ATINER CONFERENCE PRESENTATION SERIES No: COM2023-0295

ATINER's Conference Paper Proceedings Series
COM2023-0295
Athens, 29 September 2023

A Case Study on using Microservice Patterns in an Embedded System

Till Hänisch

Athens Institute for Education and Research
9 Chalkokondili Street, 10677 Athens, Greece

ATINER's Conference Paper Proceedings Series
COM2023-0295
Athens, 29 September 2023
ISSN: 2529-167X

Till Hänisch, Professor, DHBW Heidenheim, Germany

A Case Study on using Microservice Patterns in an Embedded System

ABSTRACT

Microservice architectures, initially a consequence of devOps organization, are nowadays the most fashionable architecture for enterprise or web scale applications. By separating functionalities into small, easily understandable, and maintainable parts, connected by a clearly defined interface, they provide several desirable characteristics. A microservice can be scaled, developed or updated independently from the others since the implementation of a service is by nature opaque. Organizing teams along services, the individual team can be small and communication overhead reduces to the absolute possible minimum, i.e., the interface. These are the same characteristics that electronic components have, esp. ICs, the building blocks of practically every electronic system in use today. It is therefore even more surprising that this architecture pattern is not used to the same extent by electronics engineers, especially in embedded systems. This paper presents a case study of dividing an embedded system in several small – and easy – components, each of them running on a small and simple system, all of which are connected by simple interfaces, in this case serial connections.

Keywords: *Microservice architecture, embedded systems, architecture patterns*

Introduction

Microservices are independently releasable services, that encapsulate a business relevant functionality and provide a well-defined interface to access it over the network (Newman, 2021). The key difference to Service Oriented Architectures (SOA) is, that the individual services are deployable separately. They run as separate processes, typically inside of a container or virtual machine. Access is typically through a REST-API. The implementation is completely language or platform agnostic. They are smaller than the services of a service-oriented architecture and used as building blocks to construct applications. Microservices are a child of cloud centric application development where the distribution, deployment and scalability of services is key. To allow that, it is important, that services are self-contained, that means especially, don't use shared state, often in form of a shared database, and be - as much as possible - independent of other services. That means, that microservices can be replaced or replicated without affecting other microservices in the system (Sommerville, 2021). Achieving this goal seems to be easier, when a microservice is related to a single business function (ibid). Obviously, there is a strong relation between microservices and Domain Driven Design, a method described in (Evans, 2003). Especially the bounded contexts of DDD do form a natural boundary for microservices.

Of course, there are ideas and publications about using microservices in embedded systems, see for example (Reglin, 2020) but most of them deal with the case of running many or all services on a single computer, often even in the same image. This would be considered a monolith in normal systems. Other authors like (Rabault, 2022) go farther and use the same technology (containers, standard interface protocols like REST etc.) in embedded systems, because the problems in embedded software engineering are very similar to those of the rest of the world: moving targets require flexibility and redesign of the system. With embedded systems that often means changing hardware and starting over from scratch. While it is possible to transfer the ideas and technologies of microservice architectures to embedded systems at least for some domains, see for example (Dobaj, 2018) about industrial internet of things, in many applications the complexity and the resources needed for using these techniques are not available because of cost, size or energy constraints.

Even in those cases, where these resources are available, the distributed and independent nature of microservices brings additional complexity with it such as complex deployment processes, complex error detection and handling, complex debugging, communications failures, requiring increased monitoring, due to unexpected performance problems etc. Of course, all these problems can be solved, see for example (Gartziandia, 2021), since they apply to every use of microservices. But it must be considered, if the gain in flexibility is worth the price of complexity, especially in embedded systems where the flexibility is often limited by the hardware. We do not attempt to solve this problem in general, our contribution is to provide some insights from a real-life project using modest hardware.

There are other attempts like luos[1] running on ESP32 MCUs (luos.io) which try to implement something like services but without the overhead, which are interesting to follow in their progress.

An alternative to implement something like services is to use a real-time operating system like Free RTOS or ROS running several processes that communicate via standard inter-process communication techniques. This is the classic attempt to integrate different functions in an embedded system. From an architectural point of view, this is a conventional monolithic implementation of real-time requirements. It allows to run different tasks on the same hardware. The price is the (sometimes hidden) complexity of concurrent programming with limited resources and critical sections and paths.

We will use a different approach based on first principles of software engineering and keep things as simple as possible.

In this case study we will describe a project, that experienced exactly the challenges mentioned above. The case-study is a small sized prototypical example. The total effort is split equally between mechanical engineering and software development. By using ideas from the microservice world, we could avoid restarting from scratch even when faced with major changes in the requirements. Instead of using REST, containers, Kubernetes and all the other attractive current solutions, our approach relies on classic approaches of electrical and software engineering and simple hardware components. Clearly defined interfaces help to maintain lower complexity while tackling the profits described above.

In the next section we will describe the project on a high-level view and derive our architecture from general best practices of software engineering. By analyzing the problems (and their solutions) that occur during development we will highlight the key findings of this case study.

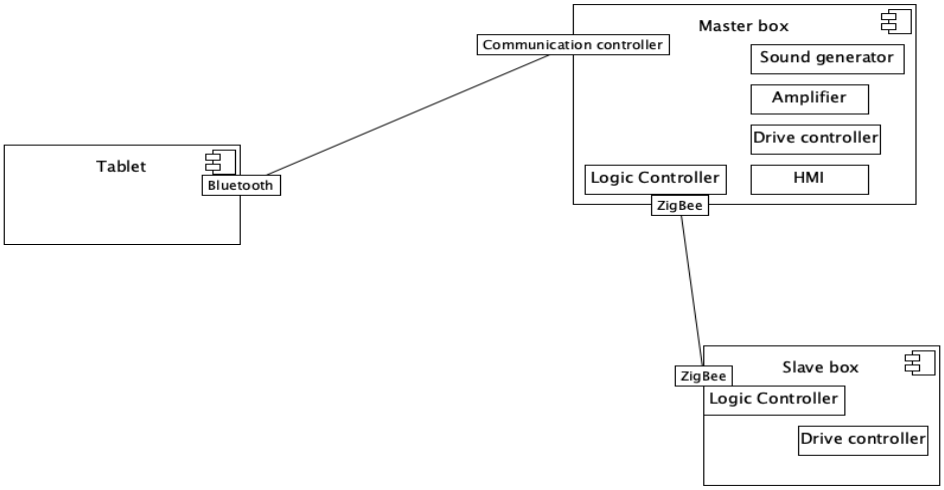
Background

The project covered in this paper was to build the control system for a high-end audio speaker with electromechanical components (drives, sensors etc.), controlled by an App with added measurement functionality for automatic room adaption. The technical details were unclear at the beginning of the project since it was a research project started in 2019. The system was designed for a life span of more than 10 years and to be built in small quantities, typical for the price tag of more than hundred thousand Euro.

The structure of the system is relatively simple and shown in Figure 1.

[1]https://www.luos.io/

Figure 1. *Components of the System*



The user controls the system either via an App running on a tablet or smartphone that sends commands to the master box via Bluetooth. The master box communicates with the slave box via ZigBee.

The development was done by a small team of4 persons for the hard- and software, and 1 person for mechanical engineering.

Development was explorative and the requirements volatile. Software and Hardware (electronics and mechanics) had to be developed independently in parallel. The pandemic required individuals to work separately due to contact limitations.

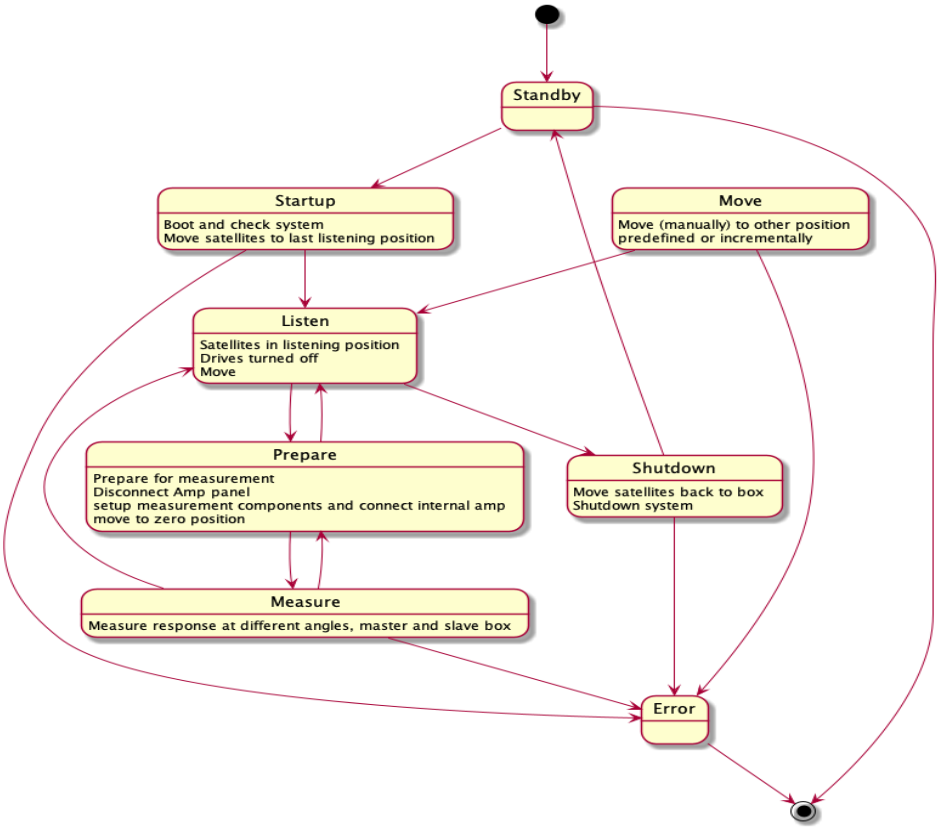
The ideas guiding the development of the architecture were:

- Maximize flexibility,
- loosely coupled components,
- low complexity,
- keep every important functionality on an independent, simple controller, instead of using a complex central embedded system with a real-time operating system,
- communication via simple (ASCII-text) messages transmitted via a serial line (UART), and
- process control for complex actions at one, isolated point, instead of being distributed over the system.

The guiding metaphor was to implement “hardware objects” communicating via messages (Details see Section 3).

The simplified state chart of the system showing the possible flow of events is shown in Figure 2. The coordination of the activities described here is done by the “Logic Controller” component, see Figure 3.

Figure 2. *Simplified State Chart of the System*



The components of one box are shown in Figure 3.

Measurement mode is completely controlled from the tablet. The app is written in Java for Android.

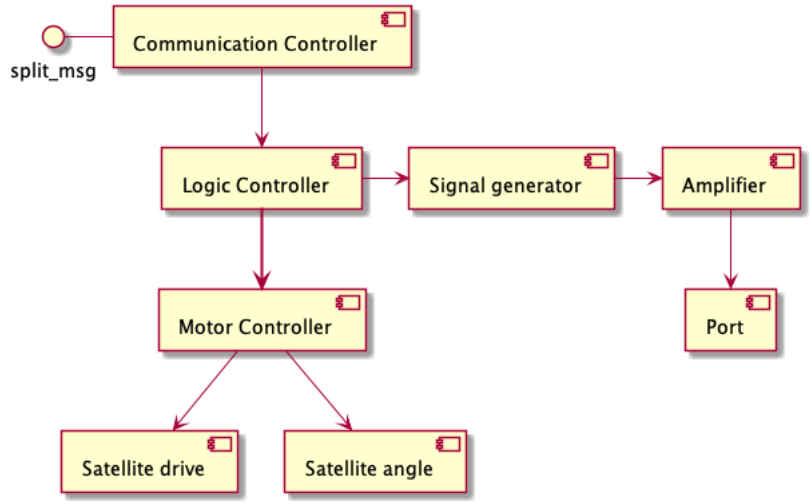
In measurement mode the satellites are moved to their starting position first. Then a few measurements are performed for fixed steps. Each step consists of the following tasks: 1) The test signal is played for a fixed time 2) Audio is captured and analyzed. Steps 1 and 2 are repeated several times. If the difference between the measurements is too large, the measurement has to be repeated, due to issues with the equipment.

Before every measurement the system tests, if the box is disconnected from the external amplifier. If yes, the amplifier is activated, and the playback of the test signal is started. As soon as the tablet signals, that the signal was detected and the measurement has finished, the playback is stopped. If this message is not received in a predefined time, the measurement mode is terminated, the amplifier deactivated, and the error is signaled to the user.

At the end of the measurement cycle the sound generator and the amplifier are deactivated and the box is connected to the external amplifier port. The box is in listening mode, this is signaled to the user.

The tablet isn't needed for listening mode. While in listening mode, the satellites can be adjusted via the tablet in manual mode. Manual mode is activated from the App.

Figure 3. *Component Diagram of One Box*



Each box (component) in Figure 3 represents a separate controller unit.

The communication controller processes the incoming message and forwards them to the logic controller.

The logic controller interprets incoming messages and creates commands for the sound generator, the amplifier and its control, motor control and object recognition. The logic controller monitors the condition and state of the system and starts error handling if it detects problems.

The logic controller is designed to be as easy as possible to allow it to (re-) start as fast as possible to react to incoming messages. During listening it can be put in sleep mode to reduce EMI problems. It will be activated by the other components when needed.

The sound generator, amplifier control and relays are needed only in measurement mode and can be deactivated during listening or manual mode.

Logic controllers and motor controllers are Arduino compatible teensy boards. Both motor controls are autonomous, in the sense that they receive and process commands like “move to this state”.

The motivation for this architecture is to need as few electronic components active while listening to music. All unnecessary components can be turned off or at least in sleep mode while listening. Ideally all digital components are turned off while hearing, but this requires a method for reactivating the control system, when the tablet should be used.

Applying Microservice Patterns to Embedded Systems Design

There are many ways to use the central ideas of Microservices in embedded systems. But at the very core of those ideas is the idea of encapsulation and information-hiding. These are very old and broad ideas: Compared with the definition of UNIX-characteristics by (McIlroy, 1978) “Make each program do one thing well. To do a new job, build afresh rather than complicate old

programs by adding new "features.", the parallels to the Open Closed Principle and the characteristics of microservices are obvious.

Typical characteristics of Microservices (Fowler, 2014):

1. Communicating with lightweight mechanisms
2. Using services as components, explicit interfaces
3. Smart endpoints and dumb pipes (esp. no ESB)
4. Running in its own process
5. Built around business capabilities
6. Design for failure
7. Evolutionary Design
8. May be written in different programming languages
9. Use different data storage technologies
10. Independently deployable
11. Products not projects (“you built it, you run it”)
12. Decentralized Governance
13. Decentralized Data Management
14. Infrastructure Automation

Numbers 9-14 are organizational and/or infrastructure aspects which are either irrelevant for embedded systems or at least for this project, so we will ignore them. Number 8 is obviously nice to have and, since part of the project is an App written in Java, was a requirement for the project anyway. Since we won’t cover the development process in this case study, number 7 is beyond the scope of this case study. Number 6 is obviously an important part of functional safety in embedded systems and made up a large part of the motion control in this project but is too specific in detail to be covered here. Number 5 needs some adaption, because business capabilities are not a common concept in embedded systems, but if we interpret them as functional groups, then we used exactly that for structuring the components (see figure 3). We interpreted “running in its own process” (number 4) as running on its own node to reduce complexity. Numbers 1-3 and 5 (in our interpretation) are the most interesting aspects and we will cover them in the next sections in more detail.

Key for the successful application of services is being able to change some implementation (or operational) aspect without affecting other service or clients. This is a very old goal of software engineering regarding components independent of the implementation mechanism (for example modules, objects, or services) is used.

The importance of loosely coupled modules (as they typically called and implemented components back then), described by low coupling is at the foundation of systematic program design, see for example (Yourdon, 1979): “The more that we must know of module B in order to understand module A, the more closely connected A is to B. The fact that we must know something about another module is a priori evidence of some degree of interconnection even if the form of the interconnection is not known.” In fact, we don’t want to know anything about component B when using component A. A more modern

view (using objects) of that is the Dependency inversion principle, the “D”-Part of SOLID (Martin, 2003).

The same goes for high cohesion or “intramodular functional relatedness” (Yourdan, 1979, p. 95), the more modern variant being the Single Responsibility Principle, the “S”-Part of SOLID.

Loose coupling and high cohesion are essential for the independence of services. The same goes for high cohesion, see (Sommerville, 2021, p.165).

Both can be achieved or at least supported by being dependent only on abstract interfaces and not on implementation (details). That means numbers 2 and 4 of our list of characteristics follows direct from basic principles of software engineering that are well tested since over 40 years.

High cohesion is not immediately on our list of characteristics. But Number 5 gives at least a hint to the right direction: Deciding, what a single microservices does based on the business context in the Domain Driven Design perspective leads to high cohesion.

An important factor leading to the recent success of Microservices is probably late binding (extreme loose coupling), a very popular concept.

A general definition of “late binding” relates to fixing the value of a variable or address of a function not at compile time but at run time. A good explanation including implementation in ANSI-C is given in (Schreiner, 1994).

For our purpose, we interpret late binding as: The resolution of the name of service being called is done at the latest moment possible – when it is called - not before that. That gives us the flexibility to change many properties of a service at run time, like replication or sharding it or using a different implementation etc. That means, the caller doesn’t have to know much about a service: the name of the service and the interface being used. When called, a request is made, that sends something looking like a message to the callee. The nice thing (and maybe the reason, why it works so well) is, that the typical implementation of a microservice uses REST-calls via HTTP. To find the service, DNS is used to resolve the address of the server given in the URL. That fits nicely to the definition of object-oriented programming by Alan Kay (Kay, 2003). Compared with the techniques used before (SOAP via whatever, XML over ESB) these protocols easily qualify as lightweight mechanisms (our number 1 of characteristics). Number 3 also supports this point.

We started with a list of characteristics of microservices and discussed, which of them apply to embedded systems. We found that some of them are not applicable (or relevant) in this area. Others are basically rules of solid software design. In the discussion we were not very explicit about the “lightweight communication mechanism”, we will cover that in the next section.

Consequences for the architecture of the system under investigation

At the begin of the project we discussed intensively the pros and cons of using a central powerful controller with a real time operating system (FreeRTOS or ROS in this case) as a platform. The pros are obvious, see for example

(Lethaby, 2013): using a well-tested platform for scheduling and communication makes life much easier. Explicit prioritization of tasks and guaranteed response times make life much easier and therefore it is industry standard to use a RTOS in such projects. There are several smaller cons like additional complexity from an additional platform, more resources required to run the OS and the additional effort of having to maintain the platform over a long lifespan (10-20 years), which is even harder when using the open- source variants like ROS which are changing quickly. But the main point, which led to the decision not to use a RTOS is: if we separate the "business" tasks not only in separate processes but on separate nodes, we didn't have to schedule anything, because on every node (except the App which runs on the Android OS, so it isn't relevant here) there is only one task with a well-defined flow of events.

By splitting the system in independent "hardware services" (every service has its own hardware), we don't have any parallel activities running on one node except the communication, especially receiving messages from the other node. And this problem can be solved by a simple interrupt service routine. The main reason for this being possible, is the use of smart motor drives which receive a trajectory and control the acceleration and deceleration of the drives independently. So, controlling the drives (two axes) was also implemented by a "hardware service", which we bought as an off-the-shelf component.

Using a sperate controller for each hardware service might look like a waste of resources at the first glance. But considering the history of components in electronic engineering it does not seem that eccentric: In electronic engineering it is common to use components like operational amplifiers in analog circuit design or logic gates (or combinations of them like PALs or FPGAs) as building blocks of larger components. The interfaces, in this case mechanical cases, pinouts, supply voltages or logic levels are highly standardized. Only this way of reusing existing designs in form of components, which are separately packed and connected in the appropriate way to realize a required function made the quick innovation cycles possible we know today. Even though that approach uses more resources (transistors in this case), the gain in productivity and reliability is worth it.

So, the first important decision was to use hardware services: one service per device. That materializes independent services, built around business capabilities, platform independence and explicit interfaces (together with the communication mechanism discussed next).

The second question was, how the services should communicate, "communicating with lightweight mechanisms" being number one on our list of characteristics of microservices. There are several possibilities, all of them commonly used. Starting with the standard techniques in all computer systems, (wired) Ethernet and Wifi which are commonly used for the communication between complex systems in bigger (number and size) systems like industrial shopfloor automatization. We didn't need the power of these and want to avoid the resource requirements of these computer network techniques. Communication between individual controllers with low bandwidth requirements in industrial (near) real time systems is often implemented with CAN (Controller Area

Network), for example in automotive applications. We used that to communicate with the drive controls.

Board level communication is typically done via serial protocols like SPI (Serial Peripheral Interface) for higher bandwidth requirements (for example for DAQ-systems), I2C for lower bandwidth.

We didn't need the performance of SPI and I2C is not good at transmitting variable length messages, so we took the easiest approach: simple ASCII-text messages for communication between all components using UART (Universal Asynchronous Receiver/Transmitter). It requires only two wires and is available on all microcontrollers. Additionally, the data paths are fixed in this point-to-point topology, so we don't even need addressing.

All messages are of the form <command> {parameter}. Typically, the commands have only one parameter, in some cases two or three but not more. So, the individual messages are short (commands are one or two characters each). That means, buffering messages does not require extensive memory in case a controller cannot process a message immediately.

An important advantage of using UART-communication is, that it can be implemented transparently over many other protocols like Bluetooth, ZigBee or USB, all of them we used in the project.

The second important decision was to implement communication via messages, transmitted via simple protocol (ASCII over UART).

Conclusions

We traced back our architecture to solid principles of software architecture and drew the connection to the characteristics of microservices. Transferring these ideas to embedded systems is possible even without sticking to the technical implementation (and the overhead), microservice architectures typically use.

The independent, parallel development of the individual components (App, process control, motor control) with mockups for the missing parts worked well. Using messages transmitted via serial lines added some complexity (buffering to avoid losing messages) but allowed very simple debugging using just terminal emulation and/or simple scripts. Redesigning the complete external communication from Bluetooth/ZigBee to Wifi and the topology from master/ slave to fully connected proved to be very simple (only a few days of work).

References

Dobaj, Jürgen, Iber, Johannes, Krisper, Michael, Kreiner, Christian, A Microservice Architecture for the Industrial Internet-Of-Things, Proceedings of the 23rd European Conference on Pattern Languages of Programs (EuroPLoP '18). Association for Computing Machinery, New York, 2018

Evans, Eric, Domain Driven Design: Tackling Complexity in the Heart of Software, Addison Wesley, 2003, p. 161

Fowler, Martin, Microservices, 2014, Available online https://martinfowler.com/articles/microservices.html

Gartziandia, Aitor, Microservice-based performance problem detection in cyber-physical system software updates, Proceedings of the 43rd International Conference on Software Engineering: Companion Proceedings (ICSE '21). IEEE Press, 147–149, 2021.

Kay, Alan, Mail exchange about what OOP is, available online at https://www.purl.org/stefan_ram/pub/doc_kay_oop_en", 2003

Lethaby, Nick, Why Use a Real Time Operating System in MCU Applications, Texas Instruments Whitepaper, 2013, available online https://www.ti.com/lit/wp/spry238/ spry238.pdf?ts=1681631181658

Martin, Robert C, Agile Software Development, Principles, Patterns and Practices, Prentice Hall, 2003

McIlroy, M. D., Pinson, E. N., Tague, B. A., Unix Time-Sharing System: Foreword". The Bell System Technical Journal Vol. 57, No. 6, Bell Laboratories, 1978

Newman, Sam, Building Microservices, O'Reilly, 2021, p. 1

Rabault, Nicolas, How microservices enhance agility in embedded systems development, Embedded.com, available online https://www.embedded.com/how-microservices-enhance-agility-in-embedded-systems-development/, 2022

Reglin, Frank, Verteilte Embedded Systems mit leichtgewichtigen Microservices realisieren, Elektroniki Praxis, available https://www.embedded-software-engineering.de/verteilte-embedded-systems-mit-leichtgewichtigen-microservices-realisieren-a-165e68421d1b0bda5c9e5f33ac1a7c42/, 2020

Schreiner, Axel-Tobias, Object-Oriented Programming With ANSI-C, Hanser, 1994, p.15

Sommerville, Ian, Engineering Software Products, Pearson, 2021.

Yourdan, Edward, Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design, Prentice Hall, 1979, p. 76