A Framework for Remotely Sharing Experimental Environments

Scott C. Livingston

Abstract— **Reproducibility is fundamental to robotics research, but access to appropriate hardware is a major limiting factor of reproducing experiments. This limitation usually arises from custom-built solutions without sufficient documentation and, simply, high costs. We propose that the best way to improve reproducibility is by removing barriers to safely sharing hardware. Towards this, we present a framework for making experimental environments remotely accessible. Modern virtualization tools like Linux containers are leveraged to enable reproducible and isolated access. The basic idea is to run user code in a container and transport all inputs and outputs through proxy programs that monitor for unsafe states. This interface is low-level: proxies can operate across serial lines, TCP, UDP, or HTTP connections. Thus, users are not locked into any particular programming library or application-level messaging system. The framework is shown in four case studies: the social robot Misty, the mobile robot Kobuki (base of the popular TurtleBot 2) with a LiDAR, wireless sensor networks, and Autoware.**

I. Introduction

Reproducibility is a pillar of science: objectivity is established because the same result can be obtained by different people. Robotics, however, presents unique challenges to having reproducible research. Practical robots involve specialized (rather than commodity) hardware interacting with physical environments. The whole system used in an experiment can be very complex, involving many custom scripts and local configuration that are not practical to include in a "Materials and Methods" section of a paper. Even within application domains that would seem to have a sufficiently narrow scope to have standardized hardware, such as mobile manipulation, different labs can have drastically different hardware for accidental reasons, e.g., financial constraints, or legacy platforms that are not available for purchase.

Historically, the research community has compensated for having less independent reproduction of experiments by having a culture of video demonstrations. Videos of real robots show readers a practical existence proof: the paper might not have enough detail to reproduce results, but the results truly were produced at least once, so with enough time and effort someone else should be able to do the same. In contrast, authors who only provide simulations are at risk of unintentional cheating by adjusting runtime parameters until the simulation trials succeed. Shared simulation benchmarks help prevent this, e.g., Meta-World [1] or the SubT Challenge [2], but typically these are only treated as prerequisite to the goal of experiments on real hardware.

The other major approach to reproducibility in robotics research is through benchmarks and standard development platforms. The complete list is too long to include here, but several recent examples of projects with open source code and hardware designs include TurtleBot [3], F1tenth [4], Donkey Car [5], Crazyflie [6], and Autoware [7], [8]. Even if a particular robot becomes standard in some topic area, reproducing results can still be difficult because there are many contextual aspects of experiments, such as how to configure Linux hosts, that are easily forgotten or considered too idiosyncratic to include in published source code. Also, being open source does not address factors like financial constraints: space in which to operate the robot, and the cost of the hardware itself.

In this paper, we propose a general framework for sharing robots and experiments. We improve reproducibility by removing the need to acquire expensive hardware or operating space. Instead, researchers can allow remote users to safely access their robot and directly experience the environment. The basic idea is to leverage advances in virtualization by making access to robots much like cloud computing: users request time, then get exclusive temporary access, and finally the user's instance is deleted; then, the hardware is prepared for the next user. Physical separation often reflects logical separation: controller software runs on a separate computer from the software that translates throttle, brake, and steer commands into actual motion. Thus, we can insert a filter between components of the robot to perform logging and apply filter rules according to formal specifications.

A central principle in the proposed system is to be "low-level" in the following respect: source code that is known to conduct an experiment will work without modification inside instances provided via our system. We achieve this by filtering raw device interfaces: UDP, TCP, HTTP, and serial buses. Following this principle makes the contribution of this work important because it is not tied to a particular messaging framework like ROS [9].

The main contributions of this work are the following: a simple yet general system for provisioning remote access across the Internet, a framework of proxies for bounding states reachable by users, and analytics about low-level activity from proxy data.

II. Prior Work

Relevant work can be organized into three categories: (1) robots available via the World Wide Web without a competitive or performance-measuring aspect, (2) education or competition-oriented testbeds that support some remote access, and (3) commercial systems for "cloud robotics."

Over the past 25 years, there have been various academic projects that allow remote users to access robots through

S.C.L. is based in California, USA, and has the email address scott@rerobots.net.

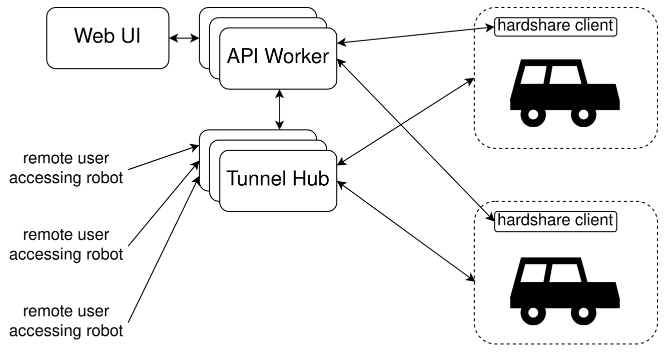

Web UI
API Worker
Tunnel Hub
hardshare client
hardshare client
remote user accessing robot
remote user accessing robot
remote user accessing robot

Fig. 1. Main components of remote device sharing: each client reports whether its associated robot (car, arm, ...) is available; API Workers act as brokers between remote users and clients; hubs manage VPN and SSH connections; the Web UI provides graphical menus, video streaming, etc.

turn-based or "headless" processes. In this context, "headless" refers to the absence of a real-time user interface: code is loaded on the testbed, executed locally, and then users get results in return. In particular, latency is not critical because there is no long distance control loop. Work from the early days of the World Wide Web leveraged technologies of that time, e.g., Java applets and request-response Web forms [10]. More recent work has been built on ROS or provides custom APIs against which your code must be built in order to work on the robots [11], [12]. Other work is similarly constrained but with a focus on education and competitions, e.g., Duckietown [13], the MPI Challege [14], and F1tenth racing [4]

There are also commercial providers of tools to access and monitor code on remote robots. However, these focus on fleet management in certain industries like warehouse robots or they are built exclusively for ROS-based robots, e.g., by Rapyuta Robotics [15] and Husarion [16].

III. Framework

In this section, we present the framework for sharing hardware. Source code is available in repositories of the rerobots organization on GitHub. Readers are advised to begin with the client repository at github.com/rerobots/hardshare. Practical case studies are given in Section IV.

The framework has two basic aspects: a system for provisioning devices and a set of proxies that filter access to sensors and motors.

A. Provisioning

As with any service where there is more than one potential user, we need a system for ensuring mutual exclusivity: at most one user accesses the robot at a time. The system should also provide queues, reservations, and other features typical of sharing a limited resource. To support reproducibility, each user should begin with the robot in an expected set of initial states. Finally, the persons who are sharing their hardware must be able to interrupt remote access and, optionally, enforce permissions or other constraints on users, for example, during peer review.

The overall architecture is shown in Figure 1. There are four components. First, the *hardshare client* is software that runs locally in the experimental environment and sends relevant status information to the server processes. The client is the main point of managing access for the person who is sharing hardware. Through it, capabilities are declared, availability of the robot is advertised, and interface contracts are enforced. Interfaces and proxies are presented in Section III-B. Clients also run initialization and termination code on robots to prepare them for each user.

The second component is the *API worker*, which manages the server-side of provisioning. Clients connect to API workers via WebSocket. API workers enforce permissions of remote users and guarantee that at most 1 user can reach a robot at a time. When a new instance is started, API workers create a tunnel (SSH or VPN) in the *tunnel hub* between users and the experimental environments. Tunnels are simple from the user's perspective: an IP address and port for SSH connections, or a client certificate to join a VPN.

The fourth component is the website. Users go to it to find relevant robots, to request connections, and to manage existing access. The graphical interface is not required: API workers also accept requests via a public HTTP-based API, where users can POST new instances, GET instance details, etc.

The isolated access of a user to a robot is referred to as an *instance*. The lifecycle of an instance is as follows:

1) User requests time on a robot. If it is available, then an instance is started.
2) Instance initialization includes health checks on the hardware. If something is wrong, the instance is marked as failed, and user is notified to try again later. Else, local initialization scripts (unique to the experimental environment) are run.
3) Instance is marked as "ready," and the user is given access credentials.
4) When the instance expires, or when the user declares that they are done, the instance terminates: local clean-up scripts (again, unique to the experimental environment) are run.

The instance includes a host that is presented to the remote user as root-access to a Linux machine. As typical for robots, hardware is accessible from this host via serial lines (character device files in Unix) or as TCP/IP peers. It is not obvious from the user's perspective, but they are inside a Linux container. The implemented framework supports popular container runtimes, including Docker, Podman, or LXD, which are chosen for the client according to needs of the lab or experimental environment. From inside the container, the user can only reach the robot as permitted by proxies, discussed in the next section.

B. Proxying

A *proxy* is a program that passes messages between two processes, acting as a wrapper around one of the processes. In practice, proxies are often transparent, performing common

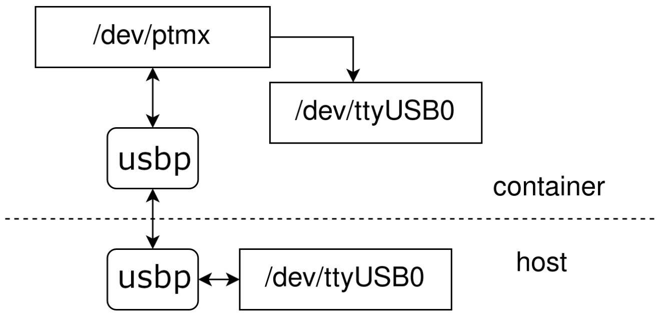


Fig. 2. USB serial proxy (usbp) between a Linux container and host. Users within the container read and write to a pseudoterminal (ptmx), and traffic is forwarded via TCP across the container network, unless unsafe states are detected. The proxy program usbp decodes messages and, combined with external pose tracking, decides when to stop access.

Internet tasks like load-balancing Web servers and delivering cached content without the user knowing that a proxy handled their request instead of the intended process.

Except for simple network features like NAT (network address translation), the principle of proxying has not been explored at lower levels of robot architectures. Thanks to recent advances in computational speed and bandwidth, this is now possible and achieved in the framework presented here. In the following subsections, different proxies are presented according to the communication layer across which they operate.

Proxying may require some custom code for each targeted device. For example, a TCP proxy for a Dexter HD arm is going to be different than a TCP proxy for a UR5 arm. Both are robot arms that have similar mechanics, but the details of the robot APIs are different. Given that proxies made for popular robots are open source, we expect that new hardware will be straightforward to support by building on existing proxy code.

There are three kinds of proxy behavior. First, the no-op proxy forwards all messages without modification and without exception. This supports cutting off access quickly from the remote user when the instance terminates, but otherwise, it does not interject controls. Second, the simple blocking proxy forwards messages that satisfy a set of rules and rejects all others. This is useful when the robot is safe on its own, but some commands should be unavailable to remote users, e.g., rebooting the robot or using character encodings outside of ASCII or UTF-8. Finally, the safety proxy monitors messages and enforces a boundary contract to prevent unsafe states from being reached.

The framework includes proxies with all of the above behaviors and implemented for the following protocols.

1) Serial Filters: UART serial buses are widely used in embedded devices, e.g., Hokuyo LiDAR [17] and Dynamixel motors [18]. The proposed filter architecture is shown in Figure 2, where the device is presented inside the Linux container as a pseudoterminal [19], and all serial traffic is monitored. This filter is demonstrated in Sections IV-B and IV-C.

2) HTTP: While HTTP is widely known for transferring webpages, it is also popular as the medium for APIs of most modern web apps. In robotics, HTTP APIs are sometimes used for behaviors that do not have real-time deadlines, such as in social robots. In summary, HTTP is organized in a request-response pattern, so proxies act on requests and return rejection responses with informative status codes like 403 (permission denied). An HTTP proxy that blocks some commands and enforces boundary safety is demonstrated in Section IV-A.

3) TCP and UDP: Many robot sensors and actuators communicate via TCP or UDP. TCP is connection-oriented with guarantees such as delivery and ordering of messages. It is used in the interfaces of the Universal Robot arms. UDP is a connectionless protocol with few guarantees (messages can arrive out-of-order or not at all), but it is fast and robust in practice. In robotics, it is used with very high bandwidth sensors such as the Velodyne spatial LiDARs, as we did for integration with Autoware (Section IV-D).

IV. Applications

In this section, we present case studies of applying the framework to different robots. These robots represent several major areas of research: human-robot interaction, indoor mobile robots, and autonomous cars. The provisioning system (Section III-A) is the same for all environments. However, the proxies required additional development uniquely for each robot. Nonetheless, we expect adding support for similar robots in each domain will be straightforward.

A. Misty 2 Social Robot Platform

1) Robot and Environment: The social robot Misty is programmed using an HTTP-based API [20]. It has differential drive dynamics with significant slip and two short arms that cannot grasp objects but are emotionally expressive. The head has full spherical motion, and the body has many sensors: microphone array, high fidelity camera, a structured light depth sensor, contact sensors in the front and back, near-distance IR range finders, speakers, and a color screen for displaying animated faces. Misty is shown in Figure 3.

For remote users, Misty is placed in a flat, indoors area with external camera and speaker. There is LiDAR-based pose tracking outside of the working area that supports the filter (described in the next section). Finally, there is a pad for wirelessly charging the battery in Misty. There are also several photographs of human heads on the walls to support experiments with face recognition.

2) HTTP Filter: Misty is controlled by sending motion commands via HTTP requests and receiving sensor data streamed via WebSocket. At all times, Misty's pose is tracked by an external system. As such, we developed a proxy that filters access in two respects:

1) a set of rules defining accepted HTTP paths and parameter ranges,
2) a kill-switch that drops all incoming requests if Misty's pose enters the boundary of the workspace and that

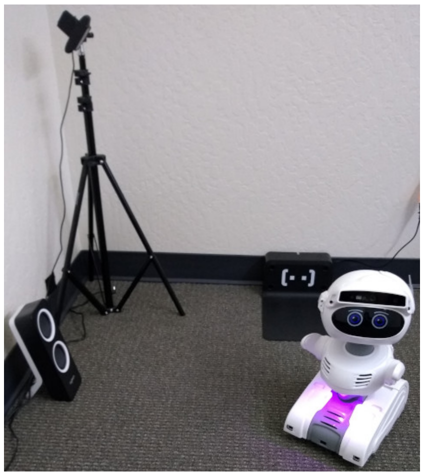


Fig. 3. Misty and the experimental environment, including speaker, overhead camera, and charging pad, as described in Section IV-A.

automatically moves Misty back into the safe operating region.

The rules can be changed at runtime according to experimental needs, though we did find a motivation for allowing administrative commands like rebooting Misty. The "kill-switch" behaves as a barrier certificate that forces the robot away from the walls surrounding the space and thereby prevents damaging collisions.

3) Termination: When a user's access ends, Misty is automatically docked onto the charged pad. The external pose tracking allowed us to reliably dock Misty, making the pad the expected initial state for the next user.

4) Results: We connected two Misty robots into the framework for more than 1 year. The provisioning system treats the environments as interchangeable, improving availability for experiments and education: if one Misty is busy, an incoming user can be directed to the other Misty. During this time, anonymous users from around the world safely learned and tested code on Misty. Some of the users were professors teaching children about robots [21]. Other users programmed Misty with Python, Java, and JavaScript. Because the framework is implemented below the level of language tools or libraries, everyone's code ran without modification.

B. Kobuki Mobile Base with LiDAR

1) Robot and Environment: The Kobuki mobile base by Yujin Robot had much popularity several years ago as part of the TurtleBot 2 development platform promoted in the ROS community. The TurtleBot 2 had a low-cost structured light depth sensor. For this experimental environment, we instead mounted (higher cost) Hokuyo LiDAR sensors. Two Kobuki robots with LiDAR are in a flat, indoors area as shown in Figure 4. The workspace is visible to users from two external cameras.

2) Serial Filter: Most users will have experience with moving the Kobuki through velocity commands via ROS messages (in TCP packets). Internally, the ROS nodes are

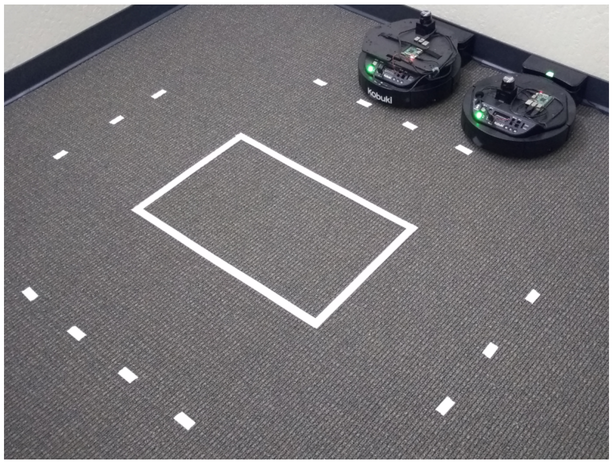
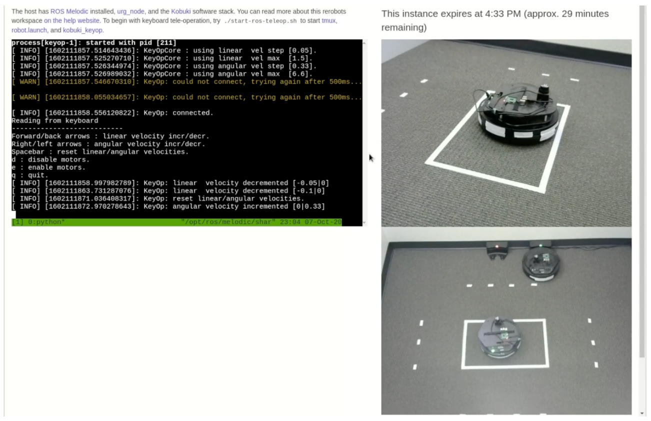


Fig. 4. Two Kobuki robots with RaspberryPi boards and Hokuyo LiDAR, each on a battery-charging pad. Floor markings support experiments by remote users.



The host has ROS Melodic installed, urg_node, and the Kobuki software stack. You can read more about this rerobots workspace on the help website. To begin with keyboard tele-operation, try `./start-ros-teleop.sh` to start tmux, robot.launch, and kobuki_keyop.

```
process[keyop-1]: started with pid [211]
[ INFO] [1602111857.514643436]: KeyOpCore : using linear  vel step [0.05].
[ INFO] [1602111857.525270710]: KeyOpCore : using linear  vel max  [1.5].
[ INFO] [1602111857.526344974]: KeyOpCore : using angular vel step [0.33].
[ INFO] [1602111857.526989032]: KeyOpCore : using angular vel max  [6.6].
[ WARN] [1602111857.546670310]: KeyOp: could not connect, trying again after 500ms...

[ WARN] [1602111858.055034657]: KeyOp: could not connect, trying again after 500ms...

[ INFO] [1602111858.556120822]: KeyOp: connected.
Reading from keyboard
---------------------------
Forward/back arrows : linear velocity incr/decr.
Right/left arrows : angular velocity incr/decr.
Spacebar : reset linear/angular velocities.
d : disable motors.
e : enable motors.
q : quit.
[ INFO] [1602111858.997982789]: KeyOp: linear  velocity decremented [-0.05|0]
[ INFO] [1602111863.731287076]: KeyOp: linear  velocity decremented [-0.1|0]
[ INFO] [1602111871.036408317]: KeyOp: reset linear/angular velocities.
[ INFO] [1602111872.970278643]: KeyOp: angular velocity incremented [0|0.33]

[1] 0:python*                    "/opt/ros/melodic/shar" 23:04 07-Oct-20
```

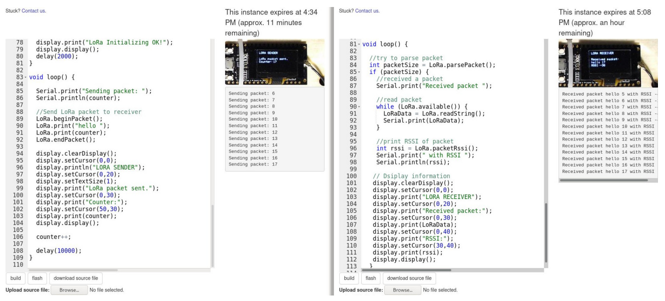
This instance expires at 4:33 PM (approx. 29 minutes remaining)

Fig. 5. Screenshot of a simple web app that leverages the framework to control the robots via the `kobuki_keyop` ROS package. The right column has video streams from both external cameras of the same workspace.

communicating with the Kobuki motor controllers via a serial protocol. We implemented a proxy for this serial protocol as illustrated in Figure 2. As with the HTTP proxy for Misty, this proxy filters access in two respects: a set of rules defining accepted commands, and a kill-switch that stops motion when a boundary of safety is reached.

The main intent was to enable low-speed multi-robot navigation experiments, so the filter prevented commands that affect unrelated behaviors (e.g., sound sequences), while some commands were bounded, in particular maximum speed. An external pose tracking system keeps Kobuki from crossing the safety barrier.

3) Results: Leveraging the framework, we built a web app with a browser-embedded terminal to support remote users without requiring them to understand SSH or VPN (Figure 5). Internally, the app uses the tunnel created from provisioning (Section III-A). The container image includes ROS already installed with relevant packages, so users can easily begin with keyboard control via the ROS package `kobuki_keyop`, as shown in a demonstration video [22].


Stuck? Contact us.

```
  display.print("LoRa Initializing OK!");
  display.display();
  delay(2000);
}

void loop() {

  Serial.print("Sending packet: ");
  Serial.println(counter);

  //Send LoRa packet to receiver
  LoRa.beginPacket();
  LoRa.print("hello ");
  LoRa.print(counter);
  LoRa.endPacket();

  display.clearDisplay();
  display.setCursor(0,0);
  display.println("LORA SENDER");
  display.setCursor(0,20);
  display.setTextSize(1);
  display.print("LoRa packet sent.");
  display.setCursor(0,30);
  display.print("Counter:");
  display.setCursor(50,30);
  display.print(counter);
  display.display();

  counter++;

  delay(10000);
}
```

This instance expires at 4:34 PM (approx. 11 minutes remaining)

```
Sending packet: 6
Sending packet: 7
Sending packet: 8
Sending packet: 9
Sending packet: 10
Sending packet: 11
Sending packet: 12
Sending packet: 13
Sending packet: 14
Sending packet: 15
Sending packet: 16
Sending packet: 17
```

Upload source file: No file selected.

Stuck? Contact us.

```
void loop() {

  //try to parse packet
  int packetSize = LoRa.parsePacket();
  if (packetSize) {
    //received a packet
    Serial.print("Received packet ");

    //read packet
    while (LoRa.available()) {
      LoRaData = LoRa.readString();
      Serial.print(LoRaData);
    }

    //print RSSI of packet
    int rssi = LoRa.packetRssi();
    Serial.print(" with RSSI ");
    Serial.println(rssi);

   // Dsiplay information
   display.clearDisplay();
   display.setCursor(0,0);
   display.print("LORA RECEIVER");
   display.setCursor(0,20);
   display.print("Received packet:");
   display.setCursor(0,30);
   display.print(LoRaData);
   display.setCursor(0,40);
   display.print("RSSI:");
   display.setCursor(30,40);
   display.print(rssi);
   display.display();
  }
```

This instance expires at 5:08 PM (approx. an hour remaining)

```
Received packet hello 5 with RSSI -
Received packet hello 6 with RSSI -
Received packet hello 7 with RSSI -
Received packet hello 8 with RSSI -
Received packet hello 9 with RSSI -
Received packet hello 10 with RSSI
Received packet hello 11 with RSSI
Received packet hello 12 with RSSI
Received packet hello 13 with RSSI
Received packet hello 14 with RSSI
Received packet hello 15 with RSSI
Received packet hello 16 with RSSI
Received packet hello 17 with RSSI
```

Upload source file: No file selected.

Fig. 6. Screenshots of a pair of Web browsers, each showing a device from the array and example programs: sender and receiver. The devices are programmed through an interface across the container-host boundary illustrated in Figure 2.

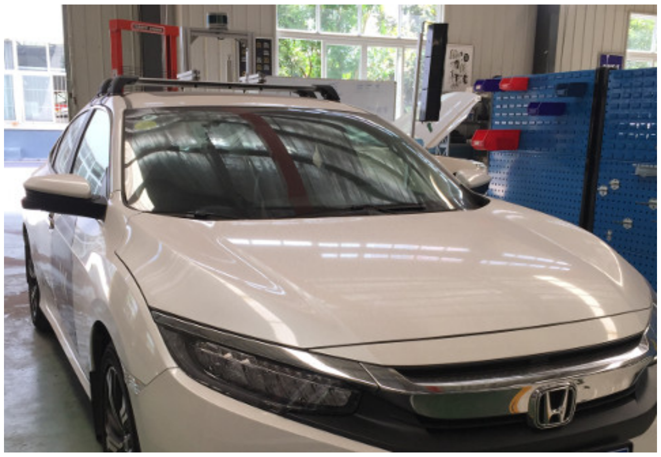


Fig. 7. Drive-by-wire car with Velodyne LiDAR and running Autoware. The hardware configuration was by PIX Moving in Guiyang, China.

C. *LoRaWAN Array*

1) Summary: LoRaWAN is a protocol for low power and long range wireless communications. While not a "robot" per se, devices with LoRaWAN are used for sensing and automation in many industries. Wireless behavior is difficult to simulate, so in practice, it is critical for experiments to be conducted on real hardware.

Thus motivated, we applied the proposed framework to share an array of TTGO LoRa32 SX1276 boards. All boards connect to a USB hub and are programmed from a single host running the hardshare clients (Section III).

Camera images are cropped such that each user can only see the board associated with their instance. Termination is easier for these devices compared to freely moving robots: it suffices to erase flash memory at the end of each user's access.

2) Results: As with the previous applications, a simple website was built on the framework. It includes a C++ code editor and terminal for streaming serial output from the devices. An example of two remote users sending and receiving LoRaWAN wireless packets is shown in Figure 6.

D. *Autoware*

1) Summary: A simple configuration for Autoware consists of a car that accepts brake, throttle, and steer software commands and that has a LiDAR (laser range finder) and color camera. The car shown in Figure 7 had such a configuration with a single Linux host for running all of the autonomy software. A supplemental power source with it was installed in the trunk.

2) Results: We installed the client software for the framework. The Velodyne LiDAR was made accessible through a UDP proxy (Section III-B.3). Instances were available through SSH and VPN. Instances of remote access have a container with Autoware already installed and point clouds streamed from the sensor as shown in Figure 7. Motion commands (steering, throttle, brake) are not supported yet. Thus, the vehicle allowed passive access via the framework, where remote users could receive sensor data and visualize through RViz.

V. Analytics

In experimental robotics, logs are critical for incremental software improvements, machine learning training, reviewing faults during field tests, etc. Logs typically are tied to the library or messaging system used, e.g., ROS bags for robots developed with ROS. Furthemore, logs are often recorded by nodes that are peers in the relevant network, consuming additional resources such as an additional TCP connection for each relevant sensor on the robot. With the presented framework, we can capture logs of messages directly by the proxy, without requiring more network connections. For each proxy, these logs only involve the proxied device. These low-level traces are useful in ways where application-level logging, e.g., ROS bags, is not. For example, a practical challenge of reading ROS bags recorded from previous software versions is to know which topics are relevant for which parts of the robot and to ensure the rest of the ROS environment is ready to play back these messages. By contrast, proxy traces correspond to messages in protocols that rarely or never change, so playback is straightforward.

Consider the Misty experimental environment from Section IV-A. The HTTP proxy can record all commands without any modification at the application level. For example, during the month of October 2020, one of the Misty workspaces had 18 instances. Of these, 13 instances had a remote user who ran some code. In other words, 5 instances were launched but then abandoned. (Recall that instances terminate automatically after an expiration time.) Among instances where users controlled the robot, 5 included both head and wheel motion commands. Interestingly, 7 instances did not include wheel motion commands, i.e., the robot did not leave the charging pad, but other actions were taken including head and arm motions or sensor data were read. Note that Misty has largely been used by a community not familiar with ROS, so other logging options are not readily available.

Figure 8 shows the number of head and base motion commands sent to Misty by anonymous users. Recall that

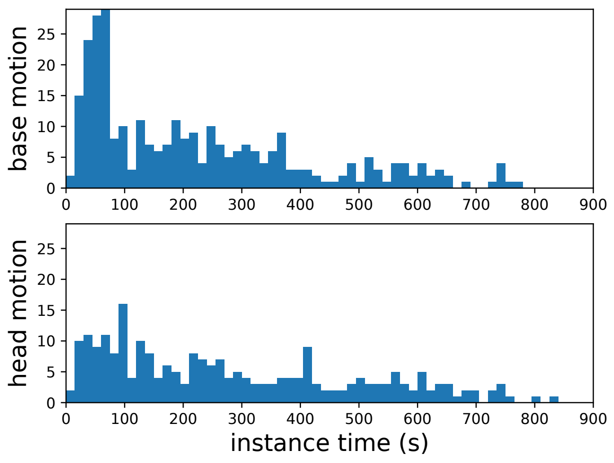


Fig. 8. Histograms of commands sent to move the robot base and head. Bin width is 15 seconds. Data are from 50 instances where remote users from around the world ran code on Misty (case study in Section IV-A). Anonymous users had instances of duration 900 seconds (15 minutes).

each instance begins at a known initial state: the charging pad. As such, the initial maximum in the histogram of base motion has an intuitive explanation: no matter the interests of the anonymous user, their code first had to move off of the charging pad. Charts like these are interesting because the data are not tied to the libraries and programming languages in which the users' algorithms were implemented. For other robots with ROS support, low-level interface data can complement application-level ROS logs. If the robot is supported by more than one library, traces from the presented framework provide a uniform basis for comparing performance.

VI. Conclusion and Future Work

To improve reproducibility of robotics research, we proposed a general framework for making robots and experimental environments remotely accessible. Our approach is to combine a provisioning system with device proxies that make access by untrusted users safe and that do not constrain users to special library or language tools. This handles the two limiting factors to reproducibility: cost of hardware and custom, undocumented infrastructure in labs. After describing the overall architecture, we presented several examples that show the versatility of the approach.

There are many directions of future work. While we built this with human users in mind, it can be used by programs to automatically access robots, run code, collect results, and repeat. Since access is low-level, not limited to a domain-specific library or toolset, future work will leverage this repeatability for continuous integration testing as well as machine learning. As more sensors, motors, and robot platforms are added, we plan to build a library of ready-to-use proxies together with the open source client.

References

[1] T. Yu, D. Quillen, Z. He, R. Julian, K. Hausman, C. Finn, and S. Levine, "Meta-world: A benchmark and evaluation for multi-task and meta reinforcement learning," in *Proceedings of the Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, L. P. Kaelbling, D. Kragic, and K. Sugiura, Eds., vol. 100. PMLR, 30 Oct–01 Nov 2020, pp. 1094–1100. [Online]. Available: http://proceedings.mlr.press/v100/yu20a.html

[2] "SubT challenge virtual competition." [Online]. Available: https://www.darpa.mil/news-events/2020-02-07

[3] "TurtleBot." [Online]. Available: https://www.turtlebot.com/

[4] M. O'Kelly, H. Zheng, D. Karthik, and R. Mangharam, "F1TENTH: An open-source evaluation environment for continuous control and reinforcement learning," in *Proceedings of Machine Learning Research*, vol. 123, 2020, pp. 77–89. [Online]. Available: https://f1tenth.org/index.html

[5] "Donkey car." [Online]. Available: https://www.donkeycar.com/

[6] "Crazyflie quadrotor by BitCraze." [Online]. Available: https://www.bitcraze.io/

[7] 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, 2015.

[8] "Autoware - the world's leading open-source software project for autonomous driving." [Online]. Available: https://github.com/autowarefoundation/autoware

[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, vol. 3, no. 3.2, 2009.

[10] K. Goldberg and R. Siegwart, Eds., *Beyond Webcams: An Introduction to Online Robots.* MIT Press, 2001. [Online]. Available: https://mitpress.mit.edu/books/beyond-webcams

[11] B. Pitzer, S. Osentoski, G. Jay, C. Crick, and O. C. Jenkins, "Pr2 remote lab: An environment for remote development and experimentation," in *2012 IEEE International Conference on Robotics and Automation*, 2012, pp. 3200–3205.

[12] D. Pickem, P. Glotfelter, L. Wang, M. Mote, A. Ames, E. Feron, and M. Egerstedt, "The Robotarium: A remotely accessible swarm robotics research testbed," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 1699–1706. [Online]. Available: https://www.robotarium.gatech.edu/

[13] L. Paull, J. Tani, H. Ahn, J. Alonso-Mora, L. Carlone, M. Cap, Y. F. Chen, C. Choi, J. Dusek, Y. Fang, D. Hoehener, S.-Y. Liu, M. Novitzky, I. F. Okuyama, J. Pazis, G. Rosman, V. Varricchio, H.-C. Wang, D. Yershov, H. Zhao, M. Benjamin, C. Carr, M. Zuber, S. Karaman, E. Frazzoli, D. Del Vecchio, D. Rus, J. How, J. Leonard, and A. Censi, "Duckietown: An open, inexpensive and flexible platform for autonomy education and research," in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, pp. 1497–1504. [Online]. Available: https://www.duckietown.org/

[14] M. Wüthrich, F. Widmaier, F. Grimminger, J. Akpo, S. Joshi, V. Agrawal, B. Hammoud, M. Khadiv, M. Bogdanovic, V. Berenz, J. Viereck, M. Naveau, L. Righetti, B. Schölkopf, and S. Bauer, "TriFinger: An open-source robot for learning dexterity," *CoRR*, vol. abs/2008.03596, 2020. [Online]. Available: https://arxiv.org/abs/2008.03596

[15] "Rapyuta Robotics." [Online]. Available: https://www.rapyuta-robotics.com/

[16] "Husarion." [Online]. Available: https://husarion.com/

[17] "Hokuyo urg-04lx-ug01 product description." [Online]. Available: https://www.hokuyo-aut.jp/search/single.php?serial=166

[18] "Dynamixel protocol." [Online]. Available: https://emanual.robotis.com/docs/en/dxl/protocol2/

[19] "pty - pseudoterminal interfaces, linux programmer's manual." [Online]. Available: https://man7.org/linux/man-pages/man7/pty.7.html

[20] "Misty REST API reference." [Online]. Available: https://docs.mistyrobotics.com/misty-ii/web-api/overview/

[21] S. M. Hadfield, S. C. Livingston, J. Woo, M. Slavin, D. Buraglio, C. Kulp, C. Richardson, and M. Hageman, "On-line steam outreach with remote robot access," Poster at The 52nd ACM Technical Symposium on Computer Science Education, March 2021.

[22] "multi_kobuki sandbox demonstration." [Online]. Available: https://vimeo.com/465989844