ZTL: Lightweight communication patterns for HRI

Patrick Holthaus
p.holthaus@herts.ac.uk
University of Hertfordshire
Hatfield, United Kingdom

Trenton Schulz
trenton@nr.no
Norwegian Computing Center
Oslo, Norway

Lewis Riches
l.riches@herts.ac.uk
University of Hertfordshire
Hatfield, United Kingdom

Claudia Andreea Badescu
University of Oslo
Department of Informatics
Oslo, Norway

Farshid Amirabdollahian
f.amirabdollahian2@herts.ac.uk
University of Hertfordshire
Hatfield, United Kingdom

ABSTRACT

Human-robot interaction (HRI) programmers often struggle with operating older robot hardware due to the short support period provided by manufacturers and difficulties integrating modern software solutions. This paper introduces the ZTL Task Library (`ZTL`), a lightweight communication framework and protocol designed to decouple robot hardware from the operating platform via socket communication, thereby increasing robot lifetime. We present a task-based communication protocol facilitating the co-design of robot behaviours with non-programming experts. Our approach has been shown across different platforms to effectively mitigate incompatibilities between middlewares, simplifying control and usability, allowing for simultaneous addressing of multiple devices.

ACM Reference Format:
Patrick Holthaus, Trenton Schulz, Lewis Riches, Claudia Andreea Badescu, and Farshid Amirabdollahian. 2026. ZTL: Lightweight communication patterns for HRI. In *Proceedings of the 2026 ACM/IEEE International Conference on Human-Robot Interaction (HRI '26)*. ACM, New York, NY, USA, 5 pages. https://doi.org/XXXXXXX.XXXXXXX

1 BACKGROUND AND SUMMARY

HRI researchers often face challenges putting and keeping physical robots into operation, given that the operating systems that commercially available robots rely upon quickly reach the end of their support. Ubuntu Xenial, which enables robots like Care-O-bot 4 or TurtleBot 3, for example, was supported for five years from 2016 until 2021. With Care-O-bot also being released in 2016, this leaves a five-year window to procure the robot, train operators, and put the robot to use. At the same time, robotics researchers need to update software to incorporate new algorithms, novel approaches and follow research developments, e.g. in rapidly evolving fields like machine learning, artificial intelligence, and robotics. Integrating these updated solutions is time-consuming and can negatively affect the operational window of the device. Sometimes, incompatibilities between the robot platforms and modern software solutions prevent the adoption of such solutions on existing devices.

We present a software library to detach older and more exotic technology from the platform they need to be operated on, effectively increasing their lifetime and cost efficiency. Our approach allows researchers, developers, and users to benefit from modern hardware, operating systems, and algorithms, while the robot continues operating in its own ecosystem. This solution is lightweight, has minimal dependencies, and is compatible with many programming languages, including end-of-life versions. Moreover, our protocol abstracts commands from robot hardware, facilitating reproducibility between experimental sites, easier adoption of solutions developed elsewhere, and allowing for co-designing robot behaviour with people less acquainted with programming.

Most robots rely on some form of middleware for inter-process communication, where different components exchange messages to operate the device. While there have been multiple attempts at enabling this communication [8, 15], many robots use some version of Robot Operating System (`ROS`) [10]. Besides providing a plethora of packages and commonly used data types, `ROS` and its successor `ROS 2` [7] enable basic inter-process communication via a publish/subscribe middleware, defining complex actions and data that can be published via different nodes to subscribers, some specifically supporting HRI [9]. Subscribers can then handle the information they receive and provide services on their own. `ROS` also provides bindings in Common Lisp, C++, and Python, which are popular languages for programming robotics, but they have also been steadily updated, so one needs to have more recent versions of Python or a compiler toolchain to continue to run the latest tools.

Supporting older robots on newer `ROS` stacks can become an issue due to incompatibilities in toolchains, libraries, and language definitions (e.g. differences in C/C++ standards or Python versions) between the robot's operating system and existing software stack and modern toolchains. This might result in excessive porting and is only possible if the original source code is available. Alternatively, one can introduce inter-process communication or create nodes on a more modern system to work as a bridge between newer and older systems, but this also adds latency and extra complexity in the system. Tools like `rosbridge` [3] aim to provide such a solution but are often limited in application as they still require modern compiler toolchains to ensure consistency between data types and are hence unsuitable to provide an interface to very outdated or non-standard systems.

At the lower level, communication between services is commonly realised using network and local sockets, which are often basic components of operating systems. Using sockets directly requires

HRI '26, March 16–19, 2026, Edinburgh, Scotland, UK

ACM ISBN 978-1-4503-XXXX-X/2018/06.
https://doi.org/XXXXXXX.XXXXXXX

that you also handle connections, define protocols, and marshal the data between services. While this offers great flexibility, it requires strong skills in architecture and design, programming, and testing—especially if different programming languages are involved.

Communication frameworks like the Spread Toolkit [13] and ZeroMQ [14] that are shipped in most operating systems provide a higher-level interface to sockets while being light on requirements. Such frameworks help with connections and marshalling primitive data like numbers and strings, but not with complex data structures. Thus, additional protocol work is still required.

With the ZTL Task Library (ZTL), we establish a novel framework and low-level library that is light on requirements and can be made widely available, including older and more exotic systems. Dependencies are restricted to ZeroMQ [14] as the underlying communication handler and a YAML [2] interpreter[1] to provide simple but user-friendly scripting and configuration functionalities. Using ZeroMQ for communication allows ZTL to wrap primitive sockets to handle setting up connections and sending and receiving data as atomic instructions. ZeroMQ also provides bindings to over 30 programming languages and good compatibility between different library versions, thus sparing users marshalling work and much of the other drudgery of low-level protocol handling. On top of ZeroMQ, ZTL provides a simple interface for task lifecycle management, inspired by ROS's actionlib [12] and Task-State Patterns [6]. However, it does not guarantee type safety or defined data types. Instead, message content is modelled solely in YAML to provide flexibility between and independence from specific datatypes used in the underlying robot control software.

2 PURPOSE

The purpose of ZTL is to enable light-weight and widely compatible remote task execution by providing a task protocol on top of ZeroMQ that encapsulates YAML data. For that, we present a communication architecture, a simple message exchange protocol and a notation to model task lifecycles.

Basic communication is thereby modelled as remote procedure calls relying on three core components: (i) a *client* that can initiate remote tasks, query the task's current state and outcomes; (ii) a *server* that dispatches the task specification to a *controller* that manages the task lifecycle (described below, see Fig. 3) which is then started and monitored by an *executor*; and (iii) a *handler* that will then act upon the task description. All three components are independent of each other, but it is generally assumed that the server and handler components exist within the same ecosystem, typically operating a robo, while interactions between client and server are realised via ZeroMQ socket communication to allow the client to solely rely on ZTL for operating the target system.

Figure 1 depicts an overview of the architecture, with the client-side class RemoteTask plus server components TaskServer, and TaskExecutor as core parts of ZTL that can be instantiated and executed. In contrast, some parts of the server, i.e., TaskControllers and ExecutableTasks, must be inherited and implemented specifically for the target hardware so that task specifications can be interpreted and executed locally in their native environment. Controllers simply have to implement initialisation, status, and abort methods, while tasks need to provide an execute and abort function to support the task lifecycle outlined below.

[1] https://pypi.org/project/oYAML/

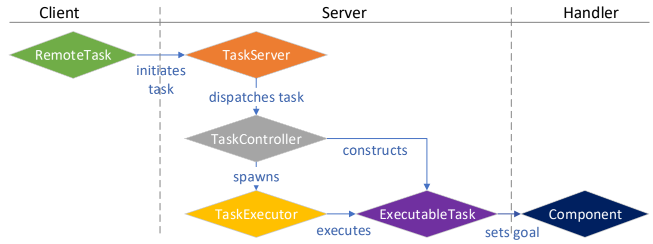

Figure 1: ZTL architecture overview, displaying client (green), server (amber, grey, yellow, purple) and handler (blue) components.

The message exchange protocol between the *client* and *server* components requires the specification of a *scope* (similar to ROS topics) to support message dispatching as well as the description of the task, including an ID, the task's current state, and a payload containing the task specification or results. Figure 2 gives an example task communication with a message sent from a client (Fig. 2a) and the corresponding reply from the server (Fig. 2b). Note that while this example shows a structured payload (described below), the protocol allows for arbitrary payloads to be exchanged. However, it explicitly requires metadata describing the task lifecycle. Moreover, while the client initiates a task, its ID is specified by the server in the controller component, which is managing the lifecycle of a task.

scope	state	id	payload
/example	INIT	<empty>	

handler	component	goal
cob	head	left

(a) Example of an initialisation request. A client aims to initialise a task at a server, specifying a dispatcher that would listen to the scope */example*.

scope	state	id	payload
/example	INITIATED	1	cob:head:left

(b) Example Server reply to initialisation request confirming that the task has been received and dispatched to a handler component.

Figure 2: ZTL Protocol specification

In ZTL, we model a task's lifecycle from a client's requests to a server, its active runtime, and terminal states as displayed in Figure 3. After a client requests to INIT (initiate) a task by sending a task specification payload and the server indicates a successful handshake via the INITIATED state, the server will try to invoke the task at the handler, setting its state to ACCEPTED or deny its execution by replying with a REJECTED state, according to the handling component's response to the task specification given in the payload. A task can then be terminated by the client, sending an `ABORT` signal, resulting in the `ABORTED` final state. Otherwise, the server can communicate the end of a task as either `COMPLETED` or `FAILED`, depending on its outcome determined by the handler.

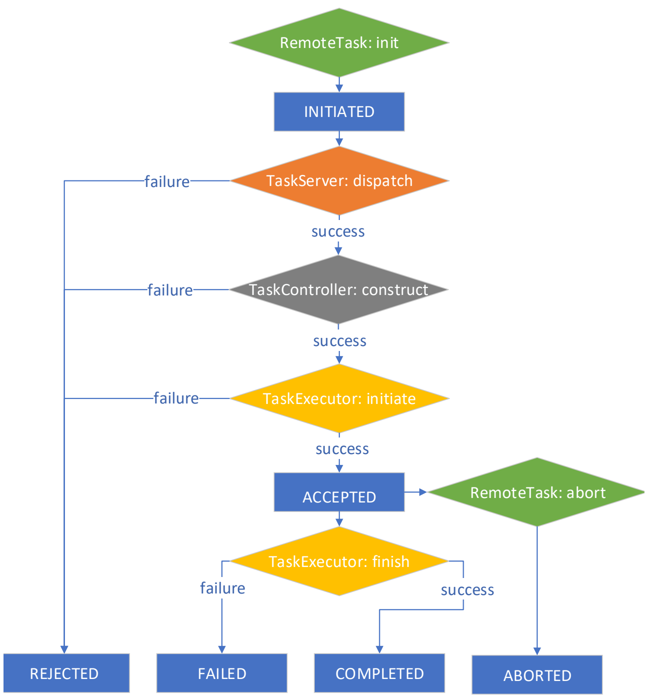

Figure 3: Task lifecycle as modelled in ZTL. States are depicted in blue; signals sent by the client are indicated in green, and server components in amber, grey and yellow.

To enable interpretable behaviour design for HRI together with non-programming experts, `ZTL` also provides a `Task` specification, allowing clients to present a structured payload with a `handler`, `component`, and `goal`. When controlling robots, such a payload can be dispatched by a server to send goals for specific components of individual robots. In Figure 2a, for example, the client requests Care-O-bot (`cob`) to move its head to the `straight` position.

Moreover, `ZTL`'s scripting engine allows designers to specify sequences of tasks to be triggered at multiple servers from a single client, expressed in a `YAML` file. We define a `scene` as a sequence of `steps` that each can contain multiple `handlers` with a list of actions (`components` and their `goal` specifications, all to be triggered in parallel). Scenes, steps, and actions can be monitored for completion or delayed if required, as indicated in `scene` below.

```
scene:
  step(wait = bool; delay = int):
    handler:
      component: goal
```

In the example scene below, the second step `greet` is executed two seconds after the `initialise` step without awaiting its completion. The steps contain actions for `handlers` controlling `cob` and `fetch` robots. Their goal specifications, which need to be interpreted by the handler, trigger their text-to-speech engine (`tts`), `arm` or mobile `base` components. Here, they are given as an $[x, y, \theta]$ position for `cob`'s base, goals to look up for `fetch`'s arm and base, a text for `cob` to say and a command triggering a routine in its arm.

```
example scene:
  initialise(wait = False):
    cob:
      base: [0, 0, 0]
    fetch:
      base: serving_position
      arm: tucked_position
  greet(delay = 2):
    cob:
      tts: "Hello and welcome to HRI!"
      arm: wave
```

3 CHARACTERISTICS

Our proposed solution aims to strike a balance between facilitating inter-process communication on a high level whilst offering most of the flexibility of lower-level solutions. It is designed alongside the principles of **versatility and portability**, i.e. the ability to use it on a wide range of systems and the potential to encapsulate existing middleware protocols. Moreover, the resulting protocol should be **easy to interpret** for expert programmers and non-experts alike.

For end-users, our approach effectively separates runtime and development environments and thereby facilitates rapid prototyping by providing the following benefits:

(1) Interactive behaviours can be largely implemented solely relying on the `ZTL` library. Since `ZTL` itself is low-level and light on requirements, it is available for most operating systems, allowing developers to use their familiar work environments while the containerised robot control or simulation can be installed, for example, on a more powerful machine.
(2) The simple, task-based communication protocol provided by `ZTL` allows for hardware abstraction and simulation and thus facilitates working without any robot and makes drop-in replacements easier, for example, when the project focus shifts or hardware becomes unavailable.

So far, `ZTL` has been used as a way to communicate between different Python versions and to communicate between multiple robots with different hardware and operating systems (a NAO 6 and a Misty II) [1]. The framework also facilitates communication between a modern virtual reality device (Meta Quest Pro) and a simulated fetch mobile manipulator to enable the experimentation of a project to facilitate interaction fluidity between humans and robots [4]. `ZTL` further supports the simultaneous operation and demonstration of multiple robot platforms (Pepper, Fetch, Care-O-bot 4, ...) and smart actuators supporting ongoing research projects like Hospital@Home, HRI experimentation [11], and public live presentations[2] in the University of Hertfordshire's Robot House.

[2]https://robothouse.herts.ac.uk/news/robot-lab-live-2023/

4 CODE/SOFTWARE

ZTL is currently available in Python versions 2 and 3 and can be installed, ideally in a virtual environment[3], via the Python Package Index[4] using pip, see README.md for details. The source code is also hosted publicly at https://gitlab.com/robothouse/rh-user/ztl and can be used under the Simplified (2-Clause) BSD License. It is organised into core classes and scripting functionality to easily trigger a series of tasks. ZTL further provides example files, and testing classes using pytest (not displayed).

```
src
├── core
│   ├── client.py
│   ├── protocol.py
│   ├── server.py
│   └── task.py
├── example
│   ├── sample_conf.yaml
│   ├── sample_script.yaml
│   ├── simple_client.py
│   ├── simple_server.py
│   ├── task_client.py
│   └── task_server.py
└── script
    └── run_script.py
```

5 USAGE NOTES

The ZTL package contains example server and client scripts that can be used to verify correct installation and to explore the protocol. First, start the server component, here on port 12345, listening to tasks on scope /test:

```
> ztl_task_server -p 12345 -s /test
```

The output should indicate that the server is listening at the specified port and scope. In another terminal, execute the client to send a task to the server on the local machine as follows:

```
> ztl_task_client -r localhost -p 12345 -s /test \
  some-handler:executing-component:goal-state
```

Upon successful communication, the example task will execute while the server and client output details about the communication between them. After approximately five seconds, the client will report completion of the task as finished.

The following example can be used to demonstrate ZTL programmatically in Python, where a TaskServer from ztl.core.server redirects request to a controller called SomeController:

```
server = TaskServer(12358)
server.register("/robot", RobotController())
server.listen()
```

The RobotController has to implement init(), status(), and abort() to create tasks for each handler, report their status, and offer aborting them, according to the protocol (Fig. 3). A TaskExecutor then runs and monitors the task in a thread and reports exceptions. When given 'xbot' as a handler, they might spawn the following ExecutableTask to control a ROS-based robot's kinematic joints:

[3]https://docs.python.org/3/library/venv.html
[4]https://pypi.org

```
class RobotKinematics(ExecutableTask):
  def __init__(self, goal):
    self.j_pub = rospy.Publisher("...", \
      sensor_msgs.JointState, queue_size=1)
    self.j_pos = check_positions(goal)
  def execute(self):
    msg = sensor_msgs.JointState()
    msg.position = self.j_pos
    self.j_pub.publish(msg)
```

Sending a request to this server from a client requires importing protocol.Task and client.RemoteTask from ztl.core. The client can then abort or query the status of the task by calling abort(), status(), or wait for it to finish as follows:

```
task = RemoteTask("localhost", 12358, "/robot")
request = Task.encode("robot x", \
  "kinematics", [-3, 14, 1.5, 9, -2])
task_id, reply = task.trigger(request)
state, reply = task.wait(task_id, timeout=5)
```

5.1 Limitations

Operating legacy hardware and software comes with many security risks that are outside of the scope of the provided solution, in the same way that many security aspects are outside the scope of ROS, for example. Despite ROS2 and ZeroMQ providing authentication and encryption mechanisms [5, 7], adopting those might lead to compatibility issues (e.g. server and client need to support the same protocols) and other risks of running a robot with vulnerable legacy software remain. Users should thus keep in mind that, while ZTL makes it possible to work with them, they should exercise caution and only use older robots in controlled or isolated environments.

Our solution already offers a useful set of strong support mechanisms for developing and controlling high-level HRI behaviours, in particular for Wizard-of-Oz-based experimentation and demonstrator setups, and when designing robot behaviours with lay persons. However, we are still working on integrating lower-level protocols, specifically when it comes to processing sensory data from a robot, such as spoken language (audio signals) or people perception (video signals) via socket communication or a structured integration with virtualisation software with bi-directional data processing to enable digital twinning and simulated user interaction.

6 CONCLUSION AND FUTURE WORK

This paper introduced ZTL, a freely available library providing a simple server-client architecture to decouple robot hardware from higher-level control logic by implementing a simple, task-based protocol for robot control. Our software has efficiently extended the lifetime of robot hardware and facilitated the co-design of robot behaviours with non-programming experts in multiple contexts. Essential future work, integrating lower-level protocols to stream sensory data from robots to clients, will enable instrumental aspects of bi-directionality and widen the applicability of our approach.

ACKNOWLEDGEMENTS

We kindly acknowledge financial support of the Dinwoodie Charitable Company for the Hospital@Home project and the Research Council of Norway as part of the ROSA project (Grant No: 321821).

REFERENCES

[1] Claudia Andrea Badescu. 2024. *Teach Me How To Dance: An Exploration of Motion Translation Methods Using the ZTL Framework for Use with Children with Autism in Educational Settings*. Master's thesis.

[2] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. 2009. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008* 5, 11 (2009).

[3] Christopher Crick, Graylin Jay, Sarah Osentoski, Benjamin Pitzer, and Odest Chadwicke Jenkins. 2016. Rosbridge: Ros for non-ros users. In *Robotics research: The 15th international symposium ISRR*. Springer, 493–504.

[4] Carlos Baptista De Lima, Julian Hough, Frank Förster, Patrick Holthaus, and Yongjun Zheng. 2024. Improving Fluidity Through Action: A Proposal for a Virtual Reality Platform for Improving Real-World HRI. In *International Conference on Human-Agent Interaction (HAI 2024)*. ACM, Swansea, UK. https://doi.org/10.1145/3687272.3690881

[5] P Hintjens. 2013. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media.

[6] Ingo Lütkebohle, Roland Philippsen, Vijay Pradeep, Eitan Marder-Eppstein, and Sven Wachsmuth. 2011. Generic middleware support for coordinating robot software components: The Task-State-Pattern. 2 (01 2011).

[7] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. 2022. Robot Operating System 2: Design, Architecture, and Uses in the Wild. *Science Robotics* 7, 66 (May 2022), eabm6074. https://doi.org/10.1126/scirobotics.abm6074

[8] Giorgio Metta, Paul Fitzpatrick, and Lorenzo Natale. 2006. YARP: yet another robot platform. *International Journal of Advanced Robotic Systems* 3, 1 (2006), 8.

[9] Youssef Mohamed and Séverin Lemaignan. 2021. ROS for Human-Robot Interaction. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 3020–3027. https://doi.org/10.1109/IROS51168.2021.9636816

[10] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[11] Lewis Riches, Kheng Lee Koay, and Patrick Holthaus. 2024. Evaluating the Impact of a Personal Data Communication Policy in Human-Robot Interactions. In *The Seventeenth International Conference on Advances in Computer-Human Interactions (ACHI 2024)*. IARIA, Barcelona, Spain, 123–128. https://www.thinkmind.org/index.php?view=article&articleid=achi_2024_3_100_20052

[12] Higor Barbosa Santos, Marco Antônio Simões Teixeira, André Schneider de Oliveira, Lúcia Valéria Ramos de Arruda, and Flávio Neves. 2017. *Control of Mobile Robots Using ActionLib*. Springer International Publishing, Cham, 161–189. https://doi.org/10.1007/978-3-319-54927-9_5

[13] Spread Concepts LLC. 2016. The Spread Toolkit. http://www.spread.org/.

[14] The ZeroMQ authors. 2023. ZeroMQ. https://zeromq.org/.

[15] Johannes Wienke and Sebastian Wrede. 2011. A middleware for collaborative research in experimental robotics. In *2011 IEEE/SICE International Symposium on System Integration (SII)*. IEEE, 1183–1190.