The Utility of Service Oriented Architectures (SOA) and Microservice Architectures in Naval Systems

By Dale McIntosh, Forward Slope Inc., Chief Technology Officer,
dmcintosh@forwardslope.com

There is much confusion in the distinction between service-oriented architectures, microservices architectures and service mesh when discussing Navy software architectures. This confusion is reasonable considering that the (RedHat, 2020)terms used to describe each architecture pattern (e.g. services, decoupling, cohesion, etc.) often overlap. The defining difference between SOA and microservices relates to scope. SOA is an **enterprise** pattern for loosely coupled Enterprise applications. SOA applications generally communicate via an Enterprise Service Bus (ESB) that manages communications between enterprise applications (IBM Cloud Education, 2019). Microservices, on the other hand, is an **application** pattern for a single application composed of loosely coupled and independently deployable services (IBM Cloud Education, 2019). (A., 2019) A service mesh provides a mechanism for controlling how different parts of an application communicate and share data with each other (RedHat, 2020).

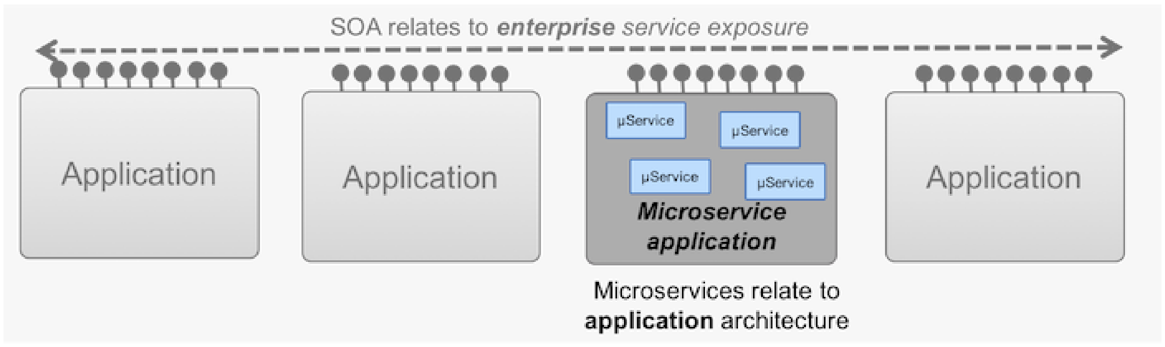
SOA relates to *enterprise* service exposure
Application
Application
µService
µService
µService
µService
Microservice application
Application
Microservices relate to **application** architecture

Figure 1: SOA vs. Microservice Architecture Scope (IBM 2020)

The Navy has embraced the use of SOA architectures since early in the 21st century. The Navy Meteorological and Oceanographic (METOC) community deployed service-oriented architectures as early as 2003 – 2005. The Command and Control Rapid Prototyping Continuum (C2RPC), a SOA-based Command and Control (C2) system was deployed as an operational prototype to Commander Pacific Fleet (CPF) in 2011. The use of SOA was very successful within C2RPC which allowed vendors to independently develop components (sub-systems) such as Open Track Manager (OTM) and Intellex with minimal interaction with developers of other components. These sub-systems were able function as a single system to produce desired results via communications through an ESB. While there are many advantages to SOA, this architectural patten does not meet the reliability and dynamic scalability needs of modern naval systems for the following reasons:

- SOA principals do not address the construction of adaptive resilient systems that can dynamically scale themselves to meet ever-changing resource requirements (e.g. compute, memory, etc.). The use of artificial intelligence and machine learning algorithms in modern military systems can be sporadically resource intensive. In a typical SOA architecture, each service must have resources statically allocated. There are components (e.g. load balancers) that can be incorporated into a SOA-based system to mitigate this deficiency, but it is not native to SOA. If the resources required by a subsystem exceed the allocated resources, the services will fail. The ability to have aspects of a system dynamically reallocate resources to meet temporary processing needs and then return excess resources to the environment when no longer required (elasticity) is outside the scope of the SOA domain.

- SOA systems can be horizontally scaled by manually provisioning multiple instances of a system or subsystem. However, this can be challenging if one or more applications within the service composition maintain state. Stateful applications would need to implement a mechanism to ensure that all calls within a single transaction are directed to the same service instance or the system must be redesigned to ensure that state is universally accessible to all instances of a service. There are additional challenges to horizontal scaling of a SOA system related to packaging and distribution for Java-based systems which tend to be packaged as either a WAR or EAR file to be deployed in an application server (e.g. JBoss). Significant rework is required to enable multiple instances of a WAR file to be deployed in a single JBoss Instance and manage a shared state.

Systems based on microservice architectures generally use containers. These typically run in a Docker or Kubernetes environment, such as RedHat's OpenShift. Each microservice runs in its own container and a bundle of microservice containers can be deployed together as a cohesive microservice application. Each microservice application can be tuned based on differing scalability requirements depending on their operational environment. This enables the environment to dynamically replicate containers to meet temporary resource demands (scale up) and destroy unneeded copies when resource requirements are reduced (scale down) as seen in Figure 2 below.

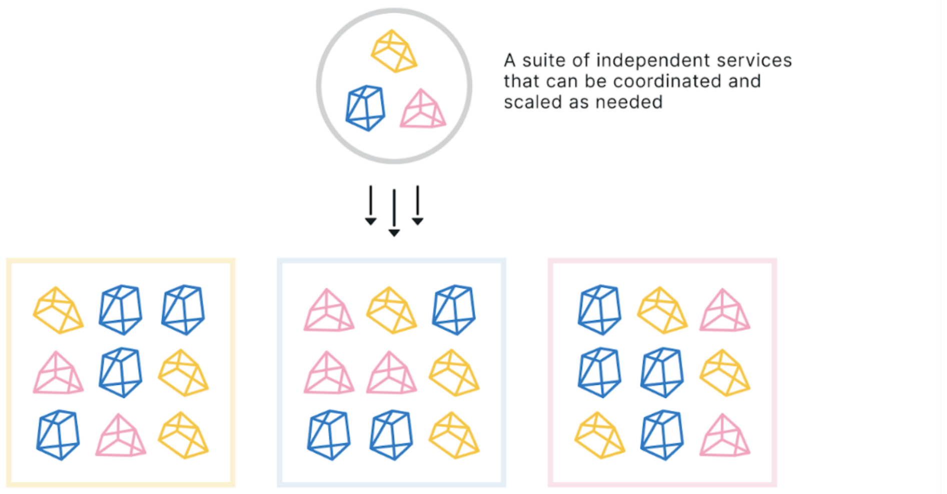
A suite of independent services that can be coordinated and scaled as needed

Figure 2: Microservice architectures facilitate selective and dynamic scaling (O'Rear 2020)

It is important to note, that elasticity generally requires the microservice application to be stateless to enable the load balancer to select a service instance based on availability and load vs access to state information. However, RedHat’s OpenShift (Kubernetes) provides StatefulSets and asynchronous replication to support scalability with stateful pods (pods containing stateful services). A pod is the smallest execution unit in Kubernetes and contains one or more containers. StatefulSets persist state information and makes it available to the services within the pod and its “copies” (Kubernetes, 2020).

There are additional advantages to microservice architectures such as reliability and security. Many of these advantages are dependent on the microservice execution environment. For instance, OpenShift (Kubernetes) has implemented Liveness and Readiness Probes to make services more robust and resilient.

- Liveness Probes are used to determine if the service has degraded and no longer able to process requests within acceptable limits. This can be very useful for determining if the service is unable to fulfill requests to meet service level requirements. When a liveness probe fails, Kubernetes will automatically destroy the offending container and start a new instance. Automated container restart can resolve many system issues and the administrator can be notified of the container restart so that detailed investigation can be performed when convenient to resolve the root cause of the problem.
- Readiness probes are used to determine when a service is available for accepting traffic. This is key during system starts/restarts and when elastically spinning up containers. Network requests/web traffic will not be sent to the container until all applicable readiness probe conditions have been met. This is a significant capability relating to microservice architectures and helps ensure that the system starts and scales reliably and predictably. Many characteristics can alter the amount of time required to start a container and the services within it. If we assume that a container will start in a fixed amount of time and delay the initiation of dependent containers for that amount of time, we run the risk of the dependent containers sending a request before the designated service is available to process the input. This can cause random errors during system startup an impact system stability.

Another significant advantage of microservice-based architectures is dependency isolation. Typical SOA environments generally are packaged with multiple applications/subsystems within a single environment. In a typical Java-based virtual environment, an application service (e.g. JBoss) would be hosted in a Linux server to provide the execution environment for multiple components. In this configuration, the subsystems running in a single application server share a single JBoss environment and any conflicts with library versions and other dependencies must be deconflicted. In addition, any condition that causes the Java Virtual Machine to fail (e.g. memory overrun), causes each of the applications/subsystems within the VM to fail.

In a typical microservices environment, each component/subsystem would be housed in a separate container, each containing its own execution environment, which eliminates configuration conflicts. In addition, since each execution unit is self-contained, when a single container fails, only the functionality contained within that container is impacted and the system can continue to function. If this failed container were configured with liveness probes, the container would be automatically restarted, and system could continue to run without interruption

Microservice dependency isolation provides a distinct advantage when performing system upgrades. Individual subsystems or services can be upgraded or even completely replaced with minimal impact to the overall system. In many cases, services can be replaced while the system is running without any noticeable impact to system users, provided the service interfaces do not change. Predominate container implementations (e.g. Docker) can streamline the upgrade process by only transferring code fragments that have changed without uploading unmodified components. This can be especially significant in an afloat environment with limited bandwidth.

Microservice architectures are generally paired with a service mesh technology (e.g. Consul) which implements capabilities, such as, security, observability, service discovery and message routing. Service meshes secure inter- and extra-service communication using "sidecar" proxies. These "sidecar" proxies bolt onto each microservice and enforce identity-based access and security policies to create a zero-trust network. These policies define the microservice application's boundaries and prevents services from outside the microservice application from communicating directly with any of the application's, providing an additional layer of security. The service mesh's sidecar proxies also have built in support for request tracing, metrics collection, and abstract service discovery out of the microservice. A service mesh is comprised of two components:

- Data Plane – The data plane transmits data between services. Each service instance has an associated sidecar network proxy. The data plane is implemented within the sidecar proxies. All network traffic between service instances flows through the service's sidecar proxy. Service instances are not network aware and each service only knows of its own network proxy. The sidecar proxy performs the following tasks: service discovery, health status checking, routing, load balancing, authentication, authorization, and observability. (Klein, 2017)
- Control Plane – "The control plane takes a set of isolated stateless sidecar proxies and turns them into a distributed system." (Klein, 2017) The control plane maintains system state and the service registry that identifies each service endpoint. This information is provided to the data plan to enable routing. (HasiCorp, 2018)

While both SOA and microservice architectures leverage services, they differ greatly in scope. SOA is an enterprise architecture pattern and microservices is an application pattern. The advent of SOA architecture was significant in getting away from monolithic application development but can be plagued by an inability to support elasticity and lack of resiliency.

Microservice-based architectures can be much more reliable and performant than SOA due to the utility of packaging services to facilitate scalability and reliability. The use of probes can enhance reliability by enabling the system to monitor and respond to service state and failure. While microservice-based architectures avoid the primary pitfalls of SOA, they can be very complicated and must be well architected.

References

A., R. (2019, November 12). *Fantastic Probes And How To Configure Them — A Kubernetes Story*. Retrieved from Medium.com: https://medium.com/swlh/fantastic-probes-and-how-to-configure-them-fef7e030bd2f

HasiCorp. (2018, Sept 17). *Service Mesh and Microservices Networking*. Retrieved from hashicorp.com: https://www.hashicorp.com/resources/service-mesh-microservices-networking

IBM Cloud Education. (2019, October 23). *Microservices*. Retrieved 11 02, 2020, from IBM Cloud Learn Hub: https://www.ibm.com/cloud/learn/microservices

IBM Cloud Education. (2019, July 17). *SOA (Service-Oriented Architecture)*. Retrieved 11 02, 2020, from IBM Cloud Learn Hub: https://www.ibm.com/cloud/learn/soa

IBM Cloud Team. (2020, November 2). *SOA vs. Microservices: What's the Difference?* Retrieved from IBM.com: https://www.ibm.com/cloud/blog/soa-vs-microservices

IBM Cloud Team. (2020, September 2). *SOA vs. Microservices: What's the Difference?* Retrieved from ibm.com: https://www.ibm.com/cloud/blog/soa-vs-microservices

Klein, M. (2017, October 10). *Service mesh data plane vs. control plane*. Retrieved from blog.envoyproxy.io: https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc

Kubernetes. (2020, 11 19). *StatefulSets*. Retrieved from Kubernetes: https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/

O'Rear, E. (2020, April 23). *SOA vs Microservices: Perspectives in Architecture*. Retrieved from lightstep.com: https://lightstep.com/blog/soa-vs-microservices-perspectives-in-architecture/

RedHat. (2020, December 3). *Microservices - What's a Service Mesh*. Retrieved from RedHat: https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh