The latest developments in advanced architectural patterns: a survey

*

Abhijit Taware
Computer Science (Masters student)
UNB
Fredericton, Canada
ataware@unb.ca

Luigi Benedicenti
Computer Science (Dean)
UNB
Fredericton, Canada
luigi.benedicenti@unb.ca

***Abstract*—With the advent of paradigm like microservices, containerized applications, cloud computing etc., the software developement process is being influenced heavily. This evolved the software developement and triggered the emergence of new architectural patterns. In this document, the common software architectural patterns are brifely discussed. The document further describes latest advancements in service oriented architecture, microservices, reactive programming and resilient software development.**

***Index Terms*—SOA, microservice, reactive programming, resilient software**

I. Architectural design patterns

Large enterprise needs software that scales with ever chaging and increasing needs of the business. Selecting the right architecture before diving into the actual work is crucial to the success of the application and enterprise. This section explores various architectural patterns used in the industry. The pros and cons will be discussed for each of the pattern.

A. Layered architecture

It is the most common architecture style, that organize similar modules into horizontal layers. The layers are independent of others and inteact using exported APIs. An application can be designed using any number of layers. The network protocol stack is a good example of layered architecture. The in upper layer is transmitted to lower layers using encapsulated packets. A layer don't have to know the inner working of other layer and communication happens through a set of APIs exposed by each layer. Another example of bussiness application, that is divided into presentation, logic and data tiers. Following of some of the benefits offered by this architecture.

- Layers can be developed and tested independently.
- Changes made in one layer doesn't affect the other layer, hence maintainable.
- Low coupling and high cohesion
- Lower layers have no dependency on higher layer and hence reusable.

The disadvantages can be summarized as follow.

Identify applicable funding agency here. If none, delete this.

- A change to any component may trigger a redeployment of the entire application.
- Each layer can have separate physical deployment or an entire application can be replicated. It is too coarse grained from deployment perspective.
- Communication across layers can be a performance bottleneck for certain applications.

B. client-server architecture

It consists of a server and multiple clients. The server keeps listening to the client requests. The server responds to any new client requests i.e., provides a service to those clients. E.g. the encryption key control server [6] provides encryption keys to the requesting clients over network. This model is prone to denial of service attack. The scalability requires replicating the server components with load balancing, failover and failback mechanisms.

C. Pipe and filter architecture

This approach is suitable for large applications that can be broken down to multiple steps. Each step refers to a filter. The filter applies a specific function to the data and can work asynchronously as well. The pipes refer to the connectors between these filters. The output of one filter serves as an input for the next filter on the pipeline. The common example is Unix pipes.

- Adding a new step is easy by adding a new filter and adding it to existing pipe stream.
- It is easier to reuse of filters doing generic actions.
- Promotes concurrency of different filters do not depend on each other.
- The errors gets propogated across the filters, which is a downside of this architecture.
- A broken filter leads to a complete broken pipe.

D. Peer to peer architecture

A peer-to-peer (P2P) architecture consists of a decentralized network of peers i.e. nodes. Unline client-server architecture, a node in this architecture can act both as a server and a

client. The workload is split into small chunks that can be reassembled later, allowing peers to work simultaneously on a task. E.g. P2P file sharing, where a file is split into chunks that allows many chunks to be downloaded from diffrent peers at the same time.

- Need for centralized server is eliminated.
- There is no single point of failure, unless the number of peers are too few.
- The increase in number of peers can be handled easily i.e., scalable.
- The model is prone to security issue, as an infected peer can affect the whole network.
- Fairless garuntees are difficult to enforce as many leeches could benefit free riders.
- Instant messaging, file sharing, collaboration apps use P2P architecture. E.g. Bitcoin, BitTorrent, napster etc.

E. Event based architecture

The callbacks mechanism describes this architecture well. It consists of source, listener, and a bus. The event source send a message i.e. an event on the bus to other component. The listener responds by performing some action. The components comminicate only via the event bus. The linux device driver interrupt handlers employ this mechanism.

- Changing name or type of an even requires changes to the listener.
- Too many event sources or listeners lead to bottleneck for the bus.
- Following the control flow is difficult due to asynchronous nature of events.
- The producer and consumer of the event need not be aware of each other allowing for loose coupling
- Loose coupling makes it easy for a component to evolve independently.
- Message passing over the bus introduce additional abstraction and may not be efficient.

F. Interpreter

It consists of a program to be executed, an interpreter which runs such a program, program state and memory component required any storage needs. The interpreted languages like Awk, Perl, Python or Java are prime example of this architectural style of development.

- Program development becomes easy with the help of interpreter.
- Allows portability across various platforms.
- Debugging become easier.
- An extra level of indirection makes execution slower.
- Interpreter can gaurd against malicious instruction

G. Blackboard

It is similar to boardroom where people solve the problem using a whiteboard. The component blackboard acts as a global information store. It allows various components to collaborate towards the final solution. The controller component monitors the blackboard and schedules individual knowledge sources. The knowledge sources are specialized workers with its own representation of the problem. The communication happens through the blackboard. The CAD software is an example of blackboard design.

- Efficient scheduling of tasks and resource management across a distributed network.
- Better suited when a problem can be split into multiple sub-problems.
- Not always easy to break down a task into subproblems.
- Everything is shared and can cause unwated information flows.
- Controller design can become overly complex and unmaintainable.
- Knowledge sources being independent, allows for reusability.

H. Publish Subscribe Architecture

I. Micro Kernel Architecture

II. New architectural styles

The monolithic applications causes lot of bloated code with dependencies and slower deployment cycles. This also leads to more buggy applications. All such issues leads to newer architectural paradigm shift, which is described in the following subsections. Following patterns are discussed exclusively in this context.

- Service Oriented architecture (SOA)
- Microservices
- Containers
- Reactive programming
- Resilient software development

A. SOA

SOA [3] allows application to be modularized into services. A centralized component is responsible for the service integration and comminication. This is achieved using an Enterprise service bus (ESB), which acts as an *orchestrator*. The data model for SOA applications evolve by an agreement between many services. However, such a canonical data model being shared by many services, which limits the evolution. The services often reflect the comminication pattern of the organization, a.k.a. Conway's law [5].

The SOA is often implemented as web services which use following communication methods.

- Simple Object Access Protocol (SOAP) for synchronous communication with other web services, typically over HTTP. Components that use SOAP are often described in a Web Services Description Language (WSDL). The WSDL is an XML-based interface description language that is used for describing the functionality offered by a web service.
- The REpresentational State Transfer (REST) protocol too can be used for basic request/response operations between the services.

Most organizations need to deal with existing legacy applications before adopting the SOA architecture. The key

bussiness processes depend on such legacy systems and hence a step by step solution must be adopted to move towards SOA. Following are some of the options strategies to deal with the change.

- Use commercial off the shelf components to replace existing system. This can be more costly in future.
- Wrap the current legacy system with a middleware that can offer the legacy system interface through a Web service.
- Redevelop the legacy application to achieve the optimal levels of decoupling. However, legacy code can be complex and lack of documentation could make it more challenging.

SOA sometimes is referred to as "simple service and smart pipes" [2].

B. Microservices

Microservices [1] allows applications to be made up of small, self-contained independent units i.e., services working together through APIs interface. Martin fowler [1] describes following key characteristics for of the microservice.

- *Componentization* : Services are independent units that communicate with other services using web requests or remote procedure calls.
- *Business capabilities* : Services represent the business capability, with responsibility for a complete stack.
- *Product rather than projects* : Microservices treat services as products and claim ownership. The ongoing maintainance requires complying with business needs.
- *Dumb pipes and smart endpoints* : Microservices do not use ESBs. The services are intelligent and communication channel is userd merely for message passing. The communicationnication channel does not implement any service functionality i.e., the channel implementation is simple.
- *Decentralized governance* : There is no orchestrator, instead it evolves towards choreagraphy and event collaboration.
- *Bounded context* : Domain-driven design complex domain up into multiple bounded contexts and maps out the relationships between them. This leads each service to have its own localized version of data model promoting loose coupling.

Microservices architecture constatly introduce new services that requires DevOps to ensure component updates in production environment. It has disadvantages such as code duplication, interfaces mismatch, operations overhead and the challenge of continuous testing of multiple systems. However, the benefits of creating loosely coupled components by independent teams using a variety of languages and tools far outweigh the disadvantages. Some of examples of microservice adoption are listed below.

- *Netflix* : Moved away from single war file to full fledged microservice approach.
- *The gaurdian* : This website maintains their core monolithic system but use microservice for new components.
- *Gilt groupe* : Uses massive microservice approach to their shopping site, allowng various teams working on different services.

C. Containers

Containers [7] [8] are often termed as light weight virtual machines. An example is lxc containers built for linux systems. They provides an isolated environment with exclusive access to resources like memory, network, storage etc. The following components enable the containers to provide isolation and bare-metal performance.

- *Kernel namespaces* — A namespace wraps a global system resource making it appear to the processes as their own isolated instance.
- *SELinux* — provides access control
- *SeComp* — restrict malicious usage of system calls
- *chroot* — allows process to change root i.e., restricts process to specific directory structure.
- *Kernel capabilities* — Distinguish between priviledged and un-priviledged process and allow access accordingly
- *CGroups* — Limits, accounts for and isolates process resource usage.

The virtual machines (VM) are widely used to host the applications. A single physical machine can have multiple VMs running allowing for a service per instance. The virtual machines have their own operating system (OS) and are resource heavy. Contrast this with containers which share the kernel with the host OS and provide resource isolation equivalent to VMs. This makes containers more attractive to deploy the microservices. Container can be equipped to have its own IP address allowing it to be the best option in cloud environments like Amazon EC2. Nothing restricts one to create containers insides a VM hosted on a third party cloud.

The containerized applications need to account for failure too. It should have a mechanism to recover from failures, i.e., restarting the service on a different host or respawing the containers. Today the distributed system development, with microservice architectures built from containerized software components [9] is dominating the industry.

D. Reactive programming

Reactive programming (RP) is a new paradigm that deals with the data flows and the propagation of change. It is possible to express static or dynamic data flows with ease in the reactive programming languages, and that the underlying execution model will automatically propagate changes through the data flow. For example, in a modelviewcontroller (MVC) architecture, RP can facilitate changes in an underlying model that are reflected automatically in an associated view [10].

Traditionally such a task was achieved using observer pattern [11]. This pattern is mainly used to implement distrubuted event driven systems, consisting following characteristics.

- A "subject" and "observer" are the main objects.
- Any change to subject is automatically propagated to the observers.

- Observers should be registered with the subject to receive the change notification.
- Subject maintains list of observers and calls update method on all of them in the event of change.

The observer pattern is criticized for lack of composability, inversion of the logical relation among reactive entities and limited readability [12]. Also, it can be a source of memory leaks especially when the observer object fails to unregister itself. During such a situation the observer object can not be garbage collected.

The following pseudocode [13] illustrates the reactive paradigm well. This code snippet receives the mouse.clicked event and the current mouse.position .

1) clicked: Event = mouse.clicked
2) scaledPosition: Signal[(Int,Int)] = mouse.position X 0.5
3) lastClick: Signal[(Int,Int)] = clicked.snapshot(scaledPosition)

The mouse position is scaled by 0.5 and saved to scaled-Position variable. On every mouse click, the snapshot of the position is saved to lastClick. Any change to mouse position will reflect to scaled position in conventional imperative programming language. Same is true about the lastClick variable. However, RP converts the line 2 into a constraint. The RP runtime identifies the dependency between scaledPosition and mouse.position, making realtime updates possible. This technique can be applied to MVC architecture to get the view updated every time something changes in the model.

- There are no bugs because of forgotten updates
- No redundant computations in case programmers code defensively and update too much,
- Applications are easily extensible as constraints can be composed, i.e., built on top of other constraints.

E. Resilient software development

REFERENCES

[1] J. Lewis and M. Fowler,Microservices-A definition of this new architectural term, 2014. https://martinfowler.com/articles/microservices.html

[2] Cerny, Tom J. Donahoo, Michael Pechanec, Jiri. (2017). Disambiguation and Comparison of SOA, Microservices and Self-Contained Systems. 228-235. 10.1145/3129676.3129682.

[3] Gold, N.E., Knight, C., Mohan, A., Munro, M. (2004). Understanding service-oriented software. IEEE Software, 21, 71-77.

[4] Marquez, Gaston Astudillo, Hernan. (2018). Actual Use of Architectural Patterns in Microservices-based Open Source Projects.

[5] M. Conway, Conways law. https://en.wikipedia.org/wiki/Conway

[6] Key Control Server. https://www.hytrust.com/products/keycontrol/

[7] LxC Containers. https://linuxcontainers.org/lxc/introduction/

[8] Kubernetes. https://kubernetes.io/

[9] Brendan Burns and David Oppenheimer. 2016. Design patterns for container-based distributed systems. In Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'16). USENIX Association, Berkeley, CA, USA, 108-113.

[10] Model-View-Controller and the Observer Pattern http://peak.telecommunity.com/DevCenter/Trellismodel-view-controller-and-the-observer-pattern

[11] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley. pp. 293ff. ISBN 0-201-63361-2.

[12] Maier, Ingo et al. Deprecating the Observer Pattern. (2010).

[13] G. Salvaneschi, A. Margara and G. Tamburrelli, "Reactive Programming: A Walkthrough," 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, 2015, pp. 953-954.