networking conditions. RoboVac offers an efficient workflow for the developers to improve a robot's performance under failures. With RoboVac, we were able to inject failures in the software, ROS message passing, networking, and ad-hoc components. During the TritonBot evolution, we founded many unseen design flaws using fault injection, and we used RoboVac to verify our fixes. We are continuing to work on RoboVac to offer a fuzz-testing scheme that enables automatic error discovery on service robots.

6.7 Toolbox for Reliable Long-term Autonomous Robots

While building and maintaining the TritonBot system, we have been using various tools and following a few rules to make the robot system more reliable. This section presents a series of practical rules and design principles that contribute to the long-term reliability and maintainability of TritonBot. These guidelines and practices may be used as a toolbox to build reliable long-term autonomous service robots.

Coding

- Use a version control system for the source code to track the changes in long-term development.
- Follow a consistent coding style. Furthermore, using a version control system with code formatter will make it easier to track the changes in the long-term evolving code.
- Leverage compiler checks. Complier analysis, like thread-safety analysis warns about potential race conditions in code. Code sanitizers such as AddressSanitizer [118], ThreadSanitizer (tsan), and MemorySanitizer (msan) help the developers to discover hidden coding errors. Modern compiler like Clang provides many useful tools to ensure the correctness of the code [119].
- Use automated unit tests to ensure the correctness between code changes.
- Try and adopt appropriate other software engineering methods and best practices. Most general coding rules apply to service robots programming.

Components

- Clearly define the interfaces and dependencies of a software component. Document a robot system design using the experience from the software industry.

- Use automated tools to manage the dependencis and deploy a software component. Tools like Docker and Rorg (Chapter 4) provide a replicable environment to build the software, and Rorg even manages inter-component dependencies.

- Expose the problems. It is better to crash a program and expose the problems than to hide them.

- Leverage tools like continuous integration/delivery (CI/CD) to find potential breakings during the development.

- Restarting a program should help it to enter a clean and healthy state. Therefore restarting a program will always be a viable last resort to restore an unhealthy component.

Integration

- Choose the appropriate granularity for loosely coupled components. Too fine- or too coarse-grain component partition both reduces scalability of a system.

- Architect the ROS system. Malavolta et al. summarized 49 evidence-based guidelines from 335 ROS-based systems [120], and some of these guidelines are practical to implement on service robots.

- Think outside ROS-ecosystem. Many ROS design decisions are made from a large-scale, ambitious, and community-driven open-source project perspective. They do not always align well with long-term autonomy.

- Any component should tolerate transient failures from other components. In this way, the robot will only go through temporary performance degradation, instead of total collapse, in case of minor issues.

- Isolate failures. In a multipurpose robot, the robot may be able to perform some tasks in case of partial failure (TritonBot can be a receptionist even if it can not guide tours).

- Design for testing. Take testing a subsystem into consideration when designing its architecture.

- Use fault injection to test a robot's resilience to certain kinds of failures. RoboVac (Chapter 5) provided experiences of using fault injection on robot systems.

- Take scalability into account while building the system. By making the components loosely coupled with each other, the developers can iterate faster without the fear of a potential system overhaul.

Robot System

- Communicate with the user. Users of service robots are sometimes tolerant and cooperative. Good communication mitigates the outcomes from failures.

- Use long-term knowledge to improve the robot system over time. A long-term autonomous service robot has the advantage of accumulating experiences that are valuable to long-term reliability. Taking advantage of these experiences makes the robot more robust over time.

- Automate monitoring, self-diagnose, and alerting. As an autonomous system, a robot can always self-diagnose and send out alerts during failures. A fleet of robots can use the same method to be monitored and diagnosed at scale.

- Keep the logs. A robust and well-rounded logging system provides useful and vital information from past failures. This informaiton can be used to perform a thoughtful post-mortem analysis, which will improve the system and reduce/eliminate future failures.

- Be prepared to live with failures. Failures are unavoidable in any engineered system, including service robots. However, a service robot can operate normally when the failures are rare, minor, and gracefully handled.

6.8 Conclusion

This chapter presents a high-level view of robotic reliability engineering. We discuss our continued experience and reliability engineering practices in the context of TritonBot, a tour guide and receptionist robot in a university building.

We presented the deployment history and the problems encountered during the deployment within the last two years. As a long-term autonomous service robot, it has challenges in the three aspects of long-term autonomy: scalability, resilience, and learning. TritonBot optimizes data compatibility, software architecture, and resource management to retain scalability. TritonBot tolerates transient failures, avoids single-point failures, and leverages monitoring to improve its resilience. TritonBot also learns from the experience and takes advantage of software fault injection. All these efforts increase the reliability of TritonBot. We also summarized a few practical rules to guide building a reliable long-term autonomous service robot system.

The complete TritonBot source code is available at https://github.com/CogRob/TritonBot. In our process of updating TritonBot to the next generation of the ROS ecosystem — ROS2, we incorporate our experience from previous deployments and take advantage of new features such as a more efficient communication mechanism from ROS2 to improve TritonBot further. We hope TritonBot will inspire more robust long-term autonomous service robot designs.

Acknowledgements

Chapter 6 contains material from “Robotic Reliability Engineering: Experience from Long-term TritonBot Development,” by Shengye Wang, Xiao Liu, Jishen Zhao, and Henrik I. Christensen, which appears in “Field and Service Robotics: Results of the 12th International Conference (FSR 2019).” The dissertation author is the primary investigator and first author of this paper. The material in this chapter is © 2020 Springer Nature Singapore Pte Ltd.

Chapter 7

Conclusion

This dissertation represents many years of study and research around the reliability and system engineering aspects in the robotics field. The interest in this topic was initially inspired by an incident: The author and his teammates built a “selfie robot” demo and took it to a fund-raising campaign; the robot operated flawlessly for an entire night except when the author and his advisor were showing the robot during a live interview. That embarrassing moment eventually led to further research in the reliability engineering of services robots and the contributions in this dissertation.

7.1 Summary of the Dissertation

This dissertation contains four primary contributions: (1) We built TritonBot as a realistic example of a tour-guide robot, and we summarized reliability challenges and lessons in TritonBot that represents a class of long-term autonomous service robots. (2) We showed that software encapsulation and dynamic orchestration reduce maintenance burden and resource contention in service robots; we implemented Rorg, a Linux container-based scheme to manage software components for service robots, to support this claim. (3) We showed that simulating a broad spectrum of rare failures at system level exposes design flaws and assists developers in improving the robustness of service robots. We built RoboVac, a fault injection framework for service robots, to support this claim. (4) We summarized of design principles from our experience with TritonBot, Rorg, and RoboVac. They will guide reliable service robot design.

In addition to these contributions, we believe the underlying concepts and principles are not only applicable to service robots, but they are also generalizable to other cyber-physical/ computer systems. These concepts include software encapsulation and dynamic orchestration, as well as broad-spectrum system-level failure simulation as a testing method.

TritonBot: A Long-term Autonomous Service Robot

We built TritonBot as a realistic example of a long-term autonomous service robot in an open environment. TritonBot works as a building receptionist and a tour guide. It recognizes people's faces, talks to people, and shows people the labs and facilities in the building. We deployed TritonBot for more than three months to discover failure modes in long-term deployments of a robot that actively interact with people in an open environment.

In the initial TritonBot deployment, TritonBot served the guests for 108.7 hours in total, actively talked and walked with people for 22.1 hours, provided 150 tours, traveled 9901.1 meters, listened for 4.0 hours ($2,616$ counts) and spoke for 10.7 hours ($14,037$ counts). We took a peek at the common failures in the long-term deployment of an autonomous service robot, and we learned the initial lessons in hardware failures, network connectivity, software failures, software deployment, navigation in open environment, speech and dialogue, face recognition, long-term log collection, safety, etc.

Hence, TritonBot became the primary experiment platform for our reliability engineering research. The following projects, Rorg and RoboVac, are both tested on TritonBot. We open-sourced the entire stack of TritonBot and hoped it would serve as an experiment platform for many robotics research projects in the community.

Scalable Software Management and Orchestration

From the experience of managing numerous software components on TritonBot and adding more features to it, we concluded that software encapsulation and dynamic orchestration reduce maintenance burden and resource contention in service robots. As TritonBot evolves and expands, the complexity of its software components is becoming a pressing issue that challenges TritonBot's scaling up. Comparing software management on robots and other computer systems (e.g., smartphones and datacenter applications), developing software for robots has two primary challenges: maintenance burdens and resource limitations.

With the recent rise of container technology, we proposed Rorg, an open-source Linux container-based scheme to manage, schedule, and monitor software components on service robots. Rorg allows developers to pack software into self-contained images and runs them in isolated environments using Linux containers; it also allows the robot to turn on and off software components on demand to avoid resource contention.

Rorg adopts a Linux container engine, Docker [59], to run robot software in individual containers. Rorg works with ROS, the de facto standard robotic middleware. It provides default configurations to run ROS software without modification. Furthermore, Rorg organizes the robot software into multiple Linux containers and models the static and dynamic relationships between them. Rorg provides an interface to create, query, update, delete, start, stop, and restart these containers manually or programmatically. Rorg allows the software components to time-share computing resources: It pauses or shuts down inactive services to save computing resources but reactivates them when they are required for an upcoming task. Rorg monitors the system and keeps the resources utilization at a reasonable level.

We evaluated software encapsulation and dynamic orchestration on TritonBot using Rorg. With software encapsulation, TritonBot went through 126 software version updates and 71 configuration changes within a year. Rorg now manages 41 services (containers) on the robot. Dynamic orchestration reduced resource consumption on TritonBot significantly compared the original static orchestration: The robot consumes 89.4% of CPU time and 3.41 GB of memory on average without dynamic orchestration. With dynamic orchestration, the average CPU drops to 48.7%, and the memory usage is 2.85 GB, and we notice the robot responds much faster. These numbers are derived from TritonBot specifically, but we expect similar improvements by switching to dynamic orchestration from static orchestration in other multi-purpose service robots.

The experience of Rorg inspired us that many tools in other computer systems have great potential to be adapted in service robot systems. The proper identification of unique challenges in robotics will lead to solutions that improve various aspects of service robots, including system and reliability engineering.

Testing Robots using Broad-spectrum System-level Failure Simulation

In the process of finding potential errors in TritonBot and fixing them to improve TritonBot's reliability, we found that simulating a broad spectrum of rare failures at system level exposes design flaws and assists developers in improving the robustness of service robots.

In the long-term deployment of TritonBot, we faced a few repeated but rare failures that are nondeterministic and hard to reproduce. As a result, the developers had to rely on hidden clues in a massive amount of log to debug, and verifying a fix is non-trivial when the condition that triggers the bug is random or undetermined. Fault injection offers developers with a practical approach to study the long-term reliability of service robots. But most of the existing fault injection works in robotics do not scale well to the increasing failure patterns of modern and future service robots.

In response to these issues, we built RoboVac, an extensible and convenient fault injection framework that works at the system level and covers many of the failure patterns seen in long-term autonomous service robot deployments. Currently, RoboVac can inject four types of failures: middleware-layer failures, software process failures, network failures, and application-specific failures. These failure types are consistent with failure modes we observed on TritonBot.

RoboVac works with ROS, one of the most popular robot middlewares. By replacing the basic ROS library with a RoboVac-enhanced version, all existing ROS programs attain fault injection capability from RoboVac without recompiling. RoboVac leverages operating system features, middleware communication channels, and other general mechanisms to provide a broad-spectrum coverage of failure patterns. It is capable of injecting failures without specific knowledge about the robot application. RoboVac takes the robotics context into account and provides flexibility such as scripting and multi-failure injection.

We applied RoboVac on TritonBot and demonstrated our practices to confront the challenges in TritonBot deployment. RoboVac helped us to reproduce previously seen but rare failures and verify the fixes; it also revealed hidden design flaws in the system; the performance overhead of fault injection has little effect on the overall performance of the robot system. Through the application of RoboVac on TritonBot, we concluded that simulating a broad spectrum of rare failures at system level exposes design flaws and assists developers in improving the robustness of service robots.

Design Considerations in Long-term Autonomous Service Robot

We continued to develop TritonBot and deployed it for an extended period. TritonBot received new features such as leg detection, a more sophisticated dialogue system, and better logging. We did an extended deployment of TritonBot in stationary mode totaling $4,333$ hours, as well as a regular tour-guide mode deployment of 195 hours. From the TritonBot experience, we summarized the design principles for a long-term autonomous service robot.

After the two-year long-term experience with TritonBot, we concluded three significant aspects of reliability engineering for long-term deployment of autonomous service robots: scalability, resilience, and learning. Scalability enables a robot system to grow and gain more features smoothly. Resilience allows a robot to adapt to environmental changes and tolerate transient errors. Learning helps a robot to benefit from experiences and become more capable over time. In each of these aspects, we demonstrated TritonBot's design consideration to make it more capable of long-term autonomy.

We concluded a few design principles as a "toolbox" to build reliable long-term autonomous service robots. These design principles stand on different levels from programming, single component, subsystem integration, and all way up to the entire robot system. These design principles help guide robust and reliable long-term autonomous service robot designs.

7.2 Future Work of This Dissertation

Many opportunities remain open in reliability engineering for long-term autonomous service robots, either as the continuance of the contributions in this dissertation or as individual projects. This section provides some insight into on-going and future work related to this thesis.

TritonBot is designed as a general platform for service robot experiments. In addition to reliability engineering research, we anticipate TritonBot to be a suitable platform for human-robot interaction research as well. A separate on-going research project is attempting to transfer the experience of guiding tours to follow people; with much experience from observing visitors following itself, we hope TritonBot will be able to follow humans more naturally and robustly. With TritonBot being open-source, we hope it will become a common testbed for robot reliability and other research.

Rorg is a Linux-container-based software management system for a service robot that relives maintenance burdens and reduces resource contention. We have a follow-up project, "fast recovery for autonomous vehicles": taking advantage of services that are encapsulated in containers in Rorg, we try to periodically snapshot the self-contained services to help them to return to normal operation in case of failure as fast as possible; we plan to apply this technology to autonomous driving vehicles where software failures introduce prolonged degraded operation.

The RoboVac project finally provides a seamless framework to inject various kinds of failures into a robot system. There is a great potential to use RoboVac to do fuzz-testing on service robots. In essence, we can inject random combinations of failures into the robot at scale and discover design flaws in a service robot design. Similarly, the resistance to random failures can be used as a metric to evaluate the robustness of a service robot quantitatively.

To gain more experience in long-term autonomous service robots in real life, we are continuing the deployment of TritonBot. We are also working on making TritonBot a permanent, always-on fixture of our building at UC San Diego, which will provide us with insights of long-term autonomy at the scale of years.

7.3 Future of Reliability Engineering for Robots

Much of previous robotics research efforts were focused on “traditional” robotics topics such as perception, planning, control, etc. System and reliability research remained overlooked in robotics until service robots recently started to assist people in everyday life. Works in this dissertation call for the attention of reliability and system engineering in service robots.

Innovations in many computer science fields can inspire reliability engineering for service robots. For example, Rorg is based on Linux containers, which is an innovation in the computer system. Robot systems are similar to traditional computer systems in many aspects; therefore technologies in computer science such as computer system, computer architecture, embedded system, and security may all have impacts on robotics. While experiences in computer science and engineering may not be directly applicable to service robots, they can be tailored to fit the unique requirements in robotics.

Besides bringing in existing ideas in computer systems, thinking of the unique opportunities in long-term autonomous service robots can also contribute to their reliability. For example, the robots can use their long-term experience to improve their reliability (Section 6.6). Or, with the slightly larger tolerance of failures, the robots can perform reinforcement learning over a long period and at scale. The distinctive characteristics of long-term autonomous service robots can inspire novel ideas to improve their reliability.

As robotics technologies are becoming more mature, we hope reliability engineering research in long-term autonomous service robots will pave the path toward the broad application of service robots in home and office settings. We look forward to seeing robots assisting us in daily life in the very near future.

Bibliography

[1] H. I. Christensen, Ed., *A Roadmap for US Robotics: From Internet to Robotics*, Nov. 2016.

[2] J. Biswas and M. Veloso, “The 1,000-km challenge: Insights and quantitative and qualitative results,” *IEEE Intelligent Systems*, vol. 31, no. 3, pp. 86–96, May 2016.

[3] N. Hawes, C. Burbridge, F. Jovan, L. Kunze, B. Lacerda, L. Mudrova, J. Young, J. Wyatt, D. Hebesberger, T. Kortner, R. Ambrus, N. Bore, J. Folkesson, P. Jensfelt, L. Beyer, A. Hermans, B. Leibe, A. Aldoma, T. Faulhammer, M. Zillich, M. Vincze, E. Chinellato, M. Al-Omari, P. Duckworth, Y. Gatsoulis, D. C. Hogg, A. G. Cohn, C. Dondrup, J. P. Fentanes, T. Krajnik, J. M. Santos, T. Duckett, and M. Hanheide, “The strands project: Long-term autonomy in everyday environments,” *IEEE Robotics Automation Magazine*, vol. 24, no. 3, pp. 146–156, Sept 2017.

[4] P. Khandelwal, S. Zhang, J. Sinapov, M. Leonetti, J. Thomason, F. Yang, I. Gori, M. Svetlik, P. Khante, V. Lifschitz, J. K. Aggarwal, R. Mooney, and P. Stone, “Bwibots: A platform for bridging the gap between ai and humanrobot interaction research,” *The International Journal of Robotics Research*, vol. 36, pp. 635–659, 02 2017.

[5] D. Bohus, C. W. Saw, and E. Horvitz, “Directions robot: In-the-wild experiences and lessons learned,” in *Proceedings of the 2014 International Conference on Autonomous Agents and Multi-agent Systems*, ser. AAMAS ’14. Richland, SC: International Foundation for Autonomous Agents and Multiagent Systems, 2014, pp. 637–644.

[6] S. Wang and H. I. Christensen, “Tritonbot: First lessons learned from deployment of a long-term autonomy tour guide robot,” to appear in 2018 27th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN), Aug 2018.

[7] M. J.-Y. Chung, J. Huang, L. Takayama, T. Lau, and M. Cakmak, “Iterative design of a system for programming socially interactive service robots,” in *Social Robotics*, A. Agah, J.-J. Cabibihan, A. M. Howard, M. A. Salichs, and H. He, Eds. Springer International Publishing, 2016, pp. 919–929.

[8] S. Andrist, D. Bohus, E. Kamar, and E. Horvitz, “What went wrong and why? diagnosing situated interaction failures in the wild,” in *Social Robotics*, A. Kheddar, E. Yoshida,

S. S. Ge, K. Suzuki, J.-J. Cabibihan, F. Eyssel, and H. He, Eds. Springer International Publishing, 2017, pp. 293–303.

[9] M. Tonkin, J. Vitale, S. Herse, M.-A. Williams, W. Judge, and X. Wang, “Design methodology for the ux of hri: A field study of a commercial social robot at an airport,” in *Proceedings of the 2018 ACM/IEEE International Conference on Human-Robot Interaction*, ser. HRI ’18. New York, NY, USA: ACM, 2018, pp. 407–415.

[10] M. Wise, M. Ferguson, D. King, E. Diehr, and D. Dymesich, “Fetch & freight: Standard platforms for service robot applications,” in *Workshop on Autonomous Mobile Service Robots, International Joint Conference on Artificial Intelligence*, July 2016.

[11] M. Quigley, K. Conley, B. 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*, vol. 3, no. 3.2. Kobe, 2009, p. 5.

[12] S. Wang, X. Liu, J. Zhao, and H. I. Christensen, “Robotic reliability engineering: Experience from long-term tritonbot development,” in *12th Conference on Field and Service Robotics (FSR 2019)*. IEEE, 2019.

[13] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at Google with Borg,” in *Proceedings of the European Conference on Computer Systems (EuroSys)*, Bordeaux, France, 2015.

[14] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, “Mesos: A platform for fine-grained resource sharing in the data center,” in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI’11. Berkeley, CA, USA: USENIX Association, 2011, pp. 295–308.

[15] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, “Borg, omega, and kubernetes,” *Commun. ACM*, vol. 59, no. 5, pp. 50–57, Apr. 2016.

[16] H. Alemzadeh, D. Chen, A. Lewis, Z. Kalbarczyk, J. Raman, N. Leveson, and R. Iyer, “Systems-theoretic safety assessment of robotic telesurgical systems,” in *Proceedings of the 34th International Conference on Computer Safety, Reliability, and Security - Volume 9337*, ser. SAFECOMP 2015. New York, NY, USA: Springer-Verlag New York, Inc., 2015, pp. 213–227.

[17] A. L. Christensen, R. O’Grady, M. Birattari, and M. Dorigo, “Fault detection in autonomous robots based on fault injection and learning,” *Autonomous Robots*, vol. 24, no. 1, pp. 49–67, Jan 2008.

[18] B. Lussier, M. Gallien, J. Guiochet, F. Ingrand, M.-O. Killijian, and D. Powell, “Experiments with diversified models for fault-tolerant planning,” *IARP07, Roma, Italy*, 2007.

[19] G. Juez, E. Amparan, R. Lattarulo, A. Ruíz, J. Pérez, and H. Espinoza, "Early safety assessment of automotive systems using sabotage simulation-based fault injection framework," in *Computer Safety, Reliability, and Security*, S. Tonetta, E. Schoitsch, and F. Bitsch, Eds. Cham: Springer International Publishing, 2017, pp. 255–269.

[20] I. Malavolta, G. A. Lewis, B. Schmerl, P. Lago, and D. Garlan, "How do you architect your robots? state of the practice and guidelines for ros-based system," in *Proceedings of the 42nd International Conference on Software Engineering: Software Engineering in Practice*, 23-29 May 2020, to appear.

[21] S. Rosenthal and M. M. Veloso, "Mixed-initiative long-term interactions with an all-day-companion robot." in *AAAI Fall Symposium: Dialog with Robots*, vol. 10, 2010, p. 05.

[22] S. Rosenthal and M. Veloso, "Mobile robot planning to seek help with spatially-situated tasks," in *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, ser. AAAI'12. AAAI Press, 2012, pp. 2067–2073.

[23] I. R. Nourbakhsh, J. Bobenage, S. Grange, R. Lutz, R. Meyer, and A. Soto, "An affective mobile robot educator with a full-time job," *Artificial Intelligence*, vol. 114, no. 1, pp. 95 – 124, 1999. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0004370299000272

[24] W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun, "Experiences with an interactive museum tour-guide robot," *Artif. Intell.*, vol. 114, no. 12, p. 355, Oct. 1999. [Online]. Available: https://doi.org/10.1016/S0004-3702(99)00070-3

[25] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hhnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz, "Probabilistic algorithms and the interactive museum tour-guide robot minerva," *The International Journal of Robotics Research*, vol. 19, no. 11, pp. 972–999, 2000. [Online]. Available: https://doi.org/10.1177/02783640022067922

[26] M. Montemerlo, J. Pineau, N. Roy, S. Thrun, and V. Verma, "Experiences with a mobile robotic guide for the elderly," in *Eighteenth National Conference on Artificial Intelligence*. USA: American Association for Artificial Intelligence, 2002, p. 587592.

[27] R. Siegwart, K. Arras, S. Bouabdallah, D. Burnier, G. Froidevaux, X. Greppin, B. Jensen, A. Lorotte, L. Mayor, M. Meisser, R. Philippsen, R. Piguet, G. Ramel, G. Terrien, and N. Tomatis, "Robox at expo.02: A large scale installation of personal robots," *Robotics and Autonomous Systems*, vol. 42, pp. 203–222, 03 2003.

[28] Gunhee Kim, Woojin Chung, Kyung-Rock Kim, Munsang Kim, Sangmok Han, and R. H. Shinn, "The autonomous tour-guide robot jinny," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, vol. 4, Sep. 2004, pp. 3450–3455 vol.4.

[29] V. Kulyukin, C. Gharpure, and J. Nicholson, “Robocart: toward robot-assisted navigation of grocery stores by the visually impaired,” in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Aug 2005, pp. 2845–2850.

[30] D. Rodriguez-Losada, F. Matia, R. Galan, M. Hernando, J. M. Montero, and J. M. Lucas, “Urbano, an interactive mobile tour-guide robot,” in *Advances in Service Robotics*, H. S. Ahn, Ed. Rijeka: IntechOpen, 2008, ch. 14. [Online]. Available: https://doi.org/10.5772/5950

[31] F. Faber, M. Bennewitz, C. Eppner, A. Gorog, C. Gonsior, D. Joho, M. Schreiber, and S. Behnke, “The humanoid museum tour guide robotinho,” in *RO-MAN 2009 - The 18th IEEE International Symposium on Robot and Human Interactive Communication*, Sep. 2009, pp. 891–896.

[32] H. . Gross, H. Boehme, C. Schroeter, S. Mueller, A. Koenig, E. Einhorn, C. Martin, M. Merten, and A. Bley, “Toomas: Interactive shopping guide robots in everyday use - final implementation and experiences from long-term field trials,” in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2009, pp. 2005–2012.

[33] K. Yelamarthi, S. Sherbrook, J. Beckwith, M. Williams, and R. Lefief, “An rfid based autonomous indoor tour guide robot,” in *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2012, pp. 562–565.

[34] R. Stricker, S. Müller, E. Einhorn, C. Schröter, M. Volkhardt, K. Debes, and H.-M. Gross, “Konrad and suse, two robots guiding visitors in a university building,” in *Autonomous Mobile Systems 2012*, P. Levi, O. Zweigle, K. Häußermann, and B. Eckstein, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 49–58.

[35] R. Kmmerle, M. Ruhnke, B. Steder, C. Stachniss, and W. Burgard, “Autonomous robot navigation in highly populated pedestrian zones,” *Journal of Field Robotics*, vol. 32, no. 4, pp. 565–589, 2015. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/rob.21534

[36] R. Triebel, K. Arras, R. Alami, L. Beyer, S. Breuers, R. Chatila, M. Chetouani, D. Cremers, V. Evers, M. Fiore, H. Hung, O. A. I. Ramírez, M. Joosse, H. Khambhaita, T. Kucner, B. Leibe, A. J. Lilienthal, T. Linder, M. Lohse, M. Magnusson, B. Okal, L. Palmieri, U. Rafi, M. van Rooij, and L. Zhang, *SPENCER: A Socially Aware Service Robot for Passenger Guidance and Help in Busy Airports*. Cham: Springer International Publishing, 2016, pp. 607–622. [Online]. Available: https://doi.org/10.1007/978-3-319-27702-8_40

[37] K. Song, Y. Chiu, S. Song, and K. Zinchenko, “Scheduling and control of a cloud robot for reception and guidance,” in *2017 International Automatic Control Conference (CACS)*, Nov 2017, pp. 1–6.

[38] Y. Chen, F. Wu, W. Shuai, and X. Chen, “Robots serve humans in public placeskejia robot as a shopping assistant,” *International Journal of Advanced*

Robotic Systems, vol. 14, no. 3, p. 1729881417703569, 2017. [Online]. Available: https://doi.org/10.1177/1729881417703569

[39] A. Birolini, *Reliability engineering*. Springer, 2007, vol. 5.

[40] B. Beyer, C. Jones, J. Petoff, and N. Murphy, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Incorporated, 2016.

[41] P. O'Connor and A. Kleyner, *Practical reliability engineering*. John Wiley & Sons, 2012.

[42] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft, "Recovery oriented computing (roc): Motivation, definition, techniques,," USA, Tech. Rep., 2002.

[43] D. Hutchins, A. Ballman, and D. Sutherland, "C/c++ thread safety analysis," in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, Sep. 2014, pp. 41–46.

[44] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI08. USA: USENIX Association, 2008, p. 209224.

[45] E. Stepanov and K. Serebryany, "Memorysanitizer: Fast detector of uninitialized memory use in c++," in *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 46–55.

[46] International Organization for Standardization, "ISO 26262: Road vehicles – Functional safety," 2011.

[47] Y. Chen, A. M. Gillespie, M. W. Monaghan, M. J. Sampson, and R. F. Hodson, "On component reliability and system reliability for space missions," in *2012 IEEE International Reliability Physics Symposium (IRPS)*, April 2012, pp. 4B.2.1–4B.2.8.

[48] W. Burgard, A. B. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, and S. Thrun, "The interactive museum tour-guide robot," in *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*, ser. AAAI '98/IAAI '98. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1998, pp. 11–18.

[49] S. Thrun, M. Bennewitz, W. Burgard, A. B. Cremers, F. Dellaert, D. Fox, D. Hahnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz, "Minerva: a second-generation museum tour-guide robot," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, vol. 3, 1999, pp. 1999–2005 vol.3.

[50] R. Kirby, J. Forlizzi, and R. Simmons, “Affective social robots,” in *Robotics and Autonomous Systems*, vol. 58, Pittsburgh, PA, March 2010.

[51] T. Kanda, M. Shiomi, Z. Miyashita, H. Ishiguro, and N. Hagita, “A communication robot in a shopping mall,” *IEEE Transactions on Robotics*, vol. 26, no. 5, pp. 897–913, Oct 2010.

[52] T. Komatsubara, M. Shiomi, T. Kanda, H. Ishiguro, and N. Hagita, “Can a social robot help children’s understanding of science in classrooms?” in *Proceedings of the Second International Conference on Human-agent Interaction*, ser. HAI ’14. New York, NY, USA: ACM, 2014, pp. 83–90.

[53] B. Amos, B. Ludwiczuk, and M. Satyanarayanan, “Openface: A general-purpose face recognition library with mobile applications,” CMU-CS-16-118, CMU School of Computer Science, Tech. Rep., 2016.

[54] W. Hess, D. Kohler, H. Rapp, and D. Andor, “Real-time loop closure in 2d lidar slam,” in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, May 2016, pp. 1271–1278.

[55] E. Marder-Eppstein, E. Berger, T. Foote, B. Gerkey, and K. Konolige, “The office marathon: Robust navigation in an indoor office environment,” in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 300–307.

[56] Google LLC. (2017, May) Cloud speech api – speech to text conversion powered by machine learning. [Online]. Available: https://cloud.google.com/speech

[57] Y. Fu, Y. Xu, and T. S. Huang, “Estimating human age by manifold analysis of face pictures and regression on aging features,” in *2007 IEEE International Conference on Multimedia and Expo*, July 2007, pp. 1383–1386.

[58] O. A. I. Ramrez, H. Khambhaita, R. Chatila, M. Chetouani, and R. Alami, “Robots learning how and where to approach people,” in *2016 25th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, Aug 2016, pp. 347–353.

[59] Docker Inc. (2018, Nov.) Docker is an open platform to build, ship and run distributed applications anywhere. [Online]. Available: https://www.docker.com

[60] R. White and H. Christensen, “Ros and docker,” in *Robot Operating System (ROS): The Complete Reference (Volume 2)*, A. Koubaa, Ed. Springer International Publishing, 2017, pp. 285–307.

[61] T. Stivers, N. J. Enfield, P. Brown, C. Englert, M. Hayashi, T. Heinemann, G. Hoymann, F. Rossano, J. P. de Ruiter, K.-E. Yoon, and S. C. Levinson, “Universals and cultural variation in turn-taking in conversation,” *Proceedings of the National Academy of Sciences*, vol. 106, no. 26, pp. 10 587–10 592, 2009.

[62] National Institute of Standards and Technology. color feret database. [Online]. Available: https://www.nist.gov/itl/iad/image-group/color-feret-database

[63] Google Inc. (2018, Apr.) Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. [Online]. Available: https://developers.google.com/protocol-buffers/

[64] Y. M. Youssef and D. Ota, "A general approach to health monitoring & fault diagnosis of unmanned ground vehicles," in *2018 International Conference on Military Communications and Information Systems (ICMCIS)*. IEEE, 2018.

[65] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada, "An open approach to autonomous vehicles," *IEEE Micro*, vol. 35, no. 6, pp. 60–68, Nov 2015.

[66] H. Bruyninckx, M. Klotzbücher, N. Hochgeschwender, G. Kraetzschmar, L. Gherardi, and D. Brugali, "The brics component model: a model-based development paradigm for complex robotics software systems," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. ACM, 2013, pp. 1758–1764.

[67] N. Hochgeschwender, S. Schneider, H. Voos, H. Bruyninckx, and G. K. Kraetzschmar, "Graph-based software knowledge: Storage and semantic querying of domain models for run-time adaptation," in *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR)*, Dec 2016, pp. 83–90.

[68] N. Hochgeschwender, G. Biggs, and H. Voos, "A reference architecture for deploying component-based robot software and comparison with existing tools," in *2018 Second IEEE International Conference on Robotic Computing (IRC)*, Jan 2018, pp. 121–128.

[69] M. Foughali, B. Berthomieu, S. Dal Zilio, P.-E. Hladik, F. Ingrand, and A. Mallet, "Formal verification of complex robotic systems on resource-constrained platforms," in *FormaliSE: 6th International Conference on Formal Methods in Software Engineering*, 2018.

[70] J. Wienke and S. Wrede, "Autonomous fault detection for performance bugs in component-based robotic systems," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Oct 2016, pp. 3291–3297.

[71] Canonical Ltd. (2018) Snapcraft: Snaps are universal linux packages. [Online]. Available: https://snapcraft.io/

[72] H. Bruyninckx, P. Soetens, and B. Koninckx, "The real-time motion control core of the orocos project," in *Robotics and Automation, 2003. Proceedings. ICRA'03. IEEE International Conference on*, vol. 2. IEEE, 2003, pp. 2766–2771.

[73] A. Mallet, C. Pasteur, M. Herrb, S. Lemaignan, and F. Ingrand, "Genom3: Building middleware-independent robotic components," in *Robotics and Automation (ICRA), 2010 IEEE International Conference on*. IEEE, 2010, pp. 4627–4632.

[74] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[75] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, “An updated performance comparison of virtual machines and linux containers,” in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 2015, pp. 171–172.

[76] C. Ltd. (2018, Nov.) What’s lxd? [Online]. Available: https://linuxcontainers.org/lxd

[77] R. Mabry, J. Ardonne, J. N. Weaver, D. Lucas, and M. J. Bays, “Maritime autonomy in a box: Building a quickly-deployable autonomy solution using the docker container environment,” in *OCEANS 2016 MTS/IEEE Monterey*, Sept 2016, pp. 1–6.

[78] Y. Xu, Z. Yan, S. Wang, C. Yang, Q. Xiao, and Y. Bao, “Avalon: Building an operating system for robotcenter,” *CoRR*, vol. abs/1805.00745, 2018. [Online]. Available: http://arxiv.org/abs/1805.00745

[79] F. Lier, J. Wienke, A. Nordmann, S. Wachsmuth, and S. Wrede, “The cognitive interaction toolkit–improving reproducibility of robotic systems experiments,” in *International Conference on Simulation, Modeling, and Programming for Autonomous Robots*. Springer, 2014, pp. 400–411.

[80] J. Weisz, Y. Huang, F. Lier, S. Sethumadhavan, and P. Allen, “Robobench: Towards sustainable robotics system benchmarking,” in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*. IEEE, 2016, pp. 3383–3389.

[81] A. Pörtner, M. Hoffmann, and M. König, “Swarmrob: A toolkit for reproducibility and sharing of experimental artifacts in robotics research,” *arXiv preprint arXiv:1801.04199*, 2018.

[82] Google LLC. (2018, Nov.) grpc: A high performance, open-source universal rpc framework. [Online]. Available: https://grpc.io/

[83] A. Leigh, J. Pineau, N. Olmedo, and H. Zhang, “Person tracking and following with 2d laser scanners,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 726–733.

[84] J. M. Voas and K. W. Miller, “Using fault injection to assess software engineering standards,” in *Proceedings of Software Engineering Standards Symposium*, Aug 1995, pp. 139–145.

[85] J. W. Duran and S. C. Ntafos, “An evaluation of random testing,” *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 438–444, July 1984.

[86] P. Koopman, E. Tran, and G. Hendrey, “Toward middleware fault injection for automotive networks,” in *Fault Tolerant Computing Symposium*, June 1998, pp. 78 – 79.

[87] B. ISO, "14971 medical devices–application of risk management to medical devices," *Switzerland: International Organization for Standardization*, 2007.

[88] J. F. Broenink, C. Kleijn, P. G. Larsen, D. Jovanovic, M. Verhoef, and K. Pierce, "Design support and tooling for dependable embedded control software," in *Proceedings of the 2Nd International Workshop on Software Engineering for Resilient Systems*, ser. SERENE '10. New York, NY, USA: ACM, 2010, pp. 77–82. [Online]. Available: http://doi.acm.org/10.1145/2401736.2401745

[89] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. L. Goues, and P. Koopman, "Robustness testing of autonomy software," in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, May 2017, pp. 276–285.

[90] J. O'Keefe, D. Tarapore, A. Millard, and J. Timmis, "Adaptive online fault diagnosis in autonomous robot swarms," *Frontiers in Robotics and AI*, vol. 5, 11 2018.

[91] R. Woodman, A. F. Winfield, C. Harper, and M. Fraser, "Building safer robots: Safety driven control," *Int. J. Rob. Res.*, vol. 31, no. 13, pp. 1603–1626, Nov. 2012. [Online]. Available: http://dx.doi.org/10.1177/0278364912459665

[92] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer, "Fault injection techniques and tools," *Computer*, vol. 30, no. 4, pp. 75–82, April 1997.

[93] H. Ziade, R. A. Ayoubi, R. Velazco *et al.*, "A survey on fault injection techniques," *The International Arab Journal of Information Technology,*, vol. 1, no. 2, pp. 171–186, 2004.

[94] R. Natella, D. Cotroneo, and H. S. Madeira, "Assessing dependability with software fault injection: A survey," *ACM Comput. Surv.*, vol. 48, no. 3, pp. 44:1–44:55, Feb. 2016. [Online]. Available: http://doi.acm.org/10.1145/2841425

[95] J. A. Duraes and H. S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, Nov 2006.

[96] J. Calhoun, L. Olson, and M. Snir, "Flipit: An llvm based fault injector for hpc," in *Euro-Par 2014: Parallel Processing Workshops*, L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, Eds. Cham: Springer International Publishing, 2014, pp. 547–558.

[97] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," *SIGPLAN Not.*, vol. 47, no. 4, pp. 123–134, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2248487.2150990

[98] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Oct 2016, pp. 1–14.

[99] A. Avizienis, J. . Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan 2004.

[100] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[101] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur, "Fate and destini: A framework for cloud recovery testing," in *Proceedings of NSDI11: 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011, p. 239.

[102] P. Broadwell, N. Sastry, and J. Traupman, "Fig: A prototype tool for online verification of recovery mechanisms," in *Workshop on Self-Healing, Adaptive and Self-Managed Systems*. Citeseer, 2002.

[103] D. Costa, H. Madeira, J. Carreira, and J. G. Silva, *Xception^{TM}: A Software Implemented Fault Injection Tool*. Boston, MA: Springer US, 2003, pp. 125–139. [Online]. Available: https://doi.org/10.1007/0-306-48711-X_8

[104] Razorcat Development GmbH. (2019, Jan.) Tessy - automated testing of embedded software. [Online]. Available: https://www.razorcat.com/en/product-tessy.html

[105] Beyond Security. (2019, Jan.) bestorm software security testing tool - beyond security. [Online]. Available: https://www.beyondsecurity.com/bestorm.html

[106] ThemeREX. (2019, Jan.) Exhaustif – a grey box testing tool. [Online]. Available: http://www.exhaustif.es/

[107] Security Innovation. (2019, Jan.) Holodeck. [Online]. Available: https://github.com/SecurityInnovation/Holodeck

[108] J. Carlson and R. R. Murphy, "Reliability analysis of mobile robots," in *2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422)*, vol. 1, Sep. 2003, pp. 274–281 vol.1.

[109] S. Wang, X. Liu, J. Zhao, and H. I. Christensen, "Rorg: Service robot software management with linux containers," in *Robotics and Automation (ICRA), 2019 IEEE International Conference on*. IEEE, 2019.

[110] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones, "Socks protocol version 5," United States, 1996.

[111] J. G. R. III, A. J. B. Trevor, C. Nieto-Granda, A. Cunningham, M. Paluri, N. Michael, F. Dellaert, H. I. Christensen, and V. Kumar, "Effects of sensory precision on mobile robot localization and mapping," in *Experimental Robotics - The 12th International Symposium on Experimental Robotics, ISER 2010, December 18-21, 2010, New Delhi and Agra, India*, 2010, pp. 433–446. [Online]. Available: https://doi.org/10.1007/978-3-642-28572-1_30

[112] J. Bohren and S. Cousins, "The smach high-level executive [ros news]," *IEEE Robotics Automation Magazine*, vol. 17, no. 4, pp. 18–20, Dec 2010.

[113] A. Marzinotto, M. Colledanchise, C. Smith, and P. gren, "Towards a unified behavior trees framework for robot control," in *2014 IEEE International Conference on Robotics and Automation (ICRA)*, May 2014, pp. 5420–5427.

[114] A. Tate, "Generating project networks," in *Proceedings of the 5th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI77. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1977, p. 888893.

[115] G. Rodola. (2008, Apr.) psutil: Cross-platform lib for process and system monitoring in python. [Online]. Available: https://github.com/giampaolo/psutil

[116] The Regents of the University of California. (2019, Mar.) grpc: Catkinized grpc package. [Online]. Available: https://github.com/CogRob/catkin_grpc

[117] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures–a technology review," in *2015 3rd international conference on future internet of things and cloud*. IEEE, 2015, pp. 379–386.

[118] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[119] The Clang Team. (2007) Clang 11 documentation. [Online]. Available: https://clang.llvm.org/docs/index.html

[120] I. Malavolta, G. Lewis, B. Schmerl, P. Lago, and D. Garlan, "How do you architect your robots? state of the practice and guidelines for ros-based systems," in *Proceedings of the 42th International Conference on Software Engineering (ICSE'20)*, Seoul, South Korea, Oct. 2020.