robotics

MDPI

Article

Robotic Systems of Systems Based on a Decentralized Service-Oriented Architecture

Lennart Siefke [1,*], Volker Sommer [1], Björn Wudka [2] and Carsten Thomas [2]

[1] Department of Computer Science and Media, Beuth University of Applied Sciences Berlin, Luxemburger Str. 10, D-13353 Berlin, Germany; sommer@beuth-hochschule.de

[2] Department of Energy and Information, Hochschule für Technik und Wirtschaft Berlin—University of Applied Sciences, Wilhelminenhofstraße 75A, D-12459 Berlin, Germany; bjoern.wudka@htw-berlin.de (B.W.); carsten.thomas@htw-berlin.de (C.T.)

* Correspondence: lennart.siefke@beuth-hochschule.de

Received: 10 July 2020; Accepted: 23 September 2020; Published: 27 September 2020

Abstract: Multi-robot systems are often static and pre-configured during the design time of their software. Emerging cooperation between unknown robots is still rare and limited. Such cooperation might be basic like sharing sensor data or complex like conjoined motion planning and acting. Robots should be able to detect other robots and their abilities during runtime. When cooperation seems to be possible and beneficial, it should be initiated autonomously. A centralized cloud control shall be avoided. Using software patterns belonging to service-oriented architectures, the robots are able to discover other robots and their abilities during runtime. These abilities are implemented as services and described by their interfaces. Composition of services can be done easily and flexibly. The software patterns originally belonging to cloud computing could be successfully adopted to decentralized multi-robot systems. The developed concept allows autonomous systems to cooperate flexibly and to compose multi-robot systems during runtime.

Keywords: autonomous robotics; multi-robot systems; systems of systems; service-oriented architectures; multi-agent systems

1. Introduction

Mobile multi-robot systems will be more and more popular in the future. During design time of the systems, the emerging multi-robot systems will not always be predictable. This can be the case, when the systems are from different manufacturers and have no knowledge about each other. An example are self-driving cars, which decide to build up a platoon [1]. A platoon enables better usage of available space, less wind resistance or time-efficient simultaneous start up at green traffic lights. Such a system consisting of autonomous systems building up spontaneously is called a System of Systems (SoS) [2,3]. The member systems have to be designed to be able to interact with unknown systems. This leads to higher demands on each systems' software stack. SoS-functionality must be safe, easy to initiate and controllable by the members of the SoS. In this paper, we introduce the application of modern concepts and patterns belonging to the category of service-oriented architectures in order to design systems of systems.

2. Related Work

Applications including multiple autonomous robots traditionally fall in the category of multi-agent systems (MAS), respectively multi-agent robotic systems (MARS). Multi-agent systems themselves belong to the category of Distributed Artificial Intelligence [4]. In a MAS, agents are defined as autonomous entities collaborating together. The agent itself is defined by Russel and

Norvig as "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators" [5]. Similar to Multi-Agent Systems is the field of research "Systems of Systems" (SoS). A SoS is a structure which consists of autonomous systems, can build up spontaneously and has an extended functionality compared to the single systems [2,3].

According to Bruneo et al., a middleware in a distributed system "is a software layer that hides all the implementation details of the communication infrastructure" [6]. By using a middleware, the developer can focus on the application and relay on an established networking stack. The meaning of the term middleware depends on the context, it differs in distributed systems and robotics. To avoid confusion in this paper, we call a middleware in the context of distributed systems "communication layer". Development of robotic systems can be supported by using a Robotic Framework (RF), which delivers a basic software architecture including reusable modules for common tasks like SLAM or navigation. The most used RF is Robot Operating System (ROS) [7]. The essential concepts of ROS are its inter-process communication capabilities, which include publish/subscribe and remote procedure calls, and an own ecosystem of software packages. However, ROS is focused on single robot systems [8]. Its communication layer lacks support of unreliable connections and is centralized. The multimaster_fkie extension allows to connect different ROS systems and setup multi-robot functionality [9]. After all, this extension still does not support important features like Quality of Service. ROS2 as a recent successor has a focus on multi-robot systems [8]. Its communication layer supports more stable communication with external systems, including Quality of Service settings. Lacking is multi-robot support in the application layer for high level functionality like platooning or lane merging in road traffic. In the application layer, the ROS2 discovery mechanism is lacking information about the system-membership of a node, description of functionality and seperation of low and high level functionality amongst others. Nevertheless, ROS2 is already used as a communication layer in distributed systems. In [10] the authors use ROS2 as a communication layer to send reliable messages through large distributed automated systems. The authors of [11] examine the suitability of ROS2 in distributed real-time applications. Other examples for available communication layers are implementations of the standards Data Distribution Service [12] and OPC-UA [13]; gRPC [14], Apache Kafka [15] or Coaty [16].

JADE is a framework that is designed for development of multi-agent systems [17]. By supporting communication between agents, JADE fulfills the role of a communication layer. Aside from that, JADE also addresses the application layer by providing features like an agent communication language, an discovery system for agents or behavior models. Nevertheless, the communication layer of JADE is quite outdated and missing important features like Quality of Service. The mechanism to discover other agents is a distributed service [18]. In a MAS setup, each system contains a discovery unit, which implements the described functionality. In [19], the authors propose a software architecture which integrates ROS and JADE for autonomous transport vehicles in cyber-physical factories. ROS is used for classic robotic functionality like localization and navigation, whereas JADE is used as a communication layer for multi-robot functionality like tasks allocation or discovery of other robots. Their multi-agent robotic system architecture consists of four layers. The upper layer, called "Social Layer", contains the distributed application whereas the bottom layers contain classic robot functionality and the software adapters to the social layer. In contrast, the authors of [20] use ROS2 as a communication layer for multi-agent systems. They focus on the application layer by developing the Belief-Desire-Intention (BDI) behavior model on top of ROS2. The Belief-component stores the knowledge which can be obtained through sensors or communication with other agents. The Desires-component stores the agent's goals. Goals can be conflicting, so a decision has to be made which leads to a plan. The plan is stored and executed by the Intention-component of each agent.

From the scope of cloud computing and software engineering, new approaches for developing distributed applications came up. Service-oriented architectures (SOA) are a way to modularize distributed applications [21]. Services are self-contained software functionality, which are accessed through clearly defined interfaces. By composing services, complex software functionality can be

accomplished. Composition of services can be done by the orchestration- or choreography pattern [22]. With orchestration, the services are composed by a central coordinator. With choreography, the services compose themselves without a central coordinator. Service-oriented architectures are available in different variants. For example, older SOA often rely on an "Enterprise Service Bus", which is a central component connected to all services and mediating data between them [23]. In contrast, modern SOA like microservices are stricter on modularization and services are more independent of other components [24]. To discover services and the location of their network addresses during runtime, the service discovery pattern is applied [25]. The combination of cloud computing and robotics lead to the field of research called cloud robotics [26]. The concept of a robot connected to the cloud by providing its functionality as services is called Robot as a Service (RaaS) [27]. In RaaS, the robots services are consumed by connected clients, for example a central control entity. The combination of cloud computing and robotics leads to a few problems with terminology. In robotics, a service is often associated with a ROS-Service. A ROS-Service is not a service in the domain of software architecture, but instead a remote procedure call (RPC). A service in the domain of software architecture is a module in a distributed system, which internal details are hidden behind an interface. Applying the term service into the domain of ROS, a service can correspondent to a ROS-Node or more likely multiple ROS-Nodes, as services in SOA often implement higher level functionality. Lastly, a ROS-Service as an RPC can be part of the interface of a service.

The development of automated multi-robot systems is suitable with approaches of both categories, MAS and SOA [28]. When comparing these different approaches, their particular roots are noticeable. The MAS approaches as a sub-category of distributed artificial intelligence are focused on the agents behavior, the persue of their goals and their acting in unknown environments. Agents are self-contained and not depending on centralized control mechanisms. Service-oriented architectures have their roots in software engineering and cloud computing. Interfaces of services are clearly defined and implementation details are encapsulated. Services are self-contained software functionality in their specific domain. Approaches of Cloud Robotics often include a central control of the distributed system. MAS and SOA approaches are not mutually exclusive when building multi-robot systems. Multi-robot systems, consisting of autonomous robots and built up with a decentralized and flexible SOA, are nevertheless missing in the literature.

At this point, it is interesting to take a look at the automotive domain, as usage of service-oriented architectures for distributed systems is very common within and additionally, road traffic is an environment with a lot of building up and dissolving systems of systems. In the automotive domain, Adaptive Platform of the AUTOSAR partnership is a standard for intelligent vehicles launched in 2017. Adaptive Platform specifies a service-oriented architecture for connecting components inside the network of a vehicle [29]. In Adaptive Platform, the car is treated as a distributed system consisting of several subsystems. However, a service-oriented architecture for cooperation with multiple vehicles enabling high level cooperative functionality is currently not within the standard. As advanced driver assistance systems become more and more sophisticated and cooperation is highly beneficial in road traffic, it is expected that cooperation of (semi-)autonomous cars also gets standardized in the future.

3. Service-Oriented Architecture for Systems of Systems

We propose an adoption of the service-oriented architectures for developing peer-to-peer robotic systems of systems (SoS). Service-oriented architectures are typically used for web-applications in cloud environments. However, there are essential differences between robotic systems and cloud systems. At first, robots are location-dependent physical entities. As cooperation often depends on the location of the robots, the knowledge of the other robots position is important. Further, robots often share space with humans, thus their use is way more safety-critical. Another difference is, that it is more likely to have few knowledge about cooperating systems during design time in multi-robot systems. The other robots might be developed by another company or do not even exist at design

time. Furthermore, in contrast to cloud applications executed in data centers, the wireless network connection between moving mobile robots is less reliable.

In this new approach, the local and social functionality is implemented in services. Local functionality corresponds to single-robot applications whereas social functionality corresponds to multi-robot applications. A service is a software component in a distributed application. The entire application consists of connected services. A service is typically connected with other services through interfaces. Services can provide interfaces and consume other service interfaces in a flexible way. Systems are able to cooperate, when they own providing and consuming services of the same interfaces. In robotic systems of systems, services typically provide sensing, acting or computing functionality. Beside the services, we introduce a component in each system to make the systems aware of each other, and another component to compose the SoS by connecting the services decentralized across the systems. The first component is called SystemDiscovery, the second component is called Reconfiguration. The robotic systems are autonomous, safety-relevant and must offer high availability and reliability. The distributed applications may demand real-time requirements. Because of this, we decided against a centralized cloud approach. The systems communicate peer-to-peer with each other. This avoids introducing an additional point of failure between the systems.

To detect other systems and their services during runtime, an adoption of the discovery-pattern is applied. This leads the systems to build up SoS with other systems, which are not known during design time. The discovery-pattern is applied in common service-oriented architectures, because of the location transparency of services in the network. This is also the case in robotics, but in addition, services belong to a specific system. This leads to the discovery of systems including their services, instead of discovery of services only. We call the component implementing the discovery-pattern SystemDiscovery. SystemDiscovery is a distributed application, which exchanges information of systems in the same network. There is one instance of SystemDiscovery per system. The services perform self-registration with their descriptions at the SystemDiscovery instance on the same system. In comparison to 3rd-party registration, the services have more control on their state. SystemDiscovery exchanges the collected information with external systems. As a result, it refreshes an up-to-date catalog of systems and their internal services, called SystemCatalog. The concept relies on client-side discovery, as it leads to less network hops than server-side discovery. In addition to the service descriptions, the services notify their current state. Like in common SOA, the discovery pattern is combined with the service health pattern. The services regularly report their health status as part of their state to SystemDiscovery. Example health status codes might be healthy, unhealthy or not_provided. Depending on the use case, more expressive health codes can be defined. The state also includes the IDs of the connected services, which allows to observe the current composition of the SoS. If there are a lot of systems in the same network, a filtering mechanism for the SystemCatalog might be necessary.

The reconfiguration component is responsible for connecting services across the systems. It is also a distributed application with one instance per system. When there are changes in SystemCatalog, the optimal configuration of the system of systems will be analyzed. The configuration corresponds to the composition of services across the system. If the desired configuration does not match the active configuration, it will be communicated with the other systems. The systems aim to reach a common agreement of their configuration. If an agreement is reached, the services will be activated according to the chosen configuration. This process is called reconfiguration, details will be published elsewhere [30]. Composition of services in a service-oriented architecture can be done by orchestration- or choreography-pattern. When taking a view on the systems, they do choreography. There is no controlling instance, the systems are autonomous and decide on their own what to do. When taking a look at the services of one system, their composition is done by reconfiguration component. This matches the orchestration-pattern. The benefit of orchestrating the services is the greater amount of control, especially in a non-ideal situation. For example, if there is a failure in a robot's sensor, sensor sharing with the most suitable available robot can be triggered. Another non-ideal situation

could be an unexpected high CPU-load, which leads to disabling less important social functionality or outsourcing tasks with high computing load to another system.

The services of each system are started automatically and register themselves at SystemDiscovery, but are not in an active state after startup. If registering was successful, a service waits for the activating command. Otherwise, the service waits a little and tries registering again. The registration involves providing the description of the service. This description contains at least an ID and lists of provided and consumable interfaces. Interacting services do not need to know each other but their interfaces. By just knowing the interface, the services are more decoupled of the practical implementation. A service providing an interface must present the network address of the interface. The list of provided and consumable interfaces must contain information about the multiplicity of the connections. For example, a service which provides an interface to direct control of an actor can probably be connected to exactly one consuming service. Instead, a service providing information can probably be connected to n consuming services. Interacting services must provide and consume the same interfaces. Beside that, a service description might contain more metadata like information about safety or quality of service. There is a lot of freedom in the internal implementation of a service. Different functionality demands different patterns and beside this, the development-team of a service knows best about suitability of technology and pattern of their problem. It is important to document the interface of a service properly. Interfaces should be used with asynchronous communication in most cases. Synchronous communication instead would introduce the disadvantages of less availability during the calls and tighter coupling of services.

The focus of this approach is support of robotic systems of systems with high level cooperative functionality. For example, autonomous cars have to perform lane merging and start cooperative path planning and navigation for more efficient driving. In addition, the cars share their perception of the environment to increase safety during lane merging. Of course, systems can also be of other types. As robotic systems are often commanded by humans, the system of systems may include devices for human-machine interaction. This can be laptops, smartphones or voice user interfaces amongst others. Such devices would also discover available systems with SystemDiscovery and can consume available services. The heterogeneity of the systems would be noticeable in the type of services a system can consume and/or provide. The approach is agnostic to a specific implementation of a communication layer, it can be put on top of different communication layers. Suitable communication layers might be ROS2 or gRPC, for example. The choice of the communication layer depends on the specific use cases. For example, some use cases demand real-time requirements on the communication whereas other use cases do not demand them.

4. Prototypic Implementation with ROS2

An overview of the assembled components and their interaction is given in Figure 1. The transition of a single autonomous robot to a system of systems can be summarized as follows. When the cooperating systems are unknown during development, the first step is to discover other systems. SystemDiscovery is used to exchange information about the systems during runtime. When they know each other's social functionality, decisions have to be made whether a cooperation is desired or not. The basis for decision-making is the SystemCatalog, as the systems need suitable service providers and consumers to cooperate. The decision is made decentralized by each system in their reconfiguration components. If a cooperation is desired, the decision triggers communication between the reconfiguration components by submitting the proposal of connected services to the respective systems. If the systems reach an agreement, each system activates its services as proposed. The functionality of the new multi-robot system is executed by the activated services. During operation, the reconfiguration component stays active to react to events in the systems or environment. To react to changes during cooperation, the systems are able to change the composition of services continuously.

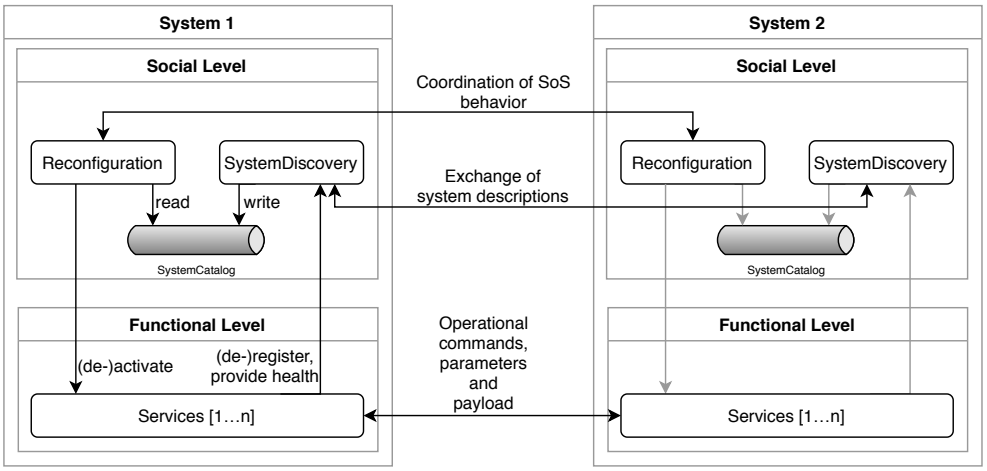

Figure 1. A system of systems consisting of two systems presented with their internal architectures. SystemDiscovery makes the systems aware of each other. Reconfiguration composes the services across the systems in a decentralized manner. The services implement functionality of the individual systems and the functionality of the emerging system of systems. SystemCatalog is designed as an event channel to achieve greater decoupling of the components and avoid polling of Reconfiguration.

In this section, we present a prototype to demonstrate the usage of the introduced concept. A stripped-down use case is implemented, which is focused on position and sensor sharing across multiple robots. There are several robots which are able to receive 3D sensor data and fuse them into their mapping components. Some of the robots have their own 3D sensors, they are additionally able to share pointclouds. If the robots meet and at least one of them has an 3D sensor, they start exchanging pointclouds. As stated in the previous section, the concept is not bound to a specific communication layer. ROS2 was chosen as a communication layer and robotic framework. The tight integration of communication and robotic functionality allows fast prototyping. Every component presented in Section 3 is implemented as a ROS2 node. This prototype is executed in a simulated environment with Gazebo.

The functionality of sharing pointclouds between the robots is implemented with two services. One service provides the pointclouds and the other one receives them and performs sensor fusion. The interface for exchanging pointclouds is called PointcloudShare. It is provided by the service PointcloudProvider and consumed by the service PointcloudConsumer. The services do not need to know each other. By just knowing the interface, the information of PointcloudProvider could be used by other suitable services too. The command-query-event pattern helps to design interfaces. A command is initiated by the consumer and can trigger a change in the state of the service provider. A query retrieves data without changing the state of the provider and is initiated by the consumer of the information. An event is triggered by the provider of the information. The interface PointcloudShare consists of one command, one query and one event. The command requestPointcloudStream(height, width, frequency) requests continuously updated pointclouds. If the provider accepts the request, the pointclouds are transmitted with the pointcloudUpdate-event which is triggered by the provider. Last part of the interface is the getActiveConfigurations-query which returns the configuration of the active pointcloud streams. As the application of sharing pointclouds is rather simple and is mainly used for demonstration purposes, we do not go much further into detail at this point. When having the unknown number of systems in mind, the multiplicity of consumers and providers of the interface becomes interesting. The service PositionConsumer can consume n providers of the interface. The service PointcloudProvider can offer the interface to n consumers. There is one state per consumer storing the respecting configuration. Each configurated pointcloud stream gets its own event channel. The provider is able to deliver the same event channel to different consumers, if they requested the same configuration. There could be performance issues from the provider and the network, when different consumers requesting different configurations. Each differently configured stream generates

more load. Because of this, the provider is more likely to deny requests the more consumers it is serving. From a consumers view, it can be more attractive to request an already active configuration to not get the request denied.

The services implementing the exchange of positions are designed very similar to the services implementing sharing of pointclouds. The interface is called PositionShare. Despite sensor data is relative to the position of the robot and its sensor, there should be no dependency between PointcloudShare and PositionShare. The service providing the PointcloudShare-interface sends the pointcloud itself, but in addition also the position of the robot and its sensor. This leads to a synchronization in time of the pointcloud and the geometric transformation, which delivers best performance. Besides this, the avoidance of runtime-dependency of PointcloudShare to PositionShare leads to higher availability and reliability. Whereas dependency between services might help to achieve complex applications, in this simple functionality it is preferable to avoid such dependency.

In this prototype, a service description contains the entries id, provided_interfaces and consumable_interfaces. A provided interface contains three entries: the name of the interface, its network address and the maximum number of connected consumers. A consumable interfaces contains the name of the interface and the maximum number of connected providers. Listing 1 shows the descriptions of the services implementing sharing of pointclouds, PointcloudProvider and PointcloudConsumer. With these information in the service descriptions, the services can be composed flexible during runtime.

Listing 1. Service descriptions and states of PointcloudProvider and PointcloudConsumer. Presented in YAML for better readability.

```yaml
---
description:
  id: PointcloudProvider
  provided_interfaces:
    - name: PointcloudShare
      network_address: /robot1/
         PointcloudProvider/
         PointcloudShare
      max_connected_consumers: 'n'
  consumable_interfaces: []
state:
  health: healthy
  connected_consumers:
    - system_id: robot2
      service_id: PointcloudConsumer
  connected_providers: []
```

```yaml
---
description:
  id: PointcloudConsumer
  provided_interfaces: []
  consumable_interfaces:
    - name: PointcloudShare
      max_connected_providers: 'n'
state:
  health: healthy
  connected_consumers: []
  connected_providers:
    - system_id: robot1
      service_id: PointcloudProvider
```

At this time, there is just a simple reconfiguration algorithm implemented in the reconfiguration component. It binds every service consumers and providers of the interface PositionShare across the robots. The functionality of sharing positions does not generate much load and the respective services will always be composed. In contrast, the functionality of sharing pointclouds generates extensive load on the provider, the consumer and the network. Because of this, the robots are configured to use this functionality only if the robots are nearby to each other. The agreement during reconfiguration is easy to achieve, as the optimal configuration of the system of systems can be determined by any system with the same result. Nevertheless, we built up a skeletal structure for future research on safety-relevant reconfiguration algorithms. A built up system of systems consisting of two robots is shown in Figure 2. The emerging application of sharing pointclouds is visualized in Figure 3. Two robots got nearby at a crossover in an indoor place and composed their services to get a better view.

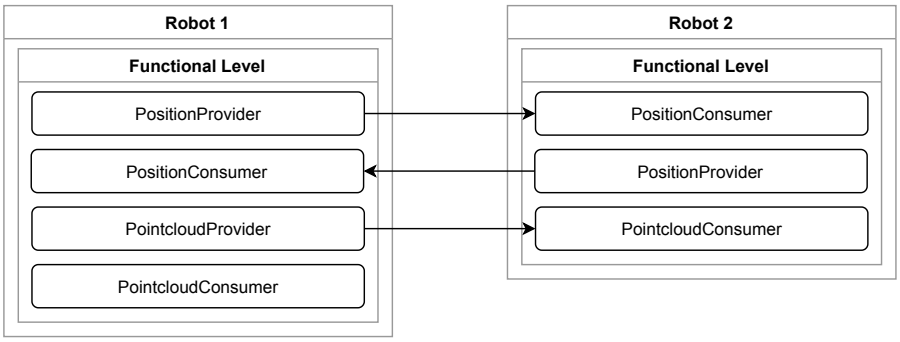

Figure 2. Composition of services across two robots after reconfiguration. The arrows present the data flow of the payload of the services. The service PointcloudConsumer of robot 1 is inactive, as the second robot is not able to offer pointclouds. The presented functional levels fit into the architecture presented in Figure 1.

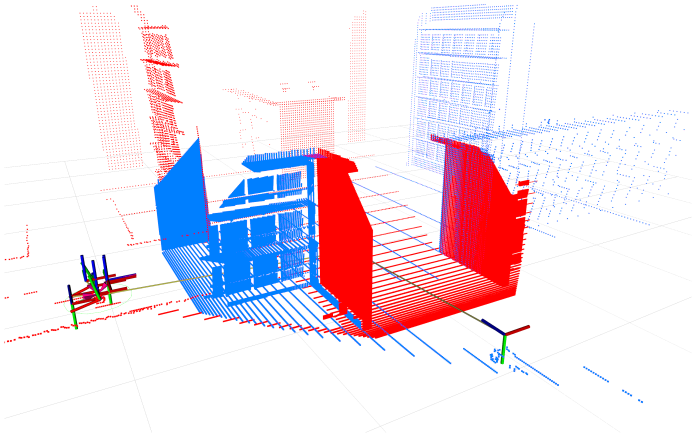

Figure 3. Two robots built up a system of systems to exchange pointclouds and fuse these with their own data. This figure shows the perception of the robot in the left. Its own sensor data is visualized in blue and the received sensor data is visualized in red. The colored threepart structures represent the positions of the robots and their components. By using PointcloudShare, the robot is also aware of the position of the 3D sensor serving the pointclouds.

5. Conclusions and Future Work

In this paper, we introduced a new software design approach for implementing flexible systems of systems in robotics. Patterns of the service-oriented architectures are adopted to support emergent cooperation between autonomous robots. The robots do not need to know each other in advance. As each service presents its provided and consumed interfaces, a very flexible composition of services is possible during runtime. Availability and reliability of the system of systems is increased by decentralized peer-to-peer communication and control. Not depending on a centralized cloud-control is an essential part of the concept. In future work, we examine safety aspects and the performance of the concept in sizable multi-robot environments on real hardware.

Author Contributions: Conceptualization, L.S., V.S., B.W. and C.T.; methodology, L.S.; software, L.S.; validation, L.S., V.S., B.W. and C.T.; formal analysis, L.S.; investigation, L.S. and V.S.; resources, L.S.; data curation, L.S.; writing—original draft preparation, L.S.; writing—review and editing, V.S., B.W. and C.T.; visualization, L.S.; supervision, V.S. and C.T.; project administration, V.S. and C.T.; funding acquisition, V.S. and C.T. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Berlin Institute for Applied Research (IFAF) within the SiReSS-project. The APC was funded by IFAF.

Acknowledgments: We are grateful to Expleo Group SAS, InSystems Automation GmbH and samoconsult GmbH for the ongoing cooperation including the productive discussions with their industrial perspective on the developed technical concepts. Our thanks are also going to Berlin Partner for Business and Technology GmbH for the support in connecting local science and industry.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Bergenhem, C.; Pettersen, H.; Coelingh, E.; Englund, C.; Shladover, S.E.; Tsugawa, S. Overview of platooning systems. In Proceedings of the 19th ITS World Congress, Vienna, Austria, 22–26 October 2012.
2. Baldwin, W.C.; Sauser, B.J.; Boardman, J. Revisiting "The Meaning of Of" as a Theory for Collaborative System of Systems. *IEEE Syst. J.* **2015**, *11*, 2215–2226. [CrossRef]
3. Boardman, J.T.; Sauser, B. The Meaning of System of Systems. In Proceedings of the 2006 IEEE/SMC International Conference on System of Systems Engineering, Los Angeles, CA, USA, 24–26 April 2006; pp. 118–123. [CrossRef]
4. Dorri, A.; Kanhere, S.S.; Jurdak, R. Multi-Agent Systems: A survey. *IEEE Access* **2018**, *6*, 28573–28593. [CrossRef]
5. Russell, S.J.; Norvig, P.; Davis, E.; Edwards, D. *Artificial intelligence: A Modern Approach*; 3rd ed.; Prentice Hall Series in Artificial Intelligence; Pearson: Harlow, UK, 2016.
6. Bruneo, D.; Puliafito, A.; Scarpa, M. Mobile Middleware: Definition and Motivations. In *The Handbook of Mobile Middleware*, 1st ed.; Auerbach Publications: New York, NY, USA, 2016; pp. 145–167.
7. Tsardoulias, E.; Mitkas, P. Robotic frameworks, architectures and middleware comparison. *arXiv* **2017**, arXiv:1711.06842.
8. Open Source Robotics Foundation, Inc. Why ROS 2? Available online: https://design.ros2.org/articles/why_ros2.html (accessed on 6 July 2020).
9. Juan, S.H.; Cotarelo, F.H. *Multi-master ROS Systems*; Technical Report; Institut de Robòtica i Informàtica Industrial: Barcelona, Spain, 2015.
10. Erős, E.; Dahl, M.; Bengtsson, K.; Hanna, A.; Falkman, P. A ROS2 based communication architecture for control in collaborative and intelligent automation systems. *arXiv* **2019**, arXiv:1905.09654.
11. Gutiérrez, C.S.V.; Juan, L.U.S.; Ugarte, I.Z.; Vilches, V.M. Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications. *arXiv* **2018**, arXiv:1809.02595.
12. Object Management Group, Inc. DDS Portal—Data Distribution Services. Available online: https://www.dds-foundation.org/ (accessed on 6 July 2020).
13. OPC Foundation. Available online: https://opcfoundation.org/ (accessed on 6 July 2020).
14. Google. gRPC. Available online: https://grpc.io/ (accessed on 6 July 2020).
15. Apache Software Foundation. Apache Kafka. Available online: https://kafka.apache.org/ (accessed on 6 July 2020).
16. Siemens AG. Coaty—The Lightweight Open-Source Framework for Collaborative IoT. Available online: https://coaty.io (accessed on 6 July 2020).
17. Bellifemine, F.; Poggi, A.; Rimassa, G. JADE—A FIPA-compliant agent framework. In Proceedings of the PAAM, London, UK, 30 April 1999; Volume 99.
18. Campo, C. *Directory Facilitator and Service Discovery Agent*; Technical Report; Universidad Carlos III de Madrid: Madrid, Spain, 2002.
19. Martin, J.; Casquero, O.; Fortes, B.; Marcos, M. A Generic Multi-Layer Architecture Based on ROS-JADE Integration for Autonomous Transport Vehicles. *Sensors* **2019**, *19*, 69. [CrossRef] [PubMed]
20. Alzetta, F.; Giorgini, P. Towards a Real-Time BDI Model for ROS 2. In Proceedings of the 20th Workshop "From Objects to Agents" (CEUR-WS.org), Parma, Italy, 26–28 June 2019; Volume 20, pp. 1–7.
21. Richardson, C. Microservices Patterns: With Examples in Java, Chapter 1. In *Microservices Patterns: With Examples in Java*, 1st ed.; Manning Publications: Shelter Island, NY, USA, 2018.
22. Lemos, A.L.; Daniel, F.; Benatallah, B. Web Service Compositionn: A Survey of Techniques and Tools. *ACM Comput. Surv.* **2015**, *48*, 1–41. [CrossRef]

23. Schmidt, M.T.; Hutchison, B.; Lambros, P.; Phippen, R. The Enterprise Service Bus: Making service-oriented architecture real. *IBM Syst. J.* **2005**, *44*, 781–797. [CrossRef]
24. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*; Mazzara, M., Meyer, B., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 195–216. [CrossRef]
25. Richardson, C. Microservices Patterns: With examples in Java, Chapter 3.2.4. In *Microservices Patterns: With Examples in Java*, 1st ed.; Manning Publications: Shelter Island, NY, USA, 2018.
26. Chen, W.; Yaguchi, Y.; Naruse, K.; Watanobe, Y.; Nakamura, K.; Ogawa, J. A Study of Robotic Cooperation in Cloud Robotics: Architecture and Challenges. *IEEE Access* **2018**, *6*, 36662–36682. [CrossRef]
27. Chen, Y.; Du, Z.; Garcia-Acosta, M. Robot as a Service in Cloud Computing. In Proceedings of the 2010 Fifth IEEE International Symposium on Service Oriented System Engineering, Nanjing, China, 4–5 June 2010; pp. 151–158. [CrossRef]
28. Ribeiro, L.; Barata, J.; Colombo, A. MAS and SOA: A Case Study Exploring Principles and Technologies to Support Self-Properties in Assembly Systems. In Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops, Venice, Italy, 20–24 October 2008; pp. 192–197. [CrossRef]
29. AUTOSAR. Specification of Communication Management. In *Adaptive Platform*; AUTOSAR GbR: Munich, Germany, 2019.
30. Wudka, B.; Thomas, C.; Siefke, L.; Sommer, V. A Reconfiguration Approach for Open Adaptive Systems-of-Systems. In Proceedings of the 2nd International Workshop on Governing Adaptive and Unplanned Systems of Systems, Coimbra, Portugal, 12 October 2020.