Project 2: Optimizing Instance Choice, Placement, and Communication for Cloud Performance of a Web Application Using Microservices Architecture

CMPT 756 Project Report, Group-8

Ayush Bajirao
Simon Fraser University
ayush_bajirao@sfu.ca

Amrit Paul
Simon Fraser University
amrit_paul@sfu.ca

Aman Saxena
Simon Fraser University
aman_saxena@sfu.ca

Aditya Desai
Simon Fraser University
aditya_desai@sfu.ca

Gauri Mane
Simon Fraser University
gauri_mane@sfu.ca

ABSTRACT

This paper introduces a holistic approach to optimizing instance choice, placement, and communication for web applications using microservices architecture in cloud environments. Our goal is to enhance performance, scalability, and cost efficiency through a multi-faceted optimization process. First, we propose a systematic method for selecting suitable instance types based on each microservice's unique requirements, ensuring cost-effective resource allocation while maintaining performance. We focus on minimizing latency and maximizing resource utilization by considering data locality by **pod placements**, **inter-service communication**, and **load balancing**. These strategies reduce network congestion and improve overall application performance. Additionally, we investigate communication patterns among microservices and explore various communication protocols, service discovery mechanisms, and API gateway designs to streamline inter-service communication.

KEYWORDS

Kubernetes, Microservices, Latency, Tracing, Pod Placement, Communication, Web Application

1 INTRODUCTION

The eShopOnDapr application is designed to be an implementation for building microservices based e-commerce applications using Dapr. It consists of a set of **loosely coupled microservices** that communicate with each other using Dapr messaging and state management capabilities.

The system design of eShopOnDapr is based on the microservices architecture pattern, where each microservice is responsible for a specific business capability. The application is composed of several microservices, including the product catalog, shopping cart, order processing, payment processing, and shipping services. These microservices are designed to be independently deployable and scalable, allowing developers to modify and deploy them without impacting the other microservices in the application.

CMPT 756 Project Report, 2023,
.

To ensure that the microservices in eShopOnDapr can handle a large number of transactions while maintaining performance, the application leverages Dapr's built-in capabilities for **distributed tracing**, **service discovery**, and **load balancing**.

2 SYSTEM DESIGN

In the context of the problem statement, eShopOnDapr's microservices architecture (Figure 1) and use of Dapr can help assess the impact of resource allocation on microservices performance. For example, developers can experiment with different resource allocation configurations, such as CPU and memory, and use Dapr's built-in metrics and tracing capabilities to monitor the performance of each microservice. By analyzing the data, developers can determine how changes to resource allocation affect the performance of each microservice and optimize the configuration accordingly to improve the overall performance of the web application.

Placement: To further *optimize performance* and *reduce latency*, eShopOnDapr uses a combination of local and remote caching strategies. Redis is used as a **distributed cache** to store frequently accessed data, reducing the need for repeated queries to the database. Additionally, Dapr provides an actor model for distributed computing, allowing for efficient communication and coordination between microservices, regardless of their physical location.

Communication: eShopOnDapr utilizes a variety of communication protocols, including HTTP, gRPC, and Event-driven messaging. HTTP is used for **synchronous communication** between services, while gRPC is used for high-performance remote procedure calls. Event-driven messaging is used for **asynchronous communication** between services, allowing for loose coupling and scalability.

The application consists of:-

- A Single Page Application running on Blazor WebAssembly sends user requests to an API gateway.
- The API gateway abstracts the backend core microservices from the frontend client. It's implemented using Envoy, a high performant, open-source service proxy. Envoy routes incoming requests to backend microservices. Most requests are simple CRUD

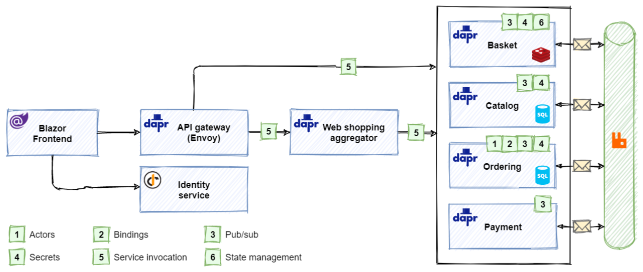

Figure 1: DAPR integration on Microservices Application

operations (for example, get the list of brands from the catalog) and handled by a direct call to a backend microservice.

- Other requests are more logically complex and require multiple microservice calls to work together. For these cases, eShopOn-Dapr implements an aggregator microservice that orchestrates a workflow across those microservices needed to complete the operation.
- The core backend microservices implement the required functionality for an e-Commerce store. Each is self-contained and independent of the others. Following widely accepted domain decomposition patterns, each microservice isolates a specific business capability:
  - The basket service manages the customer's shopping basket experience.
  - The catalog service manages product items available for sale.
  - The identity service manages authentication and identity.
  - The ordering service handles all aspects of placing and managing orders.
  - The payment service transacts the customer's payment.
  - Adhering to best practices, each microservice maintains its own persistent storage. The application doesn't share a single datastore.
- Finally, the event bus wraps the Dapr publish/subscribe components. It enables **asynchronous publish/subscribe messaging** across microservices. Developers can plug in any Dapr-supported message broker component.

3 IMPLEMENTATION

Figure 1 shows the Dapr building blocks (represented as green numbered boxes) that each eShopOnDapr service consumes.

- The API gateway and web shopping aggregator services use the service invocation building block to invoke methods on the backend services.
- The backend services communicate asynchronously using the publish & subscribe building block.
- The basket service uses the state management building block to store the state of the customer's shopping basket.
- eShopOnDapr uses the actor building block implementation. The turn-based access model of actors makes it easy to implement a stateful ordering process with support for order cancellation.

Dapr implements calls between sidecars with gRPC (Figure 2). So even if you're invoking a remote service with HTTP/REST semantics, a part of the transport is implemented using gRPC.

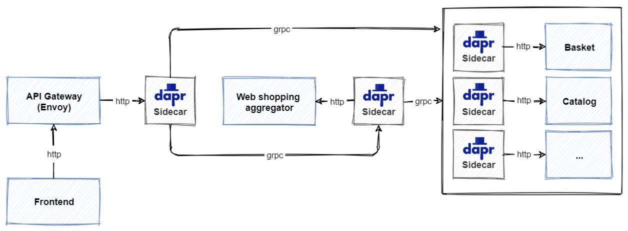

Figure 2: Dapr sidecar communication protocols

3.1 Communication Pattern

3.1.1 Service Aggregator Pattern. The pattern isolates an operation that makes calls to multiple back-end microservices, centralizing its logic into a specialized microservice. The purple checkout aggregator microservice in Figure 3, orchestrates the workflow for the Checkout operation. It includes calls to several back-end microservices in a sequenced order. Data from the workflow is aggregated and returned to the caller. While it still implements direct HTTP calls, the aggregator microservice reduces direct dependencies among back-end microservices.

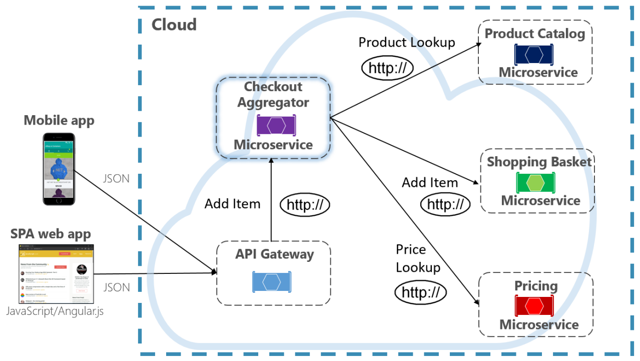

Figure 3: Aggregator microservice

3.1.2 Event Driven Pattern. Another type of message interaction uses one microservice that announces that an action had occurred while the other microservices, if interested, react to the action, or event. This is also known as the event-driven architectural style (Figure 4). For a given state change, a microservice publishes an event to a message broker, making it available to any other interested microservice. The interested microservice is notified by subscribing to the event in the message broker. Publish/Subscribe pattern is implemented for event-based communication.

4 TESTING TOOLS, METHODOLOGY AND RESULTS

In this experiment, we deployed a distributed application on Kubernetes, a widely used container orchestration system. The application was designed to run multiple pods in different regions, namely India, London, and the USA, to study the impact of geographic location on the application's performance and observe the variations

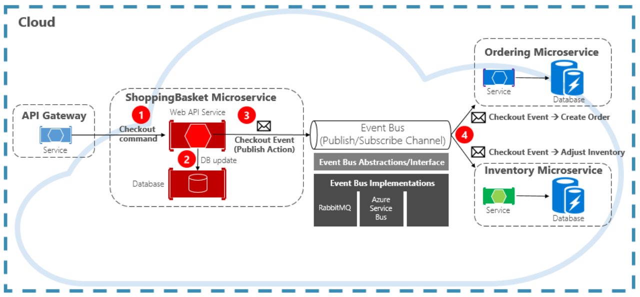
Cloud
API Gateway
Service
ShoppingBasket Microservice
1
Checkout command
Web API Service
2
DB update
Database
3
Checkout Event (Publish Action)
Event Bus (Publish/Subscribe Channel)
Event Bus Abstractions/Interface
Event Bus Implementations
RabbitMQ
Azure Service Bus
4
Checkout Event → Create Order
Checkout Event → Adjust Inventory
Ordering Microservice
Service
Database
Inventory Microservice
Service
Database

Figure 4: Event-Driven messaging

in latency between the regions.

To collect and analyze latency data, we chose Zipkin, a distributed tracing system that is often used in microservices-based architectures. With Zipkin, we collected and visualized data related to the time taken for requests to move across different parts of the distributed system. This helped us identify any bottlenecks or other performance issues caused by differences in latency between the different regions (Figure 5).

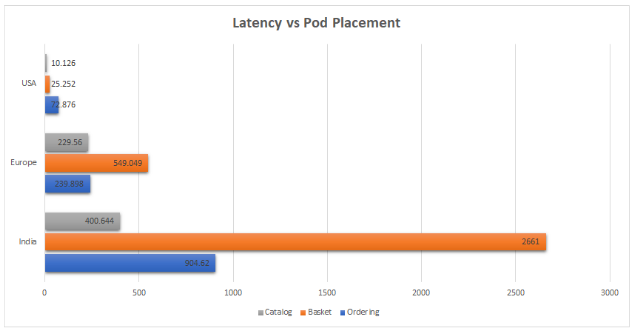

Figure 5: Latency vs Pod Placement across different regions

Using Zipkin to collect and analyze latency data from the distributed application, we were able to gain valuable insights into the performance of the system in different regions. These insights can be used to make informed decisions about how to optimize the application's performance, such as by placing pods in regions with lower latency, implementing caching mechanisms, or using content delivery networks (CDNs) to distribute content more efficiently.

Furthermore, we evaluate the efficiency of invoking a method via gRPC and HTTP, with and without Dapr, by conducting a crude microbenchmark that contrasts 'HTTP' and 'gRPC' protocols (Table 1). The primary focus of this comparison is to measure the performance of the RPC itself, rather than the complexity of the service being invoked. For this reason, a simple echo service is employed to ensure that the evaluation remains focused on the performance of the communication protocols. The repository used for this test can be found at- Github

The echo service is intentionally minimalistic and serves as a suitable means to highlight the differences in performance between gRPC and HTTP communication protocols. With the addition of Dapr, which is a runtime designed to simplify the development and deployment of microservices, the performance comparison reveals how the presence or absence of middleware impacts the efficiency of the RPC invocation. This examination provides valuable insights into the practical implications of choosing between gRPC and HTTP, as well as the role that Dapr plays in facilitating microservices communication.

Payload Testing for different communication protocols				
Test	**Hello world**	**1K text**	**10K text**	**100K text**
Plain gRPC calls	150 ms	150 ms	200 ms	550 ms
Plain HTTP calls	320 ms	355 ms	520 ms	2050 ms
DAPR wrapped gRPC calls	970 ms	980 ms	1110 ms	FAIL
DAPR wrapped HTTP calls	810 ms	820 ms	1110 ms	3160 ms
gRPC SDK calls	760 ms	760 ms	980 ms	3230 ms
HTTP SDK calls	690 ms	685 ms	800 ms	1830 ms

Table 1: HTTP vs gRPC testing for 1000 calls

5 CONCLUSION

We have found the relationship between latency and distance amongst different microservices located across the globe . This is valid for fewer requests per microservice. However, when multiple requests are made per microservice, there is a latency observed.

Interestingly, gRPC SDK uses JSON marshal/unmarshal and is still faster than DAPR wrapped gRPC calls, which is using direct gRPC but via dapr sidecars. This happens because Dapr service invocations for each publisher and subscriber events induce latency due to serialization and deserialization of message packets. For example, when you do service invocation through Dapr to call from app A to app B:

- App A sends the message to sidecar A
- Sidecar A reads the entire message in memory then serialize it
- Sidecar A sends the entire (serialized) message to sidecar B
- Sidecar B waits for the entire message to be received and then deserializes it
- Sidecar B invokes app B
- The exact data flow happens, in reverse, to send the response

The problem here is that sidecar A doesn't do anything until the full message has been read. And same for sidecar B.

The 'wrapped gRPC' call fails with context deadline exceeded when a call takes more than 2msec, which happens with a payload close to 100KB.

6 REFERENCES

[1] Distributed Application Runtime (DAPR)
[2] Google Kubernetes Engine
[3] Zipkin
[4] Jaeger
[5] Google Remote Procedure Calls