Combining Service-Oriented Computing with Embedded Systems - A Robotics Case Study

Alexander Jungmann, Jan Jatzkowski, Bernd Kleinjohann

Combining Service-oriented Computing with Embedded Systems - A Robotics Case Study

Alexander Jungmann, Jan Jatzkowski, and Bernd Kleinjohann

C-LAB, University of Paderborn,
Fuerstenallee 11, 33102 Paderborn, Germany
{global,janj,bernd}@c-lab.de

Abstract. In this paper, we introduce an approach for combining embedded systems with Service-oriented Computing techniques based on a concrete application scenario from the robotics domain. Our proposed Service-oriented Architecture allows for incorporating computational expensive functionality as services into a distributed computing environment. Furthermore, our framework facilitates a seamless integration of embedded systems such as robots as service providers into the computing environment. The entire communication is based on so-called recipes, which can be interpreted as autonomous messages that contain all necessary information for executing compositions of services.

Keywords: Embedded Systems, Service-oriented Computing, Service-oriented Architecture, Services, Behaviour-based Robotics, Mobile Robots.

1 Introduction

Embedded Systems such as mobile robots are usually restricted in their computational capabilities. Although technology is progressing and computing capacities for embedded systems are gradually increasing, algorithms applied to embedded systems usually have to be highly specialized in order to ensure feasibility. Furthermore, although sophysticated algorithms may already exist for a problem at hand, those algorithms might be too computationally expensive. Consider, e.g., an autonomous robot that has to navigate based on visual information in a non-deterministic environment. Camera images have to be processed (at least in soft real-time), natural or artificial landmarks have to be detected, and a robust localization mechanism has to estimate the robot's pose.

In our work, we investigate to what extend techniques from Service-oriented Computing (SOC) can be applied to the field of embedded systems in order to overcome these limitations and consequently increase the functionality of computationally restricted embedded systems. SOC represents a new generation distributed computing platform [4]. It is a cross-disciplinary paradigm for distributed computing that gradually changes the way software applications are designed, delivered, and consumed. For our work, that means, that computational expensive functionalities are outsourced into a distributed computing environment and provided as services to all entities in this environment. At the

same time, we investigate how embedded systems can be integrated as service providers into the entire environment, so that distributed applications can make use of them.

In this work, we make use of a concrete application scenario in the robotics domain: autonomous, mobile robots have to accomplish an objective, which they could not solve under normal circumstances due to their limited computational capabilities. Apart from the robots, the realized system makes use of several servers for computational expensive tasks. That is, functionality for tasks such as an Extended Kalman Filter (EKF) [8] based localization are provided as services. The whole system builds upon a Service-oriented Architecture (SOA), which handles the overall communication.

The remainder of this paper is organized as follows. Section 2 introduces the case study including the BeBot as embedded system and the concrete application scenario. Section 3 introduces our SOA framework as basis for the entire system. The node-based architecture that enables a BeBot to act autonomously is described in Section 4. Section 5 presents the integration of the BeBot into the SOA and briefly describes the overall system that realizes the application scenario. Section 6 finally concludes the paper.

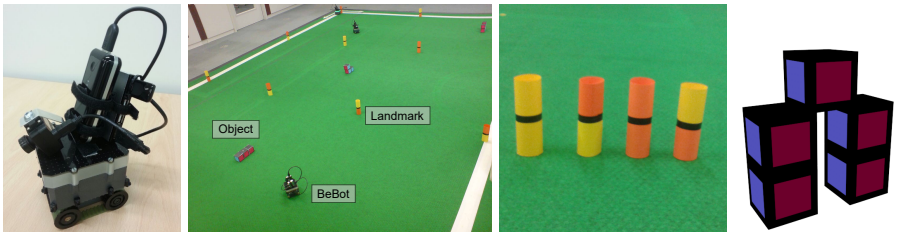
2 Case Study

The main purpose of our case study is to provide a concrete application context for identifying promising application possibilities of techniques from Service-oriented Computing to the field of embedded systems.

Embedded System: Miniature Robot BeBot. The BeBot is a miniature chain driven robot, which has been developed at the Heinz Nixdorf Institute of the University of Paderborn [5]. Despite its small size (approximately 9cm×9cm×8cm), it is equipped with modern technology such as an ARM Cortex-A8 CPU with a maximum frequency of 600 MHz accessing up to 256MB RAM of main memory for running an embedded Linux environment. By illuminating its so-called light guide in arbitrary colours during runtime, the miniature robot can express its current state to human observers and other robots. The network based TCP/IP communication is enabled by an integrated W-LAN module.

In order to extend the field of view for our application scenario, we replaced the built-in camera by a Firefly MV USB camera attached on top of the BeBot. Furthermore, we additionally replaced the integrated W-LAN chip by a customary W-LAN USB stick, resulting in a significantly higher network performance (117 kb/s vs. 2 mb/s). Attaching all devices on top of the BeBot finally leads to the rather unconventional construction depicted in Figure 1a.

Application Scenario. The objective to be solved in the application scenario is i) to utilize BeBots as mobile physical sensors in order to locate artificial objects (henceforth simply referred to as objects) in a predefined yet partially



(a) BeBot (b) Environment (c) Landmarks (d) Object

Fig. 1: Ingredients and setting of the application scenario.

non-deterministic environment and to ii) reconstruct 3-dimensional models of these objects based on images taken by BeBots. Figure 1b shows the setup of the scenario environment. The system may use several BeBots to fulfil the task stated above. In order to facilitate the localization process of the BeBots, the playground contains several artificial, colour-coded landmarks (cf. Figure 1c).

The objects consist of fixed sized cubes with blue and magenta coloured faces on the sides. The top and bottom face are coloured in black (cf. Figure 1d). Furthermore, the edges of the cubes are also coloured in black to allow a better distinction between separate cubes next to each other.

3 Service-oriented Architecture

The main goal of our Service-oriented Architecture (SOA) is to provide a distributed computing framework for all participating *entities* such as BeBots or dedicated servers. The framework enables the BeBots to outsource computationally expensive tasks, while it simultaneously enables the entire system to make use of the BeBots as physical sensors in the environment.

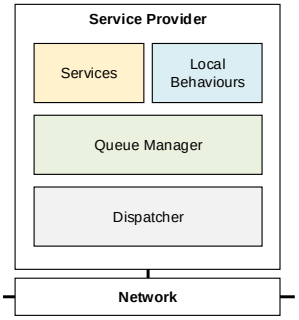
Overview. The key concept of our framework are *services*. Services are distributed and usually stateless components that encapsulate distinct functionality [3]. For addressing a composition of services, we developed a uniform and data-driven protocol based on so-called *recipes*. Recipes are autonomous messages travelling through the network containing all information and data to complete a complex task step by step. The main idea is that a service receives a recipe, extracts the required data, processes this data, appends the processed data (i.e., the result) to the recipe, and finally forwards the updated recipe to the next services defined in the recipe. Please note that we use the terms recipe and message synonymously in this work.

Another building block of our SOA are *service providers*, which realize the environment for executing services. Service providers are interconnected via network and take care of the recipe packing, unpacking, and parsing, execution management, as well as the routing and transmission of a recipe to the next

```
'recipe' :
    'id'      : 0,
    'service' : 'GaussianFilterService',
    'provider': ('192.168.0.1', 5000),
    'params'  : 'k_size' : (-1, 'k_size'),
                'image'  : (-1, 'image')
    'id'      : 1,
    'service' : 'DisplayImageService',
    'provider': ('192.168.0.2', 5000),
    'params'  : 'image': (0, 'result')

'data'  :
    (-1, 'k_size') : 3,
    (-1, 'image')  : [...]
```

(a)



(b)

Fig. 2: (a) Exemplary recipe. (b) Fundamental components of our SOA.

service-provider (if necessary). In fact, our SOA corresponds to a network of loosely coupled service providers. That is, each entity (BeBot or server) participating in the overall system features a local management unit in terms of a service provider instance.

Recipes. A recipe is a data driven construct to define i) an order in which specific services have to be executed and ii) how input and output parameters of the services have to be connected to achieve a certain goal. That is, a recipe defines and describes an orchestration of services. Initial input as well as intermediate and final result values are stored in a dedicated data section of the recipe.

Figure 2a shows an excerpt of an exemplary recipe. The `'recipe'` section contains two services. The first service (`'id': 0`) is provided by a service provider located at IP `192.168.0.1` and accessible via port `5000`. The service implements a Gaussian filter for reducing image noise. The input parameters (kernel size `'k_size'` and `'image'` to be processed) are stored in the `'data'` section. The second service (`'id': 1`) displays the processed image on a different entity in the network. The corresponding input data `'image'` is not yet available in the recipe, but will be stored with key `(0, 'result')` by the first service in the data section.

Service Provider. A service provider resides directly on top of the network and consists of multiple components on three different levels of abstraction (cf. Figure 2b). The *dispatcher* implements the application-level protocol for sending and receiving messages over the network. On top of this, the *queue manager* handles parsing of recipes, manages local services, and acts as intermediary between them. The top layer consists of individual services and so-called *local behaviours*.

Figure 3 gives a detailed overview of the processes within a service provider. The dispatcher is responsible for the communication between different service

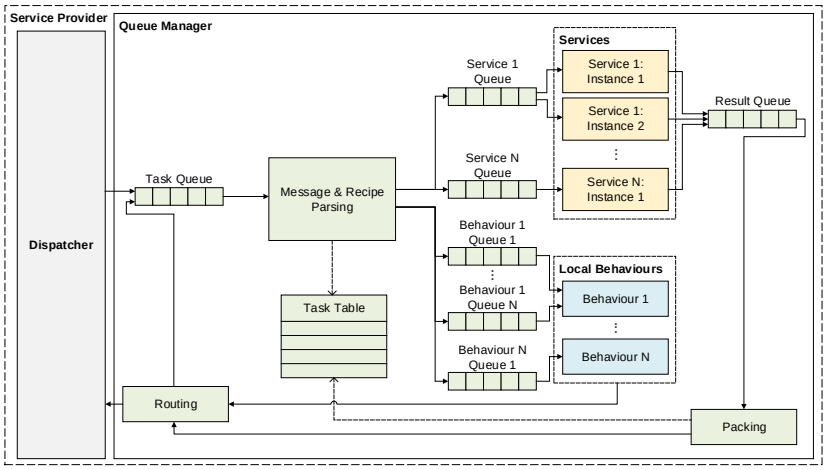


Fig. 3: Internal processes of a service provider.

provider instances among the network. Each message is serialized before it is sent across the network, and is de-serialized after it was received. In order to allow concurrent message processing, each message reception and sending task is handled in an individual thread. After de-serialization, the dispatcher puts the respective recipe into the task queue of the queue manager.

The queue manager is the heart of a service provider: Recipes are parsed and processed. That is, the next service to be executed and the associated input parameters are extracted from the recipe. The extracted information is put into the input queue of the corresponding service type. Service instances of the same type are polling on this queue. Whenever data is available in a queue, one service instance takes the data, processes it, and puts the result data into a public result queue. The queue manager appends the computed result value to the corresponding recipe. In order to keep track of which result belongs to which recipe, unique task ids are generated and stored in a so-called task table. In this way, the execution of services is strictly separated from any recipe parsing.

After being repacked, a recipe is processed by the routing component of the queue manager in order to determine the next recipe-specific processing step. If the next service is located at the same service provider, the recipe is put into the task queue of the queue manager again. Otherwise, the recipe is forwarded to the dispatcher, which takes care of sending the recipe to the respective service provider in the network.

Services vs. Local Behaviours. Within our SOA, there are two main types of computation units: services and local behaviours. These modular units provide a standardized way of computation steps that can be accessed and combined by means of recipes. They form the main logic of every application that uses our SOA for distributed computing.

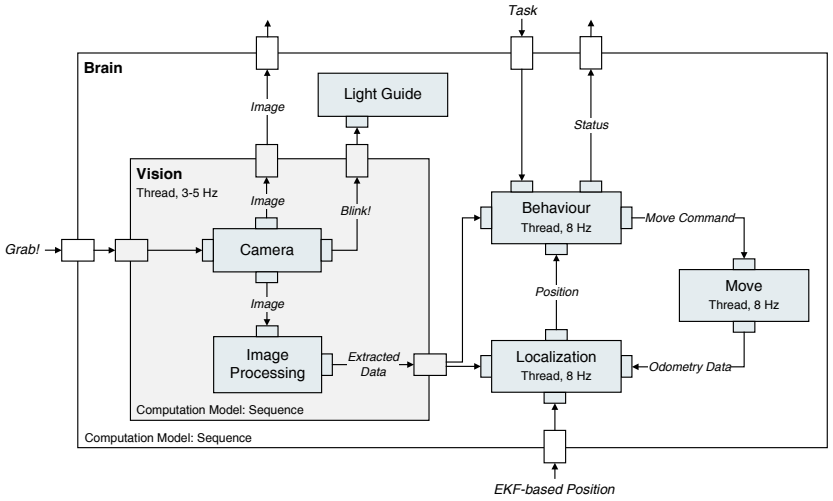


Fig. 4: Node-based software architecture of a BeBot. Ports are not labelled.

According to the design principles of Service-oriented Computing [3], services provide a stateless execution of a predefined task. However, in order to cope with inherent stateful tasks such as localization, we introduce so-called local behaviours as "stateful services". In contrast to services, which are only executed if input data is available, local behaviours can be executed periodically. Furthermore, local behaviours may have multiple input queues and have full control over them. That is, behaviours are not automatically executed when new recipes are available, but recipes are explicitly taken out of the input queues by the behaviour according to its application logic. Finally, in comparison to services, local behaviours can make use of other services by creating and emitting recipes. That is, local behaviours access the routing component of the queue manager (cf. Figure 3) and directly inject new recipes into the overall system. This concept allows to seamlessly integrate stateful and more complex functionality into the SOA framework.

4 BeBot - Basic Node-based Architecture

The robot's software system for autonomous behaviour is built based on a node-based software framework that facilitates the periodic execution of tasks. Figure 4 gives an overview of the node hierarchy of the system. Ports and port interconnections implement the data flow between nodes. The control flow corresponds to the applied computation model such as sequential execution. Furthermore, nodes can be initialized as a Thread with a distinct frequency. That is, executing a node's application logic can be decoupled from other nodes. Copying the data from one port to another, however, always depends on the defined computation model of the corresponding parent node.

Brain: The *Brain* node is the root node of the hierarchy. The Brain node's own input and output ports serve as a connection between node architecture and the SOA framework of the overall system. The computation model corresponds to the following sequence: [Vision, Localization, Behaviour, Move, Light Guide]. That is, data is copied among the ports based on the defined port interconnections according to the order of the nodes in this list. The Brain node itself is not a Thread. Its compute handler is explicitly called by a *Wrapper* that glues together node architecture and SOA framework (cf. Section 5).

Vision, Camera, and Image Processing: The functionality for capturing and processing images is split up into three nodes: Camera node, Image Processing node, and Vision node. The Camera node implements the image acquisition step. Captured images are sent to the Image Processing node. However, if triggered by its input port, single snapshots are additionally sent to the Brain node. In this manner, whenever necessary, images can be provided to services. The Image Processing node extracts scenario specific information from captured images by means of a colour-based segmentation algorithm [7] and creates landmark- and cube-data. Also, the intersection points that are used for the collision avoidance are calculated in this node.

The Vision node implements no application logic but encapsulates the distinct functionalities of the Camera node and the Image Processing node. Furthermore, the Vision node is running threaded to decouple the vision functionality from other functions such as localization. This is necessary, since the vision node is the slowest node within the hierarchy: It runs with 3-5 Hz, depending on the amount of features that were detected within an image.

Light Guide: The Light Guide node implements the interface to the robot's light guide and offers different functions. For example, the light guide gives a visual feedback to indicate that a BeBot has captured an image for the 3D reconstruction process: After an image was taken, the robot blinks three times.

Localization: The Localization node provides the currently estimated position of the robot to the Behaviour node. Due to the high computational effort of the necessary algorithms, an EKF-based estimation of the robot position cannot be done on the BeBot. As a consequence, the functionality is outsourced into a local behaviour within the SOA (cf. Section 3). The Localization node gathers the data coming from the Vision node and the odometry data from the Move node and transmits it to the localization provider.

Behaviour and Move: The Behaviour node coordinates the robot's behaviour. In each computation cycle, it executes one step of a behaviour state machine [1] according to the currently assigned task in order to obtain the next move command. A move command is composed of a translational velocity and a heading direction, and is subsequently passed on to the Move node. The Move node implements an abstraction layer for the actuator's related functionality by mapping a move command to the actual chain speeds. It also incorporates collision

avoidance techniques based on an efficient occupancy grid map approach [2]. Furthermore, the Move node gathers the raw odometry data and transmits the data to the Localization node in order to make a local pose prediction.

5 Integration: Service-Oriented Robotics

So far, we presented the application scenario, our proposed SOA framework, and the BeBot with its node-based software architecture for autonomous behaviour. We now describe how BeBot architecture and SOA framework are integrated. Furthermore, we briefly describe the overall system that realizes the application scenario.

BeBot Wrapper: A Local Behaviour. A BeBot is offering services such as "drive to position" and demanding services such as EKF-based localization at the same time. To enable a BeBot to interact with the SOA, a so-called *BeBot Wrapper* is introduced as adapter. It allows for passing messages from the SOA framework to the node hierarchy and offers an interface to the nodes for emitting recipes into the system (e.g., a recipe that contains gathered data for the EKF-based localization process). More precisely, the Wrapper implements a local behaviour in the SOA and explicitly invokes the compute handler of the BeBot's Brain node. The wrapper encapsulates the entire node-architecture as a local behaviour, which runs on a BeBot within the scope of a service provider instance. After each execution cycle, a blocked service is unblocked if data was produced for it.

A BeBot (i.e., the service provider instance running on a BeBot) also offers services. These services can pass data to the input ports of the Brain node by calling the Wrapper's delegate methods. For example, if the system wants a BeBot to drive to a specific position in the environment, the corresponding service is executed with the desired position as input values. Internally, the service uses the wrapper to delegate the task to the Brain node and blocks until it is informed by the wrapper that the task terminated.

Overall System. Figure 5 shows a schematic overview of the basic functional components. Please note, that only local behaviours are depicted. Stateless services (e.g., for image processing) are not displayed. The entire system corresponds to a network of loosely coupled service providers, which are either located on BeBots or dedicated servers. The application logic is distributed in terms of services and local behaviours among the service providers. The entire communication is based on recipes. We already introduced the Wrapper behaviour which integrates the node-architecture of the BeBot into the overall system. For that reason, let us take a closer look at the other functional components.

EKF-based Localization and Mapping: The localization and mapping behaviour encapsulates the EKF-based localization methods and maintains a global map of

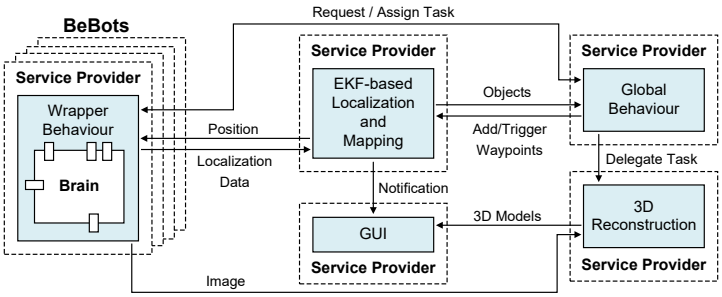


Fig. 5: Overview of the entire system.

the scenario. For each BeBot in the system, a dedicated instance of the EKF is created. Each instance receives localization data (e.g., detected landmarks) from the corresponding BeBot. More concretely, the localization node of a BeBot gathers the localization data and emits a recipe by using the interfaces provided by the Wrapper behaviour. After the computation step, the localization and mapping behaviour emits a recipe with the newly estimated position to the service provider instance of the corresponding BeBot. Furthermore, a notification recipe is sent to the GUI behaviour for updating the visualization, and the global behaviour is notified about newly tracked objects.

GUI: The GUI behaviour is created for monitoring the localization and mapping process. The behaviour receives notification updates from the localization and mapping behaviour and visualizes the current state in a two dimensional map. The actual GUI is running in a separate Thread created by the behaviour. Furthermore, the GUI provides some convenient control elements, e.g., in order to reset the EKF of a BeBot or to manually move a BeBot to a certain position in the environment.

Global Behaviour: This behaviour is responsible for coordinating the BeBots and delegating the 3D reconstruction process. It sends recipes including tasks such as "explore environment and discover objects" or "capture image of side X of object Y" to a selected BeBot, which adjusts its strategy accordingly. Whenever the global behaviour receives a notification about a newly tracked object, it organizes the reconstruction process. That is, four waypoints for taking images from each side of the object are created. The waypoints are subsequently used for assigning tasks to the BeBots. Furthermore, the waypoints are also integrated into the global map for enabling BeBots to identify waypoints in their direct surroundings.

3D Reconstruction: This local behaviour is responsible for the actual reconstruction process. If four images of the same object (an image from each side) are available, the reconstruction process takes place. The resulting 3D model is subsequently sent to the GUI in order to visualize it. A detailed description of the concrete reconstruction process, however, is beyond the scope of this paper.

6 Conclusion

In this work, we introduced an approach for combining embedded systems with techniques from SOC. That is, computational expensive as well as coordination tasks are outsourced as services and so-called local behaviours, and integrated into a distributed computing environment. While services correspond to stateless functional components, local behaviours allow for integrating statefull, periodically executed functionality into a distributed application. Furthermore, the presented approach allows for a seamless integration of embedded systems into a distributed application in order to provide distinct functionalities as services to other entities.

In the future, starting from our latest results in the On-The-Fly Computing project [6], we want to investigate to what extend processes such as service composition (recipe generation) and service integration and deployment can be automated in our SOA framework.

Acknowledgments. This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (SFB 901) and by the German Ministry of Education and Research (BMBF) through the project "it's OWL Intelligente Technische Systeme OstWestfalenLippe" (02PQ1021) and the ITEA2 project AMALTHEA4public (01IS14029J).

The authors gratefully acknowledge the contribution of Mouns R. Husan Almarrani, Maarten Bieshaar, Dominik Buse, Dominic Jacobsmeyer, Simon Merschjohann, Florian Pieper, and Christopher Skerra to the project "SoPhysticated - Service-oriented Cyber Physical Systems".

References

1. Arkin, R.C.: Behavior-based Robotics. MIT Press (1998)
2. Borenstein, J., Koren, Y.: Histogramic in-motion mapping for mobile robot obstacle avoidance. IEEE Transactions on Robotics and Automation 7(4), 535–539 (1991)
3. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall (2005)
4. Erl, T.: SOA Principles of Service Design. Prentice Hall (2008)
5. Gausemeier, J., Schierbaum, T., Dumitrescu, R., Herbrechtsmeier, S., Jungmann, A.: Miniature robot bebot: Mechatronic test platform for self-x properties. In: Proceedings of the 9th IEEE International Conference on Industrial Informatics. pp. 451–456 (2011)
6. Jungmann, A., Mohr, F.: An approach towards adaptive service composition in markets of composed services. Journal of Internet Services and Applications 6(1), 1–18 (2015)
7. Jungmann, A., Schierbaum, T., Kleinjohann, B.: Image segmentation for object detection on a deeply embedded miniature robot. In: Proceedings of the International Conference on Computer Vision Theory and Applications. pp. 441–444 (2012)
8. Thrun, S., Burgard, W., Fox, D.: Probabilistic Robotics (Intelligent Robotics and Autonomous Agents). MIT Press (2005)