Robotic Reliability Engineering: Experience from Long-term TritonBot Development

Shengye Wang, Xiao Liu, Jishen Zhao, and Henrik I. Christensen

Abstract While many researchers have built service robot prototypes that work perfectly under close human supervision, deploying an autonomous robot in an open environment for a long time is not always trivial. This paper presents our experience with TritonBot, a long-term autonomous receptionist and tour guide robot. We deployed TritonBot as an example to study reliability challenges in long-term autonomous service robots. During the past two years, we regularly do maintenance, fix issues, and roll out new features. In the process, we identified reliability engineering challenges in three aspects of long-term autonomy: scalability, resilience, and learning; we also formulated techniques to confront these challenges. Our experience shows that proper engineering practices and design principles reduces manual interventions and increases general reliability in long-term autonomous service robot deployments.

1 Introduction

While researchers continue to invent technology that extends robots' abilities to sense and interact with the environment, one of the often neglected research areas in robotics is reliability engineering. 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. The reliability issues have become an obstacle that prevents the robot from assisting people in their everyday lives. 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

Shengye Wang, Xiao Liu, Jishen Zhao, and Henrik I. Christensen
Department of Computer Science and Engineering, University of California San Diego
9500 Gilman Drive, MC 0404, La Jolla, CA 92093
e-mail: {shengye,x1liu,jzhao,hichristensen}@ucsd.edu

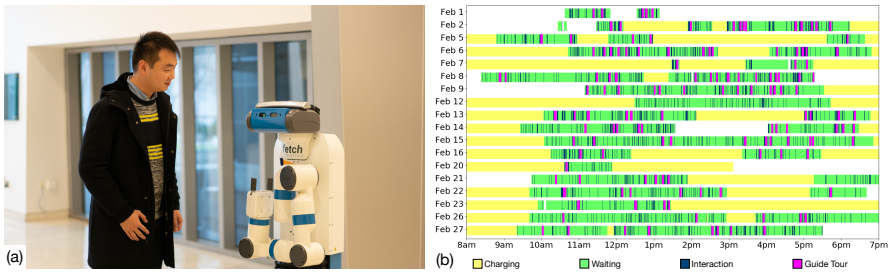

Fig. 1 (a): A visitor interacting with TritonBot. TritonBot can talk to people and show people around. (b): An one-month deployment log of TritonBot [24]. In February 2018, TritonBot worked (waiting, talking, and guiding tours) for 108.7 hours, and it actively interacted (talking and guiding tours) with people for 22.1 hours.

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 robot continues to work and evolve over an extended period.

We built TritonBot to study reliability challenges in developing and deploying a long-term autonomous service robot that interacts with people in an open environment. TritonBot (Fig. 1) is a robot receptionist and a tour guide deployed in an office building at UC San Diego. It greets visitors, recognizes visitors' face, remembers visitors' names, talks to visitors, and shows visitors the labs and facilities in the building. Our previous work [24] summarized lessons in the initial TritonBot development; this paper presents our efforts in making TritonBot more reliable during its long-term deployment.

Long-term autonomy consists of three primary factors: scalability, resilience, and learning: Scalability enables the robot system to grow and gain more features smoothly. Resilience allows the robot to adapt to environmental changes and tolerate transient errors. Learning helps the robot to benefit from experiences and become more capable over time. Our contributions in this paper are: (1) Identification of failure modes and reliability challenges in scalability, resilience, and learning using TritonBot. (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 robot. 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 paper is organized as follows: Section 2 discusses related work about long-term autonomy and reliability engineering. Section 3 introduces the overall design and functions of TritonBot. Section 4 describes our efforts in making the TritonBot system scalable in its long-term evolvement (Scalability). Section 5 shows our practices in making TritonBot resilient to failures (Resilience). Section 6 presents our attempts to improve TritonBot over time both autonomously or with the help from developers (Learning). Finally, Section 7 concludes this paper.

2 Background and Related Work

A number of projects have studied long-term autonomy. The STRANDS project [12] deployed security patrol and guide robots and reached a few weeks of autonomy without intervention. CoBots [21] navigated over 1,000 km in open environments and intensively studied long-term mapping, localization, and navigation. BWIBots [14] is a custom-designed long-term multi-robot platform for AI, robotics, and human-robot interaction that aims to be a permanent fixture in a research facility. Early "roboceptionists" and tour guide robots like Valerie [15], RHINO [4], and Minerva [23] interacted and provided tours to visitors in schools and museums with an emphasis on social competence or robust navigation in crowded environments. Previous work also reported deployment experiences and failure statistics in other robot systems [6, 7]. These work provided valuable experiences of robots under challenging environments, but few of them conclude lessons from a system-design perspective.

Reliability engineering is not a new concept in engineered systems; it is a critical study that keeps an engineered system reliable, available, maintainable, and safe [3]. Some companies in the industry even have created a *Site Reliability Engineer (SRE)* role in supporting growing Internet businesses [2]. Birolini [3] summarizes theories and provides qualitative approaches to the study of the reliability, failure rate, maintainability, availability, safety, risk, quality, cost, liability, and so on. O'Connor et al. [18] further give field-specific reliability engineering examples of mechanical systems, electronic systems, software, design for reliability, manufacturing, and more. Beyer et al. [2] from Google, Inc. discuss "site reliability engineering" and the principles and practices that keep Google's planet-scale production system healthy and scalable; they combine automated tools and appropriate engineering and emergency response workflows. None of these works 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.

3 TritonBot Overview

Our goal in the TritonBot project is to identify reliability challenges in long-term autonomous service robots and incorporate appropriate reliability engineering methods from large-scale software and other engineered systems to confront these challenges. To this end, we built TritonBot as a realistic example of a long-term autonomous service robot: TritonBot has a number of advanced features including face recognition, voice recognition, natural language understanding, navigation, and so on; but it is not overly complicated, and thus it reflects the reliability issues that may occur in a commercial service robot in the near future. In a previous work [24]

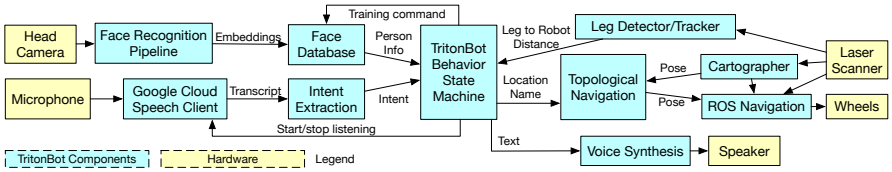

Fig. 2 A block diagram shows the primary components in TritonBot and the dataflow between them. A state machine controls the behavior of TritonBot, and standalone components including face recognition, voice recognition, leg detection and tracking, localization, and navigation support the TritonBot functions.

we presented the initial lessons learned from TritonBot in its initial deployment; we have been improving the TritonBot system to make it more robust since then.

TritonBot is based on the *Fetch Research Edition* platform from Fetch Robotics Inc. [26]. 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; these laser scanners enable 360° leg detection and tracking. 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 is deployed in Atkinson Hall, a research facility at UC San Diego. It stands in the hallway of the building, faces the entrance, and continuously detects faces in its view. When TritonBot detects a previously seen face, it greets the person by name. Otherwise, it briefly introduces itself, asks for the name of the visitor, and offers a trivia game. The simple questions allow the robot to interact with the visitor face-to-face and collect face samples. After the greetings and the games, TritonBot offers a guided tour to the user. The robot can show the visitor a few places of interest in the building, including a prototyping lab, a robot showroom, a "smart home" demo room, and a gallery. At each spot, the robot briefly introduce the place and confirms whether the visitor is still around using leg detection.

TritonBot uses the Robot Operating System (ROS), a standard middleware for robot prototypes for research. Fig. 2 shows the software architecture in TritonBot. The system leverages the open source software Cartographer [13] for localization and mapping, and the ROS navigation stack "movebase" [17] controls the robot to move around. A face recognition pipeline utilizes OpenFace [1] to generate face embeddings and calculate the similarity between two faces. An open source leg tracker [16] detects and track human leg positions around the robot using laser scan ranges. TritonBot also leverages a cloud service Google Cloud Speech API [9] to convert a user's speech to a textual transcript; a commercial software package synthesizes TritonBot's voice. To summarize, we integrated many existing components together to build TritonBot, but wrote about $100,000$ new lines of code (mostly C++ and Python) to build the entire TritonBot system.

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.

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.

4.2 Forward- and Backward-compatibility

Many robot systems use Robot Operating System (ROS) [20] 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) [11] to provide forward- and backward-compatibility. ProtoBuf is a language-neutral, 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 program. 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. We collect about 1.2 GB ProtoBuf-based logs monthly on average, 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.

4.3 Decoupling Software Components

In addition to the lack of backward- and forward-compatibility, ROS has two more limitations: the inflexible networking requirements, and the lack of security support. ROS programs (nodes) assumes bidirectional direct TCP/UDP connection between each other, so they cannot communicate over networking environments with firewalls or network address translation (NAT) devices. In addition, ROS communication lacks encryption and authentication support, which prevent it from communicating over public channels such as the Internet. These three issues contribute to the software coupling challenges in scalability.

To overcome these shortcomings, we built a part of the TritonBot system with open-source libraries and gRPC [10] 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` [22] that helps the users to generate, compile, and link ProtoBuf libraries and gRPC stubs within the ROS build environment.

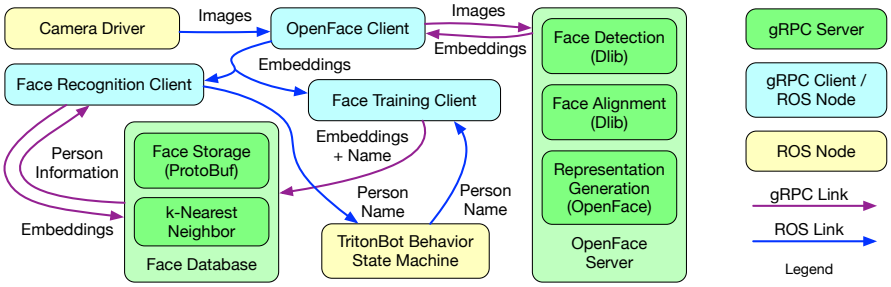

Fig. 3 The face recognition pipeline in TritonBot. A few components in ROS bridge the ROS system with standalone gRPC servers, so that TritonBot exploits the existing open-source components in ROS and backward- and forward-compatibility from gRPC. After running TritonBot for some time, we moved the "OpenFace server" to a remote computer with powerful GPUs to save computing resources on the on-robot computer.

The face recognition pipeline in the TritonBot system heavily leverages ProtoBuf and gRPC to maximize its scalability. Fig. 3 shows the components and the data flow in the face recognition pipeline. 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 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.

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.

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. The Linux container technology [19] provides a convenient approach to build, deploy, and run applications on different machines. Backed by the Linux namespaces that isolate and virtualizes system resources for a set of processes, Linux containers are much lighter-weighted than fully virtualized kernels in hypervisors or virtual machines.

Initially, we adopted Docker [8], a popular and open-source container management tool to run every software components (41 in total) as separate containers. The

isolated execution environment for each of them. When TritonBot system grows larger, resource contention becomes an issue in scaling up TritonBot system. In data centers, scaling up software system means spawning more program instances and load-balancing the tasks among them; popular tools like Kubernetes [5] leverage Linux containers to provide a unified platform to manage the programs. A service robot like TritonBot only carries limited computing resource; 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 build Rorg [25], an open source container-based software management system. Rorg not only provides an accessible mechanism to use Linux containers in robot programming environment, but it also models the component dependencies and decides what components 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 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" [2].

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 require unique customization of the tools.

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.

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; if 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.

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 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 failures 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 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 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 unavailability of another component.

5.3 Relying on Separate Subsystems

Fig. 4 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

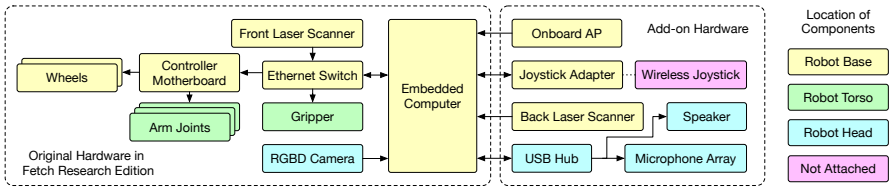

Fig. 4 Main hardware components in the TritonBot system. We added some additional hardware to the Fetch Research Edition.

built the early TritonBot prototype, we relied solely on programming and configuring the embedded computer. Soon we found that the on-robot computer become 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 systems reduces single-point failures and prepares the robot and the developer for unexpected situations.

5.4 Monitoring the System

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

The robot monitor (Fig. 5(a)) 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 of the robot and gives the developer an overview of the robot status with a simple glance. In addition to the monitoring tablet, TritonBot also analyzes its daily log and sends an E-mail about its daily work summary to the developers every mid-night. The report includes the interaction transcript and a summary about 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 also available for further analysis. Both of the monitoring methods retains the robot's autonomy, but they allow the developers and the users to understand the robot's status at different levels.

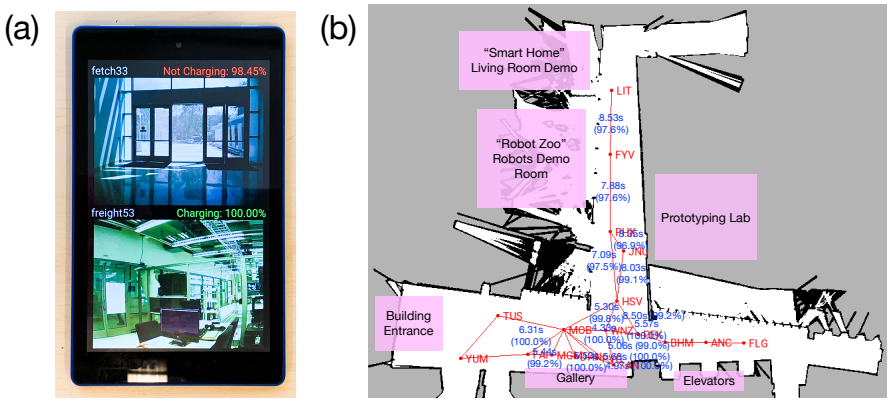

Fig. 5 (a) An Android tablet that displays the robot's eye view and battery status. The developer occasionally monitors the general status of the robot during the deployment. (b) The navigation topological map of TritonBot. The maps shows the average time to traverse an "airway" on the map as well as the success rate.

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.

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.

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 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 [17], we added a topological map layer of waypoints and paths (Fig. 5(b)). 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. Fig. 5(b) 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.

6.3 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 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 fuzz-testing scheme that enables automatic error discovery on service robots.

7 Conclusion

This paper presents robotic reliability engineering under the long-term autonomy scenario that a service robot works and evolves for an extended period. We discuss our reliability engineering practices in the context of TritonBot, a tour guide and receptionist robot in a university building. 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 increases the reliability of TritonBot.

Failures in service robots are unavoidable but manageable. TritonBot provides a realistic example of a long-term autonomous service robot in an open environment. We will continue to deploy TritonBot to provide more experiences and insights of running a long-term autonomous service robot. The complete TritonBot source code is available at `https://github.com/CogRob/TritonBot`. We hope that the reliability engineering techniques and experiences from TritonBot will inspire more robust long-term autonomous service robot designs.

References

1. Amos, B., Ludwiczuk, B., Satyanarayanan, M.: Openface: A general-purpose face recognition library with mobile applications. Tech. rep., CMU-CS-16-118, CMU School of Computer Science (2016)
2. Beyer, B., Jones, C., Petoff, J., Murphy, N.: Site Reliability Engineering: How Google Runs Production Systems. O'Reilly Media, Incorporated (2016)
3. Birolini, A.: Reliability engineering, vol. 5. Springer (2007)
4. Burgard, W., Cremers, A.B., Fox, D., Hähnel, D., Lakemeyer, G., Schulz, D., Steiner, W., Thrun, S.: The interactive museum tour-guide robot. In: Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence, AAAI '98/IAAI '98, pp. 11–18. American Association for Artificial Intelligence, Menlo Park, CA, USA (1998)
5. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Commun. ACM **59**(5), 50–57 (2016). DOI 10.1145/2890784
6. Carlson, J., Murphy, R.R.: Reliability analysis of mobile robots. In: 2003 IEEE International Conference on Robotics and Automation (Cat. No.03CH37422), vol. 1, pp. 274–281 vol.1 (2003). DOI 10.1109/ROBOT.2003.1241608
7. Chung, M.J.Y., Huang, J., Takayama, L., Lau, T., Cakmak, M.: Iterative design of a system for programming socially interactive service robots. In: A. Agah, J.J. Cabibihan, A.M. Howard, M.A. Salichs, H. He (eds.) Social Robotics, pp. 919–929. Springer International Publishing (2016)
8. Docker Inc.: Docker is an open platform to build, ship and run distributed applications anywhere. (2018). URL `https://www.docker.com`
9. Google LLC: Cloud speech api – speech to text conversion powered by machine learning (2017). URL `https://cloud.google.com/speech`
10. Google LLC: grpc: A high performance, open-source universal rpc framework (2018). URL `https://grpc.io/`

11. Google LLC: Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. (2018). URL https://developers.google.com/protocol-buffers/
12. Hawes, N., Burbridge, C., Jovan, F., Kunze, L., Lacerda, B., Mudrova, L., Young, J., Wyatt, J., Hebesberger, D., Kortner, T., Ambrus, R., Bore, N., Folkesson, J., Jensfelt, P., Beyer, L., Hermans, A., Leibe, B., Aldoma, A., Faulhammer, T., Zillich, M., Vincze, M., Chinellato, E., Al-Omari, M., Duckworth, P., Gatsoulis, Y., Hogg, D.C., Cohn, A.G., Dondrup, C., Fentanes, J.P., Krajnik, T., Santos, J.M., Duckett, T., Hanheide, M.: The strands project: Long-term autonomy in everyday environments. IEEE Robotics Automation Magazine **24**(3), 146–156 (2017). DOI 10.1109/MRA.2016.2636359
13. Hess, W., Kohler, D., Rapp, H., Andor, D.: Real-time loop closure in 2d lidar slam. In: 2016 IEEE International Conference on Robotics and Automation (ICRA), pp. 1271–1278 (2016). DOI 10.1109/ICRA.2016.7487258
14. Khandelwal, P., Zhang, S., Sinapov, J., Leonetti, M., Thomason, J., Yang, F., Gori, I., Svetlik, M., Khante, P., Lifschitz, V., K. Aggarwal, J., Mooney, R., Stone, P.: Bwibots: A platform for bridging the gap between ai and humanrobot interaction research. The International Journal of Robotics Research **36**, 635–659 (2017)
15. Kirby, R., Forlizzi, J., Simmons, R.: Affective social robots. In: Robotics and Autonomous Systems, vol. 58. Pittsburgh, PA (2010)
16. Leigh, A., Pineau, J., Olmedo, N., Zhang, H.: Person tracking and following with 2d laser scanners. In: 2015 IEEE International Conference on Robotics and Automation (ICRA), pp. 726–733 (2015). DOI 10.1109/ICRA.2015.7139259
17. Marder-Eppstein, E., Berger, E., Foote, T., Gerkey, B., Konolige, K.: The office marathon: Robust navigation in an indoor office environment. In: 2010 IEEE International Conference on Robotics and Automation, pp. 300–307 (2010). DOI 10.1109/ROBOT.2010.5509725
18. O'Connor, P., Kleyner, A.: Practical reliability engineering. John Wiley & Sons (2012)
19. Pahl, C., Lee, B.: Containers and clusters for edge cloud architectures–a technology review. In: 2015 3rd international conference on future internet of things and cloud, pp. 379–386. IEEE (2015)
20. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: Ros: an open-source robot operating system. In: ICRA workshop on open source software, vol. 3, p. 5. Kobe (2009)
21. Rosenthal, S., Veloso, M.M.: Mixed-initiative long-term interactions with an all-day-companion robot. In: AAAI Fall Symposium: Dialog with Robots, vol. 10, p. 05 (2010)
22. The Regents of the University of California: grpc: Catkinized grpc package (2019). URL https://github.com/CogRob/catkin_grpc
23. Thrun, S., Bennewitz, M., Burgard, W., Cremers, A.B., Dellaert, F., Fox, D., Hahnel, D., Rosenberg, C., Roy, N., Schulte, J., Schulz, D.: Minerva: a second-generation museum tour-guide robot. In: Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C), vol. 3, pp. 1999–2005 vol.3 (1999). DOI 10.1109/ROBOT.1999.770401
24. Wang, S., Christensen, H.I.: Tritonbot: First lessons learned from deployment of a long-term autonomy tour guide robot. In: Proceedings of the 2018 IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN), pp. 158–165 (2018)
25. Wang, S., Liu, X., Zhao, J., Christensen, H.I.: Rorg: Service robot software management with linux containers. In: Robotics and Automation (ICRA), 2019 IEEE International Conference on. IEEE (2019)
26. Wise, M., Ferguson, M., King, D., Diehr, E., Dymesich, D.: Fetch & freight: Standard platforms for service robot applications. In: Workshop on Autonomous Mobile Service Robots, International Joint Conference on Artificial Intelligence (2016)