The Soul of a Service – Petri Net Based Model for a Microservice-controller

Balakrishna Subramoney
Solutions Architect & CEO
Sun Bio America, Cary, North Carolina, U.S.A.
balu@sunbioamerica.com

Abstract

This article proposes an approach to building Microservices. This approach differs from others because of the design principles it uses. In this approach, the Microservice-controller (µSC) is the key design abstraction. A Microservice is built around the µSC. The µSC is internal to the Microservice (µS), serving as the brain or the control-unit of the Microservice. Further, the article suggests that Petri nets form a good basis for modeling the µSC. Microservices designed and constructed using this approach provide several advantages over other development methodologies. Implementation details have been omitted due to lack of space. It is the author's fond hope that the readers will be able to relate to the principles outlined.

1 Singularity, Modularity and Granularity

"Top-down and bottom-up are both strategies of information processing and knowledge ordering, used in a variety of fields including software, humanistic and scientific theories (see systemics), and management and organization. In practice, they can be seen as a style of thinking, teaching, or leadership." – Wikipedia (Tom-down and bottom-up design , 2017)

1.1 Top-down Approach

The top-down (outside-in) approach to Software engineering starts with architectural abstractions which are broken down to arrive at a design, which in turn is implemented in code. These abstractions (partitions) range from singularities at one end to granularities at the other; modularity being the golden mean. Modularity is not a point on a line it is a range of possible values (Figure 1).

In the author's opinion, Software engineering has borrowed ideas from other branches of engineering, notably civil engineering! Civil engineering is thousands of years older to Software engineering. Unfortunately, many civil engineering concepts cannot be applied mutatis-mutandis in Software engineering. Building singularities (monoliths) being one of them!

To illustrate the concept, imagine a 1000 line C program written entirely within the main function – that is a singularity (no partitioning) – difficult to build and change; if each line of the program is a function it would be an example of granularity (fine partitioning) – easy to build and change but difficult to put together. A Software is modular – C program broken down into optimum number of functions – easy to build **and** put together.

Modularity was (and sometimes is) the holy grail of Software design!

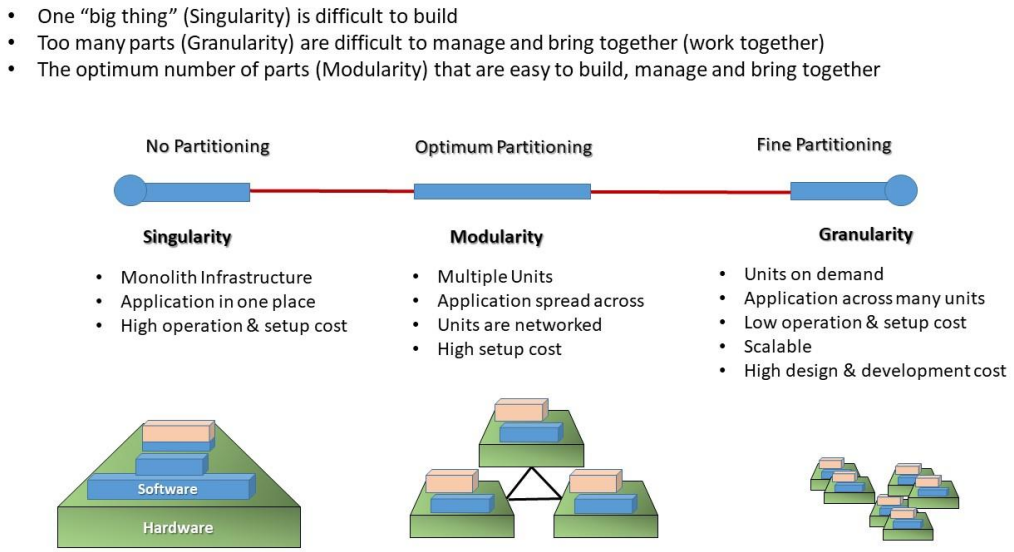


Figure 1

1.2 Bottom-up Approach

A bottom-up approach is best suited to build Microservices (granularities). The µSC (Microservice-controller) is the central abstraction, wrapped by other parts of the Microservice. Concentric circles represent a Microservice with the µSC being the center. In this approach, a list of events that can occur within the scope of the Microservice is prepared. The list of events and associated conditions along with the corresponding handlers helps define the Microservice-controller. The conditions define the controller-lattice and the event-handlers become the service routines. Please see figure 2.

Please note that a miniaturized version of a monolith (skyscraper) is not a Microservice! The macro approach is different from the micro!

Figure 2 shows the containment hierarchy of an application. The Microservice-controller models the application logic (internal events and handlers). The µSC has two distinct parts the controller-lattice and the service routines.

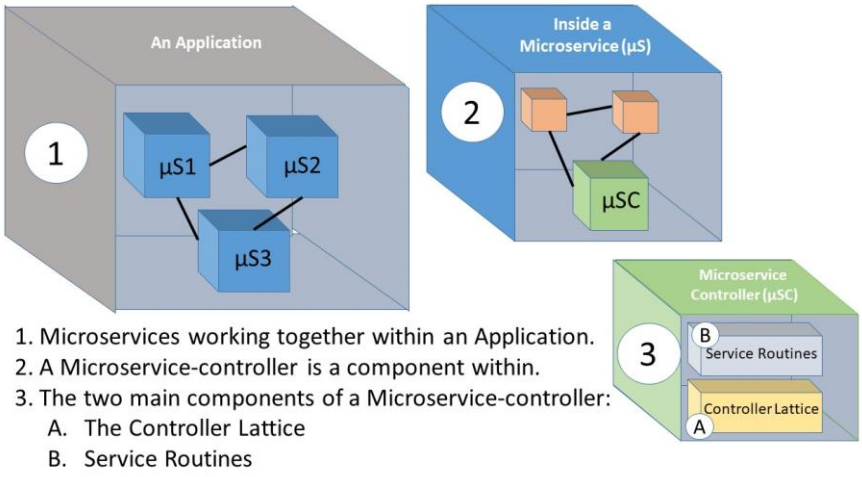


Figure 2 - Microservice-controller in Context

2 Encapsulation Vs Transparency

2.1 SOA and Encapsulation

"In SOA … A service presents a simple interface to the requester that abstracts away the underlying complexity acting as a black box. Further users can also access these independent services without any knowledge of their internal implementation." – Wikipedia (Service-oriented Architecture, 2017) Applications that employ Service Oriented Architecture (SOA) rely on Interfaces – essentially public functions. The requester communicates with the service provider via the interface. Separating the interface from its underlying implementation is an important principle in Software design called, encapsulation. The intention behind encapsulation (pun intended) was to hide the complexity of the implementation. Further, changes in the implementation did not affect the interface and hence, the requester.

The Interface (service-broker) is the mechanism to realize the service-contract between service-requestor and service-provider. Traditional Microservice architectures (subsets of SOA) use this paradigm. In this paradigm, the broker is indispensable! Brokers discover services; proxies are in charge!

2.2 Transparency – Services Sans Brokers

In the proposed approach, when two or more parts of an application (can be Microservices) need to fulfill a service contract between them; they communicate directly following these rules:

1. The service requester (part A) and the service provider (part B) have access to each other's implementation of the service contract. The parts complement each other for the implementation of that specific service contract. Further, part A and part B are always deployed as a pair (if the service contract contains more than two parties, the entire collaboration – all the parts are deployed together)

2. Part A knows the service access path (example: URL) and identity of part B and part B knows the identity of part A (This information is baked into the respective µSC).
3. Every service call returns an acknowledgement/response.
4. The interface (the service access path) provides an identity to the Microservice and defines the communication protocol (example: XML, JSON). It is however not bound to any application logic. Thus serving as a conduit. The µSC implements the application level logic.
5. The above rules are implemented/set (available) when the Microservice is instantiated. Imagine the Microservice started on a virtual machine or started as a web application by the web server. Please note that the Microservice is "built" just in time – using a build script!

Figure 3 illustrates Microservices collaborating within the scope of two applications. Note that the interface can provide a layer of security but is not bound to any application logic. The µSC establishes the runtime identity of the Microservice and the controller-lattice is the soul of the Microservice!

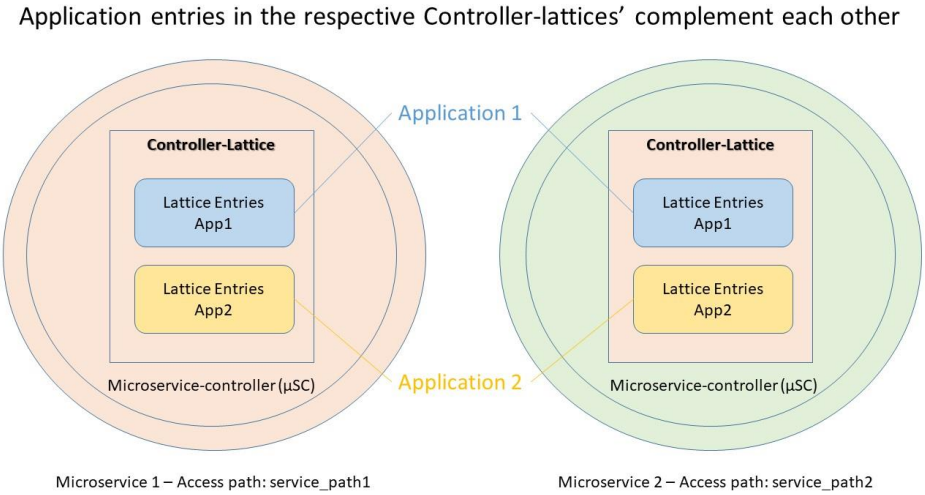


Figure 3

3 More About the Microservice Controller (µSC)

3.1 Petri net Based Model

The µSC keeps track of state and activity information within the scope of the Microservice. The author was not very successful in modeling both states of multiple variables and activities using UML (Unified Modeling Language). In the author's opinion, the UML state chart diagram and activity diagram help model states and activities respectively. Further, state chart diagrams are useful in modelling the states of a single object and tend to get out of hand for multiple dependent objects.

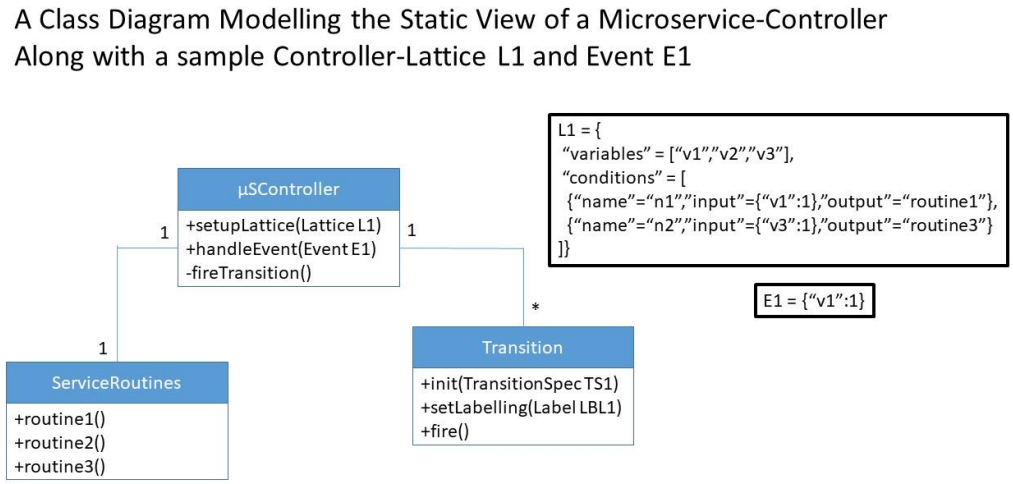
The author resorted to Petri nets, which combined the features of UML state chart and activity diagrams in an elegant way. Inspired by the Petri net implementation in Python by (Tanzer, 2012), the author modelled the µSC as a Petri net.

3.2 Behavior of The µSC

Figure 4 explains the behavioral aspects of the Microservice-controller (µSC). The Controller-lattice is a JSON (JavaScript Object Notation) structure. This JSON structure is akin to a Petri Net with the application-variables forming the places and the conditions the transitions.

Application-variables and their states denote events that can occur within the Microservice. Events are assigned event handlers. This event handling information in JSON forms the 'Controller-lattice'. The µSC receives events via a standard interface and calls the appropriate service routine.

Since, the controller-lattice is a collection of 'event-handling conditions', changing the controller-lattice changes the behavior of the Microservice! Likewise, changing the service routines changes the behavior of the Microservice. However, the two parts of µSC are loosely coupled. This enables reuse of the service routines and the controller-lattice. The service routines can be implemented in any language that can read a JSON file (the controller-lattice). The controller-lattice is baked into the Microservice during instantiation/building.



Note: Inspiration from https://johncarlosbaez.wordpress.com/2012/10/01/petri-net-programming/

Figure 4 - Illustration of Microservice-controller

4 Conclusion

Melding this approach with the SDLC (Software Development Life Cycle): 'Software requirements' are translated into 'events' that the users intend or expect of the application. Events are grouped, mapped to 'candidate services' and detailed. The 'candidate services' are analyzed to arrive at the Microservices. The 'controller-lattice' and 'service routines' required are defined and implemented (in the language of choice). The controller-lattice (the soul of the Microservice) being a JSON structure can be integrate into any instance of the Microservice independent of the language of implementation. This eases definition, construction, testing and deployment of the Microservice as a whole.

References

Control Unit. (2017, September 15). Retrieved from https://en.wikipedia.org: https://en.wikipedia.org/wiki/Control_unit

Microservices. (2017, October 4). Retrieved from https://en.wikipedia.org: https://en.wikipedia.org/wiki/Microservices

Service-oriented Architecture. (2017, September 19). Retrieved from https://en.wikipedia.org: https://en.wikipedia.org/wiki/Service-oriented_architecture

Tanzer, D. (2012, October 1). *Petri Net Programming (Part 1)*. Retrieved from Azimuth: https://johncarlosbaez.wordpress.com/2012/10/01/petri-net-programming/

Tom-down and bottom-up design . (2017, September 30). Retrieved from https://en.wikipedia.org: https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design