Arabian Journal for Science and Engineering (2019) 44:9779–9786
https://doi.org/10.1007/s13369-019-04009-z

RESEARCH ARTICLE - COMPUTER ENGINEERING AND COMPUTER SCIENCE

Developing a Portable Human–Robot Interaction (HRI) Framework for Outdoor Robots Through Selective Compartmentalization

Effective Integration of the Robot Operating System (ROS) and Android for Outdoor Robots

Sami Salama Hussen Hajjaj[1] · **Khairul Salleh Mohamed Sahari**[1]

Received: 25 April 2018 / Accepted: 25 June 2019 / Published online: 4 July 2019

Abstract
One of the challenges of outdoor robots is developing effective portable Human-Robot-Interaction (HRI) frameworks. Handheld devices offer a practical solution. By equipping these devices with robot software, they can be made to interact with the outdoor robots. Android devices are ideal as they are open source and can be integrated with robots powered by the Robot Operating System (ROS), also open source. However, due to the limits of rosjava, the mechanism that links ROS with android, and the conflicting modes of operation between ROS and android, current implementations of ROS-android offer limited robot applications that do not support advanced operations such as autonomous navigation and others. This paper implements selective compartmentalization to overcome these limitations, by combining ROS with android through a number of ROS and android bridges that would facilitate the development of advanced robot applications. Through the proposed method, authors were able to develop a portable HRI framework that allowed human operators to supervise an outdoor mobile robot while it performed an autonomous task. From their mobile devices, users were able to initialize the robot, configure its motion, and monitor its progress. Also, users were able to reprogram the robot to perform new tasks (not previously planned) through a creative use of features offered in the developed HRI framework. Also, user cognitive effort was reported to be low as evident by the positive score on the NASA-TLX scale test which was corroborated with robot performance data. This paper presents the detailed development and implementation steps.

Keywords Autonomous mobile robots · Robot Operating System (ROS) · Android · Human-Robot-Interaction (HRI) · Portable HRI frameworks · Outdoor robot applications

1 Introduction

Open-source robotics has gained a lot of interest in recent years. Leading the charge is the Robot Operating System (ROS), which offers nearly 3000 software *packages* to cover all aspects of robot operations, such as navigation, image-processing, and more [1–5]. One of the primary challenges of implementing outdoor mobile robots is developing *portable* Human-Robot-Interaction (HRI) frameworks [6,7]. An effective solution is to utilize android devices that are equipped with the needed robot software [8–12] Being open-source, the android platform is ideal for integration with ROS, allowing developers to tap into android's vast resources. Indeed, efforts to link ROS robots with android devices have been reported [8–11]. However, due to the inherent differences between ROS and android, and the limitations of rosjava, the framework linking ROS with android, these efforts have been limited. Rosjava allows only the simplest form of ROS software (Fig. 1) to be established in android. Higher-level constructs, such as the aforementioned ROS *Packages*, would need to be re-written from scratch in android, greatly limiting the scope of the developed ROS-android systems [13].

This paper presents a novel approach to developing an effective link between ROS and android, by utilizing *Selective Compartmentalization* for advanced robot systems. This paper presents a detailed development procedure, followed by an implementation on an autonomous outdoor mobile

✉ Sami Salama Hussen Hajjaj
ssalama@uniten.edu.my

[1] Centre for Advanced Mechatronics and Robotics, Universiti Tenaga Nasional, 43000 Kajang, Selangor, Malaysia

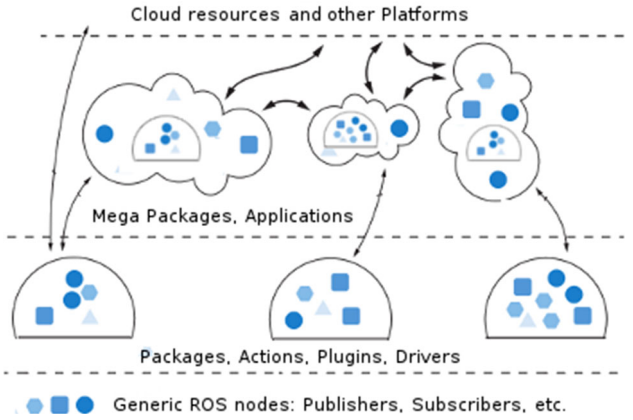

Fig. 1 The ROS Software Architecture, edited from [14]

robot. Finally, the paper evaluates the performance of the developed robot, the developed portable HRI framework, as well as the proposed approach used to implement it.

1.1 The Challenges of ROS-Android Systems

ROS and android have different modes of operations; in ROS, software components (nodes, packages, actions, etc.) launch on application start and publish/receive data continuously and simultaneously. In android, on the other hand, most modules start only when prompted by user input (screen touch, pressing a button, etc.). These conflicting modes of operation between ROS and android may prevent proper integration as ROS nodes capturing user input might start before or even without user input, causing system crashes and app failures. Some researchers developed a low-cost telepresence robot by embedding an android tablet onto an ROS-ready moving base [8]. The android device would capture user hand gestures and movements, effectively establishing a natural interface between the robot and the operator. But by embedding the android device onto the robot, its mobility was eliminated. People had to be in close proximity to interact with it which limited its range and usefulness.

Recognizing this limitation, other researchers added a second android device to act as a remote controller. Users could Teleop (remote control) the robot by re-orienting the second android device [11]. Other researchers used another ROS package, rosbridge, to extend this interaction through the Internet for outdoor robot applications [9,10]. But although they were able to Teleop the robots and receive image feedback, the interaction was very network dependent. As reported by the authors, network delays would throw user commands and robot feedback out of sync, adversely affecting its motion [9,10]. But even if we ignore networking issues, Teleop does not cater for advanced or autonomous robot applications because it is a direct motion command that the user must enter to move the robot. Without Teleop,

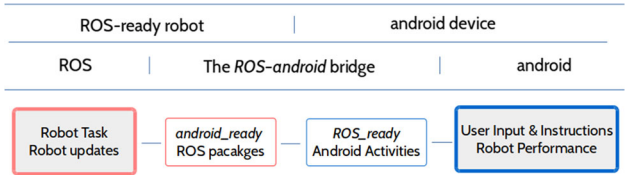

Fig. 2 Linking ROS with android through Selective Compartmentalization

the robot remains idle or in its last state. Autonomous robots, however, require a degree of autonomy to perform their tasks (patrol, agriculture, etc.) These works were limited to Teleop because implementing any higher-level applications would require re-writing ROS software in android. Aside from lost time and effort, developers need to manage the differences between C++ and Java (and/or Native C). The challenge is further complicated if the required ROS package is built on a hierarchy of other ROS packages, as is generally the case.

1.2 Selective Compartmentalization

An effective ROS–android pairing can be developed by integrating only selected aspects of ROS within android (such as ROS system requirements) and retaining core software in its habitat as shown in Fig. 2. Through this approach, ROS applications for robot software are retained on the robot's computer (or embedded system), while android activities for human interaction are retained on the android device used by the users. The *ROS-android* linking is then achieved by developing a number of ROS and android *bridges* that process and pass the required data (user input, robot data) back and forth between the ROS and android *habitats*, as shown 2. However, default configurations of rosjava do not readily facilitate this approach. Developed software would crash if default rosjava were used. So, the authors made a number of additions and original contributions to correct this and facilitate the proposed approach, effectively contributing to the growth of ROS, android, and rosjava.

The next section discusses the steps needed to implement Selective Compartmentalization to link ROS to android.

2 Methodology

In order to meet the objectives of this work and demonstrate its effectiveness, the following conditions must be met:

1. The robot implemented must be ROS-ready
2. One or more android devices need to be used
3. Networking set-up already established
4. *rosjava* already installed
5. Robot task and human involvement properly defined

One can purchase an ROS-ready robot off the shelf as ROS is supported by most robot manufacturers today. For custom-built or in-house robots, designers need to develop the needed ROS hardware abstraction software, a number of ROS packages (URDF model, tf tree, etc.) to make the robot ROS-ready; detailed instructions can be found here [15]. Wireless connectivity between the robot and the android devices must be established, forming the ROS network; this can be achieved through these instructions [16,17]. *rosjava* needs to be installed onto the developers' IDE (Integrated Development Environment, such as Android Studio), as per these instructions [18]. rosjava (and its android extension) offers the RosActivity, an android Activity that meets ROS's system requirements; it registers the android device with the ROS robot, it establishes and maintains a continuous connection with it (assuming networking was already established), and it provides the platform to develop and launch the needed ROS nodes from android. By clearly defining robot task and the level of human involvement in it, the required ROS packages and android GUI elements needed to fulfil the task can be identified, along with their meta-data (ROS topics, message types, parameters, variable type definitions, etc.).

2.1 Experimental Set-up and Robot Task

In another related project, the authors were tasked with developing an outdoor, autonomous, mobile Patrol robot; a mobile robot that autonomously patrols an outdoor field to prevent intrusions. A human operator is needed to initialize the robot, elect its task (patrol, stop, resume, goHome), define its patrol locations, configure its motion (random, loop, intermittent, etc.), and monitor its feedback (task progress, robot state). Also, the operator must be on the field, and must use own android device to interact with the robot.

As shown in Fig. 3, an empty parking lot was used to simulate the outdoor field, traffic cones placed randomly for each run as the intruders. The robot used in this experiment is a customized Turtlebot2 robot, a modified ROS-ready robot, modified to operate in the outdoors [15].

The proposed method of this paper, selective compartmentalization, is implemented to develop the needed ROS-android HRI framework as discussed next. All developed software is available on the GitHub page of this work [19].

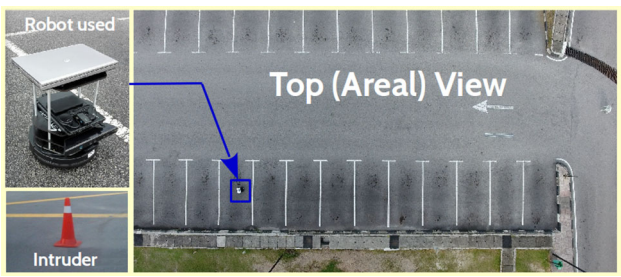

Fig. 3 Robot and field used in this work

2.2 Part I: Android-to-ROS Interactions

The objective of this interaction is to capture user input in android, transfer it to the robot in ROS, and then use it to trigger the robot into action as per user input.

In android: Based on task requirements (as defined above), a number of android GUI elements were selected to capture the required user input, as shown in Table 1.

Next, the Events (android protocols) of these elements are configured to capture user input and pass it to rosjava elements. For example, when the *goHome* button is clicked (or when the *onClick* event is triggered), the corresponding variable is assigned a pre-defined value. Similar Events are also defined when other input methods are triggered (voice command or when a point on the map is clicked).

In rosjava: based on the input captured, a corresponding rosjava publisher is called to publish that input to the robot. For example, when a user clicks on a command button, the `robotCommandPublisher` captures the corresponding commandID and publishes it to the robot (through the already established ROS network, as discussed above). Similarly, other input types are shown in Fig. 4.

In ROS: User input is divided into two types: initialization and task selection/configuration. Initialization is needed to start the robot and define its patrol locations; this input is written into a number of config files for later use. Task selection/configuration input is saved as runtime variables and used to trigger the needed robot task programs and algorithms, as shown in Fig. 4 As shown in the figure,

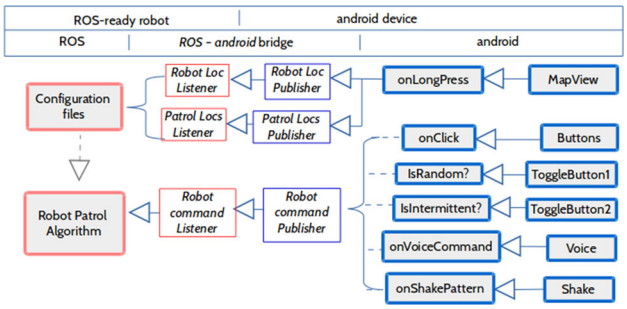

Fig. 4 android-to-ROS interactions

Table 1 Required user input and implemented android GUIs

Required user input	Android GUI
Initialize (localize) robot	MapView (with GoogleMaps)
Define Patrol locations	MapView (with GoogleMaps)
Configure robot motion	Button, Switch, CheckBox
Select robot task	Button, VoiceCommand, Shake

the `robotCommandLister` captures the commandID sent from android, and uses it to call the corresponding ROS program to perform that task, feeding it any parameters passed along as well. For example, doPatrol task corresponds to commandID = 4, and so on. The ROS programs that facilitate robot tasks (patrol, pause, resume, stop, goHome) are also developed by the author, but they are not discussed here since the focus of this paper is on the ROS–android interaction and not on ROS development. However, the source code of these programs is also available on the GitHub page of this paper [19]

2.3 Part II: ROS-to-Android Interactions

The objective of this interaction is to capture robot state in ROS, and display it in android, Fig. 5.

In ROS: To achieve this, a number of ROS publishers were developed to publish robot status, as shown in Table 2. Collectively, these updates provide a comprehensive picture of what the robot is doing. Status updates include job status, system updates, and task completion status. Other updates are self-explanatory. Part of being an ROS-ready robot, the `robot_state_publisher` publishes a comprehensive set of robot data: its current speed, sensor data, odometry, power level, etc. [20]. For this work, only speed and battery level were used.

In rosjava: The rosjava listeners are next developed to capture the information published by ROS, and convert it into data understood by android. For example, the `rsListener` captures robot status (published by the `statusPublisher` from ROS) and stores its content (statusID, statusText, etc.) as android objects. Other listeners are developed to capture other robot updates, Fig. 5.

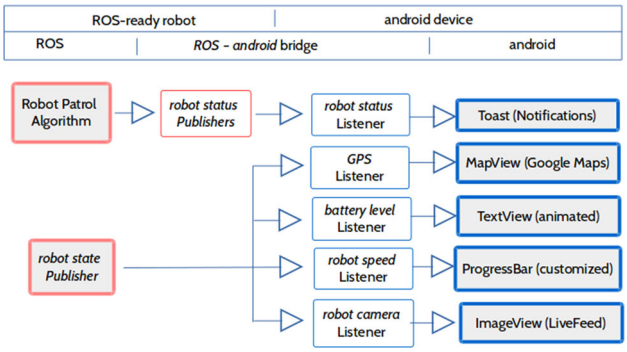

Fig. 5 ROS-to-android interactions

Table 2 Required robot output and used ROS publishers

Required robot output	ROS Publisher
status updates	`statusPublisher`
robot current Location	`GPSPublisher`
robot speed	`robot_state_publisher`
robot battery level	`robot_state_publisher`
robot image feedback	`imagePublisher`

2.4 Part III: Concurrency of Interactions

Unfortunately, the default settings of RosActivity would cause every node within the ROS–android app (all android and rosjava nodes) to start onAppStart, with all given the same default node-name, and continuously run till it is stopped by the user. This meant all notifications, all robot commands, all clicks and checks, and all robot status updates were all launched at the same time. Obviously, this resulted in errors, overlapping and conflicting processes, killed processes, over-utilized network bandwidth, and ultimately system crashes and App shutdowns.

To resolve this, default rosjava settings were overwritten to achieve the following: initialization and task selection publishers were reconfigured to run only when prompted by the user; status publishers were to publish only when there is new data (publishOnce), and finally, all nodes were launched with a uniquely identifiable name. To achieve these changes, the *init* method within RosActivity was modified, Fig. 6.

As seen in the figure, the `NodeMainExecutor` and the `NodeConfiguration` objects are set global, making them available to all other methods, such as the newly developed startNode method. Then, the `startNode` method was placed where it was needed; if a node is needed to start onAppStart, it is then called from within the init method

```
protected void init(NodeMainExecutor nodeMainExecutor) {
    this.nodeMainExecutor = nodeMainExecutor;
    nodeConfiguration =  NodeConfiguration.newPublic(
                    getRosHostname(), getMasterUri());

    startNode(rjsListener, "rjsListener_android");
    startNode(gpsListener, "gpsListener_android");
    startNode(speedListener, "speedListener_android");
    startNode(batteryListener, "batteryListener_android");
    startNode(liveFeedListener, "LiveFeedListener_android");}

protected void startNode(NodeMain node, String nodeName) {
    nodeConfiguration.setNodeName(nodeName);
    nodeMainExecutor.execute(node, nodeConfiguration);}
```

Fig. 6 Establishing Concurrency for ROS–Android

```
protected void loop() throws InterruptedException {
    if (publishOnce) {
       while (publisher.getNumberOfSubscribers()==0)
           Thread.sleep(100);
       if (i > 0) throw new InterruptedException();}
    std_msgs.Header msg = publisher.newMessage();
    msg.setStamp(command_time);
    msg.setSeq((byte) command_id);
    msg.setFrameId(command_name);
```

Fig. 7 The *robotCommandPublisher*

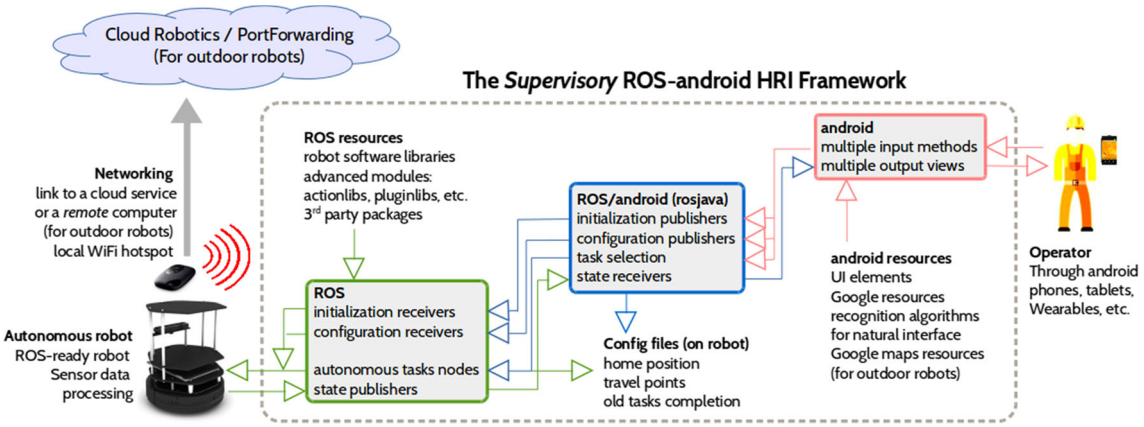

Fig. 8 The developed ROS–android HRI framework

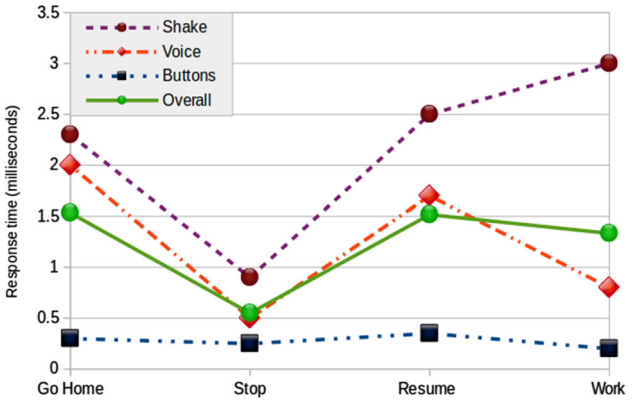

Fig. 9 Response time of *online* input methods

(default behaviour), but if it was needed only on user input, then it was called from that input's Event method, as is already discussed and shown in Fig. 4. Figure 6 shows how each node was called with a unique name to prevent naming conflicts. In ROS, publishers publish their messages continuously regardless of the status of the message (changed or not). While this behaviour is needed to facilitate robot performance, not all publishers need to behave this way. User commands, robot status updates, and notifications need to be published only when a new data arrives; republishing the same message over and over could over-burden the network, which in turn could affect robot performance. The solution was to implement the publishOnce approach, where the ROS Publisher is modified to publish just one message and then stop, Fig. 7.

The publisher first waits till it is connected with at least one listener, then it publishes the message and then shuts down. The figure also shows other modifications, such as setting the commandID, commandName, and commandTime, which are passed along to the robot. `Command_time` is used for logging and later analysis.

With this, the set-up of the ROS–android HRI framework, through selective compartmentalization, is complete and ready for testing, which is discussed next.

3 Findings

Figure 8 shows all the android-to-ROS and ROS-to-android interactions developed above. Along with the support resources from android and ROS, they form the complete ROS–android HRI framework, as developed through selective compartmentalization.

3.1 The *Android-to-ROS* Interactions

To test the effectiveness of input methods used for Task Selection and motion configuration, each was used to trigger the robot ten times. Then, using data from ROS logs (Fig. 7), the chart shown in Fig. 9 was produced.

Google recognition algorithms were implemented for *Voice* and *Shake*, but due to noise and other factors, these algorithms caused response delays and were not always successful. As seen from the chart, *Buttons* scored 100%, *Voice* 77.5%, and *Shake* 62.5%. The *Stop* command scored a 100% success rate by all methods; this happened because its voice command, the word stop was easily pronounced, and its shake pattern, a rapid left-to-right-to-left motion, were easily performed by the test subjects.

To test the effectiveness of the interactive MapView, it was put to the test and the robot was filmed in action, using a flaying camera drone flown above the field; a screen grab of that footage is shown in Fig. 10.

The figure is a composite showing the mapView from android, the aerial view of the actual robot in the field, and ROS's visualization of the robot as captured remotely in the home computer [17]. As the user selects the target locations in android, the robot captures these locations, and then it

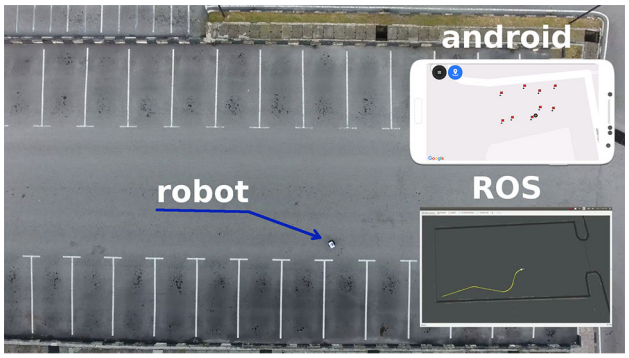

Fig. 10 Evaluating the interactive *MapView*

Tour guide robot *Delivery* robot

Fig. 11 Supporting multiple robot applications through the configuration/selection classes

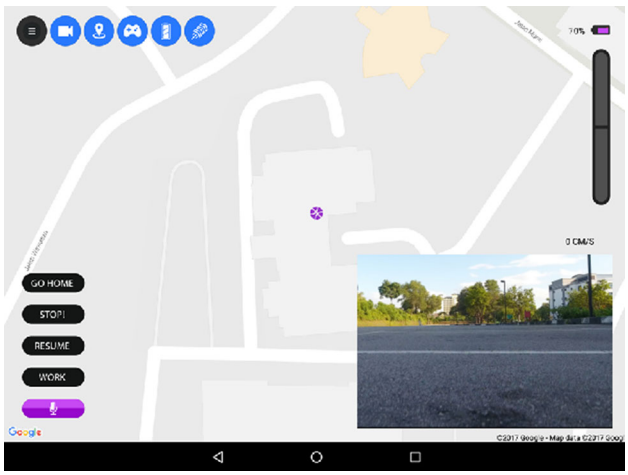

Fig. 12 The developed android interface

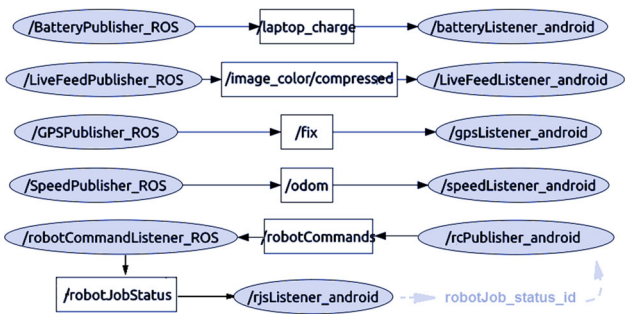

Fig. 13 Establishing ROS–android Concurrency

begins to roam through these locations as per its motion configuration, with the visualization tools following the robot and showing its location in real time [19]. As an alternative, Google's EarthView could also be used, as it provides a different look and feel of the required area.

During testing, it was found that, through a creative use of the input methods, the same robot could be made to perform other tasks (other than patrol). This unplanned feature was further tested and implemented indoors, as shown in Fig. 11. By defining just one target location (aside from home) and selecting intermittent motion, the robot became a delivery robot to that location. Also, by selecting multiple locations with the default motion (Loop), the robot became a tour guide robot. This demonstrated that the robot performance could be reconfigured from android, without re-programming or hard-coding the robot software.

3.2 The *ROS-to-Android* Interactions

The output views were successfully able to capture the robot output, as shown in Figs. 10 and 12. The developed interface is highly customizable; the user could hide, show, and group the views/input modules as well, this can be seen by comparing Figs. 10 and 12. Furthermore, the framework supported different android devices with different screen sizes; Fig. 12 shows a tablet layout, while Fig. 10 (the android component) shows a phone layout.

3.3 Concurrency of the ROS–Android Interactions

Figure 13 shows the outcome of the modifications on rosjava and RosActivity, as discussed above.

This is the ROS *Node Graph* of the framework (partial) as produced by ROS's `rqt_graph` package. The figure shows the simultaneous processes established linking between ROS and android and forming the developed framework. As seen from the figure, all processes are parallel and independent of each other, indicating concurrency.

3.4 Evaluating User Cognitive Workload

The success of an HRI framework depends heavily on its cognitive workload. An interface might offer many features, but they would be meaningless if the users required a lot of thinking or were confused when using these features. In this work, the NASA *Task Loading index (TLX) test* was administered to gauge user cognitive workload.

Each participant was required to guide the robot to patrol the field and find an intruder, three separate times. Ten final year students participated in this experiment, and none of

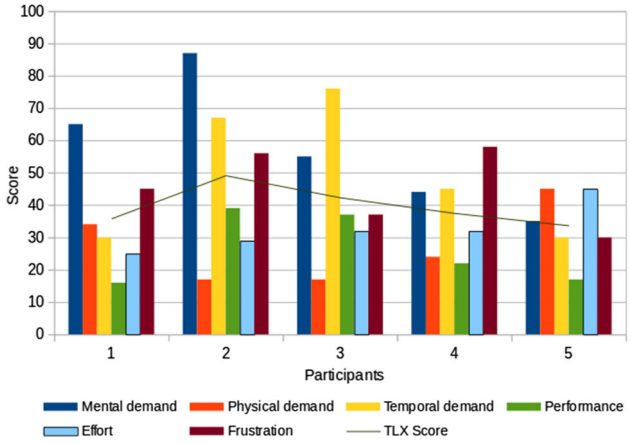

Fig. 14 Overall results of the NASA TLX test

them had prior experience with the framework. Test results showed that the developed framework scored 34.2 out of 100 on the combined TLX test scale, indicating a low-to-moderate workload requirement. In this test, *raw* TLX results were used (each TLX scale was treated equally). On its own, TLX results may be subjective, but when corroborated with the above findings, a clearer picture is established.

4 Discussion

As seen from the above, the ROS–android framework developed in this work offers a number of appealing features that cater for advanced and autonomous robots.

The framework offers multiple input methods, such as Buttons, Voice, Shake, and the interactive MapViews. These methods collectively allowed users to enter higher-level robot commands such as robot initialization, task selection, and configuration. This is far more sophisticated than any of the current implementations of ROS–android HRI frameworks, which usually offer Teleop motion commands.

Teleop robots are not truly autonomous as they require continuous and direct motion commands from user, which place a heavy burden on the wireless network and its bandwidth. That is why, the current implementations are mostly indoor or very limited outdoor systems (as reported by their authors [9,10]). On the other hand, the input commands of the developed framework do not require continuous input because they are mostly user notifications and user updates. This greatly reduces networking burden and bandwidth utilization and allows the robot to focus on its task, making it truly autonomous.

Furthermore, through a creative use of input commands (initialization, configuration, and selection), the framework can configure the same ROS robot to perform multiple jobs without re-programming or hard-coding. In this work, three applications were demonstrated using the same robot: delivery, tour guide, and patrol (outdoors). Furthermore, the framework could also support other robot jobs as well, such as Search & Rescue, agriculture operations, and a variety of social robots tasks. Current implementations do not offer the same flexibility and versatility.

Also, the platform offers a number of android views that provide a comprehensive representation of the condition of the whole system; from the state of the robot, to the task being performed and its completion status, to the current configuration of the platform itself. In contrast, current implementations offer only the robot's image feed.

Indirectly, the developed framework resulted in improved robot energy efficiency and battery life. Through shorter robot response time, Fig. 9, and the lowered cognitive effort, Fig. 14, the robot has become more time-efficient and therefore better utilizes its battery power.

Collectively, the above features offer an effective *portable* HRI framework that allows for the development of advanced robot system, that would require minimal software re-creation, would even result in improved robot efficiency and battery life, and one that requires only the use of android devices.

5 Conclusions

The paper began by reviewing current implementations of ROS–android-based HRI frameworks. After the review, it was found that current implementations focused mostly on Teleop or remote control of the robot, using ROS-android. While impressive, Teleop does not cater for autonomous robots, and is very network dependent. This greatly limits the scope and potential of these current implementations.

On the other hand, this work presents a novel approach to linking ROS with android, by developing a number of ROS and android bridges that process and pass the required data (user input and robot data) back and forth between the ROS robot and the android device, as shown in Fig. 2.

To test the validity of the developed framework, it was tested on three robot applications, an indoor delivery robot, an indoor tour guide robot, and an outdoor patrol robot. The framework supported these three applications without any further re-programming or hard-coding (aside from the remote-networking set-up for the outdoor application [17]).

The developed framework offered its users the avenue to assume a supervisory role over the autonomous robots, through the following interaction features:

– The framework supports multiple input methods, allowing users to input higher-level commands to initiate, configure, and select robot tasks.

– These input commands are notification or offline in nature, which frees the robots from user interaction while they perform their autonomous tasks.
– This also reduces networking burden and bandwidth utilization, as demonstrated by the response time data shown in the figures above.
– This has a number of output views that display a comprehensive representation of the state of the whole system: robot state, job status and completion, and others.
– The platform also offers a number of customization tools for a greater user experience; user can hide/show components, or use phones or tablets of different sizes.

With this, this work achieved its objectives, by providing a ROS–android HRI framework that truly realizes the full potential of ROS, android, and other open-source technologies, hopefully opening the door for more research in this area and facilitating more practical robot systems.

6 Future Works

There is plenty of room for further development; researchers could develop specialized input/output classes that focus on specific ROS sensors, to be developed separately and added when that sensor is used. Android support could be extended to wearables, which would allow humans to use their android watches or glasses to interact with the ROS robots, opening a variety of new and exciting implementation.

Acknowledgements The authors would like to thank the Innovative & Research Management Centre (iRMC), and College of Engineering, Universiti Tenaga Nasional (UNITEN), for their continued support of this work. Also, the authors would like to thank Mr. Ismail I. M. Al.Mahdy for his contributions in the development of some of the android GUIs and their components.

References

1. Boren, J.; Cousins, S.: New exponential growth of ROS [ROS topics]. IEEE Robot. Autom. Mag. **18**(1), 19 (2011). https://doi.org/10.1109/MRA.2010.940147
2. O'Kane, J.M.: A gentle introduction to ROS (Independently published, 2013). http://www.cse.sc.edu/~jokane/agitr/
3. G, Z.; et al.: ALLIANCE-ROS: a software framework on ROS for fault-tolerant and cooperative mobile robots. Chin. J. Electron. **27**(3), 467 (2018). https://doi.org/10.1049/cje.2018.03.001
4. Mishra, R.; Javed, A.: ROS based service robot platform. In: 2018 4th International Conference on Control, Automation and Robotics (ICCAR) (2018), pp. 55–59. https://doi.org/10.1109/ICCAR.2018.8384644
5. B.M. et al.: A two-layer network Orchestrator offering trustworthy connectivity to a ROS-industrial application. In: 2017 19th International Conference on Transparent Optical Networks (ICTON) (2017), pp. 1–4. https://doi.org/10.1109/ICTON.2017.8025148
6. Albani, D. et al.: Monitoring and mapping with robot swarms for agricultural applications. In: 2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS) (2017), pp. 1–6. https://doi.org/10.1109/AVSS.2017.8078478
7. Choubisa, T. et al.: Comparing chirplet-based classification with alternate feature-extraction approaches for outdoor intrusion detection using a pir sensor platform. In: 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI) (2017), pp. 371–379. https://doi.org/10.1109/ICACCI.2017.8125869
8. Do, H.M. et al.,An open platform telepresence robot with natural human interface. In: The 2013 IEEE 3rd Annual International Conference on Cyber Technology in Automation, Control and Intelligent Systems (CYBER) (2013), pp. 81–86. https://doi.org/10.1109/CYBER.2013.6705424
9. Speers, A. et al.: Lightweight tablet devices for command and control of ROS-enabled robots. In: The 2013 16th International Conference on Advanced Robotics (ICAR) (2013), pp. 1–6. https://doi.org/10.1109/ICAR.2013.6766481
10. Codd-Downey, R.; Jenkin, M.: Dynamic mobile interfaces for command and control of ROS-enabled robots. In: The 2015 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO), vol. 02 (2015), vol. 02, pp. 66–73
11. de A. Barbosa, J.P. et al.: ROS, Android and cloud robotics: How to make a powerful low cost robot. In: The 2015 International Conference on Advanced Robotics (ICAR) (2015), pp. 158–163
12. B.G. et al.: Android application for simultaneously control of multiple land robots which have different drive strategy. In: 2017 International Conference on Computer Science and Engineering (UBMK) (2017), pp. 724–728. https://doi.org/10.1109/UBMK.2017.8093513
13. Cerruti, J.: Using native ROS packages on android. http://wiki.ros.org/android_ndk (2016)
14. Mohanarajah, D.H.G.; D'Andrea, R.; Waibel, M.: Rapyuta: a cloud robotics platform. IEEE Trans. Autom. Sci. Eng. **12**(2), 481 (2015). https://doi.org/10.1109/TASE.2014.2329556
15. Hajjaj, S.S.H.; Sahari, K.S.M.: Bringing ROS to agriculture automation: hardware abstraction of agriculture machinery. Int. J. Appl. Eng. Res. **12**(3), 311 (2017)
16. The ROS Wiki. Running ROS across multiple machines. http://wiki.ros.org/ROS/Tutorials/MultipleMachines (2016).
17. Hajjaj, S.S.H.; Sahari, K.S.M.: Establishing Remote ROS Networks via Port Forwarding: a Detailed Tutorial. Int. J. Adv. Robot. Syst. **14**(3), 1 (2017). https://doi.org/10.1177/1729881417703355
18. Stonier, D.: The android package in ROS [indigo], ros wiki. http://wiki.ros.org/android (2015).
19. Hajjaj, S.S.H.: The GitHub Repository of this work. https://github.com/sami-s-hajjaj (2018)
20. Satellite imagery: Astrium, DigitalGlobe. The android world–hardware. https://www.android.com/ (2014)