UNIVERSITY OF TARTU

Faculty of Sciences and Technology

Institute of Technology

Allan Kustavus

Design and Implementation of a Generalized Resource Management Architecture in the TeMoto Software Framework

Master's Thesis (30 EAP)

Robotics & Computer Engineering

Instructors:

Robert Valner

Karl Kruusamäe

Tartu 2021

Abstract

Design and Implementation of a Generalized Resource Management Architecture in the TeMoto Software Framework

Autonomous robots are utilized in a wide range of domains, combining a large number of resources like sensors, actuators and algorithms to form a self-acting robotic system. Tools, such as ROS and TeMoto, have been developed to allow for handling and managing of resources composing such systems. While TeMoto is meant to handle dynamic and changing situations the current implementation of its Resource Registrar, a core TeMoto component tasked with allocating, deallocating and tracking of resources, is tightly coupled to ROS, making it difficult to modify and improve. As a result of this thesis, the Resource Registrar (RR) of TeMoto was completely redesigned to be extendable to other robotic middleware, such as ROS2, and to improve TeMoto's robustness with features, e.g., full recovery of the RR, that were unattainable with the previous design.

CERCS: T120; T125; [1]

Keywords: Robotics, programming, autonomous operation, Robot Operating System, resource management, TeMoto

Resümee

Üldise ressursihalduri disain ja teostus TeMoto tarkvara raamistikule

Autonoomsed robotid leiavad erinevates valdkondades laialdast kasutust. Kombineerides erinevaid ressursse, nagu sensorid, ajamid ning algorid, moodustavad nad iseseiseva robot-süsteemi. Tööriistad, nagu ROS ning TeMoto, kergendavad seesuguste süsteemide haldamist. Kuigi TeMoto on disainitud dünaamilisi süsteeme haldama, on selle tuumas paiknev ressursihaldur tugevalt seotud ROS funktsionaalsusega, mis teeb halduri muutmise ning edasiarendamise keerukaks. Selle lõputöö tulemusena loodi TeMoto jaoks uus ressursihaldur, mis laiendas raamistiku töökindlust varasemalt saavutamatute funktsionaalsustega, nagu ressursihalduri seisundi taastamine, ning võimaldab neid funktsionaalsuseid laiendada ka teistele raamistikele, näiteks ROS 2.

CERCS: T120; T125; [1]

Märksõnad: Robootika, programmeerimine, autonoomia, ROS, ressursihaldus, TeMoto

Contents

List of Figures

List of Tables

List of Listings

Abbreviations and constants

API - Application programming interface is an interface defining interactions between applications or applications and hardware [2].

C++ - A high-level programming language built off the C programming language and includes object-oriented features [3].

CPU - Central Processing Unit, a primary component of a computer that processes instructions [3].

IP - Internet Protocol, a communication protocol used to transport data across a network [3].

LiDAR - Light Detection and Ranging, a remote sensing method that uses laser light to measure distance [4].

ROS - The Robot Operating System (ROS), a software framework for writing robot software [5].

RR - Resource Registrar. Name for a system managing resources of a system.

TCP - Transmission Control Protocol, a protocol used by applications to communicate in a network [3].

UUID - Universally Unique Identifier. A 128-bit long number used for identifying resources in a system. It does not require a centralized authority for identification issuing [6].

1 Introduction

Robots are proliferating in multiple different domains. From vacuum cleaners and window washers to whole automated warehouses, robots are found in more and more aspects of our everyday life. With advances in technology, it becomes possible to tackle complex problems in medicine, disaster control, exploration, autonomous operation, etc. using robotics.

With the increased complexity of tasks requirements for robots systems increase. A robot can consist of a large number of resources, like sensors for perceiving the world, actuators for interacting with the world and algorithms for decision making. To cope with this robotics specific frameworks, like Robot Operating System (ROS), have been created to assist in the development of complex robotics systems.

To allow for robust autonomous operation a robots system needs to cope with dynamic changes in its environment. Efficient management of resources can increase the fault tolerance, energy efficiency and configurability of a robot. For example in the case of sensor failure, a substitute can be initiated, increasing fault tolerance. Energy efficiency can be increased when a robot powers down its navigation systems when stationary. To enable such behaviour a dynamic resource management solution is needed.

Robot specific frameworks like ROS simplify the design and creation of robot systems but are by themselves static and do not provide dynamic resource management and recovery. To tackle this problem TeMoto was developed to simplify the building of autonomous robots. TeMoto provides flexible and dynamic resource management features allowing systems to adopt to different situations.

While TeMoto is meant to handle dynamic and changing situations the current implementation of its Resource Registrar, a core TeMoto component tasked with allocating, deallocating and tracking of resources, is tightly coupled to ROS, making it difficult to modify and improve. To allow for TeMotos Resource management capabilities to improve, a complete overhaul of the RR module is necessary, which is the main goal of this thesis. As a result of this work the new RR should allow for the following:

- The system has to be framework agnostic, otherwise, it is difficult to extend and adapt the system to different situations.
- Base functionalities need to be universal across use-cases. The same functionality defined in multiple locations introduces clutter to the system.
- The system treats all resources equally and does not care for their implementation as long as they fulfil the requirements set upon them.
- When the system shuts down unexpectedly it should be able to restore itself.
- The system has to test itself. This way, when something new is added it can be validated that old functionalities stay working. Of course, everything new needs to be added to the tests.

The contents of this thesis is structured into 5 chapters. Chapter 2 gives an overview of system architectures that can be managed and benefit from resource management, introduces ROS and gives a motivation as to why TeMoto's RR requires a redesign. Chapter 3 views and compares how the problem of resource management has been tackled in other domains and different robotics specific solutions. In chapter 4 the requirements, structure and implementation of the software are laid out and described in detail. Chapter 5 gives a detailed overview of RR's ROS 1 and ROS 2 extensions. Chapter 6 presents a scenario for an autonomous robot that demonstrates the capabilities of the developed software.

2 Domain overview

Resource Registrar should not be limited to a specific robotic middleware and thus it must follow a system-agnostic design and also introduce minimal dependencies to the deployed host application. Thus architectural patterns of monolithic and distributed applications were investigated to better understand the successful design principles for the RR. Since the RR is primarily meant to work on ROS via TeMoto, the existing implementation of RR was analyzed to understand changes the new system had to make.

2.1 Monolithic systems

A monolithic system is built as a singular application. The system can consist of multiple services that come together to provide different functionalities [7]. Robots such as machines competing in RoboCup, a large international robotics competition [8], have used this approach for purpose-built solutions [9].

For example, let's take a monolithic robot that uses a camera or user input to move around while avoiding obstacles. To accomplish this it needs a behaviour module, camera data fetching and processing functionalities, modules for moving and other utilities, visualized on figure **2.1**. All these modules are integrated into the application, strongly coupling the modules to the application and each other.

2.1.1 Types

Monoliths can be categorized into 2 distinct types based on their properties [10]. They are:

- Single Process Monolith - The most common type of a monolith system where all functionalities are deployed as a single process. This is illustrated in figure **2.2** where a single piece system is addressed. There can be multiple instances of the process but fundamentally everything is packed into a single process.

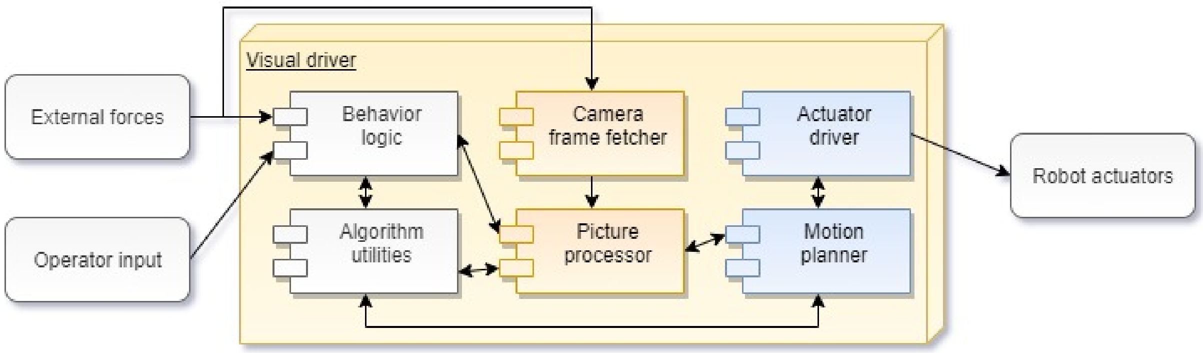


Figure 2.1: An example diagram of a monolithic robot system. It consists of a single deployable where different system parts have can have direct access to other parts. The whole system needs to be swapped out when updating parts of it.

- Modular Monolith - A monolith that has its structure is broken down into smaller modules (fig. **2.2**b). Here the single process is broken down into smaller modules, e.g., libraries, that can be worked on independently but need to be combined for the system to function.

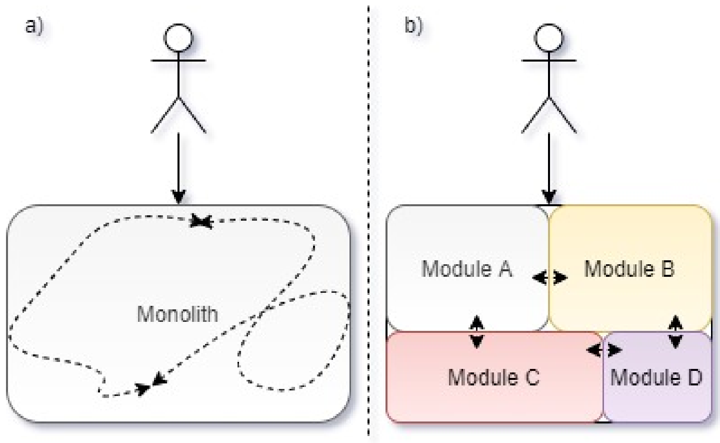


Figure 2.2: Two types of monoliths that a user executes functionality on. The dashed arrow symbolises access to system functionalities. a) single process monolith where all functionality is packaged into a single module. Functionalities can be spread across the system. b) a modular monolith where functionality is separated into responsible modules

2.1.2 Benefits

People have come to see monoliths as being associated with legacy code but it is a valid choice for a systems architecture [10]. The choice of the architecture needs to be done based on system requirements and for some systems, a monolithic design with its simplicity can be the correct choice [10].

Monoliths can be quicker to develop since the functionalities of a system can be shared between parts [10]. Functionality is tightly coupled and the code is highly interdependent [11]. This allows to initially develop features rapidly, with less time to consider architectural eccentricities [8]. A monolith is simpler to test because all the systems functionalities can be accessed when testing [12]. Additionally, the system is a single piece of software and thus there is little overhead in setup and configuration [7]. When looking at the example from before (fig. **2.1**) modules are contained inside the same application and can potentially access each other's functionalities without issues.

Modular Monolith provides the most flexibility of all the monoliths described. With well-defined module boundaries, the issue of coupling is reduced and the simplicity of a monolith retained [10]. This simplifies code changes, allows for the collaboration of different parties while keeping system setup and management overhead to a minimum.

2.1.3 Drawbacks

While being simplistic a monolith can be vulnerable to coupling - when parts of a system are intertwined in ways that make it hard to change it without affecting unrelated components of the application. With monoliths, boundaries between application responsibilities are muddled and the question of ownership can arise [10].

When system complexity increases then so do the development effort required to change one piece of the system [11]. Parts can be connected in an odd number of ways and a high level of knowledge about the code is required to avoid unintended defects. Strongly coupled modules are hard to swap out because of dependencies [12]. This rigid yet undefined coupling is a cause for frustration in many monolithic systems and is especially relevant in single process monoliths [10]. When external resources, such as third-party libraries, are used the system can be bound to this resource, making swapping or replacing it difficult.

When looking at monolith application reliability can become an issue. When a monoliths subsystem encounters a fatal error the whole system will be brought down. Unrelated modules will be affected by fatal system errors [12]. Quick fixing of such faults is also more complex because the whole system needs to be shut down to deploy a fix.

Large enough systems can start to accumulate technical debt. This term was coined by Ward

Cunningham and describes how prioritizing project requirements can compromise technical and design considerations [13]. To an extent, this debt is not bad but can start to affect development and maintenance effort [14]. This dept is not only a problem on monolithic systems but is harder to maintain. Complex monolithic systems require a deep understanding of underlying functionalities to modify them. Without this knowledge, changes can introduce unneeded complexity and design issues into the project [15]. Fixing such design issues is difficult with monolithic systems because of the high coupling and low cohesion between modules [10].

2.2 Distributed systems

A distributed architecture consists of well-defined system modules that have clear and distinguished functional responsibilities, are self-contained and easy to exchange [16]. This creates a system with extendable, modifiable and reusable modules that are loosely coupled over defined Application Programming Interfaces (APIs) [17]. Modular architecture such as micro-services has become more and more popular in software development. In robotics, this movement can also be felt where frameworks such as ROS help to decouple modules of a system [18].

Let us take the previously discussed obstacle avoidance robot and change its software architecture from a monolith (fig. **2.1**) to be distributed (fig. **2.3**). Every module was moved into its separate application that exposes an API to be used for other applications. This way a systems element can be changed out without affecting other functionalities as long as the defined API layer is kept the same.

2.2.1 Types

Distributed systems can be divided into different types. Relevant design patterns to this work are explained.

- Layered architecture - A layer is the only architectural element of a layered architecture (fig. **2.4**a). Layers bundle software functionalities up into larger building blocks. Layers themselves form blocks that can be used together to build an application. Layers connect to form a sequence of operations that are used to process requests and provide services [19]. Bottom layers provide a service to layers on top. Layers can be modified or replaced without affecting other layers [20].

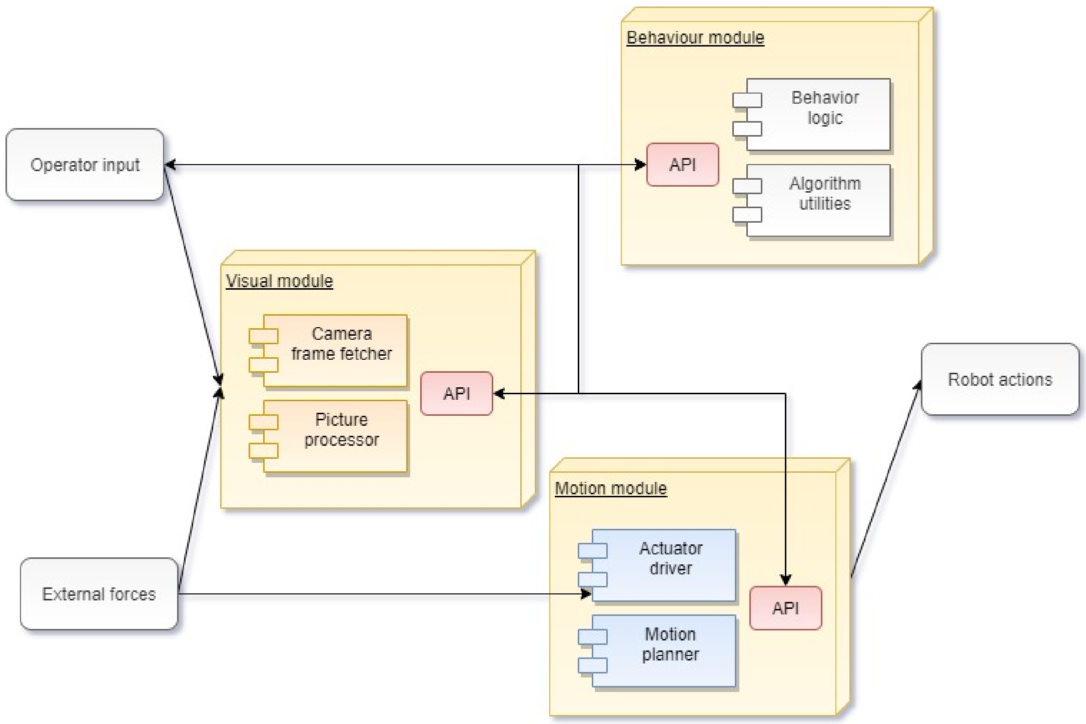


Figure 2.3: An example diagram of a distributed robot system. The system is separated into standalone modules that communicate over defined API's. Modules can be swapped out without affecting other parts of the system.

- Service-oriented architecture - Service-oriented architecture uses loosely coupled black-box components arranged in a coherent way (fig. **2.4**b) to deliver a defined level of services that are implemented inside software systems [21]. Micro-services can be classified under this group, having standalone service components with well-defined interfaces that can be used by a system to provide more complex services or behaviour.

- Event-driven architecture - Event-driven system has the capability to detect and react to events. An event is a change in the state of the system that requires actions to be taken. Components interested in events can listen for them and take action when necessary [22]. ROS uses such an architecture in its publisher/subscriber patterns where, as the name implies, a publisher emits events that subscribers can listen to and react to (fig. **2.4**c).

2.2.2 Benefits

With distributed systems functionalities are structured in independent subsystems that communicate with each other through defined APIs [23] (**2.3**). Individual micro-services are respon-

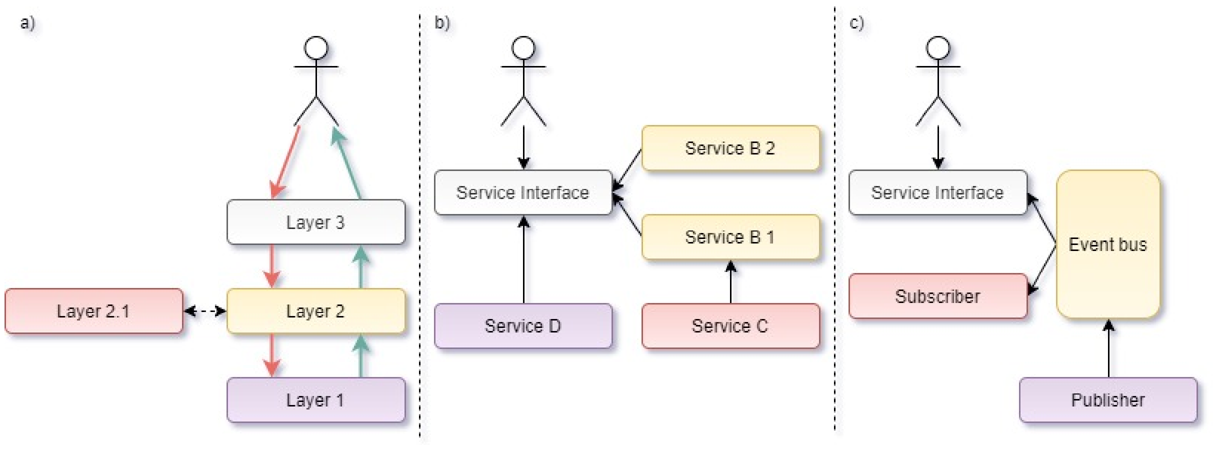


Figure 2.4: Three different distributed systems. a) Layered structure where data is processed down the layer stack and returned up the stack. b) Service-oriented system with loosely coupled modules that can be swapped out or added in when required. c) Event-driven system where components subscribe to an event bus and react to relevant events.

sible for specific functionalities, bringing their complexity down and make modifying them simpler [24]. Changes are also confined to one module and isolated. Changes do not leak outside their scope.

A distributed architecture can increase system reliability, e.g., when a fatal error occurs in one of the modules and it crashes rest of the system stays operational. Every module is independent by design and continues working. The crashed component can be restarted, restoring full system capabilities, increasing system fault tolerance [11]. If a fix has to be deployed to the system, only one component has to be swapped and overall system downtime can be reduced.

Isolated modules are simpler to optimize, e.g when a bottleneck in the system identified its implementation can be optimized without affecting other parts of the system [25]. This is because applications expose interfaces designed for communication and nothing from their internal workings [24]. Every sub-system can use appropriate technology for its functionalities. A modules internal workings can be swapped out as long as the API layer stays the same.

2.2.3 Drawbacks

Distributed systems, as with other architectures, have drawbacks that need to be taken into account. Since they consist of independent modules, management overhead for the whole system can increase, amongst other aspects.

The systems overall complexity increases because modules of a system need to have a well-defined API layer designed beforehand. Great care needs to be put into a systems design else overall system complexity can increase even when the intention was to decrease it [26]. Without constant vigilance a distributed monolith can be created, defeating initial design intentions set out by choosing a distributed pattern. A system-wide view on complexity needs to be kept in mind when designing modular code since complexity does not only come from features supplied by a system module but also from the structure of the system as a whole [27].

APIs introduce a level of abstraction to the systems where service-to-service communications can increase overall system overhead due to message translation and other factors [28]. Resource requirements for the host system can also increase. When each application module uses a constant X amount of memory and a monolith was split into Y modules then memory overhead increases from X to X*Y. Maintenance requirements also increase since every module is a standalone application [23].

Micro-services decrease local complexity: the complexity of services in a module [27]. At the same time, global complexity increases the complexity of the system made up of services. Next to code changes between services, tooling associated with modules also needs to be updated for every module that was changed. Tooling can include build utilities, monitoring systems and deployment daemons [28].

Testing provides a challenge for modules due to additional abstraction layers. Functionalities dependent on other modules need to mock external resources. While internal service behaviour tests can be simpler due to isolated responsibilities [11] service-to-service functionalities need to recreate the communications layer with all its intricacies. If mocking is flawed tests can return false-positive results, reducing confidence in test results [28].

2.3 Architectural approaches compared

Monolithic and distributed architectures both have their advantages and disadvantages. As with any other tool, consideration needs to be taken for selecting the correct architecture for a system. For systems with few functionalities that are static and resource bound monolithic patterns might be a good fit. With larger projects that require multiple teams to work on the functionalities of a system distributed patterns might be more appropriate. Table **2.1** summarizes main aspects of both architectures.

Table 2.1: Comparison between a monolithic and distributed system architecture.

	Monolithic system	Distributed system
Architecture complexity	Lower complexity	Higher complexity
Location of system complexity	Module complexity	System complexity
Management of technical dept	Difficult due to strong coupling	Simpler due to weak coupling
Maintenance effort	Higher due to strongly coupled modules	Lower due to isolated weakly coupled modules
Optimal development team size	Lower due to strong module coupling	Higher due to weak module coupling
System testing	Simpler, all features in system modules	Harder, dependencies distributed in the system
Fault tolerance	Lower, system crash affects whole system	Higher, modules are isolated and independent
System scaling	Deep	Wide
System abstraction and overhead	Lower	Higher
Monitoring complexity	Lower, one system needs to be monitored	Higher, multiple modules need to be monitored

2.4 Robot Operating System

The Robot Operating System is a framework for writing robot software. It aims to simplify the task of creating and managing complex software solutions while also providing a robust result. ROS supports a wide field of applications and a variety of robotic platforms [5], including support for many industry standard libraries such as OpenCV, MoveIt [29] or community support for many more [30].

ROS was designed to be modular and to allow a project to use as much of ROS's functionalities as it needed while skipping undesired features. It is an open-source project with a large community following with over 3000 publicly available packages for ROS ranging from proof-of-concept algorithms to industrial-grade drivers [30]. One of the core concepts of ROS is computational nodes that are standalone programs and a messaging layer that binds the nodes together into a robotic application [18].

ROS nodes are standalone programs, designed to encapsulate a specific functionality, such as object recognition or providing access to drivers of a robotic platform. ROS provides different communication patterns to connect these nodes. Multiple nodes combine into a graph and communicate with each other using topics, services or actions [18]. A simple ROS system using nodes and services is illustrated in figure **2.5**. Nodes should have well-defined responsibilities and follow modular system patterns. Isolated modules increase fault tolerance by containing system crashes to individual nodes [18]. Node-to-node communications follow defined APIs via topics and implementation logic can be hidden allowing for node swapping.

ROS allows for asynchronous topic-based publisher/subscriber messages or synchronous service-based request/response patterns. Topics are buses that nodes use to exchange messages. Nodes can connect to the data bus for one-way message consumption as seen on figure **2.5** block B. Topics are strongly typed and nodes can receive messages of the matching type. TCP/IP-based and UDP-based transportation layers are supported. ROS provides tools for topic discovery and monitoring. Communications on topics can be recorded and replayed back onto the bus that allows to replay system behaviour and simplify debugging [18].

For remote procedure calls the one-way multiple consumer approach is not appropriate. For this request/reply based services can be used [18]. A service can be defined as a request-response message pair. This structure can be seen on **2.5** block A. They are structurally independent and follow the same semantics rules as topic messages. Services use the same infrastructure as topics. Services are registered by name and clients can call the service by sending request messages and waiting for replies [18]. This call is blocking and functions as a remote procedure call. ROS provides tools for service message generation and service type management [18].

When a non-blocking goal-oriented request is needed that can take an unknown amount of time ROS Actions can be used. With actions, a goal and feedback channels are defined that are used to asynchronously update the client on the progress of a started action. Within actions use topics to exchange information. Actions can be started, queried and stopped at any time the system requires, for example, a move action can be stopped mid-execution when needed.

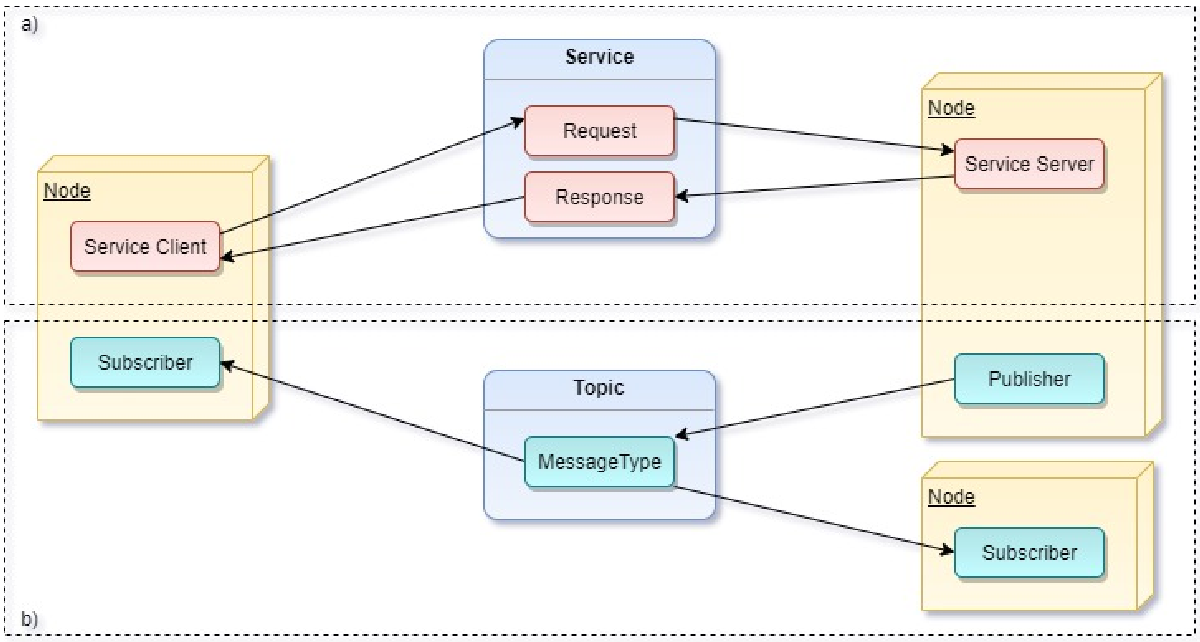


Figure 2.5: A collection of 3 ROS nodes using one service and one topic for communications. A) contains a ROS service. B) Contains a ROS topic with 1 publisher and 2 subscribers. Both communication types can be used in parallel by the same nodes [31].

2.5 TeMoto framework

TeMoto is a ROS-based software framework that is designed to facilitate building robust autonomous robotic applications. It allows for dynamic system resource allocation and management. Resource redundancy and dynamic handling of situations is the core idea of TeMoto, allowing for dynamic changes in mission scope and efficient resource managing, enabling code reuse, increasing development efficiency and system fault tolerance [32]. For example, if a remote sensing robot, equipped with numerous sensors and actuators, is deployed in a remote area with multiple zones to survey, TeMoto provides tools for a) handling the mission logic, and b) in case of sensor failure, substitute broken down resources with similar or combined resources.

2.5.1 Architecture

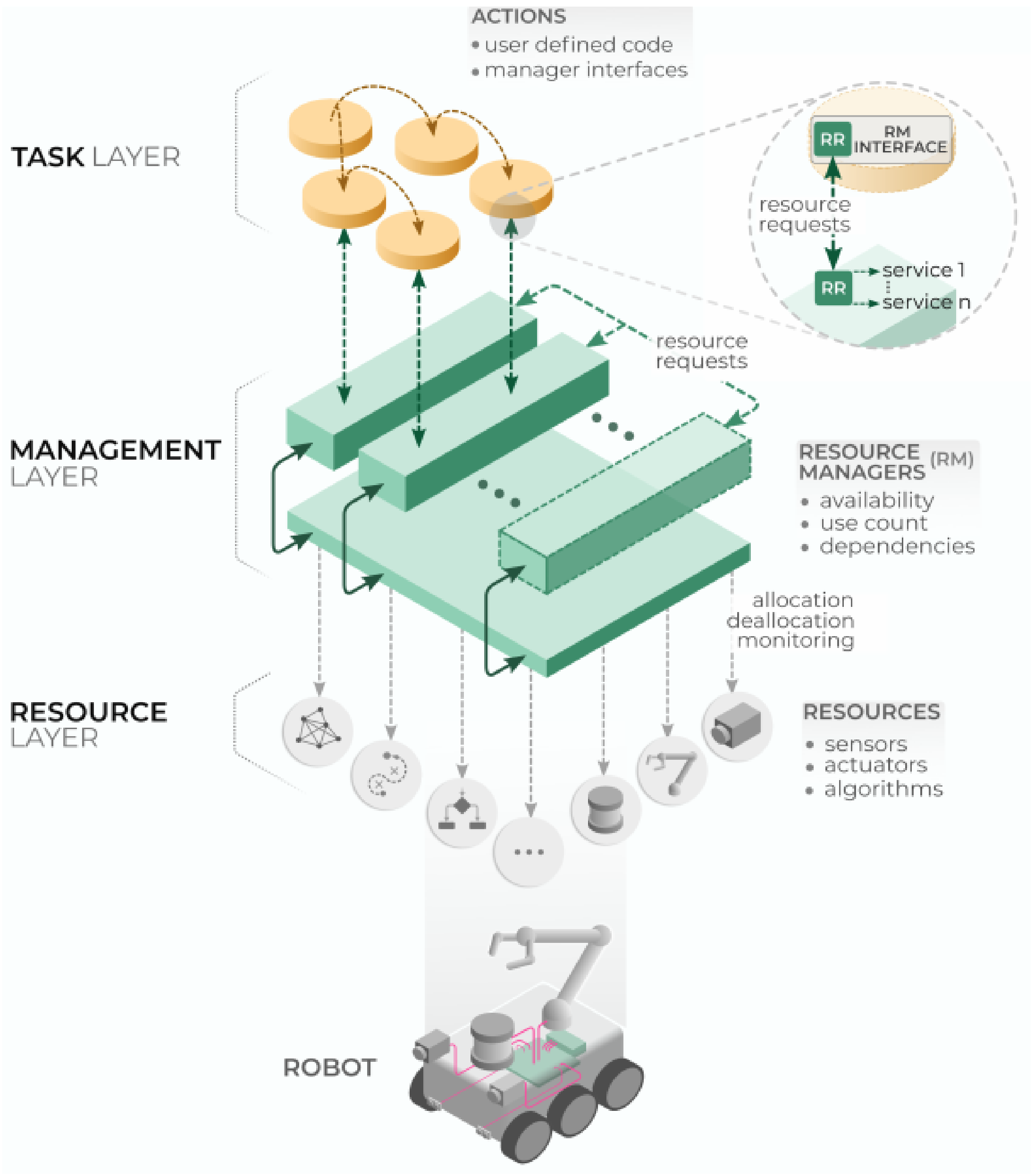


Figure 2.6: Overview of the architecture of TeMoto framework [33]

The TeMoto framework uses three distinct concepts as its foundation [34]:

- Action - A developer-defined module of code. It is executed dynamically and can contain custom logic such as device behaviour.

- Resource - Something that is allocated or deallocated on-demand, such as a camera. Resources do not need to be strictly hardware devices, software components, e.g. algorithms or combinations of sub-resources can be defined as a resource.
- Resource Manager - A subsystem responsible for system resource requests. A resource manager can handle different types of resource requests.

Figure **2.6** shows how the TeMoto framework is conceptually separated into three layers. The task layer consists of actions that are implemented as C++ shared libraries that contain developed code. Actions can utilize interfaces provided by the resource manager for access to system resources. Every manager contains several defined resources managed by a Resource Registrar (RR). RR tracks resource dependencies and their usage, allocating them on the first request and deallocating resources when the last client releases any specific resource. Resource Managers are responsible for the process of initialization and unloading of resources while RR provides a triggering mechanism for said implementations. The resource layer contains all managed resources available to be requested by a system.

2.5.2 Resource Registrar in TeMoto

In TeMoto the concept of a resource is very broad. It can be a hardware component, a piece of software or a combination of resources. To efficiently manage these resources RR needed to solve the following issues [34]:

- Resource sharing - How to share a resource without allocating it multiple times?
- Resource usage - How to make sure no clients are using a resource that is being deallocated?
- Consumer notification - How to notify a resource client if the status of a resource has changed, e.g., the resource has failed?

Every TeMoto resource manager integrates a Resource Registrar for providing access to functionalities concerning resource allocation and deallocation. All resources share the same allocation, usage and deallocation properties. Additionally, RR tracks the usage of resources and their dependencies making it vital for any subsystem that provides or requests resources. A resource registrar consists of three main components:

- Resource Catalog - Responsible for keeping track of all active in- and outbound resource queries and their inter-dependencies.
- RR Servers - Responsible for processing incoming resource requests. Each server is bound to a resource allocation and deallocation callback that is implemented inside the manager.
- RR Clients - Responsible for mediating outbound resource requests.

An illustrative example of why a RR is important can be seen in figure **2.7**. A user-defined action requires a resource consisting of multiple sub-resources: a camera and a filter algorithm. There are two resource managers in the system:

- Component manager - Maintains information about sensors, algorithms and their combinations.
- External Resource Manager - Provides capabilities to start, stop and monitor Operating System (OS) processes upon request.

First, the action requests the filtered camera resource from the Component Manager (fig. 2.7a), which knows that this resource consists of two sub-resources. Requests for the two sub-resources, to start the camera and the filter process, are sent to the ER manager. After the required processes have been initialized a response is send back to the Component Manager and from there to the requesting action that the resource is now available. Next, an independent action requests the same filtered camera from the Component Manager (fig. 2.7b) but since it knows that this resource has already been initialized and the same response as with Action_1 can be used. It is noted down that now two actions consume the same resource. Finally, when an unexpected camera failure occurs (fig. 2.7c) all consumers of this resource are notified, in this case, the Component Manager. The component manager propagates this notice to both actions, letting them know that the resource has failed. From here on it is up to Action_1 and Action_2 to decide on a course of action. They can, for example, terminate, request an alternative resource or any other appropriate action.

The internal structure of a resource registrar inside different managers is as illustrated in figure **2.8**. Based on the previous example a resource request is received by a RR Server. This server allocates the resource and stores its response data inside the Catalog. In the case this resource has other resources, RR Clients are used for sending outgoing resource request to

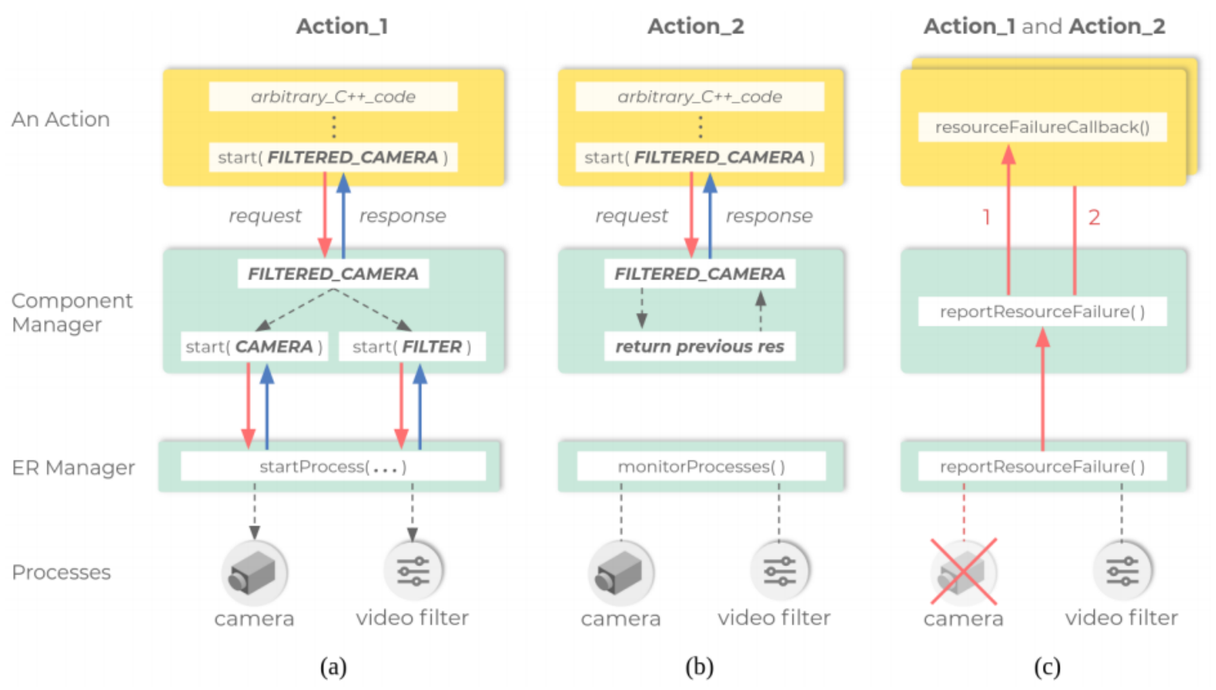


Figure 2.7: An example situation illustrating RR functionalities. (a) Action_1 requests a resource (FILTERED_CAMERA) that consists of 2 separate sub-resources - CAMERA and FILTER. (b) Action_2 is independent of Action_1 and requests the same FILTERED_CAMERA resource. (c) The camera sub-resource encounters a failure and all consumers in the dependency chain are notified of the event[TODO: REF]

other Resource Registrars, for example, to start a camera process. At this point, the dependency between a resource and its sub-resources is stored inside the Catalog.

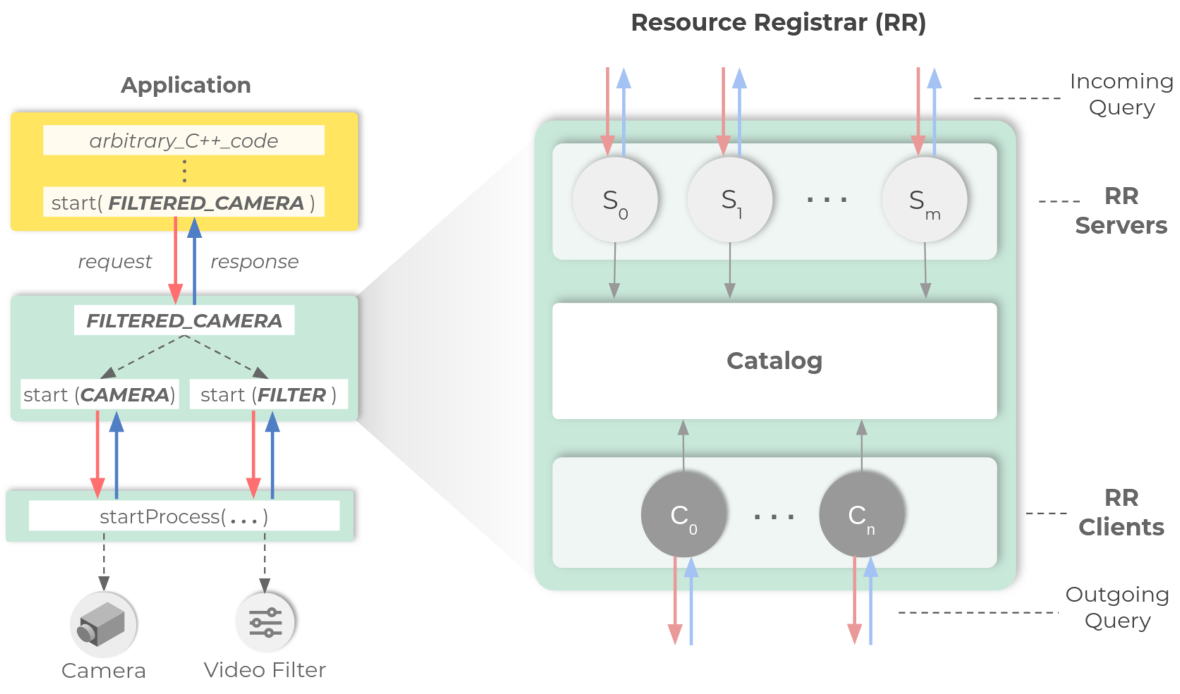


Figure 2.8: The structure of a Resource Registrar inside a application. Incoming resource queries are received by RR Servers that resolve them and store the results inside the Catalog. In the case of a resource depending on a sub-resource a RR Client executes a new resource query to allocate that resource.

2.5.3 Limitations

The current implementation of RR follows a monolithic architecture where all managers are strongly coupled to registrars. RR itself is strongly coupled to the ROS framework, making both, changing the base implementations of RR and the managers difficult because the scope of changes can not be accurately assessed. A system following modular design patterns would help to alleviate current difficulties in developing RR features. This requires a complete redesign of the Resource Registrar to be framework agnostic and must consist of only implementation-independent structures that can be extended on demand. A comparison between the current implementation and a newly proposed solution can be seen in table **2.2**.

Drawbacks of the current RR are listed below:

- Framework locked design - The RR is built with ROS in its core. All functionalities depend on ROS and it can not be used in systems that do not utilise ROS. This makes porting functionalities to other frameworks, such as ROS 2 difficult.

- No catalog backup - The current implementation does not support RR recovery after a manager crash. All data held about resource allocation and dependencies is held in memory and lost in the case of a system crash.
- Core functionality untested - The current RR does not implement unit tests that can be run to check the health of core functionalities after a change. It is difficult to validate the operation of all features and regression testing needs to be done manually.
- Code complexity - Because there are no automated tests for the RR refactoring is difficult because it is difficult to judge the scope of a change. This leads to unneeded code complexity because features can not be simplified safely.
- Circular dependency resolving - During resource allocation an allocation request can call itself, leading to a circular dependency. This is currently resolved by creating 2 RR-s inside a manager that can resolve dependencies between each other. This adds unneeded complexity to the system.

Table 2.2: Compassion of functionalities of a resource registrar.

Criteria	Before this work	This work	Optimal
Resource type agnostic	Yes	Yes	Yes
Framework agnostic	No	Yes	Yes
Extendability	Difficult	Simple	Simple
Weak coupling	No	Yes	Partial
Allocation monitoring	Yes	Yes	Yes
State restoration	No	Yes	Yes
Recursive resources requests	Partial	Yes	Yes
Managed resource amount	Unlimited	Unlimited	Unlimited
Client status callbacks	Yes	Yes	Yes
Server status callbacks	No	Yes	Yes
Automated feature validation	No	Yes	Yes
Automatic resource deallocation	Yes	Yes	Yes
Thread safety	Partial	Full	Partial

3 Related work

Resource monitoring and management is not a problem specific to robotics. Systems managing communications between multiple parties can encounter this roadblock when grown large enough. The evolution of web applications is an example where single servers are turning into multi-node network meshes to cope with the changing load patterns [35] [36] [37]. Another field requiring efficient resource management is power delivery where electricity-generating resources have to be balanced with consumption to keep the grid up [38]. With either management web applications or the power grid, mismanaged resources can bring with it disruptions or cascading failures, making them vital for everyday operations [39].

3.1 Resource management in non-robotics domains

3.1.1 Micro-service based web applications

Web applications are moving towards micro-service centric architectures where multiple independent services interconnect to form an application and its functionalities. Separate services scale horizontally and increase redundancy but increase external complexity in the infrastructure, requiring more configuration and dynamic monitoring since the service can be spread across multiple computational units.

Services of a micro-service pattern are often packaged into containers. These containers are isolated from each other using OS-level virtualization, contain software dependencies and configuration required for a service instance. The same container can be used on multiple computers and they function as resources.

For container orchestration software such as Kubernetes or Docker Swarm is used. APIs for orchestration is provided that deploy, maintain and scale applications and containers based on metrics such as CPU usage and memory allocation. Recovery routines for system applications can be defined that trigger alert routines or restart services according to need.

Multiple orchestrators can be connected to enable resource sharing between them, even if they are located on separate sites. HashiCorp Consul is used that creates and manage a service mesh that provides a service-to-service communications layer allowing to share resources between managers.

These components come together to form a dynamic web application that can configure itself based on demand and react to resource failure. These are only some tools that can be used to create and manage a micro-service based application.

3.1.2 Power grid management and load distribution

Power grid management is another domain using dynamic resource management. Electricity is consumed as it is created and balancing demand with supply is critical. Large changes in demand and supply can affect the frequency of the power being delivered. High loads lower the frequency of the grid while lower loads increase it. Power generators have a range of operating frequency that has to be sustained for safe operation. In the case of Europe, 49.0 Hz is the critical threshold before generator performance starts to deteriorate [40].

In equivalence to TeMoto, power generators and transformers can be considered the resource of the power grid. They are responsible for creating the flow of electricity and transforming it into a usable form. All resources connected to the same grid work in a synchronized manner as set by the grid parameters such as voltage and frequency.

The grid itself functions and infrastructure to transport power between resources and consumers. Power is shared with all parties that are connected to this network.

Balancing authorities provide oversight of the power grids current state. When the grid starts to go out of balance resources are requested to counteract this. Resource requests can include starting up more generators to increase available power but also to disconnect excess consumers, lowering system load.

As a result, the power grid tries to provide stable and consistent power to as many consumers as possible. Even when demand exceeds the supply smart management sheds load to keep as much of the system operational as possible.

3.2 Resource management in robotics

3.2.1 Rorg

Rorg is a toolkit leveraging the container engine Docker on Linux to run robot software in individual containers. Relationships between them are managed by Rorg exposing functionalities for starting, stopping and restarting containers. Using dynamic service allocation, similar to micro-services, it was possible to reduce the CPU utilization of a tour guide robot from 89.4% down to 48.6% with memory usage dropping from 3.41 GB to 2.85 GB [41].

A robot can be seen similar in structure to a data centres application where containers such as Docker are already used with micro-service patterns where many small services come together to provide larger functionalities. This also exists, to some extent, in ROS-based robots. Rorg provides a management layer similar to Kubernetes that can orchestrate and manage different sub-services of a robot [41].

The base element in a Rorg system is a service that represents container images that can be viewed as standalone programs. These services can request another service. This initialized the requested service and creates a relationship between them. Chains of dependencies can form this way. When a service is no longer needed it can be terminated, releasing its system resources [41].

Containers help to increase dependency reuse and reduce version conflicts. It is also fast to redeploy a service in case of execution environment corruption. At the same time, the creation and management of a Docker image increase management overhead on the overall system.

3.2.2 DyKnow framework

DyKnow is a framework built on top of ROS for configuring and maintaining resources that provide data streams. It monitors and reacts to system changes, re-configuring a system during run-time to provide clients with data they require [42].

DyKnow consists of mainly two types of entities: a DyKnow manager and computational resources that consume and produce information streams. The manager keeps track of the resource states, how they are connected and can reconfigure these connections [42]. Developers provide access to resources, such as sensors and detectors and through the DyKnow manager clients such as operators or other devices can request changes to the connection configuration.

The node and nodelet handling logic of ROS are extended to support for remapping of target topics. Internally a proxy handler is provided that connects DyKnow resource ports to ROS topics. This allows for swapping out of topics in run-time and for system reconfiguration over time. As an example scenario consider a robot that drives to its target using a depth-sensing camera. This camera provides the robot with information as to how far in front of it obstacles are. With the unexpected failure of the camera this resource is replaced in run-time with a Light Detection and Ranging (LiDAR) sensor and the robot can continue its mission. Compared to TeMoto that allows a resource to be any part of the system - a stream, an algorithm, a service, DyKnow supports only data streams as a resource.

3.3 Comparison

Resource management is not a unique problem inherent to robots systems. Table **3.1** summarized similarities between the examined systems.

Rorg manages standalone components of a system by wrapping them inside a container. In its current form, Rorg is build to fulfil a specific task in guide robots while the TeMoto framework provides a task agnostic solution.

DyKnow provides similar resource management capabilities to TeMoto when it comes to data streams. At the same time features that allow TeMoto to dynamically handle a wider range of scenarios, different managers are not replicated. Task management is not supported and the framework itself provides only partial task agnosticism.

Table 3.1: Resource management systems.

Layer	Robotics	Micro-service based web application	Electrical power management
Application and resource consumer	Robot capabilities such as decision making, actuator manipulation, telemetry fetching, etc.	Data provided by applications, such as audio and video streams or services for processing data	Electrical appliances and other electrical devices
Management	TeMoto, Rorg, DyKnow, etc.	Service mesh management systems such as HashiCorp Consul	Balancing authorities
Infrastructure	ROS, OROCOS, ROCK, YARP, etc.	Container managers such as Kubernetes, Docker Swarm, etc.	The grid and its structures
Resources	Robot sensors, actuators, behaviour routines etc.	Docker containers and other kinds of applications	Generators, transformers, etc.

4 Design and implementation of Resource Registrar

The goal of this work was to address the limitations of TeMoto's Resource Registrar. As a result, the previous RR was completely redesigned and re-implemented. The most significant change is the framework-agnostic design that allows using resource management functionalities without robotics specific frameworks. A feature comparison between the previous and current versions of the RR can be seen in table **2.2**.

Requirements for the redesigned RR are laid out in the **section 4.1**, the architecture of the registrar is described in **section 4.2** and the implementation specifics of the framework-agnostic software is given in **section 4.3**.

4.1 Requirements

4.1.1 Functional requirements

Functional requirements for the system are as follows:

- Dependency-free - RR functionalities are independent of the host application. Any implementation-specific code has to extend RR.
- Serializability - Resource queries need to be serializable so they can be stored inside the Catalog.
- Backupability - RR must be able to save its state externally and load said state on request.
- Configurability - The base functionalities of the RR are configurable without requiring the recompilation of code, such as the RR's id and backup directory.

- Individual callbacks - Every resource server needs to support separate load and unload callbacks that are triggered on the initialization and unloading of the resource.
- Run-time resource declarations - Resource servers and clients need to be created in run-time. Their configuration is not known and needs to be created dynamically.
- Resource status notifications - Resource servers need to be able to communicate with clients that have required their resources. In the case an event, for example, a fault, took place inside a resource server it must be possible to notify clients dependent on the resource.
- Graph-like sub-dependencies - Resources can consist of multiple sub-resources that form a graph. This graph needs to be resolvable by the RR.
- Testability - The functionalities provided by the RR must be covered by automated tests.
- Resource client cleanup - When a client is removed from the RR all resources associated with it are unloaded.
- Resource server cleanup - When a server is removed from the RR all resources associated with it are unloaded.

4.1.2 Non-functional requirements

Non-Functional requirements for the RR are as follows:

- ROS compatible - The system is designed to impose minimal development time for integrating with ROS.
- Generic - The base implementation can not use middleware or frameworks specific for robotics in its functionalities.
- Code follows the ROS C++ conventions - ROS has defined conventions for code style. RR Core fill follow these guides [43].
- Continuous integration - The project has a Continuous Integration (CI) pipeline to automatically validate any new functionalities once they are committed. A notification is sent out in case of validation failure.

- Error handling - When a resource encounters an error it needs to be propagated in the system to the consumer. The RR must not fail due to errors that were caused externally by the host application during resource allocation or deallocation.

4.2 Architecture

This chapter covers the architecture of the redesigned RR from its configuration and setup to system data flows. The five main structures are laid out and their inner workings described. These structures include the Configuration, Catalog, Resource Queries and the Resource Clients/ Servers handling resource calls. The Resource Registrar uses the structures to connect the applications that require a resource with applications that provide the given resource and vice versa, allowing for resource loading, unloading and propagation of resource status notifications.

Throughout this work, a *resource consumer* refers to an application that requests a resource and *resource provider* refers to an application that is responsible for loading the requested resource.

The architecture of RR was designed to handle resources as a generic concept. Anything that a host application exposes can be defined as a resource. These defined resources are bound to a RR instance and can be accessed using Resource Queries from other RR instances. Communications channels between registrars and the internal structure of a RR instance are illustrated in figure **4.1**.

4.2.1 Configuration

A RR can be initialized using a defined configuration structure. This configuration simplifies the setup of a RR inside a host system by allowing different functionalities to be enabled or disabled from one place. Configurable functionalities include the following:

- Resource Registrar ID - Unique identifier for the Resource Registrar.
- Backup location - Directory where the current state of the Catalog is saved.
- Backup interval - Time interval when the catalog is written to the backup location in seconds.
- Save on modify - Saves the Catalog every time a Resource Query modifies it.
- Erase on destruct - Deletes the Catalog backup when the RR instance is destroyed.

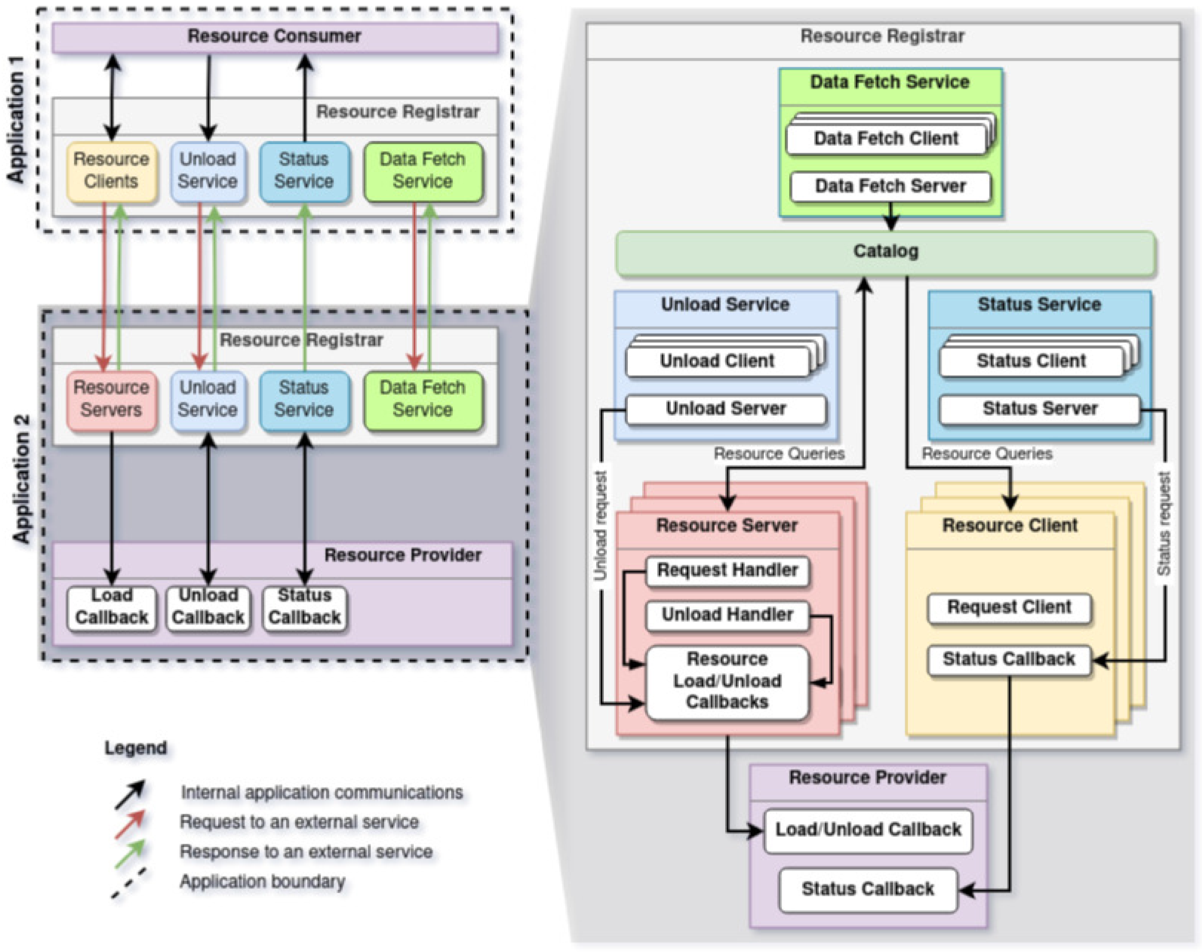


Figure 4.1: Block diagram describing the architecture of the RR. RR instances can execute procedure calls inside other RR's using services. Internally Resource Servers and Clients are connected to the Catalog and execute the callback method that was provided by the host application.

4.2.2 Resource Catalog

The Catalog holds resource queries and their dependencies. It contains all resource query data, query dependencies and responsible servers/clients for each query, illustrated on figure **4.2**. This information allows to track resource allocation by identifying if a resource has already been allocated and allows to trace a queries origin and sub-resources for message propagation. These messages can include resource unload calls that need to be sent to every resource allocated by a query or resource server status updates that need to be forwarded to every client that has the resource allocated. Additionally, the Catalog can be stored as a file and its contents can be imported into a RR instance in case of system recovery.

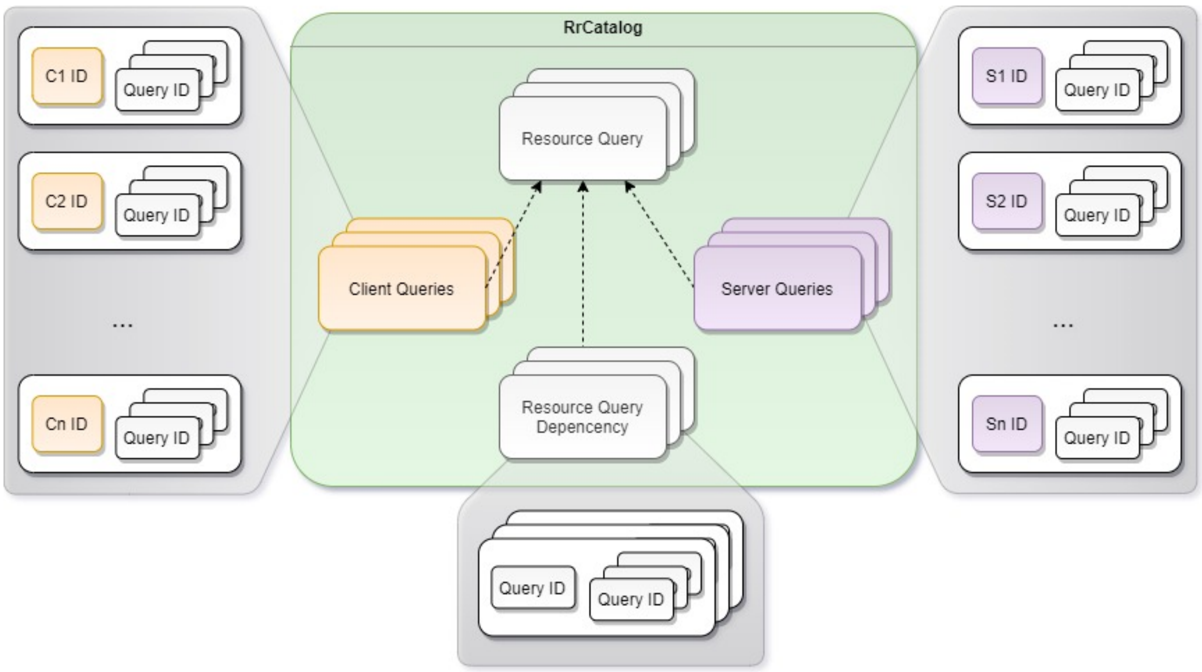


Figure 4.2: The internal structure of the Catalog. Unique resource queries are stored with their metadata is stored. Metadata includes references to Resource servers/clients and queries server/executed by them and Resource Query Dependencies connecting a query ID to its sub-resource ID's.

4.2.3 Resource life cycle

All resources follow the same life cycle of first being declared, after which they can be requested, loaded and later unloaded. Every RR has a set of resources that are defined in run-time and are accessible through Resource Registrars. Clients for communicating with Resources

Servers are created dynamically upon the first Resource Query. Resource queries and ID's are stored inside the Catalog.

Resources managed by a RR are accessible over Resource Servers. During the creation of a new Resource Server, it is connected to a host application provided resource using callbacks. These callbacks are used to relay a resource allocation or deallocation request from a registrar to the host application. Optionally a status callback can be defined for a server. The callbacks usage is described in **subsection 4.2.4**. Servers have access to the Catalog for query processing, such as uniqueness checks or allocation counts for resources. Servers store and remove resource queries from the Catalog.

Resource queries target a Resource Server and use Resource Clients inside a RR. Clients are created in run-time whenever a Resource Query is invoked by the RR resource call method. Created clients are reused for all subsequent calls targeting the same resource server. When a resource client is created a status callback can be attached, as with resource servers.

A resource can be requested using a Resource Query. It contains resource-specific data needed for its allocation and metadata required by the Resource Registrar for functionalities such as dependency resolving. This data is stored inside the Catalog and is listed in **Table 4.1**.

Table 4.1: Metadata fields inside a Resource Query.

Metadata field	Description
Query ID	A unique identifier assigned to the query by a resource server. This identifier needs to be unique across all RR's used in the system.
Origin Resource Registrar	ID of the Resource registrar a query originates from. Used for status callbacks resolving to find the target RR.
Target Resource Registrar	ID of the Resource registrar a query targets. Used for unload callback resolving to find the target RR.
Query status	Status of the finished query. Identifies if the query was successful or encountered an error.
Request metadata	Internally used request data.
Response metadata	Internally used response data.

Figure **4.3** shows the process of loading a resource. Firstly the resource consumer (user application) passes the Resource Query to the RR, which uses an RR Client to handle the query.

If a client for a said resource does not exist it is created and cached inside the RR for further use. This client sends the Resource Query to the target Resource Server. Inside the server, the query is checked against the Catalog to determine if such a resource has been initialized. In the case of a unique Resource Request the load callback, located inside the host application, is executed. The result of the query is stored inside the Catalog and also returned to the resource client. After this, the resource is successfully allocated and can be used by the resource consumer.

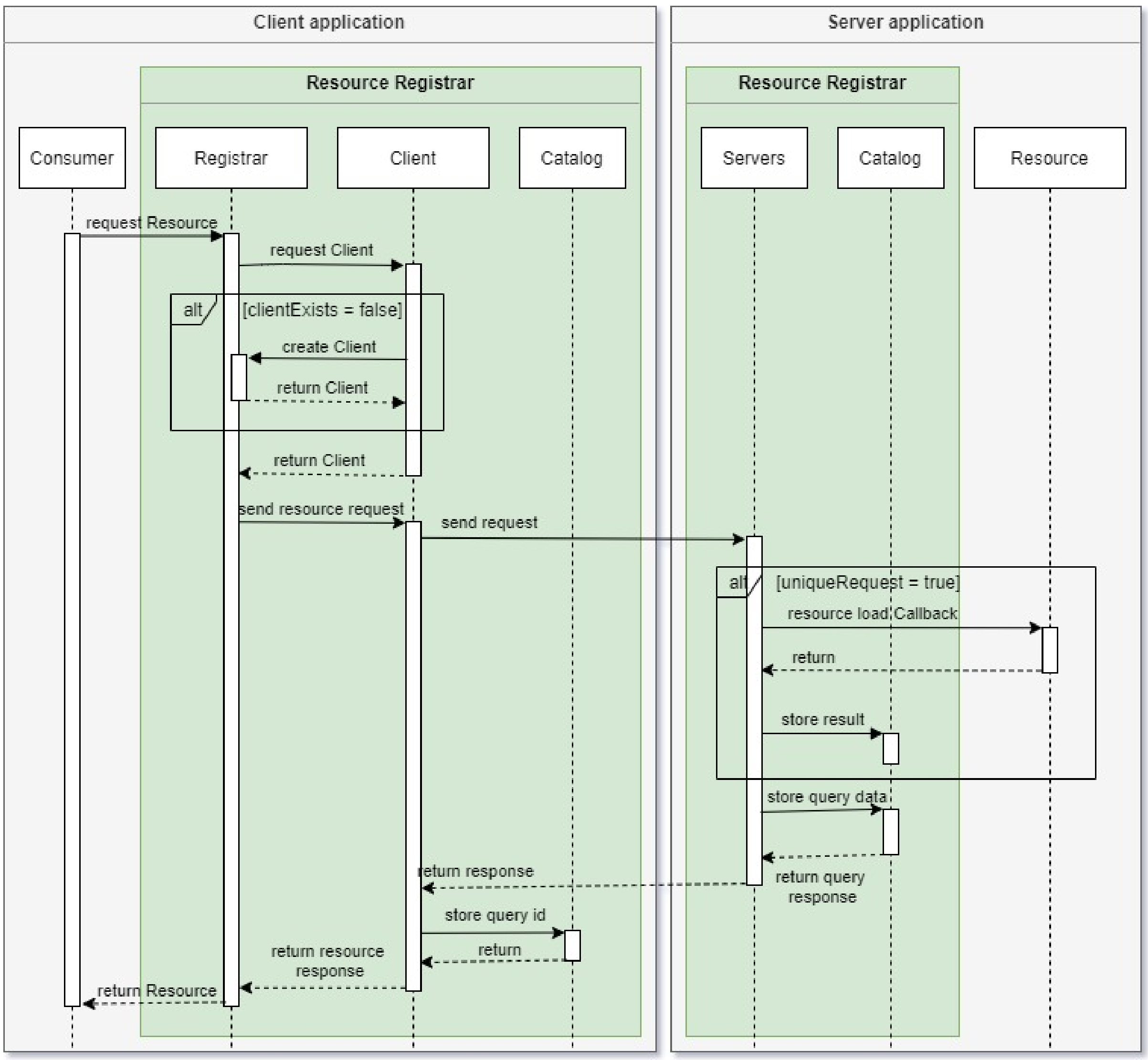


Figure 4.3: Flow diagram describing requesting of a resource.

Figure **4.4** shows the process of unloading a resource. First, the query ID of the resource-to-be-unloaded is passed to a RR. With this ID the Catalog finds the RR that is managing this resource. For every resource being unloaded a resource unload client is resolved in the same manner as with Resource Clients. When a resource server receives a unload call it removes the resource allocation entry from the Catalog and checks if any consumers for the resource remain. If there are none, the resource unload callback is executed, the resource is deallocated and the Catalog purges the Resource Query. If the resource had sub-resources and unload call is executed per each sub-resource. Every unload call in the chain is handled in the same manner until no more resources are to be resolved.

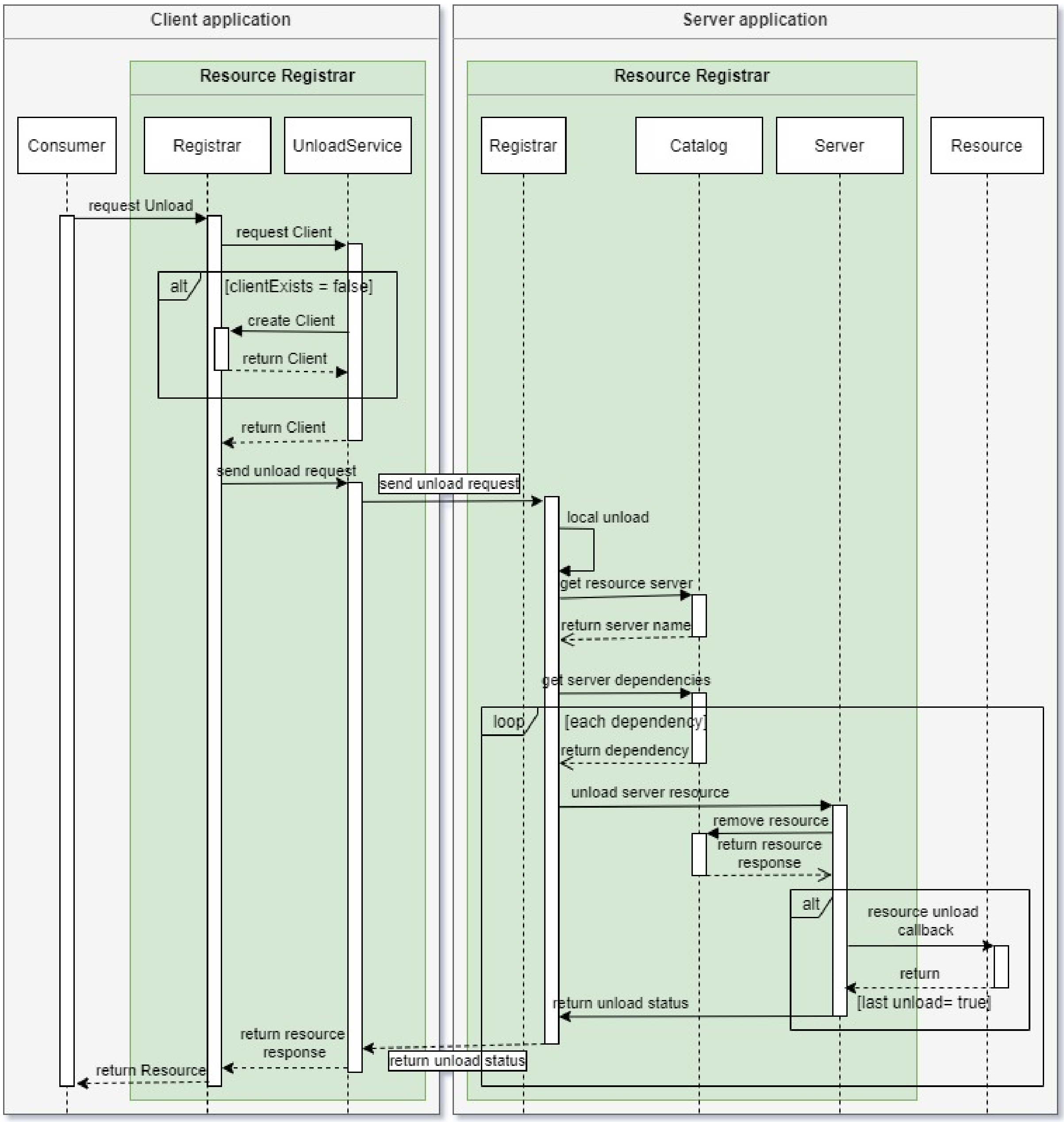


Figure 4.4: Flow diagram describing unloading of a resource.

4.2.4 Resource status notifications

Status notifications are used to propagate messages from Resource Servers to consumers, i.e., resource failure or reallocation.

Every RR has a status handler that is initialized during startup and is responsible for processing status calls. A status query contains a message and a status code that will be propagated to consumers of a resource. This process starts with the fetching of a status client using the same mechanisms as are used by resource and unload clients. The Catalog is used to find consumers of the resource and to include the resource query inside the status query. If the resource server has a status callback defined, it is executed at this point. The status query is then sent to the resources consumers. On the receiving RR's side, the Resource client responsible for this resource call is identified and the status callback is executed if defined. The same behaviour repeats until the initial resource consumer is reached. This flow is illustrated in figure **4.5**.

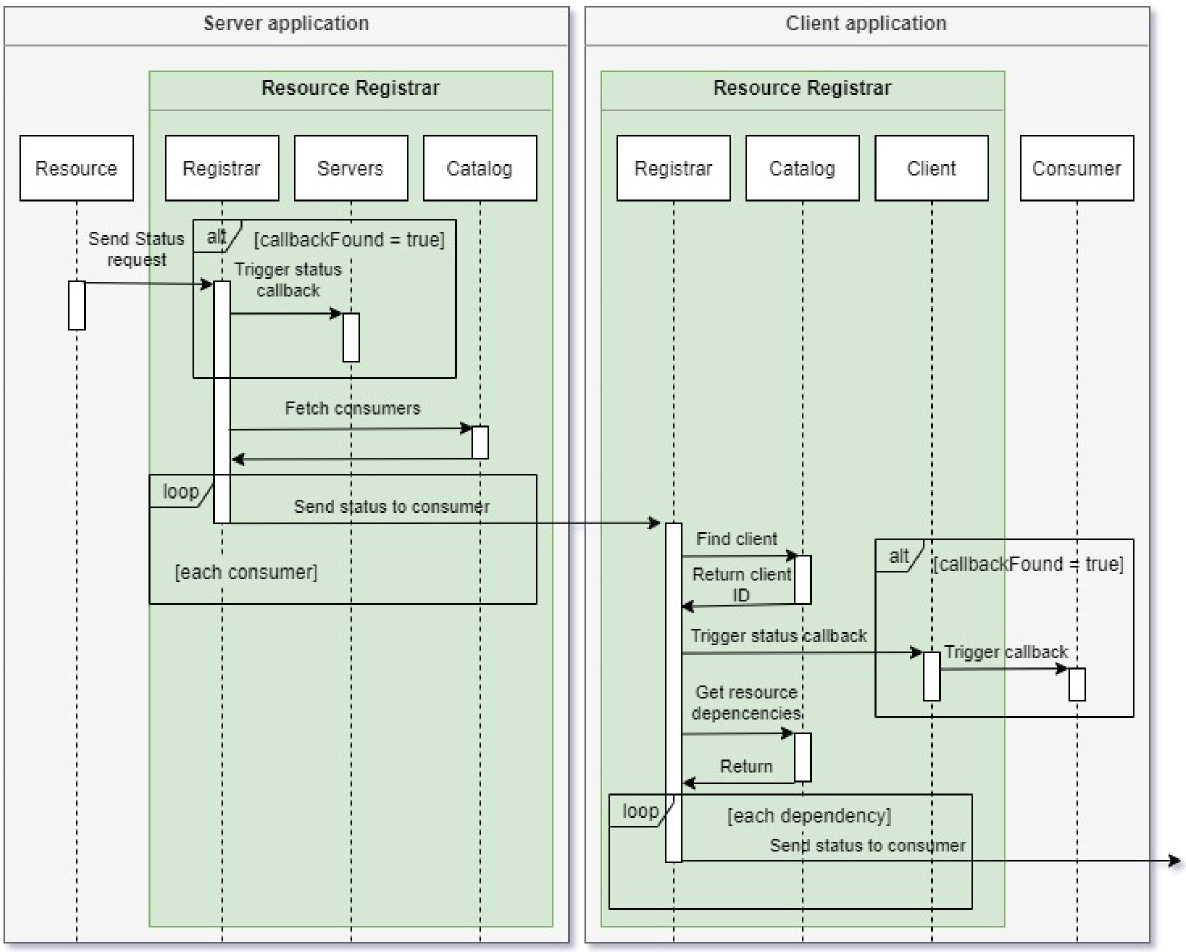


Figure 4.5: Flow diagram describing the status notification propagation of a resource.

4.2.5 Resource Query retrieval from other Registrars

To avoid data duplication issues and loss of synchronisation between RR's, the RR Catalog stores only Query ID's but not the whole queries. Yet in the case of consumer application recovery, the consumer might need to get the information about queries it has executed. For this, RR can fetch all Resource Queries from the RR instance that managed the resource.

A query fetch handled is initialized during RR startup. Query fetch clients are created and cached inside the RR. A query fetch request consists of a resource query ID and target server that determine the resource server and originating RR. All queries that originate from the calling RR are sent back as a response to the host application and they can be used to execute required recovery routines.

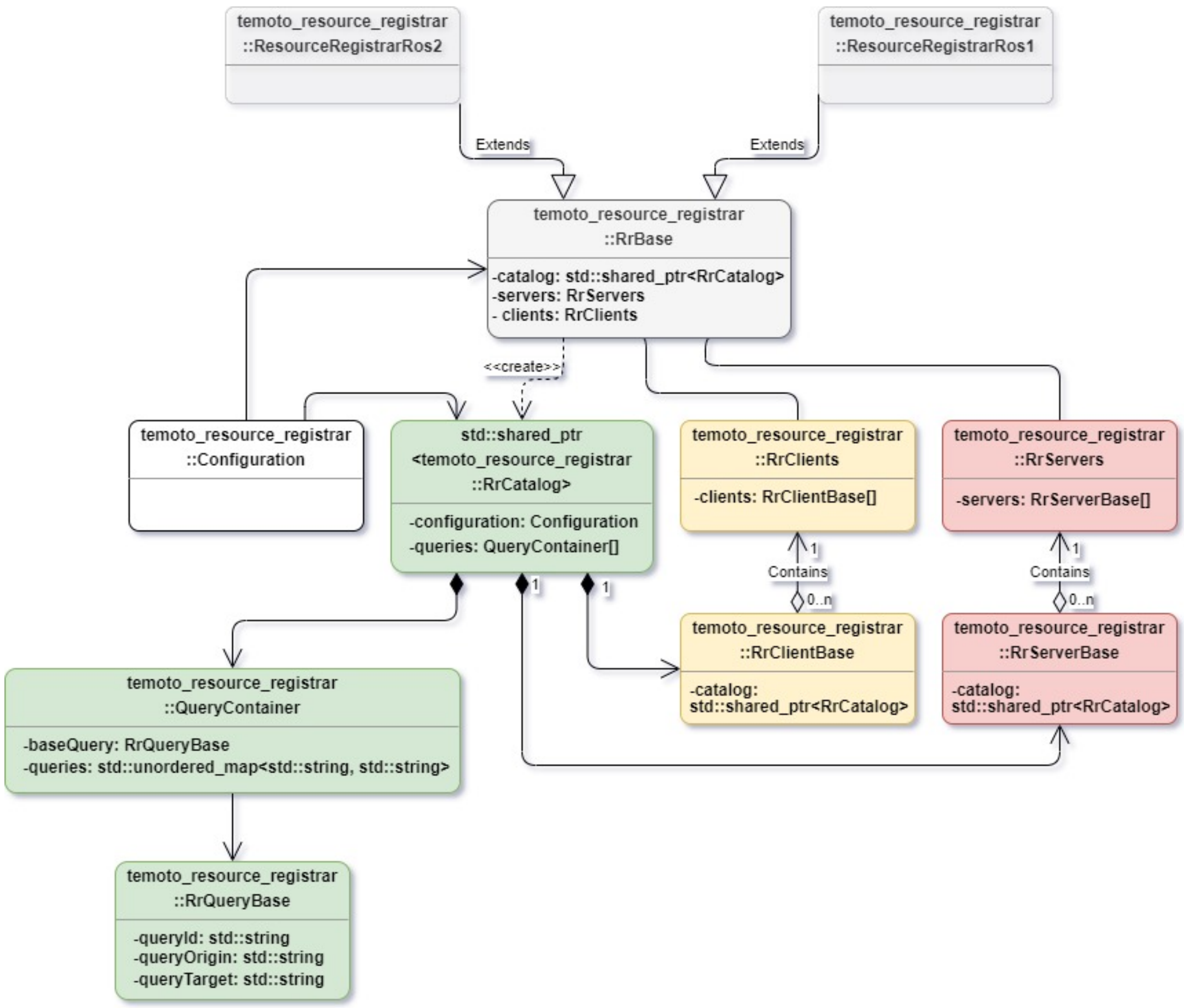
4.3 Implementation

This chapter covers the implementation-specific details of RR, which was implemented in C++, relying on the functionalities from the C++14 standard. Concepts of polymorphism, inheritance, typed methods and overwriting provided by the language are utilized to create a dynamic and extendable set of base functionalities. Test Driven Development (TDD) principles were used when developing features of the registrar where features are developed in tandem with tests. Tests are maintained during development cycles. When code changes are done they can quickly be validated with existing tests. The code is freely available on GitHub [44]. Build instructions are detailed in **Appendix A**.

4.3.1 Resource Registrar

`RrBase` is the class providing base implementation of methods required for resource handlings such as loading and unloading resources, notifying of clients, and registering of servers. Functionalities that may need expanding based on use cases are defined as abstract for overwriting. The internal structure of the class is illustrated in figure **4.6**.

The RR can be constructed either by only specifying a name for the RR or by a separate `temoto_resource_registrar::Configuration` object. When using the name constructor default values are used for the configuration. Table **4.2** shows all properties that can be programmatically adjusted, their description and default values.



temoto_resource_registrar ::ResourceRegistrarRos2
temoto_resource_registrar ::ResourceRegistrarRos1
Extends
Extends
temoto_resource_registrar ::RrBase
-catalog: std::shared_ptr<RrCatalog>
-servers: RrServers
- clients: RrClients
<<create>>
temoto_resource_registrar ::Configuration
std::shared_ptr <temoto_resource_registrar ::RrCatalog>
-configuration: Configuration
-queries: QueryContainer[]
temoto_resource_registrar ::RrClients
-clients: RrClientBase[]
temoto_resource_registrar ::RrServers
-servers: RrServerBase[]
1
Contains
0..n
1
Contains
0..n
1
1
temoto_resource_registrar ::RrClientBase
-catalog: std::shared_ptr<RrCatalog>
temoto_resource_registrar ::RrServerBase
-catalog: std::shared_ptr<RrCatalog>
temoto_resource_registrar ::QueryContainer
-baseQuery: RrQueryBase
-queries: std::unordered_map<std::string, std::string>
temoto_resource_registrar ::RrQueryBase
-queryId: std::string
-queryOrigin: std::string
-queryTarget: std::string

Figure 4.6: A diagram showing the temoto_resource_registrar::RrBase class relation to other parts of the system. RrBase has been extended to support ROS 1 and ROS 2.

Two maps of unique pointers of created servers and clients are held in the registrar. The registrar also has a shared pointer of the catalog (fig. **4.6**). This pointer is passed to every server and client instance created. The catalog pointer is updated in case contents are loaded into the catalog from disk.

Any methods that are potentially used in extended implementations are declared as virtual. This is to achieve run-time polymorphism that is needed for dynamic resource management since the type of resources being managed is unknown to the RR.

The destructor of the Resource Registrar is also defined as a virtual since other virtual methods can not be called from inside the destructor otherwise. Destructors and constructors call the final over-rider of the class, not the overriding method in a derived class. [45] This makes it impossible to use virtual methods in normal destructors. As a rule of thumb any time a class

Table 4.2: Configurable Resource Registrar properties.

Property	Setter	Description	Default
Name	setName (const std::string &name)	Defines the name of the Resource Registrar. Used as a server and client prefix.	std::string("untitled")
Location	setLocation (const std::string &location)	Defines the file that is used to load and save the catalog backup.	std::string("./catalog.backup")
Save Interval	setSaveInterval (const int &interval)	Defines the interval when the catalog is dumped onto disk. Units are in seconds.	int(60)
Save on modify	setSaveOnModify (const bool &saveOnModify)	Defines if the catalog is saved when it is modified by a load or unload operation.	bool(false)
Erase on destruct	setEraseOnDestruct (const bool &eraseOnDestruct)	Defines if the catalog is saved when it is modified by a load or unload operation.	bool(false)

has virtual methods it has to have a virtual destructor [46].

Class templates are required for all methods that directly handle resource requests. The registrar can not know what types of resource queries it has to process. To execute a resource call three template arguments need to be defined (listing **4.1**): server type, query type and status callback type. The server type is used to cast a base server pointer to the correct implementation type. The RR itself is generic and does not internally differentiate types of queries and status callbacks. Every extended call method needs to use the `privateCall` method since it implements client resolving, dependency storage and Catalog population.

The ROS 1 and ROS 2 implementation descriptions (sections **Resource Registrar on ROS 1** and **Resource Registrar on ROS 2**) describe how RR is specialized for specific robotics

```
template <class CallClientClass,
          class ServType,
          class QueryType,
          class StatusCallType>
void privateCall(const std::string *rr,
                 RrBase *target,
                 const std::string &server,
                 QueryType &query,
                 const StatusCallType &statusFunc,
                 bool overrideFunc)
{
    ...
}
```
Listing 4.1: Definition for the privateCall method that is used by all resource call methods.

middleware.

4.3.2 Resource Catalog

The Catalog is framework agnostic and holds query data that is required by all implementations in a serialized form to be generic and allow for Catalog storage. Message processing is located in the clients, servers and the registrar. They need to provide data suitable to the Catalog. The Catalog is serializable into a file using Boost [47] library functionalities.

Every data modification method inside the catalog is protected by a recursive mutex. This is to prevent data corruption since one thread can read and write into the catalog at one time. Without a mutex, race conditions can corrupt data inside the registry and data loss can occur. The used mutex is recursive to avoid dead-locks within a single thread.

Query data is used for storage, updating and checking for its existence inside the Catalog. Query data can be inserted as a whole, with request and response data or a response for a query can be supplied later on. It has to be taken into account that when a response is updated asynchronously functionalities that require response data, e.g. unloading, status messaging and existing query processing will not work as long as a response is not stored since these functionalities require data contained in the response. It is also possible to fetch a Resource Query based on its serialized form and to determine a Resource Query's initiators UUID based on any Query that has requested the same resource.

Dependency storage and checking are done using UUID-s. Every query can be fetched based on its identifier. In the catalog dependencies are stored as a map of ID-s that is connected to a list of dependencies and their Registrar instances. Dependencies can be unloaded using the original query ID.

To simplify status and unload callback handling the `RrCatalog` allows to track message identities bound to clients and servers. For status calls, this list of bound requests allows finding every unique server to send an update to. For unloading callbacks, it allows to connect a request to every target resource server and execute required subroutines for unloading.

4.3.3 Resource Server and Client

Resource servers are handled via unique pointers whose ownership is assigned to the RR during the RR Server registration process. RR Server's ID is defined as follows: resource_registrar_name + "/" + server_name. This ID can be used to communicate with the server. Clients are created when a request is first sent to a target. Every subsequent request reuses the created client.

Client objects are created when a request to a target server first takes place. Created clients are stored inside a map and can be accessed by the target server ID. Client object implementations must never use the catalog to fetch request responses. Otherwise, the server can not keep track of consumers behind a resource.

Servers respond to queries sent by resource clients. Servers are registered in the Resource Registrar beforehand. Resource load and unload callbacks need to be defined. These callbacks are used to signal a resource to start-up or shut down. These callbacks need to take the resource request and response as an argument. The server is responsible for assigning UUID-s to resource queries. These ID-s are generated using Boost provided functionalities. Catalog methods are used to check for query uniqueness and fetching of existing responses. Query metadata, such as UUID-s need to be removed from data before storing it in the catalog. Servers are responsible to sanitize data before storage. A method is implemented that cleans up the catalog when a resource is unloaded.

4.3.4 Resource request error handling

During a resource call, errors can occur that need to be propagated to the resource consumer. The `TemotoError` and `TemotoErrorStack` classes are used for this purpose. A resource load callback can throw a `TemotoErrorStack` object with a message and the origin of the error. This error is caught by the responsible resource server that stores this in the queries metadata. This in turn is picked up by the Resource registrar call that appends a new error into the stack and propagates the error, creating a stack trace. If an error is caught during a query with sub-queries already loaded resources are unloaded. The final error stack can be caught and handled by the request initiator.

4.3.5 Automated testing

Resource Registrar uses the gtest library [48] for its tests. Every major testable feature needs to be covered by automated tests. Tests are compiled as a separate executable and can be run as any other application displaying the result at the end. Tests can also be used for continuous integration or to allow checking the impact of code changes. All implemented tests are summarized in table **4.3**.

All tests are executed in the continuous integration environment. GitHub allows defining actions that run predefined scripts on certain events, such as code commits. The script compiles and runs the test executable. This validates that features included in the commit compile properly and do not break any existing functionalities. This provides an easy way to sanity test code and covers regression testing.

4.3.6 Limitations and nuances

The design of the developed solution introduced some nuances that had to be taken into account when developing and extending its functionalities. They are summarized below:

- A virtual destructors need to be redefined if the use of virtual functions is needed. This is the case for automatic resource unloading upon scope exiting.
- At the moment configuration loading needs to be implemented separately. A default implementation should be included in the base implementation.
- When many identical queries are run the calling application needs to record UUID-s of these queries for later unloading. Else it is impossible to completely unload the resource. A way to alleviate this is to have a way to hard-unload a resource but the implications of such a feature need to be analyzed as it can affect any number of resource consumers.

Table 4.3: Test suite of the RR with short descriptions.

Test ID	Description
ResourceRegistrarTest	Test for validating resource loading and unloading. Tests base functionalities of the Catalog. Uses a Consumer - Agent - Producer topology.
RegistrarConfigurationTest	Test for validating that properties set inside temoto_resource_registrar::Configuration are propagated and used properly by RrBase.
CatalogResponseUpdateTest	Test for validating that a-synchronous response updating works.
ClientUnloadTest	Validates that when a client is unloaded resources associated with it is automatically deallocated.
CallbackErrorTest	Validates resource_registrar::TemotoErrorStack propagation during a resource call. When an erroneous call has loaded dependencies, they are unloaded.
DataFetchTest	Validates that a client RrBase can fetch executed queries from a server in another RR.
RrServerStatusCallbackTest	Validates that when a status callback is triggered and propagated up the dependency chain a defined method is executed by every server and client in the chain.

5 Integration of the Resource Registrar with TeMoto

TeMoto in its current form is only supported in ROS 1. With the modular and decoupled Resource Registrar its functionalities are not locked to ROS1 and can be expanded to other frameworks, e.g. ROS 2, helping TeMoto expand to other frameworks. Implementation specifics for ROS 1 and ROS 2are discussed in this chapter.

5.1 Resource Registrar on ROS 1

Resource Registrar was extended to support ROS1 service messaging pattern. Resource Registrar defines a set of message types and services in ROS1 to support resource loading, unloading, statuses and query exchanging.

5.1.1 Message and Service types

For ROS1 one message type and three services were defined to support communications between registrars. The defined message times are `TemotoRequestMetadata` for requests and `TemotoResponseMetadata` for responses. They hold metadata required by the Resource Registrars to function. The defined services are responsible for resource loading, unloading and sharing of query data between registrar instances. Every resource handled by the RR needs to be defined as its own service.

The message types `TemotoRequestMetadata` and `TemotoResponseMetadata` need to be included by every Resource Server service. These messages type contains metadata required by the registrar to connect and store request calls. The contents of these metadata fields be seen in listing **5.1** and **5.2** for the request and response. Any additional fields required for resource requesting can freely be added to the service request and response parts. Listing

```
string servingRr
string originRr
string metadata
```

Listing 5.1: Fields defined by TemotoRequestMetadata.msg message field

```
string requestId
int16 status
string metadata #error stack support
```

Listing 5.2: Fields defined by TemotoResponseMetadata.msg message field

5.3 shows a ROS service definition of a sensor data pump and it is enhanced in listing **5.4** to support resource management.

Services used by the ROS1 resource registrar are the UnloadComponent, StatusComponent and DataFetchComponent. These services provide a way to send unload and status requests to other RRs and fetch served queries using the DataFetchComponent service. All service calls are handled internally and not intended to be used outside of the RR since they contain metadata that is contained only it the Catalog.

5.1.2 Serialization

Messages need to be serialized for storage inside Catalog. It must also be possible to convert them back into their representative object. ROS provides a mechanism to serialize its internal messages. The `MessageSerializer` class provides static methods to serialize ROS message type objects into strings and convert strings back to a specified type. This utility is heavily used in the implementation.

5.1.3 Client and Server

The `Ros1Client` and `Ros1Server` utilize ROS1 service/client patterns. A request/response pair is defined for every resource. The ROS client and server implementations create ROS specific service server and service client that can communicate with each other. Both objects are template classes. This template specifies for what service this object is responsible for.

```
string sensorId
int32 pollFrequency
---
string topicName
```

Listing 5.3: Example of a ROS 1 service declaration. A sensor can be requested to provide data with a set frequency.

```
TemotoRequestMetadata temotoMetadata
string sensorId
int32 pollFrequency
---
TemotoResponseMetadata temotoMetadata
string topicName
```

Listing 5.4: Example of a ROS 1 service declaration with TeMoto Resource Registrar specific metadata.

The `Ros1Client` class extends the base class `RrClientBase` and internally defines a ROS specific `ros::ServiceClient` instance. When the RR client object is dynamically created by the Resource Registrar a new instance of the client ROS client is initialized and used for communicating with the server. An additional status callback handler can be defined in the client that can be triggered by the Resource Registrar.

The `Ros1Server` class extends the base class `RrServerBase` and internally defines a ROS specific `ros::ServiceServer` instance. This ROS instance is used to receive client requests. The server needs to have a defined load and unload callbacks defined that are executed when required. These callbacks are defined in the extending class because their type is implementation-specific. The majority of the code is taken from the example class defined in the RR unit test. Requests must be sanitized before storage to avoid comparing messages with Resource Registrar metadata. Metadata can change uniqueness check results since message UUID-s are unique, making every query inherently unique.

5.1.4 Resource Query

`Ros1Query` extends the `RrQueryBase` class similarly as the server and client implementations and is typed by the resource service type. This object can convert a service request to a Base Query object and translate it back to the service call with all the metadata attached. Service calls must be wrapped before requests are processed by the base Resource Registrar functionalities.

5.1.5 Resource Registrar

The `ResourceRegistrarRos1` class extends RrBase to support ROS specific message methods. It needs to start-up support services responsible for unloading and statuses. Additionally, the destructor is overwritten to support automatic resource unloading when the object is cleaned up.

```
template <class QueryType>
    void call(const std::string &rr,
              const std::string &server,
              QueryType &query,
              std::function<void(QueryType, Status)>
                status_func = NULL,
              bool override_status = false)
    {
    ...
    }
```

Listing 5.5: The extended call method used inside ResourceRegistrarRos1 to execute ROS 1 specific Resource Queries.

One method that has been created to support ROS 1 is shown in listing **5.5**. This method follows the core design pattern but wraps all ROS specific objects into a more general form that can be used by the core system.

For unloading and sending status messages a similar concept is used. Before being able to be used by the Registrar the `void init()` method needs to be called. It initializes ROS services responsible for receiving appropriate messages. Client creation works the same as with resources where clients are created on the first request and reused on following executions.

5.1.6 Limitations

The ROS 1 implementation brings with it some limitations and peculiarities that need to be taken into account when starting to use it. These limitations are easy to resolve once aware of them but can become stumbling blocks for first time users. Limitations discovered during the development process were:

- As much information should be hidden from the user as possible and only essential functionalities are exposed. The current implementation does not exactly follow this principle and exposes many volatile functionalities to a developer that, when misused, can cause instability in the RR on even render the RR inoperable. At the same time, these functionalities are needed for advanced TeMoto functionalities, such as manager recovery after a shutdown.
- Recursive calls in ROS need an asynchronous spinner to be used. Every service callback inside a RR locks a thread and with recursive queries, multiple threads need to be allocated per RR. Without this, the serving of a resource query can deadlock while waiting for a thread for processing the next resource call.

- Care must be given to naming RR instances and resources inside them. ROS 1 requires all services to have unique names and when a collision occurs the previous instance is shut down. This can happen to resource servers and status, unload and query fetching services. When this happens to an existing RR its services are shut down and it becomes impossible to execute some functionalities of the RR.

5.2 Resource Registrar on ROS 2

To demonstrate the agnostic nature of the developed RR it was implemented in ROS 2 to leverage design improvements found in the new system. While ROS 2 shares many architectural patterns with ROS 1 the implementations have changed enough that porting functionalities is non-trivial. This chapter will go over the changes that were required to support ROS 2 when taking the framework established with ROS 1 as a base.

5.2.1 Message and Service types

For message and service types the structure inside ROS 2 has mostly stayed the same where msg and srv file defines the structure of data containers. Changes were done to how they are declared inside the code where messages and services now have their own namespaces to avoid collisions. As an example the `temoto_resource_registrar::UnloadComponent` is defined as `temoto_resource_registrar::srv::UnloadComponent` in ROS 2. For messages the `srv` portion of the namespace is exchanged with `msg`.

5.2.2 Serialization

Serialization in ROS 2 has been changed extensively compared to ROS 1 where a custom serialization format, transport protocol and discovery mechanisms were implemented. ROS 2 instead relies on an abstract middleware interface supporting these actions. The current ROS 2 implementation uses the Data Distribution Service (DDS) standard for its messaging layer. An adapter layer is introduced to help and map implementation-specific API's to the abstract interface, allowing ROS 2 to access the implementations functionalities. For the RR the default DDS implementation shipped with ROS 2 is utilized.

5.2.3 Clients and Servers

ROS 1 required the ROS master service, functioning as a DNS server, to be running for node discovery. With ROS 2 nodes can discover each other without the need for a master, decentralizing the system. This means that all servers and clients needed to be handled as nodes under ROS 2. Additionally, all service calls between clients and servers are asynchronous instead of synchronous.

ROS 2 defines a convention as to how to write nodes that compose a ROS system. Since resource servers and clients are part of a system they need to be fitted into these conventions. Such a preset structure keeps the implementation cleaner since it conforms to a uniform style. Objects needing to utilize the communications layer of ROS need to inherit the base Node class and its functionalities, i.e., `rclcpp::Node` for ROS 2 in C++.

ROS 2 changed the way messages are passed and handled. ROS 2 service calls are executed asynchronously and should be provided a callback function that is executed upon completion. While the ROS 1 base needed to be rewritten to support this behaviour it comes with the benefit of not blocking the node while it is processing a query. How a query was defined has also changed. While ROS 1 handled query requests and responses as references ROS 2 uses shared pointers. While it is functionally different the basic structures are kept the same, care needed to be taken while converting request handling to use the arrow(->) operator.

5.2.4 Resource Query

The query wrapper for ROS 2 was kept mainly the same as with ROS 1. RR specific metadata was stored in identically structured message types. Due to the change in the messaging mechanics, as described in the previous section, it was fitted to handle shared pointers instead of references.

5.2.5 Resource Registrar

As was the case with resource clients and servers service handling had to be rewritten because of ROS 2 structural changes. While most of the resource processing logic is handled by the base RR library, needing no additional modifications, features using communications channels, like unloading, status and data fetch services, had to be recreated to support the asynchronous nature of ROS 2 service calls. Additionally, resource call functionalities were converted over to support shared pointers.

6 Demonstration

This chapter demonstrates the outcome of this work with an autonomous navigation mission. The goal of the mission is to successfully reach a predetermined location while the robot is being subjected to a range of software and hardware faults. TeMoto is used to manage the mission logic, navigation and recovery from critical failures. TeMoto utilizes the RR for resource allocation and fault handling that is vital for mission success.

6.1 Scenario description

The scenario uses a Clearpath Jackal Unmanned Ground Vehicle (UGV) with a skid-steer drivetrain [49]. The Jackal is equipped with an Intel Core i3-4330TE CPU. For sensors, it is equipped with two 2D-LiDARS - a Hokuyo URG-04LX-UG01 (LiDAR 1) and a SICK TiM561-2050101 (LiDAR 2). The robots sensor configuration can be seen in figure **6.1**.

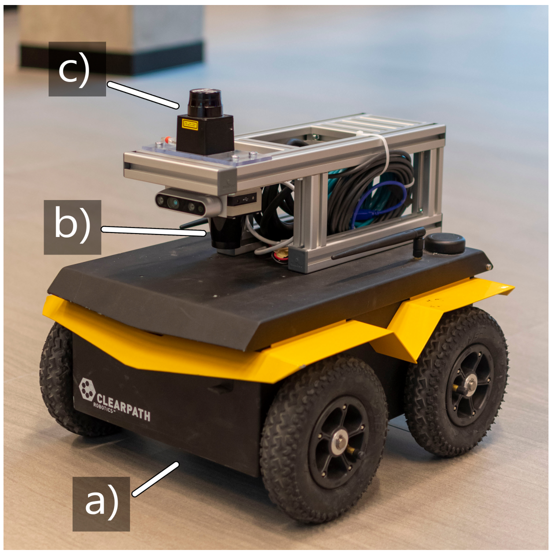


Figure 6.1: The platform used for the fault tolerant navigation demonstration. a) ClearPath Jackal UGV b) SICK TiM561-2050101 c) Hokuyo URG-04LX-UG01

As an operating system, Ubuntu 18.04 is used in conjunction with ROS Melodic Morenia. TeMoto with the developed RR is used to control and manage the robot.

For this demonstration TeMoto is set up with 3 resource managers (each resource manager is implemented as a ROS node):

- External Resource Manager (ERM) - Manages external program launching and shutdown.
- Component Manager (CM) - Manages and retains data about ROS based programs and chains of programs used in the system. Uses ERM to execute these components.
- Robot Manager (RM) - Retains information about resources that compose a robot and provides access to these resources. For example, the robot's driver and controller, which are ROS nodes, are resources that RM is managing. RM uses ERM to execute its resources.

A TeMoto Action is used to control the robot in conjunction with the managers. Firstly, the action loads the robot using RM. This fetches the configuration of Jackal that is used for loading of other components. After this a the LiDAR 1 is loaded from the CM that contacts the ERM allocating the LiDAR resource. After the robot is initialized a navigation goal is specified and the robot is instructed to reach that goal. The robot moves until it reaches its designated target.

The mission assigned to the robot consists of 3 distinct phases. At first, the robot is in an idle state waiting for an operation to start. When the navigation mission is started the robot allocates necessary resources for the goal of autonomously moving from its starting position to a predefined point. During the mission, multiple software and hardware faults are introduced to test its resource management capabilities. After reaching its goal the UGV returns to its idle state. Figure **6.2** shows the mission flow and error recovery flows can be seen in figure **6.3**.

During the mission, the robot is subjected to hardware and software faults that have to be resolved to continue with the mission. Hardware faults F_1 - F_2 that are introduced are:

- F_1 LiDAR 1 disconnect - The device was disconnected from the robot. LiDAR 1 had to be substituted for LiDAR 2.
- F_2 Motor driver stopped - The motor drivers encounter an error. The motor drivers had to be restarted to continue movements.

On the software side, TeMoto's managers are forcefully shut down using the terminal command "killall" to simulate a critical system error within the manager, after which the manager has to restart (done by enabling the *respawn* parameter in the TeMoto ROS launch file) and restore the Catalog's content. Software errors F_3 - F_5 introduced during the mission are as follows:

- F_3 ERM failure - The ERM is forcefully shut down and had to restore its previous state. The RR Catalog is used to restore the internal state of the ERM.
- F_4 CM failure - Similar to the ERM's failure the manager is shut down and has to restore its state with the RR.
- F_5 RM failure - As was with the ERM and CM the RM is shut down and has to be restored.

This scenario puts the redesigned RR's capabilities to the test, from resource loading and unloading to internal state recovery.

6.2 Results

The demonstration was carried out as shown in figure **6.4**, where it was placed into a corner of a corridor and had to manoeuvre to a designated spot behind a corner. During navigation, multiple faults were injected into the system. A complete timeline with the distance travelled by the robot can be seen in figure **6.5**.

Without any faults, the robot manages to complete the mission in 32 s. Yet without fault recovery, the robot is impaired within the first fault F_1, where the LiDAR 1 failed and the mission could not be completed. With TeMoto and RR-enabled fault recovery, the mission is completed in 43 s. At point, F_1 LiDAR 1 encountered a fault and the robot stopped for 3 seconds to substitute it for LiDAR 2. F_2 stopped the motor drivers and it took 5 seconds to restart them. F_3 - F_5 did not cause the robot to stop because resource managers and the robot's driver/controller nodes are different OS processes (ROS nodes). Internally the managers recovered their state from their RR instances to allow resource unloading, which is not possible without the recovery procedure. When the robot reached its target, it unloaded all resources and entered an idle state. Resource registering, recovery and unloading worked as predicted.

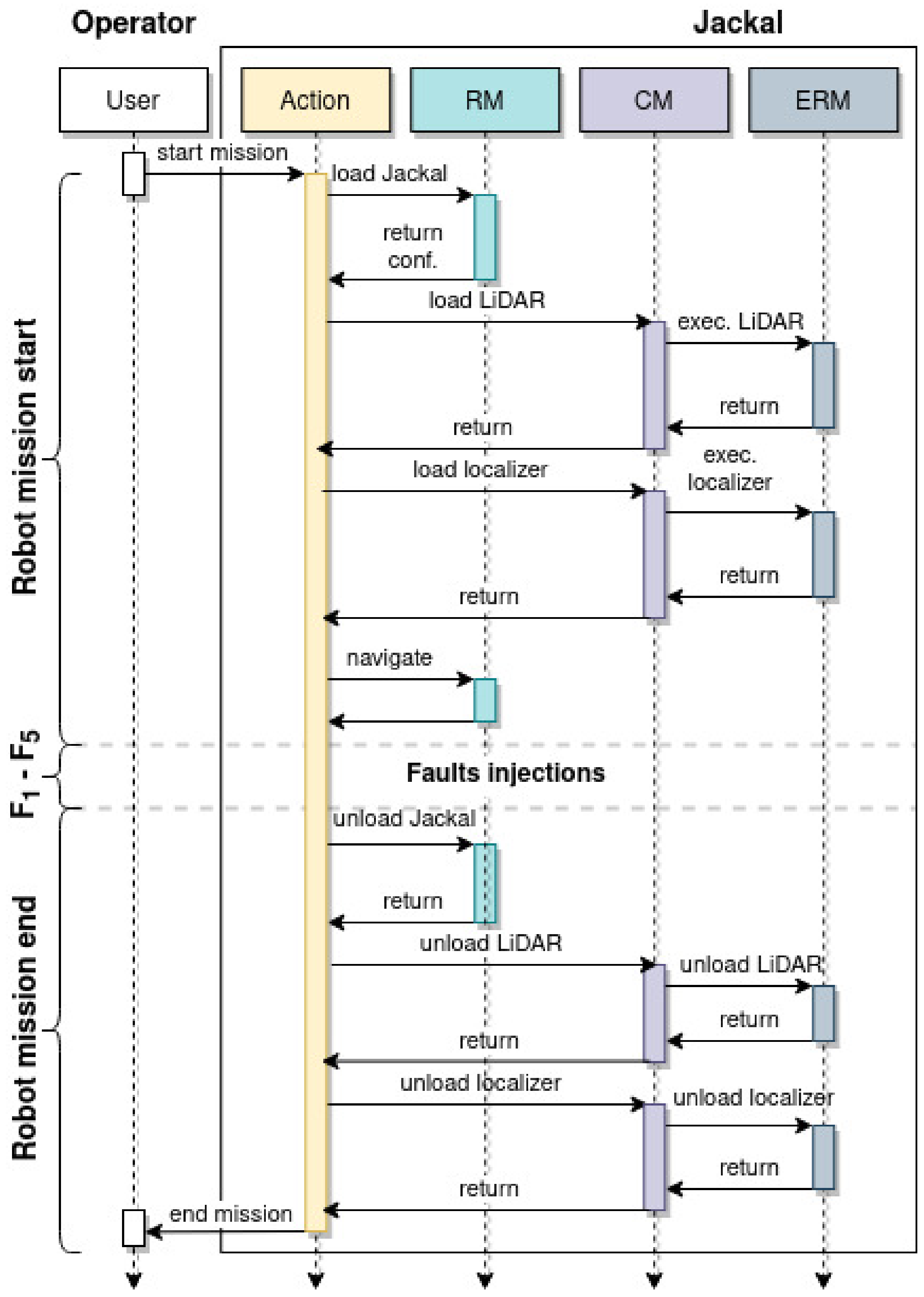


Figure 6.2: Flow diagram of a navigation mission. The robot and its resources are loaded during the start of the mission and unloaded once the mission has concluded. Faults F_1-F_5 are injected into the system during the mission.

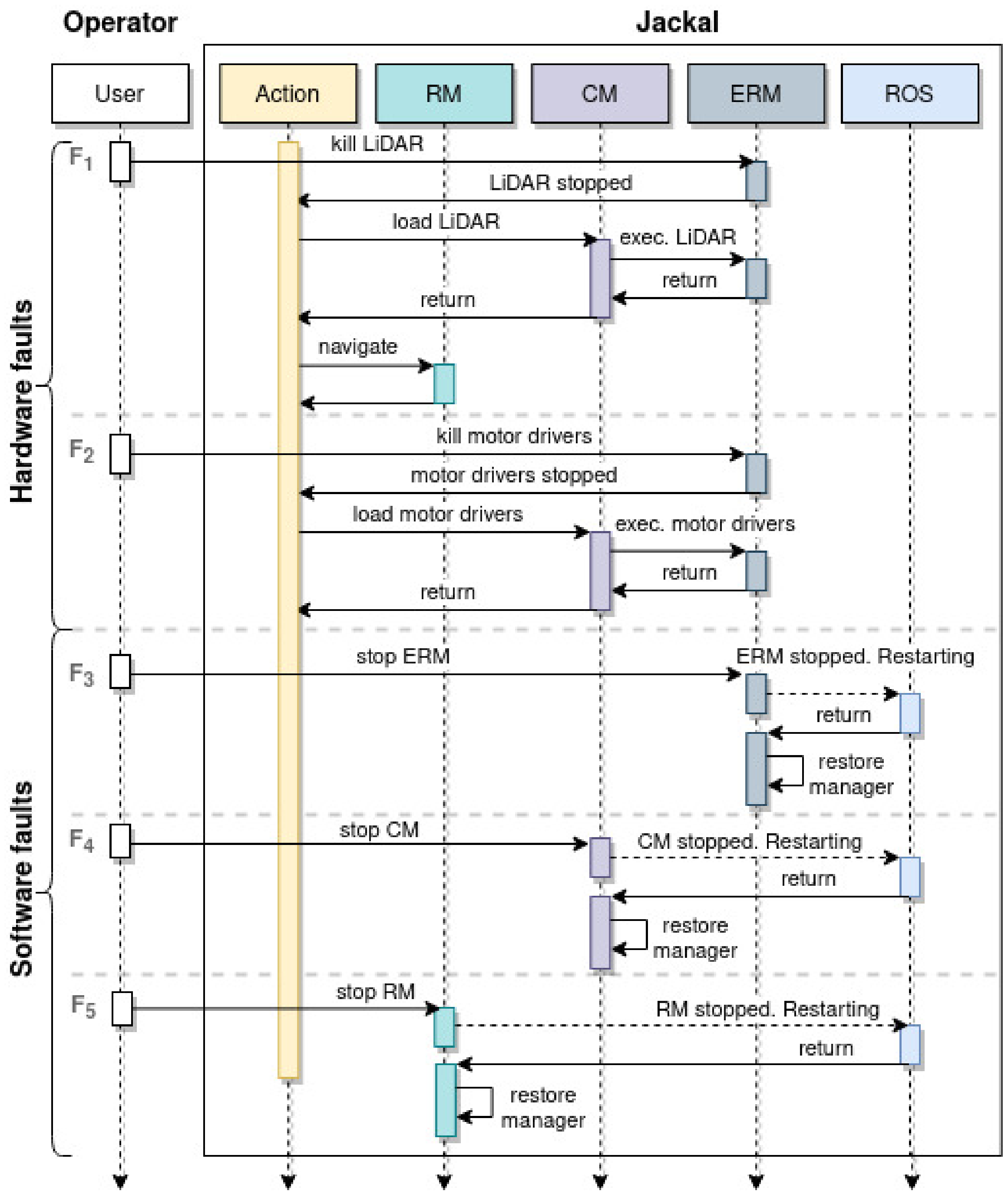


Figure 6.3: Flow diagram of two types of errors that occur during the demonstration mission. The Action is notified about a failures F_1 - F_2 and executes a fallback routine. For software errors F_3 - F_5 the ROS node is restarted and the manager recovers its state.

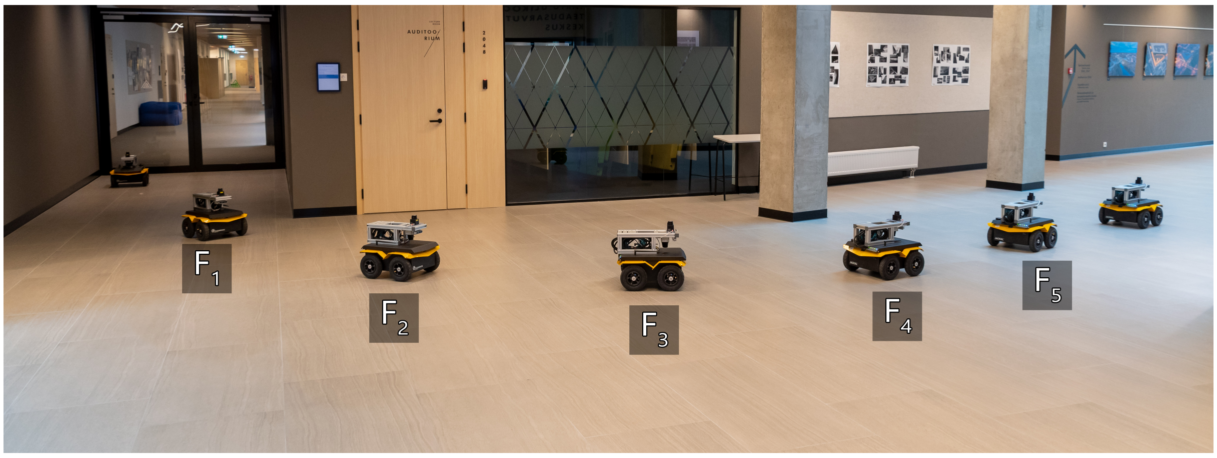


Figure 6.4: Path the Jackal took and when faults F_1 - F_5 were introduced.

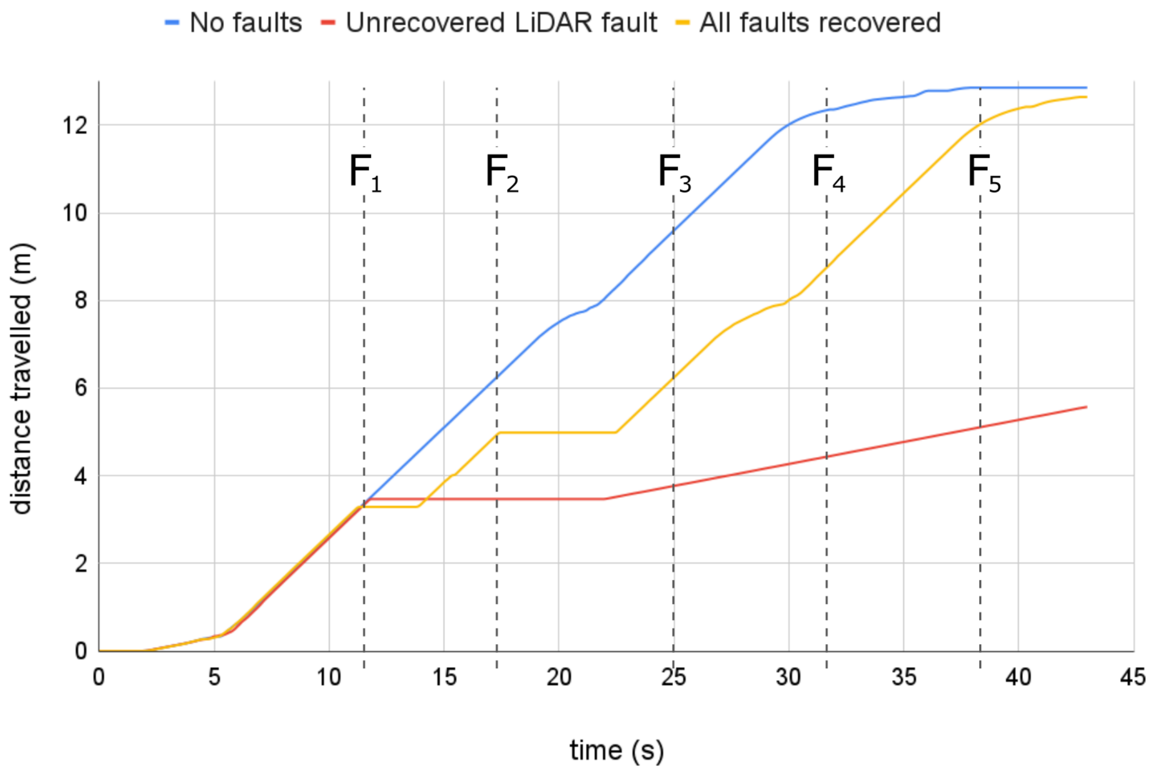


Figure 6.5: Distance travelled by the Jackal UGV during its mission. F_1 - F_5 show the moment a fault was injected into the robots system.

7 Conclusion and future work

As a result of this thesis, a robotic middleware agnostic Resource Registrar was developed. This project's source code and documentation is freely available to be modified and improved in GitHub [44]. This repository also contains example extensions of the Resource Registrar allowing it to be used on ROS 1 and ROS 2 to simplify its adoption. Improvements on previous work include making base functionalities framework agnostic, allowing for resource registrar recovery and the possibility of extending it to support multiple frameworks. The software's fault-tolerant resource management capabilities were demonstrated on the Clearpath Jackal UGV with an autonomous navigation mission. The resulting software meets the described requirements and is ready to be used in robotics systems.

Since the software is open-source and robotic middleware agnostic, it's available for adoption and improvements by developers and users worldwide. This software is in active development and multiple points of improvement have already been identified, e.g.

- Improve Catalog backup logic to avoid potential backup corruption.
- Improve thread safety of the resource request.
- Expand the RR functionalities to support additional frameworks.
- Expand automated test scope to cover more scenarios and frameworks besides the base implementation.
- General code optimization and refactoring to maintain system structure.

To maintain and improve the software package it needs to be actively employed in real applications. The RR's functionalities have been applied in TeMoto, allowing the development of features previously impossible. This increased flexibility in development allows TeMoto to continue improving and evolving.

8 Summary

The main goal of this thesis was to design and create a generalized Resource Registrar for usage in TeMoto, a framework designed to simplify the development of autonomous robots. Additionally, the software had to be platform-agnostic and extendable to provide resource management capabilities to a wide range of systems.

Firstly an overview of system architectures was given to understand the needs of systems a generalized Resource Registrar needs to fulfil. Robotic specific frameworks such as ROS and TeMoto were analyzed to identify bottlenecks in existing solutions.

Related work in other domains was analysed to identify potential solutions for dynamic resource management. Existing solutions in robotics were compared and the motivation for this thesis was presented.

The software implementation chapter defined the requirements for the proposed Resource Registrar. Additionally, an in-depth view of the architecture and implementation specifics was given. Next to the base implementation the systems integration with the ROS 1 and ROS 2 frameworks is explained.

The thesis concludes with a demonstration of a robot that utilized TeMoto and the developed Resource Registrar to complete a mission. It is concluded with the results and considerations for future work.

9 Kokkuvõte

Selle lõputöö peamine eesmärk oli ressursihalduri loomine TeMoto roboti raamistiku jaoks. Arendatud tarkvara on platvormi agnostiline ning laiendatav, et pakkuda ressursside haldamise võimekust mitmetele süsteemidele.

Esmalt antakse ülevaade olemasolevate tarkvarasüsteemide arhitektuuridest, et mõista, mis nõudeid nad ressursihalduri disainile esitavad. Robootikale spetsiifilisi raamistike, nagu ROS ning TeMoto, analüüsitakse, et tuvastada puudujääke olemasolevas süsteemis.

Teiste valdkondade seotud tööde ülevaates tutvustatakse olemasolevaid ressursihalduse lahendusi. Samuti tutvustatakse robootika-spetsiifilisi lahendusi ning võrreldakse neid käesolevas lõputöös valminud süsteemiga.

Tarkvara disaini ja arenduse peatükis defineeritakse süsteemile esitatavad nõuded. Lisaks kirjeldatakse detailselt loodud ressursihalduri arhitektuuri ja implementatsiooni. Lisaks baasimplementatsioonile tutvustatakse ROS 1 ning ROS 2 ressursihalduri laiendusi.

Viimane peatükk demonstreerib käesolevas töös valminud tarkvara autonoomse roboti navigeerimise näitel, kus robot suudab jätkata navigeerimist, hoolimata erinevatest esilekutsutud riist- ja tarkvaravigadest. Lõpetuseks vaadeldakse demonstratsiooni tulemusi ning kaalutletakse edasisi arendusi.

Bibliography

[1] "Common european research classification scheme." [Online]. Available: https://www.etis.ee/Portal/Classifiers/Details/d3717f7b-bec8-4cd9-8ea4-c89cd56ca46e?lang=ENG%5C# [Accessed: 17-Apr-2021].

[2] "Application programming interface," Mar 2021. [Online]. Available: https://www.hubspire.com/resources/general/application-programming-interface/ [Accessed: 09-May-2021].

[3] "Techterms - the computer dictionary." [Online]. Available: https://techterms.com/ [Accessed: 09-May-2021].

[4] N. O. US Department of Commerce and A. Administration, "What is lidar," Oct 2012. [Online]. Available: https://oceanservice.noaa.gov/facts/lidar.html [Accessed: 09-May-2021].

[5] "About ros." [Online]. Available: https://www.ros.org/about-ros/ [Accessed: 21-Apr-2021].

[6] R. S. P. Leach, M. Mealling, "A universally unique identifier (uuid) urn namespace," Jul 2005. [Online]. Available: https://tools.ietf.org/html/rfc4122 [Accessed: 21-Apr-2021].

[7] S. Odean, "Software architecture - the monolithic approach," Sep 2018. [Online]. Available: https://medium.com/@shivendraodean/software-architecture-the-monolithic-approach-b948ded8c333 [Accessed: 21-Apr-2021].

[8] Apr 2021. [Online]. Available: https://www.robocup.org/ [Accessed: 28-Apr-2021].

[9] G. K. Kraetzschmar, N. Hochgeschwender, W. Nowak, F. Hegger, S. Schneider, R. Dwiputra, J. Berghofer, and R. Bischoff, "Robocup@work: Competing for the factory of the

future," in *RoboCup 2014: Robot World Cup XVIII*, R. A. C. Bianchi, H. L. Akin, S. Ramamoorthy, and K. Sugiura, Eds. Cham: Springer International Publishing, 2015, pp. 171–182.

[10] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith.* O'Reilly Media, Incorporated, 2019. [Online]. Available: https://books.google.ee/books?id=iul3wQEACAAJ

[11] A. Kainz, "Microservices vs. monoliths: An operational comparison," Jul 2020. [Online]. Available: https://thenewstack.io/microservices-vs-monoliths-an-operational-comparison/ [Accessed: 21-Apr-2021].

[12] V. Shvetsov, "Monolithic vs. microservices architecture: Which is right for your app," Jan 2021. [Online]. Available: https://mlsdev.com/blog/128-microservices-vs-monoliths-how-to-understand-when-it-s-time-to-use-the-former-option [Accessed: 22-Apr-2021].

[13] S. Watkins, "Technical debt debate, with ward cunningham & capers jones," Jan 2013. [Online]. Available: https://www.castsoftware.com/blog/ward-cunningham-capers-jones-a-discussion-on-technical-debt [Accessed: 16-May-2021].

[14] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does migrating a monolithic system to microservices decrease the technical debt?" *Journal of Systems and Software*, vol. 169, p. 110710, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301539

[15] M. D. Marco, "Technical debt and the monolith myth," Aug 2020. [Online]. Available: https://medium.com/@massimodm/technical-debt-and-the-monolith-myth-aaae043fa909 [Accessed: 22-Apr-2021].

[16] O. Vogel, I. Arnold, A. Chughtai, and T. Kehrer, *Software Architecture: A Comprehensive Framework and Guide for Practitioners*, ser. SpringerLink : Bücher. Springer Berlin Heidelberg, 2011. [Online]. Available: https://books.google.ee/books?id=M3RzjWmMgBwC

[17] R. Lin, "Monolithic vs modular," Nov 2016. [Online]. Available: https://medium.com/@berto168/monolithic-vs-modular-9b6d69684a2c [Accessed: 21-Apr-2021].

[18] M. Quigley, B. Gerkey, and W. Smart, *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*. O'Reilly Media, 2015. [Online]. Available: https://books.google.ee/books?id=Hnz5CgAAQBAJ

[19] C. Lilienthal, *Sustainable Software Architecture: Analyze and Reduce Technical Debt*. dpunkt.verlag, 2019. [Online]. Available: https://books.google.ee/books?id=Eq_2DwAAQBAJ

[20] K. M. Sugathadasa, "Distributed system architectures and architectural styles," Sep 2017. [Online]. Available: https://keetmalin.wixsite.com/keetmalin/post/2017/09/27/distributed-system-architectures-and-architectural-styles

[21] J. Hurwitz, R. Bloor, C. Baroudi, and M. Kaufman, *Service Oriented Architecture For Dummies*, ser. –For dummies. Wiley, 2006. [Online]. Available: https://books.google.ee/books?id=BgNhTkdwVogC

[22] H. Taylor, A. Yochem, L. Phillips, and F. Martinez, *Event-Driven Architecture: How SOA Enables the Real-Time Enterprise*. Pearson Education, 2009. [Online]. Available: https://books.google.ee/books?id=g1318W0CIm4C

[23] C. Richardson, "Microservices pattern: Microservice architecture pattern." [Online]. Available: https://microservices.io/patterns/microservices.html [Accessed: 22-Apr-2021].

[24] S. B.-D. Delamore, "The 5 essential elements of modular software design," Jan 2021. [Online]. Available: https://shanebdavis.medium.com/the-5-essential-elements-of-modular-software-design-6b333918e543 [Accessed: 24-Apr-2021].

[25] A. Figueroa, "Monolithic vs micro-services, and all in between," Aug 2019. [Online]. Available: https://medium.com/swlh/monolithic-vs-micro-services-and-all-in-between-7d496408ad02 [Accessed: 24-Apr-2021].

[26] V. Khononov, "Untangling microservices, or balancing complexity in distributed systems," Apr 2020. [Online]. Available: https://vladikk.com/2020/04/09/untangling-microservices/ [Accessed: 22-Apr-2021].

[27] G. Myers, *Composite/structured Design*, ser. Van Nostrand Reinhold data processing series. Van Nostrand Reinhold, 1978. [Online]. Available: https://books.google.ee/books?id=OtUmAAAAMAAJ

[28] T. Hoff, "Microservices - not a free lunch!" [Online]. Available: http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html [Accessed: 22-Apr-2021].

[29] "Integration with other libraries." [Online]. Available: https://www.ros.org/integration/ [Accessed: 24-Apr-2021].

[30] "Is ros for me?" [Online]. Available: https://www.ros.org/is-ros-for-me/ [Accessed: 24-Apr-2021].

[31] "Understanding ros 2 nodes," Oct 2019. [Online]. Available: https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Nodes.html [Accessed: 28-Apr-2021].

[32] R. Valner, V. Vunder, A. Zelenak, M. Pryor, A. Aabloo, and K. Kruusamäe, "Intuitive 'human-on-the-loop' interface for tele-operating remote mobile manipulator robots." Madrid, Spain: European Space Agency, Jun. 2018, p. 8.

[33] "Github - temoto-telerobotics/temoto." [Online]. Available: https://github.com/temoto-telerobotics/temoto [Accessed: 16-May-2021].

[34] R. Valner, V. Vunder, M. Pryor, A. Aabloo, and K. Kruusamäe, "Temoto: A software framework for dependable long-term robotic autonomy with dynamic resource management," *Autonomous Robots*, submitted.

[35] E. Woods, "Software architecture in a changing world," *IEEE Software*, vol. 33, pp. 94–97, 11 2016.

[36] "Docker is an open platform to build, ship and run distributed applications anywhere." [Online]. Available: https://www.docker.com/ [Accessed: 28-Apr-2021].

[37] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *ACM Queue*, vol. 14, pp. 70–93, 2016. [Online]. Available: http://queue.acm.org/detail.cfm?id=2898444

[38] D. McDonald, *Electric Capitalism: Recolonising Africa on the Power Grid.* Taylor & Francis, 2012. [Online]. Available: https://books.google.ee/books?id=-Lrohmv15A8C

[39] U. S. C. S. S. C. on the Year 2000 Technology Problem, *Utilities and the National Power Grid: Hearing Before the Special Committee on the Year 2000 Technology Problem, United States Senate, One Hundred Fifth Congress, Second Session, on the Readiness of the Utility Industry, Including Electric and Gas Utilities, to Deal with the Year 2000 Technology Problem, June 12, 1998*, ser. S. hrg. U.S. Government Printing Office, 1998, no. v. 4; v. 22. [Online]. Available: https://books.google.ee/books?id=duHpgPhtIP4C

[40] "Commission regulation (eu) 2016/631 of 14 april 2016 establishing a network code on requirements for grid connection of generators," Apr 2016. [Online]. Available: https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=OJ:JOL_2016_112_R_0001 [Accessed: 12-May-2021].

[41] S. Wang, X. Liu, J. Zhao, and H. I. Christensen, "Rorg: Service robot software management with linux containers," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 584–590.

[42] 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)*, 2016, pp. 55–60.

[43] "Ros c style guide." [Online]. Available: http://wiki.ros.org/CppStyleGuide [Accessed: 25-Apr-2021].

[44] "Github - temoto-telerobotics/temoto_resource_registrar." [Online]. Available: https://github.com/temoto-telerobotics/temoto_resource_registrar [Accessed: 16-May-2021].

[45] S. C. Dewhurst, "Oop50-cpp. do not invoke virtual functions from constructors or destructors," Apr 2021. [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/cplusplus/OOP50-CPP.Donotinvokevirtualfunctionsfromconstructorsordestructors [Accessed: 21-Apr-2021].

[46] M. Cline, "When should my destructor be virtual?, c faq," Jul 2012. [Online]. Available: https://www.webwise-scripts.com/cfaq/virtual-dtors.html [Accessed: 21-Apr-2021].

[47] "boost - c++ libraries." [Online]. Available: https://www.boost.org/ [Accessed: 12-May-2021].

[48] Google, “Github - google/googletest.” [Online]. Available: https://github.com/google/googletest [Accessed: 12-May-2021].

[49] C. Robotics, “Jackal ugv - small weatherproof robot - clearpath.” [Online]. Available: https://clearpathrobotics.com/jackal-small-unmanned-ground-vehicle/ [Accessed: 18-May-2021].

Appendices

Appendix A - Compilation of the RrCore library

The rr_core library is a cmake project. It installs a dynamic library that can be used in external projects. By default, test compilation is disabled since it breaks in additional dependencies and slows down first time compilation. The only third party library dependency is Boost. Boost provides base functionalities for serialization and UUID generation. The make file checks for its existence in the system.

To enable test compilation the property of -Dtest=ON needs to be defined when building. This command compiles libraries like gtest and glog used in tests and builds the test executable. This file can be found in the build folder with the name rr_core_test.

Examples of commands to build the library and included tests:

- mkdir build; cd build; cmake ..; sudo make install
- mkdir build; cd build; cmake .. -Dtest=ON; sudo make install; ./rr_core_test