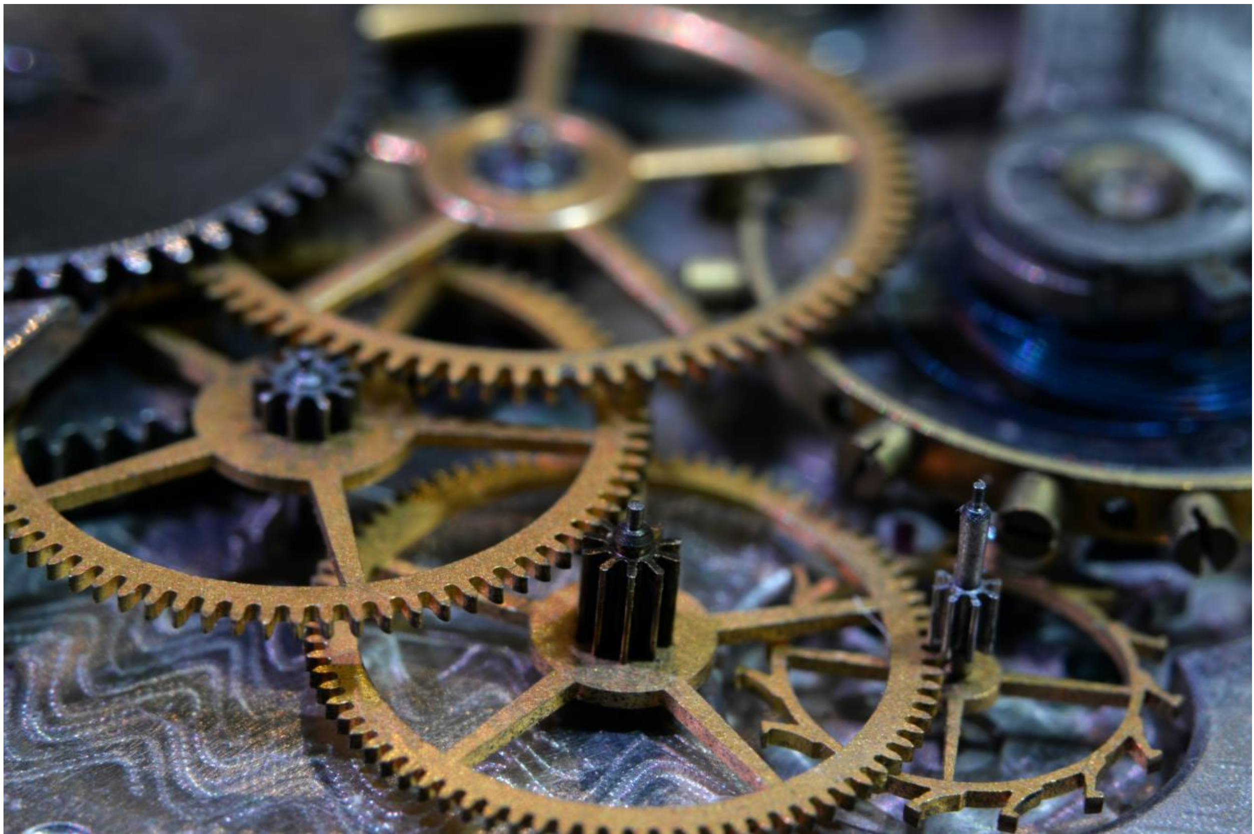
Microservices

What they are and how they benefit us



From then to now:

"In the beginning…"

1. Non-structured languages

Non-structured programming is the earliest programming paradigm used in general-purpose (Turing-complete) computer systems.

Examples of non-structured programming languages:

- Assembler / Machine Code
- COBOL
- FORTRAN
- BASIC

Usually optimized at the machine code level, using things like end call optimization.

2. Early structured languages

The next evolution was procedural programming. Procedural programming breaks code into function blocks, which are parts of the program that perform a specific purpose. A “main” function is generally used which calls each of the individual functions in sequence to perform the overall application task.

Examples of structured languages using procedural programming:

- ALGOL
- C

Usually optimized similarly to non-structured languages, but additionally using structures like heap versus stack and reducing pushes and pops on/off the stack.

3. Interpreters vs. Compilers

- Interpreters allow the programmer to distribute and run source code. The interpreter converts the code to machine language during the execution phase. The benefit is that it is simple to distribute and run a program, but the disadvantage is that it is slow.

- Compilers analyze and convert a program into machine-language before it is run. This extra step adds complexity, but it allows error-checking in advance of runtime and it also makes the program run much faster because the system doesn't have to interpret the source code at runtime.

- Compilers are faster, but interpreters are sometimes easier to use.

4. Object-oriented languages

Another form of structural programming – developed after procedural – was called object-oriented. It grouped programming units into objects and classes, with classes being a “blueprint” for a block of code that defines a specific functional item containing data definitions and functions that are useful for dealing with that data. The class is used to create an instance of one or more objects. Examples of object-oriented languages include:

- Simula (Algol family)
- Smalltalk
- C++ (C family)
- Java
- Python

Object-oriented languages allow programming concepts to be “compartmentalized” in a way that optimizes source-code maintenance.

A note about optimizations

The benefits of structured languages aren't so much for execution speed but more to make the code more manageable. Some benefits make the code arguably more robust, but mostly it is done so that complex source code can be written. It's an organizational tool.

The truth is that most of the things that make advanced programming languages useful actually slow down the compiled binary. But it is more than made up by other tools in the modern software arsenal.

As an example, consider the (often despised) global variable compared with a private member variable in a class that is exposed with getter and setter member functions.

Access to a global variable is much faster than access to a private member variable through a getter or setter function. Access to a global variable is just a matter of reading or writing a particular memory location, whereas getter and setter functions require stack manipulation, which is much more processor-intensive. But the concept of using a private member variable protects manipulation of that variable by accident or by aberrant code.

So the benefits of modern structured programming are much more for source code organization than they are for performance. The benefits are seen in speed to develop, test and maintain. And that includes developing other tools to enhance performance, like memory managers, load balancers and the like.

You can compare modern languages to superchargers. A supercharger is driven by the engine, so it robs power from it. But the power gains made from boosted intake pressure more than make up for the loss.

5. The push for *Portability*

- ### Standardized languages

  Early non-structured languages were proprietary and so even language “families” had slightly different commands. So the first attempts to make portable code focused on standardizing languages, making them have the same command set on different machines.

- ### Libraries and device drivers

  Standardized languages were a step forward, but this still left all device access specific to the platform. To standardize access to peripherals, libraries and device drivers were formed. That allowed access to devices to be abstracted from the programmer. For example, file I/O is performed with a library call that accesses device drivers specific to the system. The source code works without modification on various platforms, just by compiling on that platform.

- Interpreters within Virtual Machines

  As systems became more and more complex, the build process also became more and more complex. Also – perhaps more importantly – distribution to various platforms could be difficult where there is a mix of hardware. So in 1991, SUN Microsystems developed Java, which essentially went back to an interpreted model. It allowed the distribution of code to various machine types, each with its own machine-specific virtual machine that would run the code.

- Compiling at runtime

  Allows the system or the VM to run a compiled and/or partially-compiled and optimized program from distributed source.

6. Parallel Processing

- ## Shared-memory

  Works best for tightly coupled applications, highly application-specific, requires semaphores and/or other locking mechanisms, often used in hardware, e.g. hardware cache.

- ## Message-passing

  Works best for code that is “naturally parallelizable,” e.g. algorithms or processes that can be easily split and rejoined without locking and/or race problems.

- ## Distributed computing

  Multi-node peer-to-peer networks – essentially a high-level message-passing architecture.

- ## Multiple-core processors

  Employ shared-memory or message-passing approaches, but tend to favor shared-memory via caching.

7. Synchronous vs Asynchronous

- Synchronous code executes in a linear fashion and cannot take advantage of parallelism.

- Asynchronous code has inputs and outputs that do not need to be synchronized, so it *can* be parallelized.

8. Dependency-injection

- Much like an advanced library.
- An abstraction layer for subsystem types.
- Allows the development environment to specify a software component or even a whole subsystem type without being tied to interface details.
- Makes it relatively easy to switch between software components, services, vendors and even whole subsystems.
- Testing can be accomplished by mocking components or whole subsystems
- Vendor ties are relaxed because subsystems are abstracted, e.g. database

9. VMs evolve into Containers

- Until 2010 or so, most applications software ran either on dedicated servers or in virtual machines on servers.
- Virtual machines virtualize an entire operating system, and they have software managers to control access to physical devices.
- In 2013, Docker created a “platform as a service” product that virtualized applications and configurations as a bundle or “container” that can be run on a computer.
- Only the application is virtualized, not the entire operating system. The Docker container communicates with the operating system, so all Docker instances share the resources of the server. It is very much like a VM, but it doesn’t include the OS.
- So virtual machines are “fat” compared to containers
- Examples of containers:
    - Docker
    - Kubernetes
    - Containerd
    - Podman

10. Container Managers

- Container managers automate the creation, deployment and scaling of containers. Container management facilitates the addition, replacement and organization of containers, which is important in complex systems of large scale.
- Some available container managers:
  - Kubernetes
  - Google Cloud Platform
  - Microsoft Azure
  - Amazon AWS

11. Microservices

- Microservices bring all of this together.
- They are the latest software evolution.
- They use development frameworks like Spring Boot, which employs Java and Dependency Injection as well as several other technologies that reduce boiler plate code. For example, Lombok removes the need to write getters and setters.
- Unit and integration test tools are “built-in,” making unit tests, regression tests and integration tests more integrated with development. Of course, an application can still be written without any built-in tests, but it is part of the framework and has become increasingly embraced by developers.
- Microservices can be run on an individual developer’s system, on a test system, on a dedicated production server or on a cloud server.
- Microservices allow porting of legacy systems in small chunks. A portion of the legacy code – a service – can be coded as a microservice and the legacy application can be modified to use it.

- Little by little, a large monolithic application can be ported to the cloud by replacing subsystems with microservices in small, easily managed sprints.
- Microservices tend to be parallelizable. Asynchronous code (like WebFlux) is *highly* parallelizable. But even blocking code can be parallelized by tools like Kubernetes, given its ability to expand and contract – spinning up containers as needed.
- Reliability is enhanced because a blocked and hung (container) process can be automatically killed by Kubernetes and another spun up.
- Of course, the biggest advantages are seen in stateless processes – whether coded with blocking or flux code – because stateless processes can be asynchronously executed.
- Stateful processes require additional consideration.
- Note that a Microservice *can* be written with other tools. A C++ application can be written and compiled, launched in Docker containers, managed by Kubernetes and deployed in an on-prem or offsite cloud server.
- But the most common Microservice code is Java, because of its platform-agnostic nature.

Conclusion

One of the promises of "the cloud" is the ability to take advantage of technologies that exploit concurrency. Load balancers can expand and contract containerized code, adding services as needed and then taking them down when load relaxes. Of course, to take advantage of this, the code must be parallelizable.

A database that is written to is a stateful application of shared memory. It responds best to shared-memory techniques, like caching and striping.

However, a read-only database can be easily parallelized with multiple copies. Read-only Lookup tables and rulesets (for rules engines) can be distributed to multiple container instances.

So careful planning really helps an organization take advantage of microservice architectures.

They can be quickly and easily implemented, so by starting small, an enterprise can begin to take advantage of microservices almost overnight. A complex system can be made with multiple microservices, each with its own specific task. Changes to any of the microservices can be made without affecting any of the others, so feature changes, regression testing and maintenance releases are much less painful than in larger monolithic applications.