International Journal of Research in Engineering and Science (IJRES)
ISSN (Online): 2320-9364, ISSN (Print): 2320-9356
www.ijres.org Volume 12 Issue 1 ǁ January 2024 ǁ PP. 252-255

Revisiting Design Patterns Tailored for Distributed Systems in a Containerized Environment

Srividhya Chandrasekaran
Senior Product Manager
schandrasekaran@spotify.com

Sriram PollachiSubburaman
Senior Software Engineer
pollach@adobe.com

Abstract— *In the realm of containerization, this paper explores and re-evaluates design patterns tailored for distributed systems. Addressing scalability, reliability, orchestration, security, and deployment, the study assesses the adaptability and efficiency of these patterns in contemporary software architecture within containerized environments. Real-world case studies and empirical evidence provide insights into their practical impact, highlighting the crucial role of these patterns for architects and developers. The results highlight their importance in improving agility, resilience, and maintainability in contemporary software development practices.*

Keywords—*Design Patterns, architecture, Kubernetes, Resource efficiency, containers*

Date of Submission: 10-01-2024 Date of acceptance: 24-01-2024

I. INTRODUCTION:

In the late 1980s and early 1990s, object-oriented programming revolutionized software development, introducing modular components. Today, a similar shift is occurring in distributed systems, marked by microservice architectures built from containerized elements. Containers, akin to objects, encapsulate code and dependencies for reliable deployment across environments [1]. They embody the Single Responsibility Principle, mirroring the transformation seen in the late 20th century. Operating as "objects" in distributed systems, containers expose APIs and enforce isolation barriers, resembling design patterns formulated for objects. This paper is inspired by Google Inc. "Borg, Omega, and Kubernetes." [2] and provides a comprehensive summary of emerging design patterns for containers, featuring real-world examples and a comparative evaluation to assess their effectiveness.

II. LITERARY REVIEW

The literature review for "Revisiting Design Patterns Tailored for Distributed Systems in a Containerized Environment" centers on exploring the strengths and weaknesses inherent in various design patterns within the intersection of distributed systems and containerization. The review establishes a baseline for understanding their evolution in contemporary containerized environments by examining existing research on traditional design patterns in distributed systems. The emphasis lies on the influence of containerization on software development, prompting a meticulous examination of how design patterns have historically functioned and adapted in this evolving landscape. Through a systematic analysis, the review identifies and synthesizes the strengths and weaknesses associated with specific design patterns within containers, shedding light on their applicability and limitations. This focused exploration provides valuable insights for architects and developers navigating the intricacies of designing containerized distributed systems.

III. UNIFIED OVERVIEW OF CONTAINERIZED DESIGN PATTERNS WITH DESIGN COMPARISON FRAMEWORK

A. Sidecar pattern

Sidecars are auxiliary processes or services deployed as part of the application [3]. They share the same lifecycle as the parent application, being created and retired alongside it [4].

Exploring the Advantages of Sidecar Patterns below

Scalability: Sidecars facilitate scalability by attaching helper containers and allowing independent scaling.

Flexibility: They offer modularity and adaptability to diverse functionalities without impacting the main service.

Resource Efficiency: They may consume additional resources due to the presence of helper containers.
Resilience: Enhance fault tolerance by isolating issues within individual service-helper pairs.
Security: Provide isolation between main service and helper containers, enhancing security.
Ease of Implementation: Relatively straightforward, with changes contained within individual helper containers.
Example: Dapr [12] uses side-car implementation in their core architecture.

B. Ambassador pattern

Ambassador containers proxy communication to and from a main container.an ambassador container brokers interactions between the application container and the rest of the world. [6].
Scalability: Supports scalability by managing inbound/outbound traffic through a proxy container.
Flexibility: Requires changes in the proxy for modifications, potentially impacting flexibility.
Resource Efficiency: Requires fewer resources as a single proxy manages traffic for multiple services.
Resilience: Ensures reliability, but a single proxy failure can affect multiple services.
Security: This may require additional security measures for communication between services and the proxy.
Ease of Implementation: This may require intricate setup and maintenance due to centralized proxy management.
Ex: Kubernetes-native Ambassador API Gateway [13] built on Envoy proxy.

C. Adapter pattern

The ambassador pattern is a useful way to connect containers with the outside world. An ambassador container is essentially a proxy that allows other containers to connect to a port on localhost while the ambassador container can proxy these connections to different environments depending on the cluster's needs.[7].
Scalability: Facilitates scalability through flexible component integration and adaptability to different interfaces.
Flexibility: Promotes flexibility by allowing integration of components with disparate interfaces.
Resource Efficiency: Efficient in integrating components with varying interfaces.
Resilience: Resilient to interface changes, providing a level of fault tolerance.
Security: Focuses on integrating components; security is typically not a primary concern.
Ease of Implementation: Relatively easy to implement for integrating diverse components.
Ex: Docker compose tool [14] which simplifies the deployment and orchestration of multi-container docker applications.

D. Work Queue pattern

Work queue pattern allows the handling of arbitrary processing code packaged as a container, and arbitrary data, and builds a complete work queue system. The work queue pattern dictates that you split up a big task into smaller tasks to reduce running time. [8]
Scalability: Enables scalable task distribution by decoupling task producers and consumers.
Flexibility: Provides flexibility in handling asynchronous tasks with dynamic task distribution.
Resource Efficiency: Efficient in handling asynchronous tasks and optimizing resource utilization
Resilience: Enhances fault tolerance by decoupling task producers and consumers.
Security: Provides isolation between task producers and consumers; security measures can be implemented.
Ease of Implementation: Straightforward for task distribution; maintenance might involve monitoring task queues.
Ex: KubeMQ [15] a Kubernetes-native, high-performance, message broker and message queue system.

E. Self-Awareness pattern

This pattern describes situations where an application needs to introspect and get metadata about itself and the environment where it is running [9]. On many occasions, an application needs to be self-aware and have information about itself and the environment in which it is running
Scalability: Offers adaptability and scalability by allowing components to dynamically adjust to varying workloads.
Flexibility: Enhances adaptability by allowing components to adjust behavior based on runtime conditions.
Resource Efficiency: Aims to optimize resource usage by dynamically adjusting component behavior.
Resilience: Aims to enhance resilience by dynamically adapting to runtime conditions.
Security: Enhances isolation through dynamic adjustments; security depends on the component's design.
Ease of Implementation: Implementation complexity depends on the dynamic adjustment logic; maintenance involves adapting to changing conditions.
Ex: Autoscaler in Kubernetes [16], a component that embodies this pattern.

F. Scatter/gather pattern

The Scatter-Gather pattern is a message routing pattern that receives a request, distributes it to multiple recipients, and aggregates their responses into a single message [10] . This pattern is used when maintaining the overall flow of messages between senders and recipients who may send responses is necessary.
Scalability: Enhances scalability through parallel processing and aggregation of results.
Flexibility: Flexible in handling distributed tasks and aggregating results.
Resource Efficiency: Optimizes resource usage through parallel processing and aggregation.
Resilience: Resilient through parallel processing; fault tolerance depends on the underlying system.
Security: Isolation through parallel processing; security depends on the underlying system.
Ease of Implementation: Implementation involves parallel processing; maintenance can be straightforward
Ex: Flunetd [17] an open-source data collector that unifies data collection and consumption

G. Custom Controller pattern

The controller watches for changes to objects and acts on those changes to drive the cluster to a desired state. You can also implement custom logic and extend the functionality of the platform.
Scalability: Scalability depends on the control logic implemented in the custom controller.
Flexibility: Flexibility depends on the logic implemented in the custom controller.
Resource Efficiency: Efficiency depends on the logic implemented in the custom controller.
Resilience: Fault tolerance depends on the custom controller's implementation.
Security: Security depends on the logic implemented in the custom controller.
Ease of Implementation: Implementation complexity depends on the custom controller's logic; maintenance involves custom control logic adjustments.
Ex: Prometheus operator [18] that manages the Prometheus monitoring instances.

H. Initializer pattern

Init containers allow the separation of initialization-related tasks from the main application logic [11].
Scalability: Typically used during initialization; scalability might depend on the specific initialization tasks.
Flexibility: Typically used during initialization; adaptability might be limited to this phase.
Resource Efficiency: Typically used during initialization, focusing on resource setup.
Resilience: Typically used during initialization, with potential fault tolerance mechanisms.
Security: Security considerations during initialization tasks.
Ease of Implementation: Typically, straightforward to implement during initialization; maintenance during system initialization.
Ex: Istio service mesh [19] that leverages this concept to initialize and configure sidecar proxies.

IV. DESIGN PATTERN COMPARISON

TABLE I. COMPARING STRENGHT S AND WEAKNESSES OF VARIOUS DESIGN PATTERNS

Design Pattern	Strengths	Weakness
Sidecar Pattern	Modular, Enhanced fault tolerance, isolated functionalities	Increased resource usage, potential complexity with multiple helper containers
Ambassador Pattern	Resource-efficient, centralized traffic control.	Potential single point of failure, complexity in proxy management.
Adapter Pattern	Interface integration , Flexibility	Limited fault tolerance,Limited security
Work Queue Pattern	Task distribution, fault tolerance	Limited security features, might not fit all scenarios
Self-awareness Pattern	Dynamic adaptability, resource optimization	Complexity in dynamic adjustment logic
Scatter/Gather Pattern	Parallel processing, result aggregation	Dependency on the underlying system
Custom Controller Pattern	Custom control logic, adaptability	Dependency on custom logic, potential complexity
Initializer Pattern	Initialization tasks, simplicity	Limited scope to initialization phase, potential security considerations. how to represent this visually

V. CONCLUSION:

In summary, the review of strengths and weaknesses in design patterns for distributed systems within containerized environments provides crucial insights into their practical applications and limitations. It highlights the transformative impact of containerization on software development, guiding architects and developers in designing resilient systems. Overall, this focused exploration contributes valuable knowledge to the dynamic landscape of containerized distributed systems, fostering improved practices in modern software architecture.

REFERENCES:

[1]. Docker. "What is a Container?" [Online]. Available: https://www.docker.com/resources/what-container/
[2]. Google Inc. "Borg, Omega, and Kubernetes." [Online]. Available: https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/45406.pdf
[3]. Microsoft Azure. "Sidecar Pattern." [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar
[4]. Upnxtblog. "8 Container Design Patterns That You Should Know." [Online]. Available: https://www.upnxtblog.com/index.php/2018/12/03/8-container-design-patterns-that-you-should-know/
[5]. A. Koschel, M. Bertram, R. Bischof, K. Schulze, M. Schaaf, and I. Astrova. "A Look at Service Meshes." Jul. 2021. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9555536. DOI: https://doi.org/10.1109/iisa52424.2021.9555536
[6]. O'Reilly Online Learning. "Designing Distributed Systems - Ambassadors." [Online]. Available: https://learning.oreilly.com/library/view/designing-distributed-systems/9781098156343/ch02.html#id11
[7]. Matthew Palmer. "Multi-Container Pod Design Patterns - CKAD Course." [Online]. Available: https://matthewpalmer.net/kubernetes-app-developer/articles/multi-container-pod-design-patterns.html
[8]. TechBeacon. "7 Container Design Patterns You Need to Know." [Online]. Available: https://techbeacon.com/enterprise-it/7-container-design-patterns-you-need-know
[9]. O'Reilly Online Learning. "Kubernetes Patterns - Self Awareness." [Online]. Available: https://learning.oreilly.com/library/view/kubernetes-patterns/9781492050278/ch13.html
[10]. Tania Storm (Medium). "Scatter Gather Message Pattern." [Online]. Available: https://medium.com/@tanstorm/scatter-gather-message-pattern-43d3e6a11198#:~:text=The%20Scatter%2DGather%20pattern%20is,may%20send%20responses%20is%20necessary..
[11]. Ashokan Nagaraj (DEV Community). "Init Container Pattern." May 17, 2022. [Online]. Available: https://dev.to/ashokan/init-container-pattern-5hn1
[12]. Dapr Maintainers. "Dapr - Distributed Application Runtime." [Online]. Available: https://dapr.io/
[13]. HCL-TECH-SOFTWARE (GitHub). "Ambassador: Open source Kubernetes-native API gateway." [Online]. Available: https://github.com/HCL-TECH-SOFTWARE/ambassador
[14]. Docker Documentation. "Docker Compose Overview." [Online]. Available: https://docs.docker.com/compose/
[15]. KubeMQ. "KubeMQ: Kubernetes Message Queue Broker Platform." [Online]. Available: https://kubemq.io/
[16]. Kubernetes.io. "Horizontal Pod Autoscaling." [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[17]. Fluentd Project. "Fluentd | Open Source Data Collector." [Online]. Available: https://www.fluentd.org/
[18]. Prometheus Operator (GitHub). "Prometheus Operator." [Online]. Available: https://github.com/prometheus-operator/prometheus-operator
[19]. Istio. "The Istio Service Mesh." [Online]. Available: https://istio.io/latest/about/service-mesh/