TeMoto: A Software Framework for Adaptive and Dependable Robotic Autonomy with Dynamic Resource Management

ROBERT VALNER\textsuperscript{1}, VEIKO VUNDER\textsuperscript{1}, ALVO AABLOO\textsuperscript{1}, MITCH PRYOR\textsuperscript{2}, and KARL KRUUSAMÄE\textsuperscript{1}

\textsuperscript{1}University of Tartu
\textsuperscript{2}University of Texas at Austin

Corresponding author: Robert Valner (e-mail: robert.valner@ut.ee).

This research was in part supported by Los Alamos National Laboratory, a target grant COVSG24 funded by Estonian Research Council, IT Academy Programme for ICT Research Development funded by Estonian Ministry of Education and Research, CHIST-ERA project InDex and Estonian Centre of Excellence in IT (EXCITE) funded by the European Regional Development Fund.

ABSTRACT
For widespread deployment of robots in challenging environments (fire fighting, search and rescue, planetary exploration, etc), the software of the robot must allow for reliability and adaptability. For many existing systems, an unexpected change in mission specification, component failures, or energy conservation requires downtime for adaption (redesign of mission logic, switching sensor data processing pipeline, etc.). This is because software and hardware components for robotic applications are commonly chosen or designed based on task requirements and integrated either directly in the source code or via system configuration scripts, such as ROS launch files, leading to a fixed monolithic design. As the necessity and extent of adaptive behaviors is not always known prior to deployment, the structure of a robot’s software needs to support it by design. In this paper, we propose TeMoto, a novel architecture for adaptive autonomous robots, and a ROS-based framework of openly available software tools that implement the TeMoto architecture. TeMoto is a developer tool which combines dynamic (run-time) task and resource management, encourages modular and scalable system design and is task and platform agnostic - TeMoto provides the foundation for an adaptive robotic system. The feasibility of the TeMoto framework is qualitatively assessed via experiments on single and multi-robot setups spanning common scenarios (teleoperation, autonomous surveillance, cargo delivery, etc.). TeMoto-based systems exhibit increased fault tolerance and dynamic reconfigurability of software and hardware resources, as well as up to a 47% reduction in power consumption compared to the non TeMoto-enabled reference setup.

INDEX TERMS Fault tolerance, Long term autonomy, Reconfigurable architectures, Redundancy, Robot programming, ROS.

I. INTRODUCTION
THE diverse application domain of partially and fully autonomous robots (elderly care \cite{1}, flexible manufacturing \cite{2}, fighting wildfires \cite{3}, space exploration \cite{4}, handling hazardous waste \cite{5}, etc.) pose an equally diverse set of technical challenges for the robotics developers. A robots’ hardware and software can both be viewed as an integration of smaller modules or components. On a component level, many technical challenges (mapping, localization, communication, etc.) are common and share widely accepted software development practices and tools that simplify the development process \cite{6}. But component integration is often domain or application specific \cite{7}, which works well when prototyping or working with a well-defined and unchanging problem, such as product assembly. Yet autonomous robots are envisioned to operate in unstructured and dynamic environments, fulfill different objectives that may change over time, cooperate with humans or other robots, and cope
with potential hardware and software limitations (power constraints, etc.) or malfunctions - aspects which are still bested by human dexterity. Thus the research in robotic autonomy architectures is facing many open challenges, which commonly imply that an autonomous robot must be able to adapt. This applies to all aspects of robot development, including:

- **Adaption of design** - or ease of redesign, maintenance and feature scalability. Adapting an existing system to new domains promotes reuse and scaling it with new features should not require a complete redesign.

- **Adaption during deployment** - or run-time reconfiguration and task scalability. A deployed autonomous robot should be adaptable both in terms of mission objectives and failures in hardware or software.

It is difficult to foresee which set of architectural design principles lead to adaptive autonomous robots that efficiently cover the aforementioned design and deployment considerations. However, it is safe to assume that task specifications and requirements can change [8], the deployed hardware and software can malfunction [9], [10], computational and power resources are limited, and robotic technology evolves [11]. Thus these assumptions form the minimal set of criteria a robot software architecture must be based on.

In this paper, we present TeMoto (Fig. 1), a novel architecture for adaptive autonomous robots, and a ROS-based [12] framework of openly available\(^1\) software tools that implement the TeMoto architecture. TeMoto helps to solve several recurring problems and design challenges, such as:

- **Dynamic task management**: TeMoto separates mission strategy related code from functional resources, such as sensors and actuators. A task consists of modular actions that implement a specific behaviour, e.g., a sensing or a navigation action. Actions can contain any arbitrary user defined code and can start, stop and access resources. Actions can be connected into sequential, concurrent and cyclical graphs, thus allowing the user to implement arbitrarily complex run-time behaviours (see Experiment 3 in section IV-D).

- **Dynamic resource management**: Existing components (ROS nodes or other executables), i.e., resources such as a camera, lidar, network or CPU intensive algorithm can be programatically started, stopped and monitored for failures (see Experiment 1 and 2 in sections IV-B and IV-C), which are reported to all consumers of a resource. TeMoto provides an accounting mechanism for resources (the Resource Registrar or RR), embedded into all subsystems of TeMoto, which mediates resource queries. Thus, the RR knows how many consumers a resource has and what are its sub-resource dependencies, which is important when propagating resource failure messages to all consumers or allocating (mitigating multiple allocations) and de-allocating (making sure that there are no consumers left) resources.

- **Minimal development overhead**: TeMoto does not require any resource customization, i.e., existing ROS packages can be used via TeMoto without modification. Similar frameworks assume specific behaviour from resources in order to be used within the framework (overview in section II-B). Having minimal development overhead is important for smooth adoption and maintenance because the users do not need to fully commit their project to a specific framework and its invasive requirements.

- **Modular design**: While TeMoto contains a number of subsystems (ROS nodes), each subsystem (maintained in separate repositories\(^1\)) has minimal dependencies. This allows the robotics community to adopt only the subsystems that matter for their project and keeps code bloat at a minimum.

TeMoto is a continuously developing project and the major contribution of this paper is to establish the fundamental design aspects of TeMoto, provide proof-of-concept demonstrations, and discuss the long-term challenges TeMoto addresses. To demonstrate the features and potential benefits of integrating TeMoto, we qualitatively compare it with related works and showcase the technical capabilities in three different experiments. Parts of this work (the UMRF notation [13] and the Resource Registrar [14], [15]) are covered in more detail by our other publications. This paper extends and integrates our previous work thus completing the architecture of TeMoto.

The next section reviews contributions in the domain of adaptive software architectures for autonomous robots. Section III presents the general structure of TeMoto and describes each subsystem. A series of hardware demonstration experiments are completed and analyzed in Section IV. Section V summarizes the results, key contributions as well as ongoing and future development.

## II. RELATED WORK

Different architectural layers, e.g., task and resource management (Fig. 1), have been the focal point of numerous research efforts. For example ROSPlan [16], SERA [7] and MoBar [17] are mostly focusing on adaptive task management via deliberation techniques. On the other hand, DyKnow [18], Metacontrol [19] and Rorg [20] focus on fault tolerant resource management. The aforementioned approaches showcase the capabilities of a specific layer [16], [19], [21], [22] or robustness in specific task domain [20], [23], but do not combine task and resource management in a reusable framework for different tasks and hardware configurations. In this section, we evaluate the existing literature from the perspective of deployability as a task agnostic framework in applied robotics. In Section II-A we describe the evaluation criteria. In Section II-B each framework is then analyzed against the proposed criteria (summarized in Table 1).

---

\(^1\)https://github.com/temoto-framework
A. QUALITATIVE EVALUATION CRITERIA

For long-term deployment and reliability of autonomous robots, adaptability on the component (sensors, actuators, algorithms) and mission-level is a key enabling factor [8], [19], [21], [24], [25]. Thus a viable software framework for developing long-term autonomy in robots should have task management, dynamic resource management, and task agnostic features:

- **Dynamic task management**: Does it support or have integrations to a task management systems for dynamic deployment and reconfiguration of a robot’s mission?
- **Dynamic resource management**: Can its components, i.e., resources, be dynamically started, stopped and associated with other resources as a dependent sub-resource?
- **Task agnostic**: Is the framework for general use or is it designed for a specific task or robot’s configuration?

Next, while specific architectural design principles may be promising on paper, the practicality of the implementation ultimately determines its impact on the robotics community and ease-of-use as a tool. Thus the following metrics can be considered to describe the reusability properties of the framework:

- **Minimal development overhead**: Can third party components, managed or controlled by the framework, be used without requiring customization?
- **Publicly available**: Are the source materials or binary packages of the framework’s implementation publicly available so that it could be used by the robotics community?
- **Clean implementation**: Is the publicly available implementation reusable, i.e., does not contain any task or hardware platform specific code artefacts?
- **Applied middleware**: Middleware support for ease of
adoption by the robotics community.

B. COMPARISON OF EXISTING FRAMEWORKS

This section summarizes each framework from Table 1.

Brunner et al. [26] combine RAFCON [37] based task management with Links-and-Nodes [26] based dynamic resource management. The RAFCON tasks are augmented with a resource dependency system, where each task can define a data, world state information, or a computational resource dependency prior to the execution. However, little is disclosed on how the Links-and-Nodes (LN) based resource management works and its limitations. The only publicly available implementation is for RAFCON, which covers just the task management aspect (excluding the resource dependency augmentations). Also, resources are globally managed, meaning that resources cannot dynamically depend on sub-resources (see section III-A for an illustrative example).

ROSPan [16] and MoBar [17] are frameworks that integrate Planning Domain Definition Language (PDDL) based planning with ROS (and with Generator of Modules or GENOM in case of MoBar) and streamline the use of PDDL in robotics. Both ROSPlan and MoBar however implicitly assume that low-level robotic modules (navigation, manipulation, etc.), i.e., the functional layer, are always available, or in other words, functional resources are not managed and failures are not recoverable on the functional layer.

PLEXIL-ROS [22] is an architecture that combines the Plan Execution Interchange Language (PLEXIL) with ROS. PLEXIL-ROS supports navigation tasks and allows for the utilization of formal analysis and verification tools developed for PLEXIL. However, PLEXIL-ROS has no resource management and it supports tasks only in the navigation domain. Thus any adaptive behaviors can be realized on the planning layer, but implementation details are not provided and there is no publicly available source code.

SERA [7] is an architecture for decentralized multi-robot collaboration. SERA implements its task management via central control stations which distribute the collaborative task (described via linear temporal logic) to individual robots. Each robot’s software layout is segregated in to mission and change management layers. The former interprets the high level goal where the latter refines the goal according to the feedback provided by the component control layer. The component control layer is the biggest architectural limitation of SERA, as it is not generic and assumes a fixed layout of a robot’s components (a combination of navigation, manipulation, gesture recognition, localization, etc modules). It has no resource management and adaption on the component level cannot be achieved. The linear temporal logic based task definitions are structurally rich but lack support for cyclical behaviors. Finally the open source implementation of SERA is task-specific and does not promote reuse.

SOTER [30] architecture combines state machine programming language and code executor called P [38] the whole framework is implicitly designed for navigation tasks. The openly available implementation is demo specific and unusable as a generic framework. Also SOTER does not manage resources, i.e., all robotic components are started up separately which the custom runtime assurance modules send commands to. Thus a critical failure in components layer or Robot SDK layer (as described by the authors [30]) renders the robot unrecoverable. Finally the tasks are not dynamically invocable, i.e., tasks are precompiled and executed as a single application.

The STRANDS [21] project focuses on long term deployment of an autonomous service robot for navigation related tasks. The architecture of STRANDS is capable of task and resource management and the publicly available implementation is well organized and promotes reuse. However, the task management is limited to sequential behaviors; tasks are implemented as ROS action servers and run constantly in the background. The framework is designed for the navigation tasks and the resource management is limited to just restarting ROS nodes.

Osmosis [23] is an architecture for fault tolerant navigation applications. Effectively, Osmosis has a task management layer that allows users to define and switch between navigation missions and a fault-tree based fault management layer where the user can implement custom fault detection and recovery routines. The publicly available implementation of Osmosis does promote reuse with well structured software packaging, but it is limited to navigation tasks. Also the dynamic recovery of failed components is mentioned by the authors [23] yet the implementation details are undisclosed and do not show in the publicly available source code.

Rorg [20] addresses several key challenges in long-term deployment, such as backward and forward compatibility for data logging, secure communication, and dynamic resource management via containerization. While the work provides valuable lessons-learned, the implemented software stack is designed for the hardware layout described in [20] and lacks a methodology for defining tasks other than its original mission.

DyKnow [18] is a resource management level tool that allows robots to dynamically start, stop and join resources into pipelines/streams. DyKnow is limited to customized resources (every ROS node must use a specialized DyKnow nodehandle [18]), making it difficult to reuse existing tools (e.g., other ROS nodes, such as camera drivers).

Metacontrol [19] is a resource management level tool which allows users to define a robot’s software configurations and use them to reconfigure the robot during run-time to overcome undesired conditions (resource failures, environmental restrictions, etc.). Metacontrol however does not allow users to define new configurations for a deployed/running system; scale to concurrent configurations and resource usage overlaps, where independent tasks can simultaneously access the same resource; work with third party resources without prior modification that fulfills the System Modes [39] design.
The comparison of the related work in Table 1 shows that the TeMoto framework, by providing full stack of tools from dynamic task to resource management (Fig. 1), addresses all architectural and implementation requirements for creating adaptive robotic solutions. None of the compared frameworks alone allow combining all the features (Table 1). They either lack dynamic task or resource management, are not usable for tasks other than initially designed for (not task agnostic), or require customization of managed resources (lack of minimal overhead). While some customization is technically plausible depending on the system requirements, it increases the maintenance and development effort especially for keeping the customized resources up-to-date.

### III. TEMOTO ARCHITECTURE AND IMPLEMENTATION

TeMoto is a software toolkit which enables developers to design adaptive and resilient robotics applications. Commonly, a robotic (ROS-based) application is built for a specific task, such as autonomous navigation in a warehouse. Hardware (LIDAR, cameras, etc.) and software (localization, motion planner, mission logic, etc.) components are chosen or designed based on task’s requirements and integrated via system configuration scripts (e.g., ROS launch files). Yet a change in mission specification, component failures, or energy conservation would require a downtime for adaptation (e.g., redesign of mission logic, switching sensor data processing pipeline, powering down unused components). This may not be feasible in time-critical scenarios, i.e., fire fighting, or in environments with restricted access, such as disaster areas.

TeMoto’s architecture (Fig. 2) builds upon above described component-based applications by adding a functionality to dynamically control the components (start, stop, monitor, combine), i.e., resource management, and define tasks which control the execution flow of the robot, i.e., task management. Thus the application can be organized through tasks, where the individual user-defined and reusable actions within the task (navigation, manipulation, etc) can request, access and stop components when required.

Each action is described using UMRF markup language (Fig. 3), which allows invoking actions via any system or command interface that outputs commands in UMRF [13]. Additionally, UMRF allows describing parent-child relations between actions, thus forming a graph or UMRF graph, which is used to invoke actions in a particular order.

The resource managers fill in the gap between the tasks and components by implementing the process of allocating and deallocating a specific component (i.e., a resource). Because a resource can be requested by multiple actions and could be dynamically combined from multiple sub-resources, an accounting and error propagation system, Resource Registrar (RR), is used inside every resource manager and its interface.

The Action Engine, i.e., the run-time system for task management, and all individual resource managers (see Table 2 for an overview) are implemented independently by design and thus developers can selectively adopt the tools that matter for their project (clean and modular design). Furthermore, the core tools, e.g., Resource Registrar or Action Engine, are internally implemented without relying on ROS libraries, which are then wrapped with a ROS-based API. Thus, these tools can be adopted with relative ease by middleware systems other than ROS. TeMoto requires no customization of the managed components (minimal overhead) and task management is not constrained to any specific application domain (task agnostic). All tasks and components can be controlled during run-time or dynamically, thus forming the foundation for an adaptive robotic system.

Table 2 summarizes the available tools of the TeMoto framework, which are further described in the following sections. The RR and Resource Info Synchronizer are explained in sections III-A and III-B followed by a description of resource managers in Section III-C. Section III-D summarizes

---

**TABLE 1. Comparison of adaptive robotic frameworks.**

| Framework        | Architectural Features | Implementation Features |
|------------------|------------------------|-------------------------|
|                  | Dynamic Task Management| Dynamic Resource Management| Task Agnostic | Minimal Overhead | Publicly Available | Clean Implementation | Applied Middleware |
| Brunner et al. [26] | ✓                      | ✓                       | ✓                | unclear          | ✓                  | ✓                    | ROS, LN           |
| ROSPlan [16]     | ✓                      | ✓                       | x                | ✓                | ✓                  | ✓                    | ROS               |
| MoBar [17]       | ✓                      | ✓                       | x                | ✓                | ✓                  | ✓                    | ROS, GENOM        |
| PLEXIL-Ros [22]  | ✓                      | ✓                       | x                | ✓                | ✓                  | ✓                    | ROS               |
| SERA [7]         | ✓                      | ✓                       | limited          | ✓                | ✓                  | ✓                    | ROS               |
| Osmosis [23]     | limited                | limited                 | limited          | ✓                | ✓                  | ✓                    | ROS               |
| STRANDS [21]     | limited                | limited                 | limited          | ✓                | ✓                  | ✓                    | ROS               |
| Rorg [20]        | limited                | ✓                       | ✓                | ✓                | ✓                  | ✓                    | ROS, gRPC         |
| DyKnow [18]      | x                      | ✓                       | ✓                | ✓                | ✓                  | ✓                    | ROS               |
| Metacontrol [19] | x                      | ✓                       | ✓                | ✓                | ✓                  | ✓                    | ROS, ROS2         |
| TeMoto           | ✓                      | ✓                       | ✓                | ✓                | ✓                  | ✓                    | ROS               |
how actions are described, implemented, and executed.

A. RESOURCE REGISTRAR

The Resource Registrar (RR) is the core component of every TeMoto subsystem that provides or requires resources (a resource manager or an action). RR implements the functionalities necessary during resource acquisition since all resources have common allocation, usage, and deallocation properties. RR tracks how many clients are using a resource and its sub-resource dependencies, which makes RR a central
component of every subsystem that provides or requests resources. More precisely, the RR addresses the following challenges [14], [15]:

- How to share resource(s) with multiple clients without allocating the same resource multiple times?
- How to assure that no clients are using the resource when deallocating it?
- How to notify clients if the status of a resource changes, e.g., the resource has failed?

To illustrate the need for an RR, consider an example scenario (Fig. 4) where a user-defined action requests a resource (filtered camera image) composed of multiple sub-resources: a camera and a filter algorithm. In this example, two resource managers are implemented: the Component Manager and the Process Manager. The former maintains the knowledge about sensors, algorithms, etc., and their combinations. The latter can start, stop, and monitor OS processes on request (see Table 2 for a more detailed description). First, the action (Action_1 in Fig. 4) requests for the filtered camera (Fig. 4a) from Component Manager, which knows that this resource consists of two sub-resources. Hence the Component Manager requests the Process manager to start the camera

| Layer            | Tool                        | Description                                                                                                                                                                                                 |
|------------------|-----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| TASK Management  | UMRF                        | Unified Meaning Representation Format (UMRF). A platform independent task description format based on JSON notation. Allows defining tasks as graphs of interconnected actions, described via parent/child relations. Supports sequential, concurrent and cyclical graphs and parametrization, i.e., actions can accept and produce data. |
|                  | Action Engine               | A task management system that executes tasks based on UMRF descriptions. Tasks can be started, stopped and modified during runtime.                                                                                |
|                  | Action Designer             | A graphical tool that streamlines the process of designing UMRF-based TeMoto actions.                                                                                                                        |
| RESOURCE Management | Resource Registrar (RR)   | A software module which adds accounting (use count, dependency management) functionalities to services that load/unload resources. Allows sending resource status information (failure notifications, etc) to resource consumers. Supports OpenTracing-based distributed tracers for logging distributed resource queries and status info propagation. Essential part of every resource manager. Restorable from critical failures via back-up of the internal state. |
|                  | Resource Info Synchronizer (RIS) | A software module that streamlines the process of sharing and synchronizing databases between resource managers.                                                                                           |
|                  | Process Manager             | A ROS node, which can dynamically invoke programs, including ROS executables, ROS launch files and regular executables. Running programs are then monitored for process failures (via segmentation faults, unexpected shutdowns, etc), which are reported to the consumers of the failed resource. |
|                  | Component Manager           | A ROS node, which maintains information about components (including published/subscribed topics, package names) and can dynamically compose them based on component templates. Components are ROS based programs (nodes/launch files) which provide sensing or data processing functionalities. A component template outlines the structure of a composed component, e.g., a combination of a manipulator arm and a force-torque sensor with compliance controller. Templates can then be used to dynamically combine individual components and replace faulty ones. Components are invoked via Process Manager. RIS is used to share component info with other Component Manager instances for collaborative resource sharing (see section IV-C). |
|                  | Other managers              | Other non-core tools, such as the Visualization Manager and the Context Manager, have been developed to cover more specific use-cases. Visualization Manager uses RViz as the main platform for visualization, where each displayable data type is displayed via dynamically loadable RViz plugins. Thus data can be visualized programmatically on demand. Context Manager maintains a model of the world as hierarchical relations between maps and world objects in a tree structure. The world model is shared with other robots (other instances of Context Manager) via RIS, allowing to build and utilize a common world model in collaborative tasks. |
and the filter processes (ROS nodes). Next, another action (Action_2 in Fig. 4) independently requests for the same filtered camera stream (Fig. 4b), but since this resource has already been allocated, the Component Manager can return the same response as for the Action_1 and track that two clients are consuming this resource. Finally, the camera fails unexpectedly and all consumers of this resource are notified (Fig. 4c), which in this case is just the Component Manager. The Component Manager on the other hand knows that Action_1 and Action_2 consume a resource that depends on this failed sub-resource and thus notifies them. It is up to Action_1 and Action_2 to implement a suitable response behavior (either terminate, ask for an alternative, invoke any other routines, etc.).

The RR consists of three main parts (Fig. 5a):

- **Resource Catalog** keeps a record of all active in- and outbound resource queries and their inter-dependencies. The catalog is maintained in a serialized form and backed-up on the file system. Thus the state of a resource manager and the RR within can be restored after a critical failure.

- **RR Servers** ($S_0$ to $S_m$ in Fig. 5) process the incoming resource requests. Each server is bound to custom resource allocation and deallocation callbacks implemented inside the manager (Fig. 5).

- **RR Clients** ($C_0$ to $C_n$ in Fig. 5) mediate the outgoing resource requests.

TeMoto’s RR also integrates distributed tracing functionalities via OpenTracing [40]. Distributed tracing is a tool for aggregating complex queries and resource failure reports across multiple subsystems deployed on multiple robots, which are typically logged to a local log-file. A central trace collector then assembles the trace based on individual spans reported by the subsystems involved in the query. Hence, with distributed tracing the chronological sequence of queries and their duration can be used for debugging and performance analysis, which was utilized to analyse the performance overhead of TeMoto in Sections IV-B and IV-C.

### B. RESOURCE INFO SYNCHRONIZER

The Resource Info Synchronizer (RIS) provides tools for sharing resource-related information between robots. When TeMoto is deployed on multiple robots, one instance per robot (Fig. 6a), then each can have the same set of resource managers (see also Fig. 9). RIS allows sharing resource information between different instances of the same manager. This allows one robot to use another robot’s resources, giving an extra level of dependability and flexible use of resources (demonstrated in section IV-C). The collaborative utilization of robotic resources via RIS provides the system developer additional flexibility when designing robotic applications, which in similar scenarios would need to rely on custom and non-standard solutions.

RIS utilizes ROS messages for communicating resource information (RI). A single ROS topic is used to advertise (RI-advertise) and request (RI-request) RI. A callback function is invoked (Fig. 6b) within RIS per each RI message, after which the purpose of the message is checked (RI-advertise/RI-request). RI-advertise messages contain the name of the sender and RI data, which is used to update the RI registry in the resource manager. RI-request messages, on the other hand, indicate that a remote resource manager is requesting RI, which subsequently causes the RIS to get the current state of RI registry and publish it as a RI-advertise message. Both RI-advertise and RI-request procedures can be invoked via RIS API by the resource manager in case the RI database gets updated or the resource manager comes online.

### C. RESOURCE MANAGERS

**Resource Managers (RM)** are a set of subsystems (implemented as ROS nodes) designed to manage a specific resource type. RM’s utilize RR to manage queries and resource dependencies but, as shown in Fig. 7, each RM is responsible for providing the appropriate load, unload callbacks for the RR. Additionally, any other routines that are related to maintaining (searching, updating, etc.) a resource type are embedded in a RM (Fig. 7). In general, a RM is a subsystem for managing an arbitrary resource type and is customized based on application specific requirements. A developer is
**FIGURE 4.** An example scenario demonstrating the recurring challenges (allocation, dependency and status info management) that RR solves. Action_1 requests a resource (filtered camera) that is composed of multiple sub-resources, a camera and a filter algorithm (a). Action_2 independently from Action_1 requests also for the same filtered camera resource without causing another allocation of the sub-resources (b). A sub-resource failure is reported to all consumers in the dependency chain (c).

**FIGURE 5.** Structure of the RR (a). RR depicted as a part of a host subsystem, e.g., Process or Component Manager (b).
free to create or adopt any number of RMs. See Table 2 for an overview of available Resource Managers in TeMoto framework.

**D. ACTIONS**

Actions are code modules that embed developer-defined task logic [13]. Actions accept and provide data, i.e., parameters, which makes an action reusable in different scenarios, e.g., a “navigation” action that accepts the destination coordinates as an input. An action is not a program executed as a standalone process by the OS. It is a plug-in dynamically loaded by a host program, making both able to share system memory for quick data exchange. The Action Engine is a host program that provides dynamic loading and execution of sequential/concurrent actions as well as action loops [13]. Thus a range of tasks can be described and executed by combining primitive and reusable actions. For example, “picking up an object” task can be formed by “locate-object,” “navigate-to,” “move-manipulator-at,” and “grab” actions executed sequentially.

TeMoto actions are implemented in C++ and maintained as shared libraries. Each action is accompanied by a JSON file that contains a description in the UMRF notation. The UMRF outlines input and output parameters and other characteristics that define the action behavior. The UMRF description is used to search for actions that fulfill any specific requirements, e.g., an action that is able to perform a specific process on a point cloud and output the result in a correct form.

Fig. 8 shows how a set of actions are executed in the Action Engine based on a UMRF graph. The Action Engine first receives a UMRF graph via a ROS message that embeds UMRF descriptions as JSON strings. Each node in the UMRF graph is mapped to a respective action by Action Mapper using a reference list of known actions. The list is maintained by the Action Indexer that periodically searches actions from predefined paths. Finally, each action is loaded, instantiated, and executed by the Action Executor in the sequence defined by the UMRF graph.

The TeMoto framework provides the tools and API for starting, stopping and re-configuring tasks via the Action Engine. But TeMoto - by design - omits the control of tasks to the developer who utilizes TeMoto. Such control can be further delegated to an automated task planning system. The UMRF [13] can be used to describe tasks in JSON syntax, decoupling the Action Engine specific API from user’s control application (see Fig. 3).

**IV. DEMONSTRATORS AND EVALUATION**

In this section, we demonstrate and quantitatively show how TeMoto meets the design criteria outlined in Section II. As determined in Section II, a side-by-side quantitative comparison with the reviewed frameworks is impractical as no other framework offers the same set of technical features. To that end, we have developed demonstrators that exemplify and quantify the potential value of TeMoto to robotic system developers and show TeMoto’s eligibility as a software backbone for adaptive robotic applications.

The evaluation is completed in three experiments with each evaluating a specific set of TeMoto-enabled adaptive features. Experiment 1, a remote teleoperation task, confirms that TeMoto provides resource redundancy to ensure the success of a mission. The resource synchronization features are confirmed using redundant multi-robot system in Experiment 2 where an interruption in the visual sensing capabilities of one robot is compensated by another robot. Experiment 3 shows a deployed robot adapted for a new mission, changing from surveillance to object transportation mission and back. Additionally, common metrics such as power consumption, CPU utilization, and transition durations are presented. Videos of every experiment can be found in
the supplementary materials. The TeMoto framework\(^2\) and demonstration materials\(^3\) with detailed replication instructions can be found on GitHub.

A. OVERARCHING SETUP OF ALL EXPERIMENTS

This section introduces the common software and hardware for evaluation experiments (Fig. 9).

1) Hardware setup

Four hardware platforms, i.e., actors, complete the experimental tasks and are connected via a wireless network:

- **Clearpath Jackal**: a skid-steer robot equipped with Ouster OS1-64 3D lidar, Intel RealSense 435i 3D camera, Hokuyo urg-04lx-ug01 2D lidar and an onboard computer with Intel Core i3-4330TE CPU.
- **Robotont**: an omnidirectional drive robot equipped with Intel RealSense D435i 3D camera and Intel NUC8i5BEK2 onboard computer.
- **Ufactory xArm**: 7 degree-of-freedom manipulator controlled by an onboard computer.
- **Operator Control Station (OCS)**: a Lenovo y50-70 laptop with Intel Core i7-4720HQ CPU and a PlayStation 4 gamepad.
- **Data acquisition server (DAQ server)** - a dedicated laptop (i5-8250 CPU), connected to the same wireless network, was used for acquiring and processing the raw CPU utilization, power consumption and query tracing metrics.

2) Software and data acquisition setup

All actors used TeMoto with Ubuntu 18.04 and ROS Melodic. ROS Noetic is also supported. All actors have an individual set of TeMoto subsystems (Fig. 9) with Robot Manager and Processs Manager being common across all platforms (Please refer to the online materials of the demonstration for a detailed overview of each actor’s TeMoto configuration). CPU utilization and power consumption are measured using a background process which gathered data at 1 Hz. Samples were sent via Pushgateway v1.2.0 to the DAQ server which aggregated the data to a time-series database via Prometheus v2.9.2 displayed via Grafana v6.2.7. TeMoto related resource requests and status communication were measured with tracing features embedded in the RR (see Section III-A). The tracing API OpenTracing v1.5.0 was used with the tracing library Jaeger v0.5.0. Each robot had a Jaeger Agent running in the background which gathered raw traces and sent them to the DAQ server which displayed the traces via Jaeger UI.

3) Evaluation Methodology

CPU utilization and power consumption differences between TeMoto-enabled and reference setups were evaluated as

\[
\Delta x = \frac{x_{ref} - x}{x_{ref}} \cdot 100\%
\]

where \(x\) is the mean of the measured quantity and \(x_{ref}\) is the mean value without TeMoto-enabled dynamic resource management.

Each experiment is segregated into stages representing a specific phase, e.g., where a number of sensors are enabled, followed by a phase with a different configuration. Mean CPU utilization and power consumption values were calculated per stage where the states were unchanged for approximately 10 minutes (~10x60 data samples per metric for each actor) to provide reasonable mean values. The stage durations in the supplementary video material are shorter
to concisely present only the functional aspects of each experiment. The time overhead of TeMoto is evaluated per each stage transition. We measure the total time spent in stage transition vs portion of this time spent by TeMoto’s resource management functions by utilizing distributed tracing (see Section III-A). The results are organized into sequence diagrams (in appendix), which show how queries propagate between subsystems and robots and how much time is spent per each step.

B. EXPERIMENT 1 - FAULT TOLERANT SENSOR REDUNDANCY

This experiment evaluates the viability of dynamically loading resources to enable redundant and fault-tolerant robots. The underlying scenario depicts a remote inspection mission where the operator uses visual sensor feedback. The operator’s situational awareness is challenged by incidents that compromise the robot’s sensors. As a countermeasure, each resource failure is addressed by utilizing an alternative onboard sensor.

The OCS is used to command the Jackal via a gamepad. The OCS a priori assumes that Jackal has three functional visual sensors in a known preferential order: 3D LIDAR, a depth camera, and a 2D LIDAR.

Experiment 1 includes 4 stages. During the idle stage, Jackal and the OCS have TeMoto running in the background without any active resources or actions. During stage 1 (Fig. 10a) an action is invoked on the OCS instance which:

- Requests a 3D LIDAR feed from Jackal.
- Requests Jackal to initialize the mobile base (load the driver ROS node).
- Loads a resource that is able to visualize the 3D LIDAR feed (RViz).
- Loads a gamepad controller resource.

During stage 2, Jackal’s 3D LIDAR is compromised (Fig. 10b) and the previously invoked action on the OCS is automatically notified. This causes the action to execute a preprogrammed behavior that requests a depth camera feed from Jackal and reconfigures RViz to visualize the data. During stage 3, Jackal’s depth camera is compromised (Fig. 10c) and is replaced with a 2D LIDAR. Without TeMoto-enabled dynamic resource management the average power consumption, with all sensors enabled, was 43 W and 70 W for the OCS and Jackal respectively (reference values). Using TeMoto resulted up to 47% (Jackal) and 12% (OCS) reductions in power consumption (Table 3) and 68% reduction in CPU utilization for both systems (Table 4). The detailed sequence diagram of Experiment 1 is shown in the appendix (Fig. 14), which indicates per each resource request less than half of the time is taken up by TeMoto related operations and the rest is taken up by inter-process communication between the subsystems.

C. EXPERIMENT 2 - MULTI-ROBOT ESCORT VIA RESOURCE SYNCHRONIZATION

This experiment evaluates the resource synchronization features (Section III-B) of TeMoto in a remote inspection mission where an operator has visual feedback from the robot’s camera. Visual feedback is severed but automatically compensated by a nearby robot which escorts the compromised robot back to the operator.

The three actors are the Worker (Jackal), Escort (Robotont) and OCS. The Worker and Escort are equipped with onboard cameras and the Component Manager of each robot knows about this resource via a component info YAML-file. Through RIS, the OCS and other actors are aware of all available remote resources, including cameras. The Worker has an Augmented Reality (AR) tag used to track its location.

The experiment has three stages: idle, stage 1, and stage 2. When idling, actors have TeMoto running in the background
without any active resources or actions. All resource info synchronization events take place when a new TeMoto instance is started after which all instances remain idle. During stage 1 (Fig. 11a) an action is invoked on the OCS instance which

- Requests a camera feed from the Worker.
- Loads a resource able to visualize the Worker’s camera feed (RViz).
- Loads a gamepad controller resource used to control the Worker’s mobile base.
During stage 2, a failure is introduced to the Worker’s camera (Fig. 11b) and the previously invoked action on the OCS instance is automatically notified about the incident. This causes the action to execute a preprogrammed behavior that

- Requests a camera feed from the Escort.
- Invokes an action on the Escort which makes it follow the Worker (Fig. 11c). The action which the OCS invoked on the Escort makes the Escort
  - Load a camera and AR-tag detection resources.
  - Load an algorithm that makes the Escort follow the Worker by outputting velocity commands based on the location of the detected AR-tag.

Average power consumption and CPU utilization of the actors across the three stages of the experiment are presented in Tables 5 and 6. Without TeMoto’s dynamic resource management the average power consumption, with all sensors enabled, was 31 W, 48 W and 17 W (reference values) for the OCS, Worker, and Escort respectively. However if resources were utilized only when needed, power consumption and CPU utilization were reduced. The most drastic being the idle stage with 26%, 4% and 31% reduction in power consumption and 94%, 37% and 43% reduction in CPU utilization for the OCS, Worker and the Escort respectively.

Average power consumption and CPU utilization were reduced. The most drastic being the idle stage with 26%, 4% and 31% reduction in power consumption and 94%, 37% and 43% reduction in CPU utilization for the OCS, Worker and the Escort respectively. However if resources were utilized only when needed, power consumption and CPU utilization were reduced. The most drastic being the idle stage with 26%, 4% and 31% reduction in power consumption and 94%, 37% and 43% reduction in CPU utilization for the OCS, Worker and the Escort respectively.

The detailed sequence diagram of Experiment 2 is shown in the Appendix (Fig. 14), which indicates per each resource request less than half of the time is utilized by TeMoto related operations and the rest is used by inter-process communication between the subsystems.

### D. EXPERIMENT 3 - MISSION ADAPTIVENESS DURING LONG-TERM DEPLOYMENT

This experiment evaluates how robots can be adapted for new tasks described by a UMRF graph. The scenario depicts a mobile robot autonomously sweeping an area as its “day job” until the robot is retasked to deliver goods, transferred from another robot. After completing the delivery, the robot resumes with the surveillance task.

Conventionally, robotic tasks are embedded into monolithic scripts (e.g., ROS launch files) or programs (e.g., ROS nodes). If the task changes, this approach requires manual intervention by an operator, who stops the programs/scripts related to the old task and invokes new ones. TeMoto provides tools to programmatically start, stop and modify tasks during run-time, which can be utilized by a top-level automated task planning system, leading to a dynamically configurable autonomous system. The technical emphasis of this experiment is that a full stack of TeMoto’s tools are utilized (Fig. 2 and Fig. 3) to achieve fully dynamic and reconfigurable behaviour:

- The UMRF graph describes the mission in an interface independent JSON format keeping the system flexible to different input command modalities (Fig. 3),
- The Action Engine composes the mission during run-time, via UMRF graph, from modular and reusable actions,
- Resource managers dynamically start and stop the resources required, and
- RIS provides resource synchronization functionalities which allows for the control of resources on a remote robot (e.g., load the robot’s driver and controller) - a feature currently not supported by ROS.

This experiment involves three actors: Jackal, Ufactory xArm7 and the OCS. The Jackal utilizes 2D lidar and gmapping [41] SLAM to localize in its environment. TeMoto’s Robot Manager controls Jackal’s navigation and xArm7’s manipulation functionalities. The OCS stores the mission plans as UMRF graphs where the first graph (Fig. 12a) instructs the Jackal to cyclically navigate through three different locations and the second graph (Fig. 12b) coordinates the Jackal and xArm7 to deliver two objects.

Task 1 starts when the operator sends the surveillance UMRF graph (Fig. 12a and Fig. 13a) to the OCS’s Action Engine. The graph contains three navigation actions, each pre-configured for a specific location via UMRF input parameters. The navigation action internally instructs the Robot Manager to move the Jackal. Similar to the previous experiment the OCS’s Robot Manager knows the mobile platform is managed in Jackal’s TeMoto instance, thus relaying the navigation command to Jackal. After a few cycles of surveillance,
the operator invokes the delivery UMRF graph (Fig. 12b and Fig. 13b). This UMRF graph instructs the Jackal to navigate to the pick-up location while initializing xArm7’s ROS driver and controller via Robot Manager. After the parallel actions are finished, the xArm7 is instructed to transfer two objects to the Jackal. The Jackal is then instructed to navigate to a predefined delivery point. After delivery task, the robot continues the surveillance task.

V. DISCUSSION

These experiments provide both a practical overview of TeMoto’s feasibility as a framework for adaptive and fault tolerant applications, and they quantified performance enhancements related to power consumption and CPU utilization. TeMoto also reduced operator’s overhead by eliminating the need to manually re-configure the displays (managed via Visualization Manager) in Experiments 1 and 2. This reduces the operator’s workload and does not require technical skills to do so. The dynamic reconfigurability of resources enabled redundancy with the alternative sensors being dynamically enabled only if needed. Thus, up to 47% reduction in power consumption was achieved in the experiments illustrating the potential for greater reductions in real-world scenarios. Further increases of efficiency may be attained for complex robotic platforms by enabling components only as needed. The TeMoto framework provides a backbone for realizing fault-tolerant mechanisms and asynchronous multi-task systems.

The core features of the TeMoto framework, such as Resource Registrar, Resource Info Synchronizer, the Action Engine and all resource manager interfaces are written in C++. Support for Python-based projects is the next development task. We are actively working on ROS2 support. Thus our focus is on providing support for both ROS versions with
minimal code duplication.

The backbone of TeMoto’s resource management functionalities, the RR (see section III-A), in its current implementation is missing resource consumer prioritization features, which are needed for allowing higher priority consumers to gain control over a resource (or deny a request of a low priority consumer). For example if a manipulator arm (a resource) is initialized and controlled by a low priority task (consumer 1), a higher priority task (consumer 2), e.g., an external collision risk assessment routine, can gain control by acquiring the resource and release it shortly after the risk has been mitigated. Also extending RR status update channel to forward resource state changes from, e.g., “loading” to “up”, is future work. Such behavior is necessary for indicating resource consumers about resources that may have a long initialization period.

VI. CONCLUSIONS

TeMoto is a continuously evolving ROS-based project which reduces the complexity of developing fault-tolerant, energy-efficient, and asynchronous multi-tasking for complex robotic systems. The TeMoto framework can be applied as a backbone for a robot’s software architecture, as it encompasses both high-level aspects of robot commanding modalities and low-level aspects of managing robot’s resources. Yet the modular and layered design of the TeMoto framework supports customizing, adapting, and building upon its core tools individually when required. The results presented in this work demonstrate the feasibility of the TeMoto framework for adaptive and reliable applications. Increased energy efficiency (up to a 47% reduction in power consumption) and fault tolerance - as well as runtime reconfigurability of the robotic tasks and resources - were demonstrated on both single and collaborative multi-robot systems with teleoperation, autonomous surveillance, cargo delivery scenarios.

REFERENCES

[1] M. Shishehgar, D. Kerr, and J. Blake, “A systematic review of research into how robotic technology can help older people,” Smart Health, vol. 7-8, pp. 1–18, Jun. 2018.
[2] J. A. Marvel, S. Bagchi, M. Zimmerman, and B. Antonishek, “Towards Effective Interface Designs for Collaborative HRI in Manufacturing,” ACM Transactions on Human-Robot Interaction, vol. 9, no. 4, pp. 1–55, 2020.
[3] Y. Tamura, H. Amano, and J. Ota, “Analysis of firefighting skill with a teleoperated robot,” ROBOMECH Journal, vol. 7, no. 1, p. 26, Jun. 2020.
[4] T. Fong, J. Rochlis Zumbado, N. Currie, A. Mishkin, and D. L. Akin, “Space Telerobotics: Unique Challenges to Human–Robot Collaboration..."
in Space,” Reviews of Human Factors and Ergonomics, vol. 9, no. 1, pp. 6–56, Nov. 2013.

[5] K. Nagatani, S. Kiribayashi, Y. Okada, K. Otake, K. Yoshida, S. Tadokoro, T. Nishimura, T. Yoshida, E. Koyanagi, M. Fukushima, and S. Kurotawama, “Emergency response to the nuclear accident at the Fukushima Daiichi Nuclear Power Plants using mobile rescue robots,” Journal of Field Robotics, vol. 30, no. 1, pp. 44–63, 2013.

[6] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and N. Thanh-Hung, “Designing autonomous robots: Toward a more dependable software architecture,” IEEE Robotics and Automation Magazine, vol. 16, pp. 67–77, 2009.

[7] S. Garcia, C. Menghi, P. Pelliccione, T. Berger, and R. Wohlrab, “An architecture for decentralized, collaborative, and autonomous robots,” in 2018 IEEE International Conference on Software Architecture (ICSA), 2018, pp. 75–760.

[8] L. Kunze, N. Hawes, T. Duckett, M. Hanheide, and T. Krajnik, “Artificial Intelligence for Long-Term Robot Autonomy: A Survey,” IEEE Robotics and Automation Letters, vol. 3, no. 4, pp. 4023–4030, Oct. 2018.

[9] Y. Yokokoji, “The role of robots to respond to nuclear accidents: Applying the lessons of the past to the fukushima daiichi nuclear power station,” Annual Review of Control, Robotics, and Autonomous Systems, vol. 4, pp. 681–710, 2021.

[10] J. Delferico, S. Mintchev, A. Giusti, B. Gromov, K. Melo, T. Horvat, C. Cadena, M. Hutter, A. Ipspeert, D. Floreano et al., “The current state and future outlook of rescue robotics,” Journal of Field Robotics, vol. 36, no. 7, pp. 1171–1191, 2019.

[11] A. Di Lollo, R. Murphy, A. Krieger, J. Zhu, R. H. Taylor, and H. Su, “Medical robots for infectious diseases: Lessons and challenges from the covid-19 pandemic,” IEEE Robotics Automation Magazine, vol. 28, no. 1, pp. 18–27, 2021.

[12] A. Koubia, Ed., Robot Operating System (ROS): The Complete Reference (Volume 1), ser. Studies in Computational Intelligence. Springer International Publishing, 2016.

[13] R. Valner, S. Wanna, K. Kruusmaa, and M. Pryor, “Unified Meaning Representation Format (UMRF) - A Task Description and Execution Formalism for HRL.” ACM Transactions on Human-Robot Interaction, 2022. [Online]. Available: https://dl.acm.org/doi/10.1145/3522580

[14] R. Valner, V. Vunder, A. Zelenak, M. Pryor, A. Aabloo, and K. Kruusmaa, “INTUITE ‘HUMAN-ON-THE-LOOP’ INTERFACE FOR TELE-OPERATING REMOTE MOBILE MANIPULATOR ROBOTS,” in International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS), June 2018.

[15] A. Kustavus, “Design and implementation of a generalized resource management architecture in the temoto software framework,” Master’s thesis, University of Tartu, 2021.

[16] M. Cashmore, F. Fox, D. Long, M. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtos, and M. Carreras, “ROSPlan: Planning in the Robot Operating System,” in Twenty-Fifth International Conference on Automated Planning and Scheduling, Apr. 2015. [Online]. Available: https://www.aaai.org/ocs/index.php/ICAPS/ICAPS15/paper/view/10619

[17] P. Muñoz, M. D. R-Moreno, D. F. Barrero, and F. Ropero, “MoBar: A Hierarchical Action-Oriented Autonomous Control Architecture,” Journal of Intelligent and Robotic Systems: Theory and Applications, vol. 94, no. 3-4, pp. 745–760, Jun. 2019.

[18] D. de Leng and F. Heintz, “DyKnow: A dynamically reconfigurable stream reasoning framework as an extension to the robot operating system,” in 2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), Dec. 2016, pp. 55–60, ISSN: null.

[19] C. Hernández, J. Bermudo-Alonso, and R. Sanz, “A self-adaptation framework based on functional knowledge for augmented autonomy in robots,” Intergrated Computer-Aided Engineering, vol. 25, no. 2, pp. 157–172, Jan. 2018.

[20] S. Wang, X. Liu, J. Zhao, H. Christensen, and H. I. Christensen, “Robotic Reliability Engineering: Experience from Long-term TritonBot Development,” in 12th Conference on Field and Service Robotics (FSR 2019). EasyChair, 2019, iSSN: 2516-2314.

[21] “The STRANDS Project: Long-Term Autonomy in Everyday Environments,” IEEE Robotics and Automation Magazine, vol. 24, no. 3, pp. 146–156, 2017.

[22] H. Cadavid, A. Pérez, and C. Rocha, “Reliable Control Architecture with PLEXIL and ROS for Autonomous Wheeled Robots,” in Advances in Computing, ser. Communications in Computer and Information Science, A. Solano and H. Ordoñez, Eds. Cham: Springer International Publishing, 2017, pp. 611–626.

[23] A. Favier, A. Messieux, J. Guiochet, J.-C. Fabre, and C. Lesire, “A hierarchical fault tolerant architecture for an autonomous robot,” Jun. 2020. [Online]. Available: https://hal.laas.fr/hal-02558804

[24] R. Capilla, J. Bosch, P. Trinidad, A. Ruiz-Cortés, and M. Hinchee, “An overview of dynamic software product line architectures and techniques: Observations from research and industry,” Journal of Systems and Software, vol. 91, pp. 3–23, 2014.

[25] A. Ahmad and M. A. Babar, “Software architectures for robotic systems: A systematic mapping study,” Journal of Systems and Software, vol. 122, pp. 16–39, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121216301479

[26] S. G. Brunner, A. Domel, P. Lehner, M. Beetz, and F. Stulp, “Autonomous parallelization of resource-aware robotic task nodes,” IEEE Robotics and Automation Letters, vol. 4, no. 3, pp. 2599–2606, 2019, publisher: Institute of Electrical and Electronics Engineers Inc.

[27] “ROSPlan.” [Online]. Available: https://github.com/KCL-Planning/ROSPlan

[28] “MoBar Turtlebot.” [Online]. Available: https://github.com/ISG-UAH/mobar-turtlebot

[29] “SEERA.” [Online]. Available: https://github.com/SergioGarG/SEERA_Platform

[30] S. Shivakumar, H. Torfah, A. Desai, and S. A. Seshia, “SOTER on ROS: A Run-Time Assurance Framework on the Robot Operating System,” in Runtime Verification, ser. Lecture Notes in Computer Science. Springer International Publishing, 2020, pp. 184–194.

[31] “SOTEROnROS.” [Online]. Available: https://github.com/Droma-Org/SOTEROnROS

[32] “STRANDS.” [Online]. Available: https://github.com/strands-project

[33] “Osmosis.” [Online]. Available: https://github.com/osmosis

[34] “TritonBot.” [Online]. Available: https://github.com/CogRob/TritonBot

[35] “DyKnow.” [Online]. Available: https://github.com/dnlen/dyknow_common

[36] “Metacontrol ROS.” [Online]. Available: https://github.com/MROS-RoMoSys-ITP/MROS-Hands-On

[37] S. G. Brunner, F. Steinmetz, R. Belder, and A. Döml, “Racfon: A graphical tool for engineering complex, robotic tasks,” in 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2016, pp. 3283–3290.

[38] A. Desai, V. Gupta, E. Jackson, S. Qadeer, S. Rajamani, and D. Zufferrey, “P: Safe asynchronous event-driven programming,” SIGPLAN Not., vol. 48, no. 6, p. 321–332, jun 2013. [Online]. Available: https://doi.org/10.1145/2493370.2462184

[39] A. Nordmann, R. Lange, and F. M. Rico, “System modes - digestible system (re-)configuration for robotics,” in 2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE), 2021, pp. 19–24.

[40] “OpenTracing.” [Online]. Available: https://opentracing.io

[41] G. Grissiti, C. Stachniss, and W. Burgard, “Improved Techniques for Grid Mapping With Rao-Blackwellized Particle Filters,” IEEE Transactions on Robotics, vol. 23, no. 1, pp. 34–46, Feb. 2007, conference Name: IEEE Transactions on Robotics.
VEIKO VUNDER Veiko Vunder is a lecturer in robotics technology at the University of Tartu. He holds an M.Sc. degree in computer engineering and a Ph.D. degree in engineering and technology - both obtained from the University of Tartu. Veiko’s research is related to modeling and characterization of smart actuators for soft robotics applications and improving operators’ situational awareness in teleoperated robots.

ALVO AABLOO Alvo Aabloo received the Ph.D. degree in physics in 1994 focusing on computational research of celluloses. His postdoc research topic was computational modeling of ion exchange polymer membranes in ion-polymer batteries. Since 2004, he has been focusing on ion-polymer active materials (artificial muscles), cross-disciplinary research, computational methods, fabrication, control, and applications. He has been the Coordinator of the H2020 MSCA-ITN network and a Research Partner for FP7, H2020, ESA, EIT-Health, and IT manufacturing projects. His research interests include electrochemomechanically active polymer composite materials, computational materials, ion-polymer batteries, robotics, and space applications.

MITCH PRYOR Dr. Pryor is a Senior Research Scientist and Lecturer for the Cockrell School of Engineering at the University of Texas at Austin. Dr. Pryor earned his BSME at Southern Methodist University in 1993. He completed his Masters (1999) and PhD (2002) at UT Austin with an emphasis on the modeling, simulation, and operation of redundant manipulators. He has worked for numerous research sponsors including, NASA, DARPA, DOE, INL, LANL, ORNL, Y-12, and many industrial partners. He is a co-founder of the Nuclear Robotics Group and the Drilling Rig Automation Group. Both are interdisciplinary research efforts to deploy robotics in hazardous, uncertain environments to perform manufacturing, material handling and other tasks. He is a member of ROS-Industrial, IEEE, ASME, PGE, and ANS.

KARL KRUUSAMÄE Karl Krusamäe is an associate professor of robotics engineering at the University of Tartu. He received the M.S. degree in information technology and the Ph.D. in physics from the University of Tartu, Tartu, Estonia, in 2008 and 2012, respectively. His research interests include human-robot interaction and shared autonomy.

APPENDIX A DETAILED SEQUENCE DIAGRAMS OF EXPERIMENTS 1 AND 2

Figures 14 and 15 show the detailed sequence diagram (see Section IV-A3 for more details) of Experiment 1 (section IV-B) and Experiment 2 (section IV-C) respectively, which reads from top to bottom. All related TeMoto subsystems on each actor are shown on their own timeline (dashed vertical arrow) and communication between the subsystems is indicated via solid horizontal arrows. The time spent in each subsystem is indicated via a solid thick line along the respective timeline, where the total duration (without parentheses) and the TeMoto subsystem specific overhead of that duration (within parentheses) is indicated on the solid thick line. The communication durations are indicated on the respective horizontal lines.

The sequence diagrams (Fig. 14 and 15) shows that per each resource request less than half of the time is taken up by TeMoto related operations and the rest is taken up by inter-process communication between the subsystems. The exceptions are the “show camera feed” request in stage 1 and the response operations triggered by the “Worker camera stopped” message in stage 2. With the former, the Visualization Manager first starts RViz and then waits approximately for 2 s for RViz to come up, so that it can load a camera display plugin in RViz. With the latter, the Operator’s action initializes a ROS publisher and waits for the Escort’s Action Engine to subscribe, which takes approximately 3 s with the given setup.

* * *
FIGURE 14. Sequence diagram of Exp. 1 where the vertical dashed lines represent the timelines of each subsystem, the thick vertical lines represent time spent by each subsystem and horizontal solid lines represent communications between subsystems. Time durations are indicated above each line and subsystem specific overhead of that duration is in parenthesis. Time spans are not scaled to actual durations.
FIGURE 15. Sequence diagram of Exp. 2 where vertical dashed lines represent the timelines of each subsystem, thick vertical lines along each timeline represent time spent by each subsystem and horizontal solid lines represent communications between the subsystems. Time durations are indicated above each line and TeMoto subsystem specific overhead is in parenthesis. Time spans are not scaled to actual durations.