Received: 12 October 2023 | Revised: 8 January 2024 | Accepted: 4 February 2024

DOI: 10.1049/csy2.12112

IET Cyber-Systems and Robotics

ORIGINAL RESEARCH

ROSIC: Enhancing secure and accessible robot control through open-source instant messaging platforms

Rasoul Sadeghian | Shahrooz Shahin | Sina Sareh

RCA Robotics Laboratory, Royal College of Art, London, UK

Correspondence
Sina Sareh.
Email: sina.sareh@rca.ac.uk

Abstract

Ensuring secure communication and seamless accessibility remains a primary challenge in controlling robots remotely. The authors propose a novel approach that leverages open-source instant messaging platforms to overcome the complexities and reduce costs associated with implementing a secure and user-centred communication system for remote robot control named Robot Control System using Instant Communication (ROSIC). By leveraging features, such as real-time messaging, group chats, end-to-end encryption and cross-platform support inherent in the majority of instant messenger platforms, we have developed middleware that establishes a secure and efficient communication system over the Internet. By using instant messaging as the communication interface between users and robots, ROSIC caters to non-technical users, making it easier for them to control robots. The architecture of ROSIC enables various scenarios for robot control, including one user controlling multiple robots, multiple users controlling one robot, multiple robots controlled by multiple users, and one user controlling one robot. Furthermore, ROSIC facilitates the interaction of multiple robots, enabling them to interoperate and function collaboratively as a swarm system by providing a unified communication platform that allows for seamless exchange of data and commands. Telegram was specifically chosen as the instant messaging platform by the authors due to its open-source nature, robust encryption, compatibility across multiple platforms and interactive communication capabilities through channels and groups. Notably, the ROSIC is designed to communicate effectively with robot operating system (ROS)-based robots to enhance our ability to control them remotely.

KEYWORDS

end-to-end encryption, interactive communication, remote robot control, secure web-based communication

1 | INTRODUCTION

The emergence of web-based teleoperated robots has revolutionised a variety of sectors, including manufacturing, healthcare and environmental exploration. However, these technological advancements have created a set of challenges, particularly in terms of security, user-centredness and accessibility.

A primary concern with robots controlled via web-based platforms is their dependency on communication networks, which exposes them to significant security risks [1–6], including unauthorised access, data breaches and hacking [7]. Another critical aspect is ensuring a user-centred interface for seamless interactions between users and remote-controlled robots. The interface should be intuitive, easy to learn and require minimal training to cater to users with diverse technical backgrounds. Striking the right balance between functionality and simplicity is vital to enhancing the efficiency of robot operation. Moreover, in certain applications, multiple robots must collaborate and communicate with each other to achieve a common goal. Coordinating their actions and ensuring seamless data exchange have become critical for successful group operations [8–10].

From web-based control for robotic arms to predictive simulations and cloud-based frameworks for seamless interactions with robots, researchers have continuously strived to improve the user experience and enhance teleoperation. Within this context, in 1995, Taylor and Trevelyan pioneered web-based control for a robotic arm to manipulate coloured blocks [11]. Building upon this work, Burgard and Schulz developed a predictive simulation for visualising and managing teleoperation delays in mobile robots [12]. Another significant contribution came from Goldberg et al., who created a web-based control system that allowed users to interact with a robot for gardening purposes via the web [13]. In response to the demand for enhanced scalability, flexibility and computational power, cloud-based robots control have evolved, surpassing traditional web-based methods by utilising dedicated cloud server infrastructures. In the realm of service-oriented computing, Yinong et al. presented a cloud-based framework for seamless interactions with robots [14]. To address the challenge of interacting with ROS (Robot Operating System) for novice users, Osentoski et al. proposed rosbridge and rosjs, enabling JavaScript-based interactions with ROS topics and services [15]. Meanwhile, Alexander et al. curated a collection of open-source modules that leverage modern web and network technologies to interface with ROS [16]. Additionally, Kubaa et al. developed an asynchronous cloud-based communication protocol to facilitate seamless communication between robots and users [17]. However, despite the advantages of these internet-based (including web and cloud-based) telecommunication systems, they do have certain weaknesses. Some systems require access to unsecure public IP addresses for WebSocket clients, a communication protocol that enables two-way interactive communication between a client and server over a single, long-lived connection [18]. Others may rely on cloud-based services such as ROSLink [19], which can potentially expose systems to security risks, vendor lock-in and reduced flexibility. Controlling robots through the Internet involves various methods, each offering unique advantages and limitations. ROSBridge, a protocol used for ROS production, facilitates communication between ROS and external systems. It provides native integration with ROS, efficient WebSocket communication and support for various message types [20]. However, non-ROS applications might be less straightforward to use, and security vulnerabilities can arise if not properly configured. Message Queuing Telemetry Transport, a lightweight messaging protocol popular in Internet of Things applications, allows efficient remote control of robots. Its publish/subscribe and queuing paradigms are beneficial, but it requires additional middleware for ROS integration and may exhibit higher message latency than other methods [21]. Representational State Transfer Application Programming Interfaces Application Programming Interfaces (APIs) offer simplicity and broad support across programming languages, making them accessible even to non-ROS applications. However, their statelessness limits real-time capabilities, and extra effort is needed for security implementation [22]. While WebSockets offer full-duplex communication for real-time data exchange and are ideal for interactive applications, they require support on both the client and server sides and additional security measures may be necessary [23]. Virtual Private Networks establish secure connections between the controlling device and the robot, enabling seamless access to the robot's local network. This ensures a robust integration with existing ROS infrastructure but network latency and maintenance complexities could impact real-time applications [24]. Secure Shell (SSH) provides secure remote access to the robot's terminal, allowing command execution and remote management. SSH requires minimal setup and is widely available, but it lacks continuous streaming and real-time control capabilities [25].

Motivated by significant challenges of web-based robot control, including user-centeredness, scalability of access and security, the Robot Control System using Instant Communication (ROSIC) system is proposed to (1) enhance web-based robot control by utilising the advanced features of open-source instant messaging platforms and (2) facilitate the use of robots, making them accessible irrespective of user technical expertise. In this context, the main contributions of this paper include the design and implementation of ROSIC, a robust middleware that harnesses the features of open-source instant messaging platforms to enable secure, efficient and accessible remote control of robots to significantly reduce the relevant development time and cost. Through an innovative architectural framework, ROSIC enhances the telecommunication process, enabling users of varying technical backgrounds to remotely operate robots without requiring in-depth knowledge of robotics. This paper further demonstrates the viability of ROSIC through comprehensive scenarios, showcasing its potential to enhance the field of remote robot control and telecommunication. Figure 1 illustrates the role of the ROSIC middleware within the robots' remote control system.

The remainder of this paper is organised as follows. In Section 2, we delve into the details of the ROSIC middleware, outlining its architecture, key components and integration with open-source instant messaging platforms. In Section 3, we explain how ROSIC was integrated with Telegram, a widely used open-source instant messaging application. We show how ROSIC's abilities were merged with Telegram's features. We also describe our approach for testing the ROSIC's performance and efficiency, including the experimental setup and results of the experiments. Expanding on the obtained results, Section 4 examines the implications of ROSIC's design and its potential to enhance remote robot control and telecommunication. The paper concludes in Section 5 by summarising our contributions and insights and discussing future research directions.

2 | ROSIC CONCEPT

In the domain of internet-based teleoperated robotics, achieving seamless remote control coupled with enhancing the level of data accessibility are two primary challenges. The first involves improving the efficacy and reliability of remote control capabilities, particularly over the Internet, which is crucial

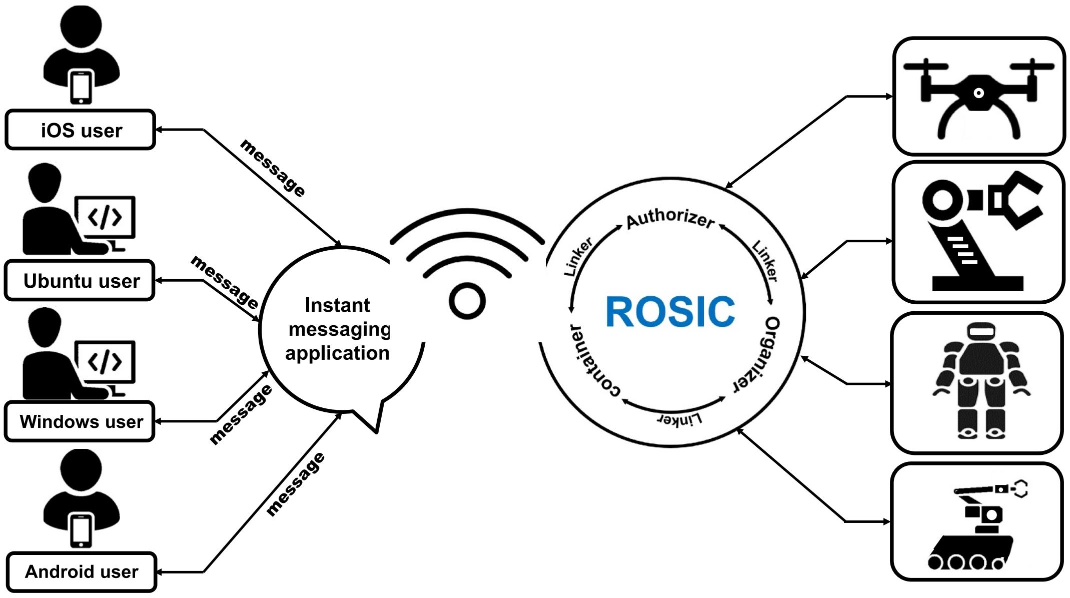


FIGURE 1 This diagram illustrates the architecture of the ROSIC, a system designed to facilitate remote control of robots via open-source instant messaging platforms. Central to the system is the ROSIC core, which is surrounded by four key components: the *linker*, which connects all system elements; the *authoriser*, which manages user authentication and command authorisation; the *organiser*, which orchestrates the command flow and user access and the *container*, which securely stores data and encapsulates commands. Notably, the architecture's design promotes data security [4], user-centeredness [9], accessibility [8] and interoperability [26], allowing seamless integration and communication between diverse robot models and platforms. In this system, users of varying technical backgrounds interact with ROSIC, sending commands that are processed and relayed to various types of robots. ROSIC, Robot Control System using Instant Communication.

for facilitating real-time interventions and operational adjustments. The second challenge pertains to enhancing data accessibility. This is particularly important in systems designed to be managed by multiple operators or utilised by numerous users, where seamless and secure access to relevant data is fundamental for efficient operation. This requirement significantly impedes their widespread adoption in this field. Moreover, a predominant advantage in the sphere of mobile robotics is the ability to control these devices through any operating system, further broadening their accessibility and usability.

The fundamental premise of ROSIC is to leverage the inherent features of open source instant messenger platforms, such as real-time communication, group chats, end-to-end encryption and cross-platform support. By utilising these features, ROSIC aims to develop a secure, swift, user-centred and cost-effective telecommunication system tailored for robotics. This approach not only streamlines the communication process but also enhances the accessibility and security of interactions with robotic entities.

2.1 | System architecture

The core architecture of ROSIC comprises four components: *authoriser*, *organiser*, *container* and *linker*. These components can be seamlessly integrated into any open-source instant messaging platform, leveraging the platform's features to craft a secure telecommunication system for robots. Algorithm 1 outlines the implementation process of the ROSIC middleware, providing a comprehensive overview of its structure and functionality.

- **Authoriser:** In ROSIC, the *authoriser* serves as a pivotal mechanism, determining access rights and permissions based on specific criteria. Within the framework of instant messaging platforms, users must first authenticate themselves by signing in with a username and password before they can send or receive messages. For developers interfacing with the platform's API, an authorisation token or API key becomes essential for accessing specific features. This token, which is a distinct combination of letters and numbers, is crucial for developers to include in their API requests. It serves not only to identify the developer or application but also to verify that they have the necessary permissions for the desired action. For those aiming to access ROSIC through a messaging platform, it is imperative to input their API token or key into the authorisation file (*container*). This file maintains a record of all authorised users or operators permitted to utilise ROSIC. Upon successful registration, ROSIC issues a unique authorisation key to the user or operator. Every interaction with ROSIC, be it reading or writing, necessitates the inclusion of this key. ROSIC then cross-references this token with its database, assessing its authenticity and the associated permissions. If the token aligns with the permissions and is authenticated, the user or operator request is executed. Otherwise, it declines.
- **Organiser:** In ROSIC, the *organiser* plays a key role, primarily in implementing detailed permissions termed as

Algorithm 1 ROSIC general structure based on its main components, *authoriser*, *organiser*, *container*, and *linker*

```
   Data: userCredentials, userCommand
   Result: "Command relayed" or "Error in processing request"
1  if AUTHENTICATE(userCredentials) then
       // Authorizer: Validates user credentials
       $authorizationKey \leftarrow$ GENERATEKEY($userCredentials$) // Authorizer: Generates a unique
       session key if CHECKPERMISSIONS(authorizationKey, userCommand) then
           // Organizer: Validates command permissions
           $processedCommand \leftarrow$ PROCESSCOMMAND($userCommand$) // Container: Refines user
           command if RELAYCOMMAND(processedCommand) then
               // Linker: Sends command to robot return"Command relayed."
2  return"Error in processing request."
3  Procedure AUTHENTICATE(credentials)
       Data: User credentials
       Result: Boolean indicating if credentials are valid
4      returncredentials are valid against the database
5  Procedure GENERATEKEY(credentials)
       Data: User credentials
       Result: Unique session key
6      returnuniqueKey for the session
7  Procedure CHECKPERMISSIONS(key, command)
       Data: Authorization key, User command
       Result: Boolean indicating if key has permissions for command
8      returnkey matches permissions for command
9  Procedure PROCESSCOMMAND(command)
       Data: User command
       Result: Refined command
10     returnoptimize and refine command for robot
11 Procedure RELAYCOMMAND(processedCommand)
       Data: Processed command
       Result: Boolean indicating if command is relayed
12     returncommand is successfully sent to robot
```

scopes. For example, a user or operator might possess the *scope* to read messages but be restricted from sending them. The *organiser* diligently verifies the scopes tied to a user's or operator's token, ensuring actions are strictly within their permitted boundaries. Additionally, to safeguard against misuse and maintain equitable usage, ROSIC enforces rate limits. This constrains the number of requests a user or operator can initiate within a specified duration. The *organiser* evaluates not only the user's or operator's access level but also whether the rate limits are adhered to. In situations where there is a hint of inappropriate use or when a user or operator's access becomes redundant, the *organiser* remains the authority to invalidate the ROSIC authorisation key, effectively terminating access for the implicated user or operator.

- **Container:** A *container* is a secure, isolated environment within the system that holds specific data, permissions and functionalities. It encapsulates user and operator information, their associated scopes, rate limits and other relevant details, ensuring that each user's or operator's interactions are confined to their designated boundaries. The core attributes of a *container* are: 1. Isolation: to ensure that each user's or operator's actions, data and permissions are kept separate from others, preventing unintended interference or data breaches. 2. Security: by encapsulating data and permissions within a container, ROSIC can offer an added layer of protection against unauthorised access or potential vulnerabilities. 3. Flexibility: to allow ROSIC to easily allocate or modify resources, permissions and functionalities for individual users or operators without affecting the overall system. 4. Efficiency: to streamline the process of managing multiple users or operators, each container can be treated as a standalone unit, making it easier to monitor, update, or troubleshoot.

- **Linker:** In ROSIC's framework, a *linker* is a tool that connects various ROSIC components to each other, ensuring seamless communication and interaction. It acts as a bridge, facilitating the transfer of data, requests, and responses between the user's or operator's *container* and the central ROSIC system. The main features of the *linker* include the following: 1. Integration: the linker ensures that all the system components, regardless of their individual configurations or data, can interact with the main ROSIC system without compatibility issues. 2. Data Flow: It manages the flow of data between containers and the main system, ensuring that requests and responses are directed to the appropriate destinations. 3. Scalability: As ROSIC grows and more users or operators are added, the *linker* can efficiently connect new *containers* to the system without disrupting existing operations. 4. Monitoring and Control: The linker provides a central point for ROSIC to monitor interactions, enforce rate limits and validate authorisation keys across all *containers*. In fact, while the *container* provides a secure and isolated environment for each user or operator, the *linker* ensures that these individual units can effectively communicate and interact with the broader ROSIC system.

The efficacy of ROSIC is underpinned by the intricate interplay of its architectural components. Each component not only performs its designated role but also seamlessly integrates with others, ensuring the system's holistic functionality. The process commences with the *authoriser*, which validates user credentials and grants a unique authorisation key for authenticated sessions. This key becomes instrumental for the *organiser*, which, building upon the authentication, regulates user interactions by cross-referencing permissions associated with the key. As commands are issued, the *container* steps in, processing these commands in a secure environment. Its functionality is deeply intertwined with the validation of the *authoriser* and the regulatory oversight of the *organiser*. Finally, the *linker* acts as the communication conduit, relaying the refined commands from the *container* to the robot. It ensures that the communication is not only accurate but also secure, drawing from the collective validations and checks of its preceding components. Together, these components of ROSIC work in tandem, offering a robust and efficient telecommunication framework for robotic systems.

2.2 | Scenarios for robot control

Several scenarios can be outlined where ROSIC serves as a telecommunication system to streamline communication between robots and their users or operators. These include:

- **One user or operator communicates with one robot:** When an operator initialises control over a robot, ROSIC's *container* functionality seamlessly integrates the robot's operational parameters and task-specific configurations, aligning them with the operator's specific needs [27]. This integration establishes a deterministic operational framework, enhancing the robot's efficiency and precision for its designated tasks. ROSIC's *linker* component stands as a pivotal bridge, enabling instantaneous bidirectional communication. As the operator issues commands or tweaks parameters, the robot's sensors promptly provide feedback, facilitating swift and dynamic modifications. This capability for immediate interaction is indispensable, especially for tasks demanding high levels of adaptability and rapid responsiveness. Additionally, ROSIC's advanced monitoring features offer the operator a comprehensive view of the robot's operational metrics, ensuring that tasks align with set benchmarks. Should any deviations arise, ROSIC's sophisticated control algorithms spring into action, recalibrating the robot's functions to meet the stipulated criteria.

Utilising ROSIC within the framework where a single operator interfaces with an individual robot offers transformative potential across a spectrum of industries. Specifically: (1) Autonomous Vehicle Fleets: With ROSIC, an operator can oversee the real-time operations of a self-driving vehicle, making on-the-fly decisions based on traffic flow, road conditions and adaptive rerouting strategies. (2) Smart Grid Management: In power grids, an operator can utilise ROSIC to control a robot responsible for tasks, such as optimal load distribution, swift fault detection and proactive maintenance. (3) Forestry Management: In forested regions, an operator can harness ROSIC to direct a robot in tasks like monitoring forest health, early fire detection, and overseeing unauthorised activities.

Algorithm 2 outlines the interaction between a single operator and a robot through the ROSIC system. The process commences with the `initializeOperator` function, enabling an operator to establish its control preferences. The `assignTask` function lets the operator assign a task to the robot only if it is currently idle. Through the `getFeedback` function, the operator can instantaneously receive data from the robot, exemplifying real-time bidirectional communication. The `adjustRobotSettings` function facilitates the operator in tweaking the robot's configurations dynamically, while the `recalibrateRobot` function ensures that the robot's functionalities can be recalibrated to maintain alignment with set benchmarks or criteria.

- **Multiple users or operators interacting with one robot:** In this scenario, ROSIC acts as the backbone of the operation, providing a structured, secure and efficient framework for multiple users to access and control a single robot [28]. Its features ensure that each user's interactions are isolated, the data are managed effectively and the robot operates

Algorithm 2 ROSIC general structure for controlling a robot by an operator

```
Data: operator = {}, robot = {"status": "idle", "data": {}, "config": {}}, tasksQueue = {}
Result: Management of a robot by a single operator
1  Function initializeOperator(operatorID, preferences)
2      operator = {"ID": operatorID, "preferences": preferences}
3  Function assignTask(task)
4      if robot["status"] == "idle" then
5          robot["status"] = "active"
6          tasksQueue.append(task)
7  Function getFeedback()
8      return robot["data"]
9  Function adjustRobotSettings(settings)
10     robot["config"].update(settings)
11 Function recalibrateRobot(calibrationData)
12     robot["config"]["calibration"] = calibrationData
```

seamlessly. One of ROSIC's standout features is its dynamic scheduling capability. By leveraging algorithmic time-slot allocations, ROSIC ensures optimal robot utilisation, minimises idle times and prevents access conflicts. Furthermore, its granular permissioning system, underpinned by a robust *authoriser* mechanism, facilitates role-based access. This means that a senior user's interaction with the robot can be qualitatively different from that of a junior user based on predefined roles and permissions. The *linker* component in ROSIC acts as a communication bridge, ensuring real-time data transfer, command execution, and feedback loops between the user interface and the robot's operational module. From a data integrity standpoint, ROSIC's container approach offers unparalleled advantages. Post-operational data are stored within the confines of the individual's container, ensuring data sanctity and preventing potential breaches.

Some of the applications of using ROSIC based on this scenario include: (1) Medical Robotics: Different medical professionals access a surgical robot for various procedures in a hospital setting. (2) Educational Institutions: Students and instructors access a shared robot for different experiments or demonstrations. (3) Research Laboratories: Multiple researchers access a specialised robot for different experiments or data collection sessions. (4) Manufacturing Units: Different technicians access a robot for various tasks throughout a production cycle. (5) Entertainment Industry: Multiple programmers or designers programme a robot for different scenes or performances.

Algorithm 3 manages multiple users' access to a single robot within the ROSIC framework. The user dictionary stores user details, with each user having a `userID` (a unique identifier), `role` (like senior or junior), `timeSlot` (a designated period for robot access) and `permissions` (allowed tasks). The `robotStatus` indicates whether the robot is `idle` or `active`. At any point, the `activeUser` variable denotes the user currently interfacing with the robot. The `addUser` function registers users, `accessRobot` checks if a user can access the robot based on the current time and their time slot, `storeData` saves user-specific data, and `endSession` terminates an active user's interaction with the robot. This structure ensures sequential non-conflicting access for users.

- **One user or operator interfacing with multiple robots:** One of the main applications of this scenario is robot fleet management. In large-scale operations, such as warehouses, logistics centres or manufacturing units, a fleet of robots is often deployed to optimise processes and improve efficiency. A single manager or operator, using ROSIC, oversees this fleet, ensuring synchronised operations, task allocations and real-time monitoring. In this scenario, the manager can utilise ROSIC's *container* feature to encapsulate configurations, tasks and data specific to each robot in the fleet. This ensures that each robot operates within its designated parameters, preventing operational conflicts. As the manager assigns tasks or modifies operations, the *linker* ensures that commands are instantly relayed to the respective robots, allowing for dynamic task reallocation based on real-time needs. For instance, if one robot faces a malfunction or is delayed, tasks can be swiftly reassigned to another robot in the fleet. Furthermore, ROSIC's monitoring capabilities enable the manager to get a bird's-eye view of the entire fleet's operations. This not only aids in identifying bottlenecks or inefficiencies but also ensures timely interventions in cases of anomalies. Leveraging ROSIC in the context of the first scenario can be transformative across various sectors, such as: (1) Agriculture: ROSIC can empower farmers to seamlessly control a fleet of robots, streamlining tasks, such as planting, harvesting and real-time crop monitoring. (2) Warehouse Management: Warehouse operators can

Algorithm 3 ROSIC general structure for one user or operator interfacing with multiple robots

```
Data: users = {}, robotStatus = "idle", activeUser = None
Result: Manage multiple users accessing a single robot
Function addUser(userID, role, timeSlot, permissions)
    users[userID] = {"role": role, "timeSlot": timeSlot, "permissions": permissions, "data": {}}
Function accessRobot(userID, currentTime)
    user = users[userID]
    if robotStatus == "idle" and user["timeSlot"]["start"] ≤ currentTime ≤ user["timeSlot"]["end"] then
        robotStatus = "active"
        activeUser = userID
        return "Access granted."
    return "Access denied."
Function storeData(userID, data)
    if activeUser == userID then
        users[userID]["data"] = data
        return "Data stored."
    return "Error storing data."
Function endSession(userID)
    if activeUser == userID then
        robotStatus = "idle"
        activeUser = None
        return "Session ended."
    return "Error ending session."
```

harness ROSIC to efficiently manage a cadre of robots, ensuring precise inventory management, timely restocking, and swift order fulfilment. (3) Environmental Exploration: In critical search missions, operators can utilise ROSIC to coordinate a diverse mix of aerial and ground-based robots, ensuring comprehensive coverage across challenging terrains and swift response times. (4) Traffic Management: Traffic controllers can leverage ROSIC to oversee a fleet of aerial robots, facilitating real-time traffic surveillance, congestion analysis, and incident reporting. In each of these applications, ROSIC's robust architecture ensures efficient communication, task allocation and real-time adjustments, optimising the performance of robotic fleets.

Algorithm 4 illustrates how a single user or operator can effectively manage multiple robots within a fleet. It starts by initialising crucial data structures, including a dictionary to represent the robot fleet, the currently active robot and a queue for tasks. The algorithm defines essential functions, such as `addRobot` to include robots in the fleet along with their configurations; `assignTask` to assign a task to a specific robot if it is idle; `switchRobot` to change the active robot; `getFeedback` to retrieve feedback data from a specific robot and `reallocateTask` to reallocate a task from one robot to another if a new robot is available. These functions collectively illustrate how a single user or operator can manage a fleet of robots by adding, assigning, switching between them, retrieving feedback and reallocating tasks when necessary through the ROSIC middleware.

- **An exclusive communication system for robot-to-robot interactions:** In expansive operations, especially in dynamic environments, the ability for robots to communicate and collaborate becomes paramount. Each robot, upon deployment, is initialised with ROSIC's *container* feature, which encapsulates its operational parameters and task-specific configurations. This ensures that each robot operates within its designated parameters, optimising its performance for the given task. ROSIC's *linker* component facilitates real-time bidirectional data transfer not only between the operator and each robot but also between the robots themselves. Furthermore, ROSIC's monitoring capabilities provide a holistic view of the entire robot's operations. This not only aids in identifying bottlenecks or inefficiencies but also ensures timely interventions in cases of anomalies in robot operations. In scenarios where a robot faces a challenge it cannot overcome alone, other robots, informed via the telecommunication system, can aid or adjust their tasks accordingly.

Utilising ROSIC within the framework of the first scenario offers transformative potential across a spectrum of industries. Specifically: (1) Autonomous Vehicle Fleets: ROSIC facilitates real-time data exchange among self-driving vehicles, enabling them to share insights on traffic flow, road conditions and adaptive rerouting strategies. (2) Smart Grid Management: Within power grids, robots can utilise ROSIC to communicate effectively, ensuring optimal load distribution, swift fault detection and proactive maintenance. (3) Forestry Management: In forested regions, robots can harness ROSIC to relay information concerning forest health, early fire detection and monitoring of unauthorised activities.

Algorithm 5 presents how the ROSIC middleware enables the coordination and control of numerous robots through integration with a telecommunication system. It commences by initialising essential data structures, including dictionaries to represent robots, a task queue and shared data. The algorithm defines crucial functions: `initializeRobot` to add robots to the system along with their parameters; `assignTask` to assign tasks to robots if they are idle; `shareData` to share data among robots; `getSharedData` to retrieve

Algorithm 4 ROSIC general structure as a robot fleet management system with single user

```
Data: robotFleet = {}, activeRobot = None, tasksQueue = {}
Result: Fleet management of multiple robots by a single user/operator
1  Function addRobot(robotID, configuration)
2      robotFleet[robotID] = {"config": configuration, "status": "idle", "data": {}}
3  Function assignTask(robotID, task)
4      if robotFleet[robotID]["status"] == "idle" then
5          robotFleet[robotID]["status"] = "active"
6          tasksQueue.append({"robot": robotID, "task": task})
7  Function switchRobot(robotID)
8      activeRobot = robotID
9  Function getFeedback(robotID)
10     return robotFleet[robotID]["data"]
11 Function reallocateTask(oldRobotID, newRobotID, task)
12     if robotFleet[newRobotID]["status"] == "idle" then
13         robotFleet[oldRobotID]["status"] = "idle"
14         robotFleet[newRobotID]["status"] = "active"
15         tasksQueue.remove({"robot": oldRobotID, "task": task})
16         tasksQueue.append({"robot": newRobotID, "task": task})
```

Algorithm 5 ROSIC general structure for the integration of multiple robots with the telecommunication system

```
Data: robots = {}, tasksQueue = {}, sharedData = {}
Result: Management of multiple robots with telecommunication system integration
1  Function initializeRobot(robotID, parameters)
2      robots[robotID] = {"config": parameters, "status": "idle", "data": {}}
3  Function assignTask(robotID, task)
4      if robots[robotID]["status"] == "idle" then
5          robots[robotID]["status"] = "active"
6          tasksQueue.append({"robot": robotID, "task": task})
7  Function shareData(robotID, data)
8      sharedData[robotID] = data
9  Function getSharedData(robotID)
10     return sharedData[robotID]
11 Function adjustStrategy(robotID, newStrategy)
12     robots[robotID]["config"]["strategy"] = newStrategy
```

shared data from a specific robot and `adjustStrategy` to modify the strategy of a robot. These functions collectively portray how the ROSIC middleware facilitates the management of multiple robots within a telecommunication system, enabling task assignment, data sharing and strategy adjustment.

3 | INTEGRATION WITH TELEGRAM

Telegram Instant Messenger is a suitable choice for implementing ROSIC for several compelling reasons, such as: (1) Open-Source Nature: Being open-source, Telegram allows developers to access its source code, facilitating customisation, improvements and integration with systems such as ROSIC without any licencing restrictions. (2) End-to-End Encryption: Telegram offers end-to-end encryption for secret chats, ensuring secure communication. This security feature is crucial when controlling robots remotely, as it safeguards commands and feedback from potential interception or tampering. (3) Cloud-Based Architecture: Telegram's cloud-based approach ensures that messages (commands for robots within the context of our research) are stored securely and can be accessed from multiple devices. This allows for a seamless and continuous control experience even if one device encounters issues. (4) Cross-Platform Support: Telegram is compatible across a variety of devices, including smartphones, tablets and computers running on different operating systems. This flexibility ensures that a wide range of users can control robots without the need for specialised hardware. (5) Large File Transfer: With the ability to send files up to 2 GB, Telegram can effectively handle large datasets or software updates for robots, facilitating on-the-fly adjustments or improvements to robot functions. (6) User-Centred Interface: Telegram is designed with a user-centred interface, making it accessible even to non-technical users. This aligns with ROSIC's goal to design robot controls with a user-centred approach. (7) Interactive Communication Capabilities: Features such as group chats and channels in Telegram can be harnessed for scenarios where multiple users need to communicate with or control a single robot or a group of robots. Given these attributes, Telegram can serve as a robust and versatile platform for implementing ROSIC, promoting secure, efficient and user-centred remote robot control.

Figure 2 illustrates the core of the Internet based telecommunication system: a setup based on the ROSIC middleware integrated with the Telegram API. As depicted in Figure 2, any device that can run the Telegram application, ranging from mobile phones and single-board computers, such as Raspberry Pi, to personal computers and laptops, can communicate via ROSIC over the Internet.

Algorithm 6 begins with the initialisation of the Telethon library, a Python framework tailored for the Telegram API, enabling our programme to interact with Telegram and extract

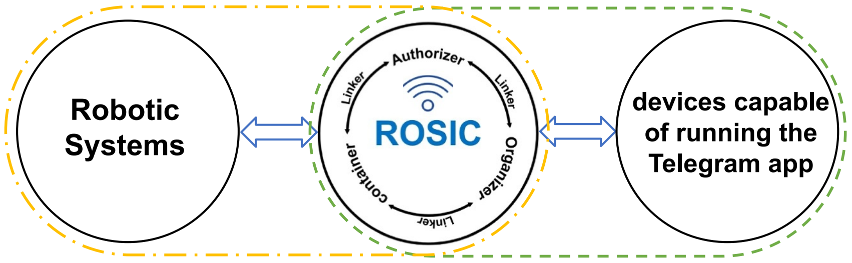


FIGURE 2 The foundational telecommunication architecture of ROSIC middleware in conjunction with the Telegram API, optimised to streamline and enhance the communication management process. API, Application Programming Interface; ROSIC, Robot Control System using Instant Communication.

Algorithm 6 Save Telegram group messages to file (*container*)

```
Data: API_ID (Your Telegram API ID), API_HASH (Your Telegram API Hash), GROUP_ID_OR_LINK (Your Telegram
      Group Link or ID), FILE_NAME (messages.txt)
Result: Messages from the Telegram group saved to a text file
1: Initialize TELETHON LIBRARY
2: Create TELEGRAM CLIENT using API_ID and API_HASH named "client"
3: while client is connected do
4:     if NEW MESSAGE in GROUP_ID_OR_LINK then
5:         SENDER ← Get sender of the message
6:         Open FILE_NAME in append mode as "file"
7:         Write "Sender ID: ", SENDER's ID, ", Sender Username: ", SENDER's USERNAME, ", Message: ", MESSAGE
             TEXT to "file"
```

messages from a group chat. During the initialisation, key data points and their unique credentials from Telegram for application authentication (API_ID and API_HASH) are established. Furthermore, we define the specific Telegram group of interest through GROUP_ID_OR_LINK and the storage location for the messages with FILE_NAME. By default, we choose 'messages.txt' as the filename (*container*).

Upon creating a Telegram client using the aforementioned credentials, the algorithm enters its primary loop, continuously monitoring the designated group for new messages. When a new message is detected, the sender's details are retrieved and the message, along with the sender's information, is appended to the specified file. This ensures a comprehensive record of both the message and its sender. The algorithm remains operational, archiving messages in real-time until an external factor, such as a disconnection or programme termination, interrupts it. Consequently, all messages from various users or operators are stored in a text file within the ROSIC system.

Algorithm 7 presents the operational framework of a Telegram bot, which is tailored to manage user interactions based on permissions and identity verification via ROSIC. Initially, the bot is initialised with a unique TOKEN, which is essential for its operation on the Telegram platform. The algorithm comprises several functions. The `verify_identity(user_id)` function assesses if a user, identified by their `user_id`, possesses permission based on a predetermined schedule. The `verify_identity(user_id)` function ascertains the user's identity, returning a Boolean value indicating its success or failure. To determine if a user has the requisite permissions to access specific information, the `has_accessibility_permission(user_id)` is defined. The start (update, context) function is activated when the bot starts or when a user initiates a conversation, checking the user's schedule-based permission and sending a corresponding reply. The `handle_message(update, context)` function manages incoming user messages, verifies user identity and responds based on the content of the message and user permissions. The `main()` function serves as the core of the algorithm. It initialises the bot using the provided TOKEN, configures command handlers, especially for the start command, and sets up a

Algorithm 7 The operational framework of ROSIC integrated with a Telegram bot: Tailoring user interactions with permissions and identity verification

```
Data: TOKEN initialized with "YOUR_TELEGRAM_BOT_TOKEN"
1  has_schedule_permission(user_id);
2  return True or False based on the schedule;
3  verify_identity(user_id);
4  return True or False based on identity verification;
5  has_accessibility_permission(user_id);
6  return True or False based on accessibility permission;
7  start(update, context);
8  user_id ← get user_id from update;
9  if has_schedule_permission(user_id) then
10     Send reply "You have permission to send and receive commands.";
11 else
12     Send reply "You do not have permission to send and receive commands.";
13 handle_message(update, context);
14 user_id, text ← get from update;
15 if not verify_identity(user_id) then
16     Send reply "Identity verification failed. Communication terminated.";
17     return;
18 if text is "request information" then
19     if has_accessibility_permission(user_id) then
20         Send reply "Here's the information you requested.";
21     else
22         Send reply "You do not have permission to access this information.";
23 main();
24 updater ← create with TOKEN;
25 Add start command handler to updater;
26 Add message handler for non-command text to updater;
27 Start updater;
28 Keep updater running until interrupted;
29 Execute main();
```

message handler for the non-command text. Once these configurations are in place, the bot is activated and remains in operation until an external interruption occurs.

Algorithm 8 initially sets up an ROS 2 node named `chat_interface`, a computational entity in ROS 2 designed for data processing and communication with other nodes. This node's primary role is to act as a bridge between the chat system and the robot. Two main functions are defined: `robot_callback(data)` and `send_to_robot(message)`. The former is crafted to manage incoming data from the robot, display the received message and offer a placeholder for further integration to relay this message to the chat system. The latter function, on the other hand, is responsible for transmitting messages to the robot, publishing them to a specific ROS 2 topic and showcasing the sent message.

To ensure continuous communication, the pseudocode establishes a subscription to the `from_robot_topic`, enabling the node to listen for messages on this topic and triggering the `robot_callback` function upon receipt of any message. To verify the connection, ROSIC also sends an initial greeting to the robot using the `send_to_robot` function. The process concludes with the node entering a spin state, signifying its readiness to perpetually process data and await any callbacks, ensuring uninterrupted interaction between the chat system and the robot.

4 | PERFORMANCE AND EFFICIENCY ANALYSIS

The performance and efficiency of ROSIC can be gauged through several critical parameters: (1) Latency, which measures the time taken for a command to reach its destination, is paramount for real-time interventions. (2) Throughput, indicates the system's capacity by assessing the number of messages processed per unit time. (3) Scalability, as ROSIC is designed to cater to multiple users and robots, its scalability, both in terms of the number of users and robots, becomes crucial. This scalability analysis should evaluate how the system performs with increasing concurrent users or robots. (4) Resource utilisation, including Central Processing Unit (CPU), memory and bandwidth consumption, provides a lens into the system's efficiency and potential bottlenecks. Additionally, from a user-centric perspective, (5) Security is always an important concern, and can be assessed by tracking the time taken for user authentication (Authentication Time) and monitoring any unauthorised access attempts.

To assess the critical parameters influencing the performance of telecommunication systems utilising ROSIC middleware, we conducted tests on an NVIDIA® Jetson Orin Nano™ platform. The test environment operated on a network with upload and download speeds of 26 Mbps each. Although pinpointing the exact latency of message transmission in ROSIC via Telegram is complex due to network variables, we approximated latency by measuring the time taken for message dispatch and response receipt. Notably, this method may not provide absolute latency precision due to inherent network speed fluctuations, server load dynamics, and other factors. Figure 3 shows the obtained results based on the latency test and the time stamps. The results are collected based on Algorithm 9. To measure the telecommunication latency of ROSIC via the Telegram API, Algorithm 9 utilises the ‘TelegramClient‘ from the Telethon library. It initialises a client using unique developer credentials (`API_ID` and `API_HASH`), sends a test message to the user's own account, and calculates the time difference between sending and acknowledgement. This process is repeated 100 times to account for network variability. Each latency measurement, paired with its timestamp, is stored in a structured list. The accumulated data are then compiled into a `DataFrame` and saved as a uniquely timestamped Excel file, providing a comprehensive record of the API's responsiveness. Figure 4 shows the results for 10 latency tests with the associated error

Algorithm 8 Communication between ROSIC middleware and ROS 2 robot

```
   Data: data (from robot), message (to robot)
   Result: Communication between chat system and ROS 2 based robot
1  begin
2      Initialize ROS 2 node as "chat_interface"
3      Function robot_callback(data):
4          DISPLAY "Received from robot: " + data
5          // Integrate logic here to send this message to the chat system
6      Function send_to_robot(message):
7          PUBLISH message to "to_robot_topic"
8          DISPLAY "Sending to robot: " + message
9      // Create a subscription to the ROS 2 topic "from_robot_topic"
10     CREATE SUBSCRIPTION to "from_robot_topic" with robot_callback
11     // For demonstration, send a greeting to the robot
12     CALL send_to_robot(message)
13     // Keep the node active and listening
14     SPIN node indefinitely
```

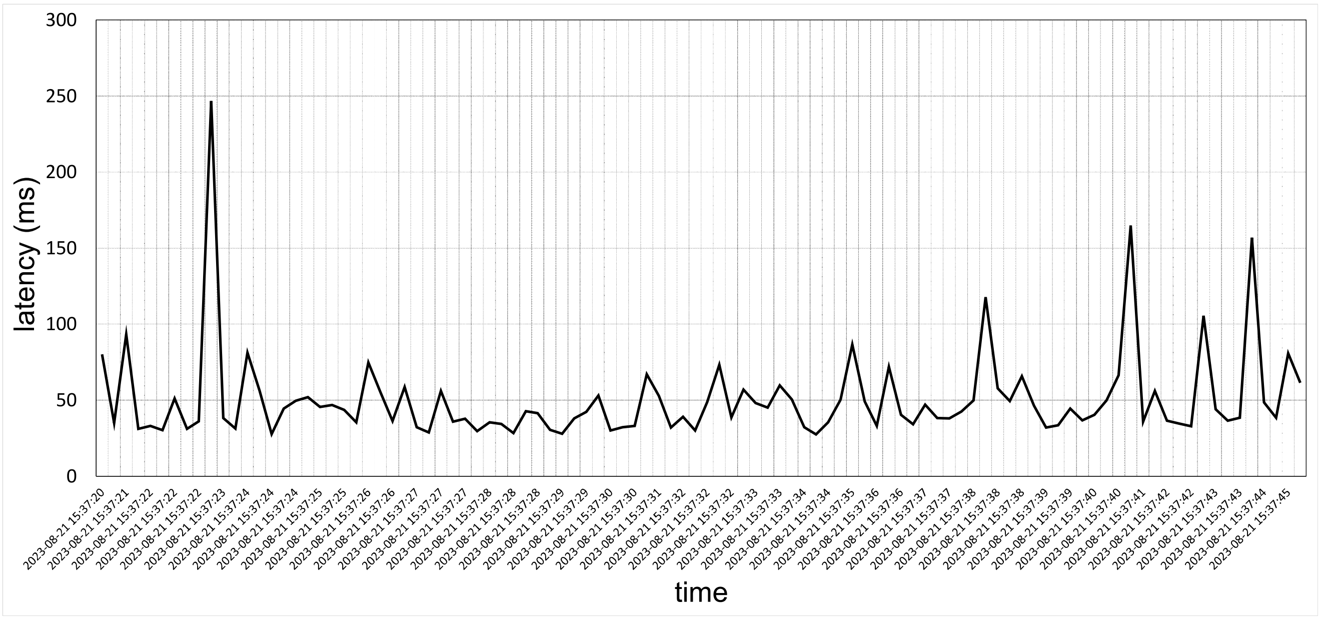


FIGURE 3 Latency test outcomes via Algorithm 9, integrating Telegram API with ROSIC. Through 100 test messages, the algorithm gauges latency, with a peak nearing 250 ms during the assessment.

Algorithm 9 Measure the telecommunication latency of ROSIC via the Telegram API

```
Data: API_ID (Your Telegram API ID), API_HASH (Your Telegram API Hash)
Result: Excel file with latency results
1: begin
2:     Function measure_latency begin
3:         client ← Initialize TelegramClient with 'session_name', API_ID, and API_HASH;
4:         Start client;
5:         start_time ← Current time;
6:         Send message to self ('me') with content 'Testing latency';
7:         end_time ← Current time;
8:         Disconnect client;
9:         latency ← (end_time - start_time) × 1000;
10:        return latency;
11:    latency_data ← Empty list;
12:    for i=1 to 100 do
13:        timestamp ← Current time in format "YYYY-MM-DD HH:MM:SS";
14:        latency ← measure_latency();
15:        Append { 'Timestamp': timestamp, 'Latency (ms)': latency } to latency_data;
16:        Print latency value;
17:    df ← Create DataFrame from latency_data;
18:    timestamp ← Current time in format "YYYY-MM-DD_HH-MM-SS";
19:    excel_file ← "latency_results_" + timestamp + ".xlsx";
20:    Save df to Excel file named excel_file;
21:    Print "Latency results saved to " + excel_file;
```

bars, where the maximum latency observed based on the employed hardware and network conditions remained below 250 ms.

Algorithm 10 presents a systematic approach to evaluate the reliability and throughput of the transmission of ROSIC messages via the Telegram platform. Initially, the algorithm establishes a connection to the Telegram servers using the API credentials. The algorithm then sets parameters such as the end time for the test, based on the given test duration, and initialises counters to track the number of messages sent, the

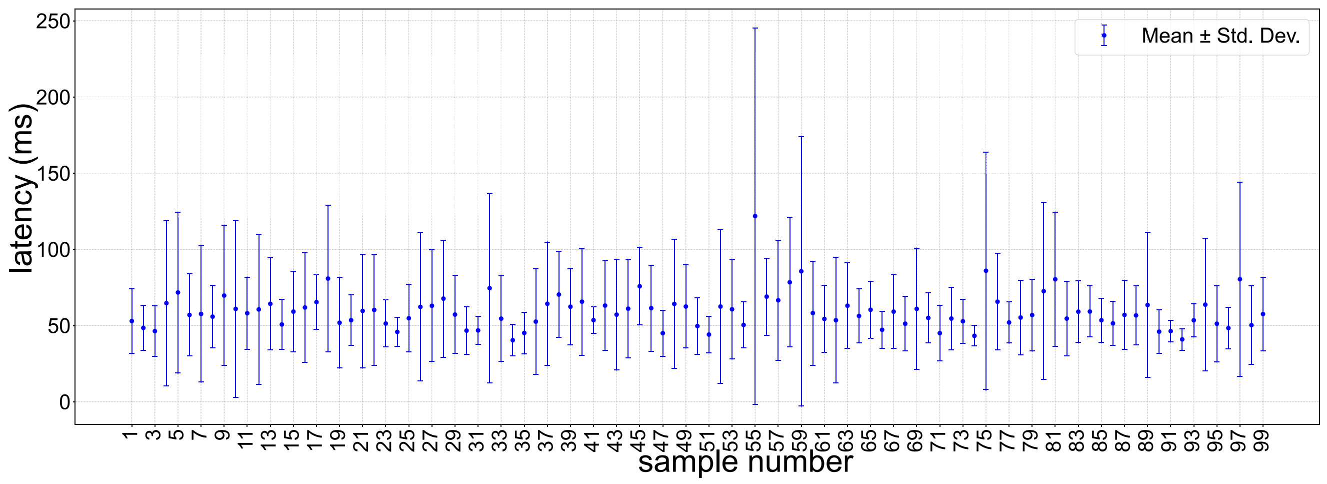


FIGURE 4 Ten latency tests depicted with corresponding error bars. In the context of the hardware and network specifications detailed in this paper for the ROSIC via the Telegram API system, the maximum observed latency remained below 250 ms.

Algorithm 10 Success rate and throughput assessment of ROSIC via the Telegram API messaging

```
Data: API credentials, recipient username, message content, test duration, send interval
Result: Success rate in percentage, Throughput in bytes/second
begin
    Initialize Telegram client with API credentials;
    Connect to the client;
    Set recipient using recipient username;
    Set end time as current time + test duration;
    num_messages_sent ← 0;
    num_messages_successful ← 0;
    total_data_sent ← 0;
    while current time < end time do
        Attempt to send message to recipient with content message content;
        if message send is successful then
            Increment num_messages_successful;
            Add size of message content to total_data_sent;
        else
            Print exception message;
        Increment num_messages_sent;
        Wait for send interval duration;
    Calculate success rate as $\left(\frac{\text{num\_messages\_successful}}{\text{num\_messages\_sent}}\right) \times 100$;
    Calculate throughput as $\frac{\text{total\_data\_sent}}{\text{actual test duration}}$;
    Print Number of Messages Attempted;
    Print Number of Successful Messages;
    Print Success Rate;
    Print Throughput;
    Disconnect from the client;
```

number of successful messages, and the total amount of data transmitted. Once connected, the recipient of the messages is determined using the specified `username`. The algorithm then enters a loop, continuously sending a predefined message to the recipient until the set test duration elapses. For each attempt, successful transmissions increment a success counter, while the total attempts, regardless of outcome, increment an overall counter.

Algorithm 11 is designed to gauge the performance of the ROSIC via the Telegram messaging platform under varying workloads. For each batch size in a predefined list, the algorithm captures the start time and initialises a list to record

Algorithm 11 **Scalability assessment of ROSIC via the Telegram API messaging**

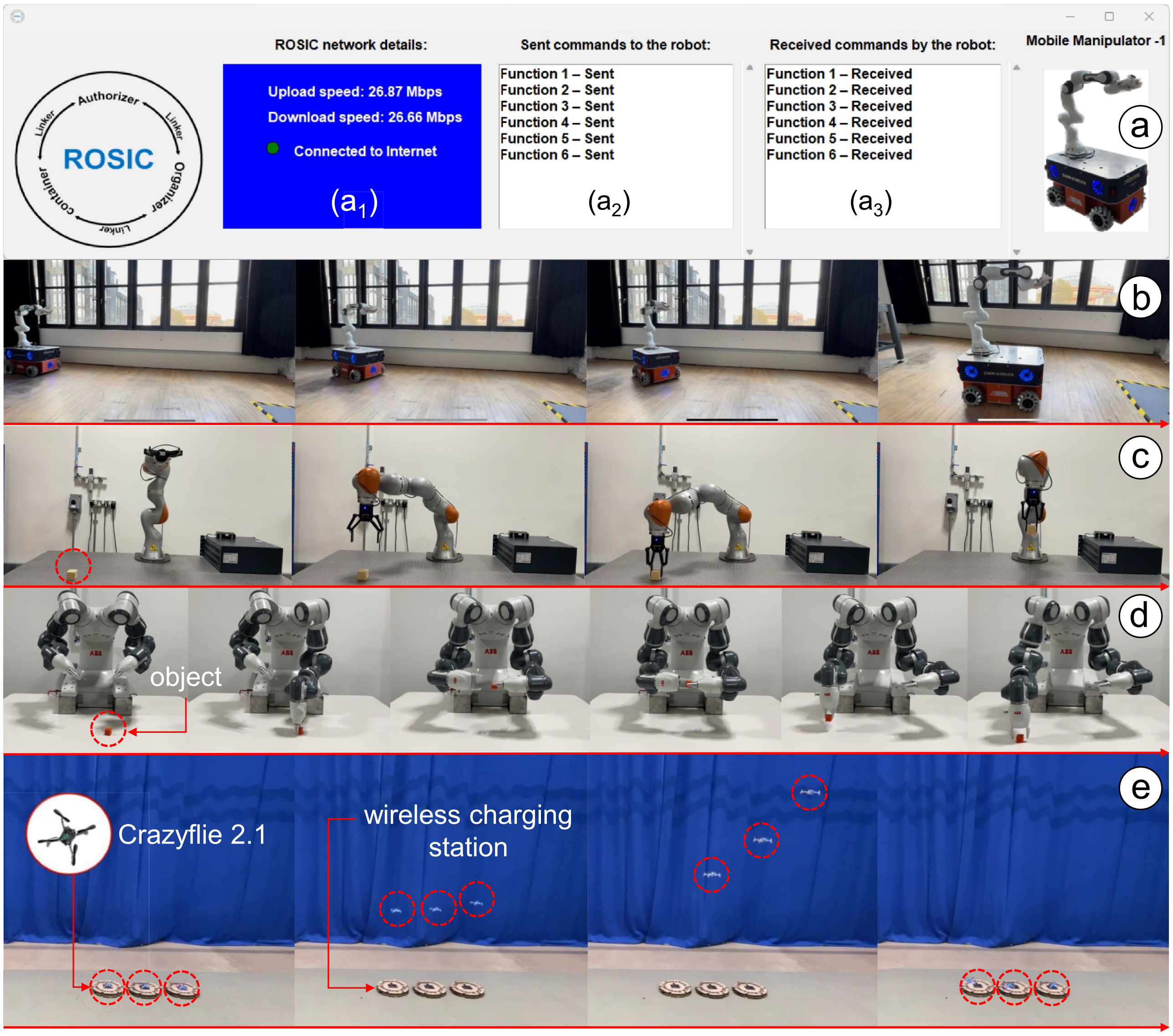
```
Data: API credentials, recipient username, message content, list of message batch sizes
Result: Throughput and average latency for each batch size
begin
    Initialize Telegram client with API credentials;
    Connect to the client;
    Set recipient using recipient username;
    for each batch size in list of message batch sizes do
        Set start time to current time;
        Initialize empty list for latencies;
        for i from 1 to batch size do
            Set send start time to current time;
            Send message to recipient with content message content;
            Set send end time to current time;
            Calculate latency as send end time minus send start time;
            Add latency to list of latencies;
        Set end time to current time;
        Calculate total time as end time minus start time;
        Calculate throughput as batch size divided by total time;
        Calculate average latency as the average of list of latencies;
        Print throughput and average latency for batch size;
    Disconnect from the client;
```

individual message latencies. As messages are dispatched, the time taken for each send operation is computed and stored. Upon completing the batch, the end time is noted and the total time for the batch is determined. Using these data, two critical performance metrics are calculated: throughput and average latency. The throughput represents the number of messages sent per unit time and is derived by dividing the batch size by the total time (similar to Algorithm 10). Meanwhile, the average latency, which is indicative of the mean-time taken for individual message deliveries, is computed from the recorded latencies. These metrics offer insights into the system's efficiency and responsiveness under the varying tested loads. The procedure is reiterated for each batch size, furnishing a comprehensive view of scalability as the messaging load escalates. The session concludes by disconnecting from the Telegram client, ensuring the session's integrity. Within the main loop, Algorithm 11 attempts to send a predefined message to the designated recipient until the test duration elapses. For each successful message transmission, the number of counters for successful messages and the total amount of data sent are incremented. If an error occurs during transmission, an exception message is printed, providing insight into potential transmission issues. After the loop concludes, the algorithm calculates the success rate by comparing the number of successful messages to the total attempted. Additionally, it computes the throughput by dividing the total data sent by the actual test duration, yielding a measure in bytes per second. This dual metric, success rate and throughput offers a comprehensive assessment of the ROSIC messaging system's performance under the tested conditions via Telegram. Moreover, the algorithm was executed with a test duration that resulted in 51 message send attempts. Remarkably, all of these attempts were successful, leading to a perfect success rate of 100%. This indicates that during the test period, the Telegram platform was highly reliable, with no message transmission failures or losses. The throughput, measured at 195.12 byte/s, provides insight into the data transfer rate. Given that the content of each message was a predefined text for throughput assessment, this rate likely reflects the combined size of the message content, metadata and any overhead introduced by the Telegram protocols.

The scalability assessment of ROSIC via the Telegram messaging platform yielded intriguing results. For smaller batches of 10 messages, the system exhibited a high throughput of 15.82 message/s with a minimal average latency of just 0.0632 s per message. This suggests that for low-volume messaging, the system is highly efficient and responsive. However, as the message batch size increased, a decline in throughput was observed. For a batch of 50 messages, the throughput dropped to 5.04 message/s, with the latency almost tripling to 0.1985 s. A further increase in batch size to 100 messages saw the throughput decrease to 3.04 message/s, accompanied by a latency of 0.3290 s. Interestingly, when the batch size was amplified to 500 messages, the throughput remained relatively stable at 3.18 message/s, and the latency slightly decreased to 0.3143 s. This suggests that beyond a certain threshold, the system's performance stabilises, indicating a potential saturation point in its scalability. Such insights are invaluable for optimising and predicting the system's behaviour under varying loads.

In Figure 5, part (a) presents the ROSIC middleware user interface, illustrating the data transmission speed (a_1), encompassing both sending and receiving functionalities, along with the connection status. Furthermore (a_2) and (a_3) within part A distinctly display the specific commands dispatched to the robot and the confirmation of their successful reception by the robot, denoted as 'Sent command to the robot' and 'Received commands by the robot', respectively. The following sections of Figure 5 visually represent robots being controlled via ROSIC, utilising ROS 2 through the Internet and integrated with the Telegram API. In particular, (b) shows an omni-directional mobile manipulator in action, (c) showcases the KUKA LBR iiwa 7 in operation, (d) demonstrates the dual-arm YuMi ® collaborative robot in a hand-over task and (e) presents the Crazyflie 2.1, an open-source flying development platform, engaged in a synchronised swarm flight.

Telegram's end-to-end encryption, especially evident in its *Secret Chats* feature, ensures that messages are encrypted on the sender's device and decrypted only on the receiver's end, safeguarding data during transit. The platform's transparency, owing to its open-source nature, allows for rigorous code reviews by independent security experts, leading to the identification and rectification of potential vulnerabilities. Furthermore, Telegram's self-destructing messages in *Secret Chats* and its policy of not storing these chats on servers can be instrumental for ROSIC, minimising data retention risks. The added layer of security through two-step verification and the ability to customise security settings further fortifies user data against



Upload speed: 26.87 Mbps
Download speed: 26.66 Mbps
● Connected to Internet

Function 1 – Sent
Function 2 – Sent
Function 3 – Sent
Function 4 – Sent
Function 5 – Sent
Function 6 – Sent

Function 1 – Received
Function 2 – Received
Function 3 – Received
Function 4 – Received
Function 5 – Received
Function 6 – Received

FIGURE 5 Utilising ROSIC for remote control of various robotic systems over the Internet, operators can activate various predefined movement functions that are designed based on ROS 2 (a) User interface of the ROSIC middleware (b) An omni-directional mobile manipulator demonstrating varied manoeuvres (c) The KUKA LBR iiwa 7 executing a grasping action (d) The dual-arm YuMi ® collaborative robot from ABB performing an object hand-over; and (e) A trio of Crazyflie 2.1 drones exhibiting synchronized swarm flight.

unauthorised access. Additionally, Telegram's robust API and bot support can be harnessed by ROSIC for automation and enhanced user interactions without compromising security. The platform's vast user base and active development ensure that any emerging security threats are swiftly addressed. Its cloud-based architecture offers users the flexibility of accessing data from any device securely, and the secure file-sharing capabilities can be pivotal if ROSIC requires file exchanges. In essence, integrating ROSIC with Telegram not only provides a suite of advanced security features but also ensures a user-centred experience, provided that the implementation is done judiciously and that users are educated on best practices.

5 | CONCLUSION

The development and proposal of the ROSIC emerged from the pressing need to address the complexities and challenges inherent in remote robot control. ROSIC is architectured to provide a comprehensive telecommunication system tailored for robots, emphasising versatility, efficiency, scalability and user-centred design. Key features of ROSIC include its modular architecture, which comprises the *authoriser*, *organiser*, *container* and *linker*, ensuring seamless and adaptive integration with diverse robotic scenarios. To advance web-based robot control methods, ROSIC implements innovative integration with instant messaging applications, substantially improving accessibility for a wide range of users and augmenting real-time control functionalities. This bypasses the complexities typically associated with cloud based approaches often used for the same purpose. Our experimental evaluations of ROSIC's performance, when integrated with Telegram application, were conducted on an NVIDIA® Jetson Orin Nano™ platform within a network environment of 26 Mbps upload and download speeds. These tests assessed key performance parameters such as latency, throughput, scalability and resource utilisation. The experimental results confirmed the effectiveness of our approach where for small batches of 10 messages the system shows high efficiency with 15.82 message/s and 0.0632-s latency. However, with larger batches, such as 50 and 100 messages, throughput drops to 5.04 and 3.04 message/s, respectively, with increased latency. Interestingly, at 500 messages, throughput stabilises at around 3.18 message/s, suggesting a scalability threshold. While ROSIC has shown promising potential, future enhancements could include elevating its security by integrating distributed ledger technology. This would further strengthen data privacy and system integrity, making it a more robust and reliable platform.

CONFLICT OF INTEREST STATEMENT

The authors declare no conflicts of interest.

DATA AVAILABILITY STATEMENT

Data sharing is not applicable to this article as no new data were created or analyzed in this study.

ORCID

Rasoul Sadeghian https://orcid.org/0000-0001-6336-4002
Sina Sareh https://orcid.org/0000-0002-9787-1798

REFERENCES

1. Ray, P.P.: Internet of robotic things: concept, technologies, and challenges. IEEE Access 4, 9489–9500 (2016). https://doi.org/10.1109/access.2017.2647747
2. Mohsan, S.A.H., et al.: Unmanned aerial vehicles (UAVs): practical aspects, applications, open challenges, security issues, and future trends. Intelligent Service Robotics, 109–137 (2023). https://doi.org/10.1007/s11370-022-00452-4
3. Naseer, F., Khan, M.N., Altalbe, A.: Telepresence robot with DRL assisted delay compensation in IoT-enabled sustainable healthcare environment. Sustainability 15(4), 3585 (2023). https://doi.org/10.3390/su15043585
4. Naseer, F., et al.: A novel approach to compensate delay in communication by predicting teleoperator behavior using deep learning and reinforcement learning to control telepresence robot. Electron. Lett. 59(9), e12806 (2023). https://doi.org/10.1049/ell2.12806
5. Altalbe, A., et al.: Orientation control design of a telepresence robot: an experimental verification in healthcare system. Appl. Sci. 13(11), 6827 (2023). https://doi.org/10.3390/app13116827
6. Zhu, Q., et al.: Cybersecurity in robotics: challenges, quantitative modeling, and practice. Foundations and Trends in Robotics 9(1), 1–129 (2021). https://doi.org/10.1561/2300000061
7. Clark, G.W., Doran, M.V., Andel, T.R.: Cybersecurity issues in robotics. In: 2017 IEEE Conference on Cognitive and Computational Aspects of Situation Management (CogSIMA), pp. 1–5 (2017)
8. Orsag, M., Korpela, C., Oh, P.: Modeling and control of MM-UAV: mobile manipulating unmanned aerial vehicle. J. Intell. Rob. Syst. 69(1-4), 227–240 (2013). https://doi.org/10.1007/s10846-012-9723-4
9. Kennel-Maushart, F., Poranne, R., Coros, S.: Interacting with multi-robot systems via mixed reality. IEEE International Conference on Robotics and Automation (ICRA), 11633–11639 (2023)
10. An, X., et al.: Multi-robot systems and cooperative object Transport: communications, platforms, and challenges. IEEE Open Journal of the Computer Society 4, 23–36 (2023). https://doi.org/10.1109/ojcs.2023.3238324
11. Taylor, A.L., Wright, J.T.: A telerobot on the world wide web. In: National Conference of the Australian Robot Association (1995)
12. Schulz, D., et al.: Web interfaces for mobile robots in public places. IEEE Robot. Autom. Mag. 7(1), 48–56 (2000). https://doi.org/10.1109/100.833575
13. Gerkey, B., Vaughan, R.T., Howard, A.: The player/stage project: tools for multi-robot and distributed sensor systems. Proceedings of the 11th international conference on advanced robotics. 1, 317–323 (2003)
14. Chen, Y., Du, Z., García-Acosta, M.: Robot as a service in cloud computing. Fifth IEEE International Symposium on Service Oriented System Engineering, 151–158 (2010)
15. Osentoski, S., et al.: Robots as web services: reproducible experimentation and application development using rosjs. IEEE International Conference on Robotics and Automation, 6078–6083 (2011)
16. Alexander, B., et al.: Robot web tools [ROS topics]. IEEE Robot. Autom. Mag. 19(4), 20–3 (2012). https://doi.org/10.1109/mra.2012.2221235
17. Anis, K.: ROS as a service: web services for robot operating system. Journal of Software Engineering for Robotic 6(1), 1–14 (2015)
18. Dalla Libera, F., Ishiguro, H.: ROSlink: interfacing legacy systems with ROS. IEEE International Conference on Robotics and Automation, 475–481 (2013)
19. Koubaa, A., Alajlan, M., Qureshi, B.: Roslink: Bridging Ros with the Internet-of-Things for Cloud Robotics, pp. 265–283. Robot Operating System (ROS) (2017)
20. Portugal, D., et al.: On the Security of Robotic Applications Using ROS, pp. 273–289. Artificial Intelligence Safety and Security (2018)

21. Profanter, S., et al.: OPC UA versus ROS, DDS, and MQTT: performance evaluation of industry 4.0 protocols. IEEE International Conference on Industrial Technology (ICIT), 955–962 (2019)
22. Koubaa, A.: Service-oriented Software Architecture for Cloud Robotics (2019). arXiv preprint arXiv:1901.08173
23. Kavas-Torris, O., et al.: V2X communication between connected and automated vehicles (CAVs) and unmanned aerial vehicles (UAVs). Sensors 22(22), 8941 (2022). https://doi.org/10.3390/s22228941
24. Chen, K., et al.: FogROS2-SGC: A ROS2 Cloud Robotics Platform for Secure Global Connectivity. arXiv preprint arXiv:2306.17157 (2023)
25. Stellios, I., et al.: A survey of iot-enabled cyberattacks: assessing attack paths to critical infrastructures and services. IEEE Communications Surveys & Tutorials 20(4), 3453–3495 (2018). https://doi.org/10.1109/comst.2018.2855563
26. Sareh, S., et al.: Interoperable Robotics Proving Grounds: Investing in Future-Ready Testing Infrastructures (2023)
27. Sadeghian, R., Sareh, S.: Multifunctional arm for telerobotic wind turbine blade repair. In: 2021 IEEE International Conference on Robotics and Automation (ICRA), pp. 6883–6889 (2021)
28. Sadeghian, R., Sareh, S. Robotic repair system. U.S. Patent Application 18/032,139, Royal College of Art (2024)

How to cite this article: Sadeghian, R., Shahin, S., Sareh, S.: ROSIC: Enhancing Secure and Accessible Robot Control through Open-Source Instant Messaging Platforms. IET Cyber-Syst. Robot. e12112 (2024). https://doi.org/10.1049/csy2.12112