

Lab 7

Distributed Software Systems
UNIBO - 2024/25

Anouk Wuyts - 1900124167

Davide De Rosa - 1186536

Dapr (Distributed Application Runtime)

Open-source runtime designed to simplify the development of **microservices** by providing **common building blocks** like service invocation, state management, Pub/Sub messaging, and bindings, enabling developers to build distributed applications more easily.

Dapr employs a **sidecar architecture** where a *Dapr sidecar* runs alongside each microservice instance to provide essential capabilities without modifying the core application code.

Dapr (Distributed Application Runtime)

Some key features of Dapr are:

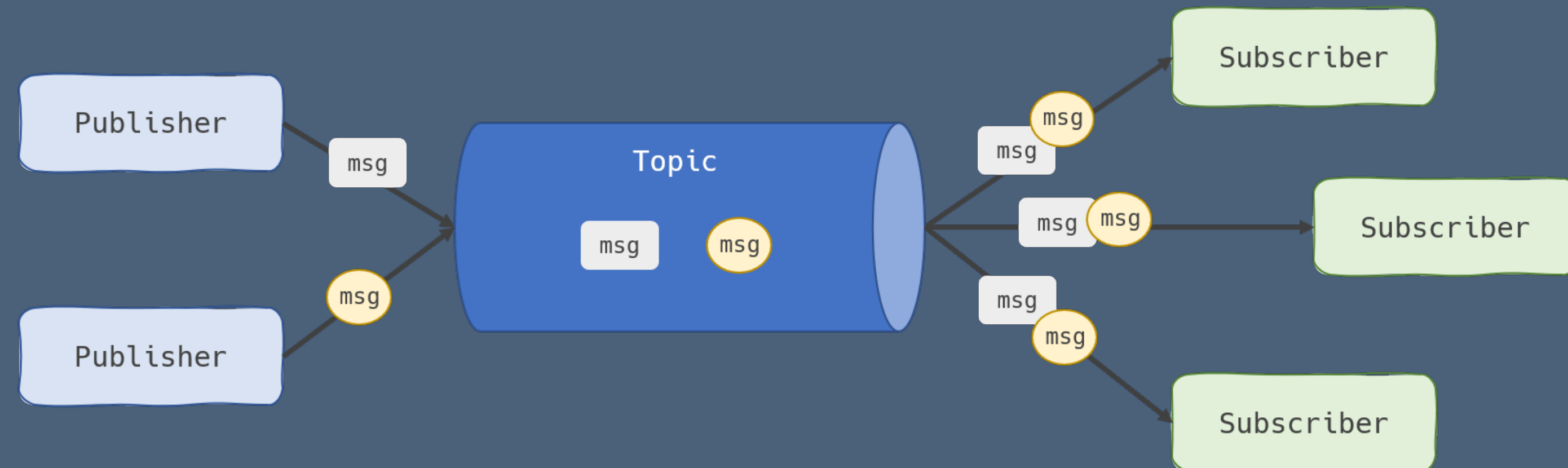
- **Decoupling:** separates functionalities like service invocation, state management, and Pub/Sub messaging from the main application logic
- **Interoperability:** works with multiple programming languages and various message brokers (e.g., Redis, Kafka)
- **Simplified Development:** reduces the complexity of building microservices by providing built-in capabilities
- **Resilience:** sidecars can handle retries and fault tolerance, enhancing service reliability
- **Independent Scaling:** each service can scale independently, while sidecars handle cross-cutting concerns

Pub/Sub Architecture

We choose to implement a **Pub/Sub architecture**, which is a form of an **Event-Driven** architecture.

A *Pub/Sub system* is a messaging pattern where **publishers** send messages to a topic without knowing the subscribers.

Subscribers express interest in one or more topics and receive messages when published, allowing for decoupled communication between components in a system.



Implementation

We started by installing **Dapr** on our *local environment*. The *Docker engine* is needed for Dapr to work correctly. That's one of the few requirements needed for Dapr to run.

Like we just said in the last slide, we chose to implement a *Pub/Sub architecture*, which is one of Dapr's **building blocks**.

We used Python with the *Dapr library* to make everything work seamlessly.

Implementation

Dapr uses **components** which can be swapped pretty easily just by changing a **.yaml file**.

For our implementation we used a Pub/Sub component which uses **Redis** for the data persistency of the system.

We could have used **Kafka** instead of *Redis*, which could have been achieved by only changing the *.yaml file* for the component.

As you can see in the code snippet, everything is setup in a declarative way.

```
apiVersion: dapr.io/v1alpha1
kind: Component
metadata:
  name: orderpubsub
spec:
  type: pubsub.redis
  version: v1
  metadata:
    - name: redisHost
      value: localhost:6379
    - name: redisPassword
      value: ""
```

Redis vs Kafka

Redis: an in-memory data structure store that functions as a key-value database, cache, and message broker. It supports simple Pub/Sub messaging but is primarily used for fast data retrieval and storage.

Kafka: a distributed event streaming platform designed for high-throughput, fault-tolerant messaging. It is optimized for handling large volumes of data in real time, making it suitable for building event-driven architectures and processing streams of records.

Implementation - Publisher

The *publisher* uses the *DaprClient* library to publish some random objects.

The key function is the *client.publish_event()*, which publishes the object to the queue for every subscriber to consume it.

```
with DaprClient() as client:
    for i in range(1, 20):
        order = {
            'orderId': i,
            'customerName': f'Customer {i}',
            'amount': i * 10.0,
            'status': 'Pending'
        }
        try:
            result = client.publish_event(
                pubsub_name='orderpubsub',
                topic_name='orders',
                data=json.dumps(order),
                data_content_type='application/json',
            )
            logging.info('Published data: %s', json.dumps(order))
        except Exception as e:
            logging.error('Error publishing data: %s', str(e))

        time.sleep(1)
```


Implementation - Subscriber

The *subscriber* exposes a “**/dapr/subscribe**” **GET** method, which is used to make Dapr know which topics the subscriber is interested in.

Once a new object is published inside that topic, a **POST** method is called by Dapr to the subscriber to consume the object.

```
@app.route('/dapr/subscribe', methods=['GET'])
def subscribe():
    subscriptions = [{
        'pubsubname': 'orderpubsub',
        'topic': 'orders',
        'route': 'orders'
    }]
    print('Dapr pub/sub is subscribed to: %s' % json.dumps(subscriptions))
    return jsonify(subscriptions)
```

```
@app.route('/orders', methods=['POST'])
def orders_subscriber():
    event = from_http(request.headers, request.get_data())
    order_id = event.data.get('orderId')

    if order_id is None:
        return json.dumps({'success': False, 'error': 'orderId missing'}),
            400, {'ContentType': 'application/json'}

    print('Subscriber received: orderId=%s, customerName=%s, amount=%s, status=%s' % (
        order_id,
        event.data.get('customerName'),
        event.data.get('amount'),
        event.data.get('status'),
    ), flush=True)

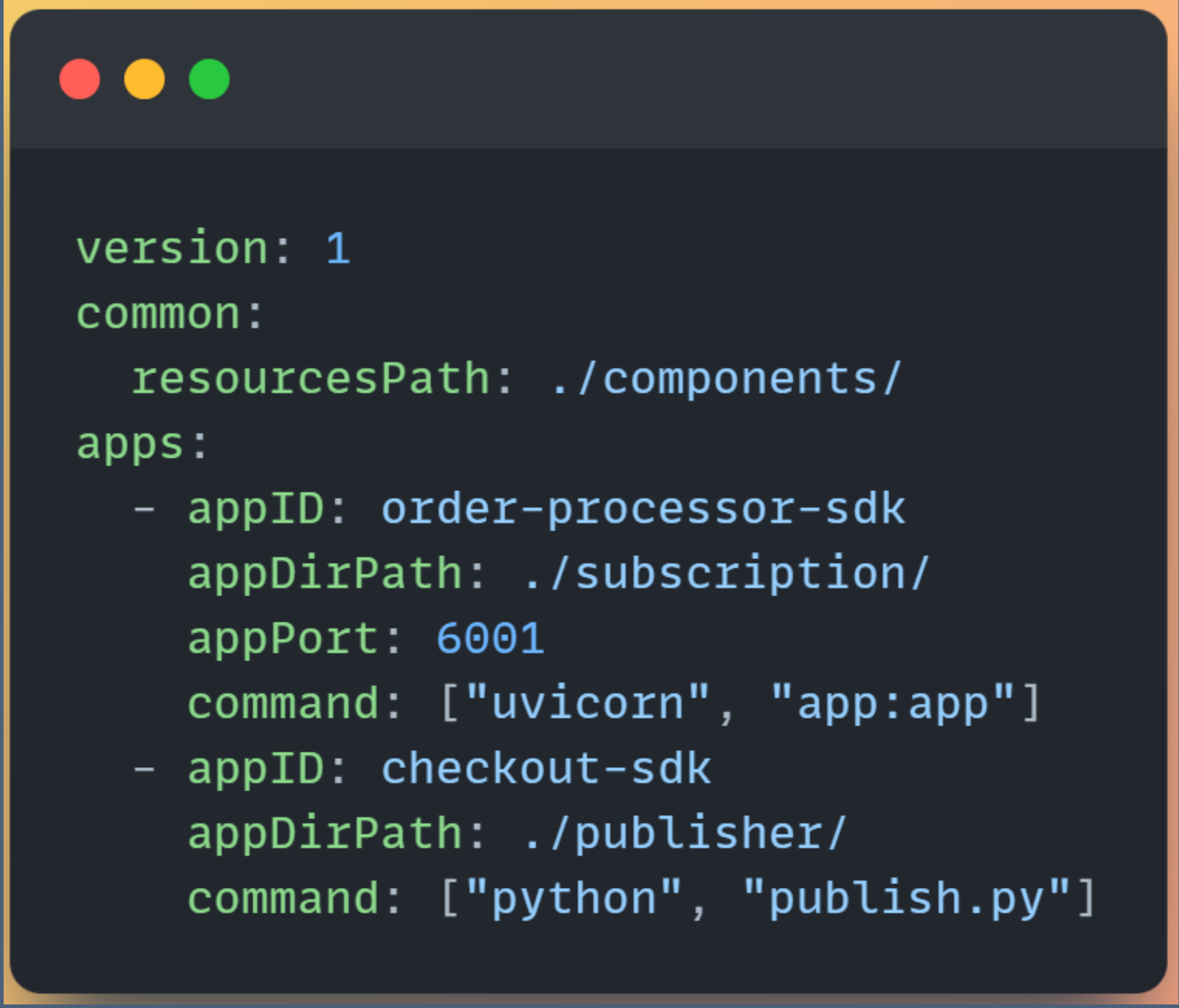
    return json.dumps({'success': True}), 200, {'ContentType': 'application/json'}
```

Usage

To run the Dapr system, we used a *dapr.yaml* file, which can execute different apps by running a single command. You can also run the apps separately.

You also have to specify the components you need for the system to work.

In our case it was only the Pub/Sub component.

A terminal window with a dark background and orange border, showing the contents of a `dapr.yaml` file. The file is a YAML configuration for a Dapr system. It includes a `version` field set to `1`, a `common` section with a `resourcesPath` set to `./components/`, and an `apps` section with two application configurations. The first application is `order-processor-sdk` with `appDirPath` `./subscription/` and `appPort` `6001`, running `uvicorn`. The second application is `checkout-sdk` with `appDirPath` `./publisher/`, running `python` on `publish.py`.

```
version: 1
common:
  resourcesPath: ./components/
apps:
  - appID: order-processor-sdk
    appDirPath: ./subscription/
    appPort: 6001
    command: ["uvicorn", "app:app"]
  - appID: checkout-sdk
    appDirPath: ./publisher/
    command: ["python", "publish.py"]
```


Usage

After executing both the publisher and the subscriber, we can see the Dapr system at work.

Once the publisher publishes something, the subscriber consumes the data received by printing it on the console.

You're up and running! Both Dapr and your app logs will appear here.

```
== APP == INFO:root:Published data: {"orderId": 1, "customerName": "Customer 1", "amount": 10.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 2, "customerName": "Customer 2", "amount": 20.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 3, "customerName": "Customer 3", "amount": 30.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 4, "customerName": "Customer 4", "amount": 40.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 5, "customerName": "Customer 5", "amount": 50.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 6, "customerName": "Customer 6", "amount": 60.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 7, "customerName": "Customer 7", "amount": 70.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 8, "customerName": "Customer 8", "amount": 80.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 9, "customerName": "Customer 9", "amount": 90.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 10, "customerName": "Customer 10", "amount": 100.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 11, "customerName": "Customer 11", "amount": 110.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 12, "customerName": "Customer 12", "amount": 120.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 13, "customerName": "Customer 13", "amount": 130.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 14, "customerName": "Customer 14", "amount": 140.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 15, "customerName": "Customer 15", "amount": 150.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 16, "customerName": "Customer 16", "amount": 160.0, "status": "Pending"}
== APP == INFO:root:Published data: {"orderId": 17, "customerName": "Customer 17", "amount": 170.0, "status": "Pending"}
```

You're up and running! Both Dapr and your app logs will appear here.

```
== APP == Dapr pub/sub is subscribed to: [{"pubsubname": "orderpubsub", "topic": "orders", "route": "orders"}]
== APP == 127.0.0.1 - - [25/Oct/2024 20:47:54] "POST /orders HTTP/1.1" 200 -
== APP == Subscriber received: orderId=1, customerName=Customer 1, amount=10.0, status=Pending
== APP == Subscriber received: orderId=2, customerName=Customer 2, amount=20.0, status=Pending
== APP == 127.0.0.1 - - [25/Oct/2024 20:47:55] "POST /orders HTTP/1.1" 200 -
== APP == Subscriber received: orderId=3, customerName=Customer 3, amount=30.0, status=Pending
== APP == 127.0.0.1 - - [25/Oct/2024 20:47:56] "POST /orders HTTP/1.1" 200 -
== APP == Subscriber received: orderId=4, customerName=Customer 4, amount=40.0, status=Pending
== APP == 127.0.0.1 - - [25/Oct/2024 20:47:57] "POST /orders HTTP/1.1" 200 -
== APP == Subscriber received: orderId=5, customerName=Customer 5, amount=50.0, status=Pending
== APP == 127.0.0.1 - - [25/Oct/2024 20:47:58] "POST /orders HTTP/1.1" 200 -
== APP == Subscriber received: orderId=6, customerName=Customer 6, amount=60.0, status=Pending
== APP == 127.0.0.1 - - [25/Oct/2024 20:47:59] "POST /orders HTTP/1.1" 200 -
== APP == Subscriber received: orderId=7, customerName=Customer 7, amount=70.0, status=Pending
== APP == 127.0.0.1 - - [25/Oct/2024 20:48:00] "POST /orders HTTP/1.1" 200 -
== APP == Subscriber received: orderId=8, customerName=Customer 8, amount=80.0, status=Pending
== APP == 127.0.0.1 - - [25/Oct/2024 20:48:01] "POST /orders HTTP/1.1" 200 -
== APP == Subscriber received: orderId=9, customerName=Customer 9, amount=90.0, status=Pending
== APP == 127.0.0.1 - - [25/Oct/2024 20:48:02] "POST /orders HTTP/1.1" 200 -
== APP == Subscriber received: orderId=10, customerName=Customer 10, amount=100.0, status=Pending
== APP == 127.0.0.1 - - [25/Oct/2024 20:48:03] "POST /orders HTTP/1.1" 200 -
```

System Scalability

Horizontal Scaling: Dapr supports horizontal scaling of services. You can run multiple instances of your application, and Dapr will manage the routing of messages effectively

Dynamic Discovery: with Dapr's service discovery, you can add or remove service instances without affecting the overall application

Multi-language Support: Dapr's ability to work with multiple programming languages makes it easier to scale teams and services across different tech stacks

System Performance

Sidecar Overhead: the Dapr sidecar introduces slight additional latency, as it processes requests between services and the broker. However, this overhead is generally minimal and well-optimized

Latency: Dapr aims for low-latency communication, and in local or optimized environments, it can achieve near-native speeds. However, network complexity in distributed setups may introduce minor latency increases

High Throughput: Dapr supports high message throughput by offloading work to efficient brokers (e.g., Redis, Kafka), allowing it to handle substantial traffic

Resilience: built-in retry and fault tolerance help ensure reliable message delivery, even under high load

Strengths of Dapr for Pub/Sub

Abstraction of Messaging Patterns: Dapr abstracts the complexity of various messaging patterns, making it easy to implement Pub/Sub without deep knowledge of the underlying message broker technologies

Interoperability: Dapr can integrate with various messaging systems (e.g., Redis, Kafka, Azure Service Bus), allowing developers to choose the best tool for their needs without being locked into a single vendor

Decoupling of Services: by using a Pub/Sub model, Dapr allows services to be decoupled, which enhances maintainability and facilitates the independent development and deployment of microservices

Resilience: Dapr provides built-in retries and fault tolerance, which enhances the reliability of message delivery, especially in distributed systems

State Management: Dapr supports state management, enabling easy tracking of message states across services, which is useful in long-running processes

Limitations of Dapr for Pub/Sub

Overhead of Sidecar Architecture: Dapr operates as a sidecar, which can introduce additional latency and resource consumption. This might be a concern in scenarios with extremely tight performance requirements

Complexity of Distributed Systems: while Dapr provides abstractions, the underlying complexity of distributed systems still exists. Developers need to consider network partitions, eventual consistency, and other challenges inherent in distributed architectures

Limited Control over Message Broker: Dapr manages the connection and messaging details, which may limit the ability to customize certain behaviors of the underlying message broker

Community and Ecosystem: as a relatively newer project, Dapr may have a smaller community and fewer third-party resources or plugins compared to more established technologies like Kafka or RabbitMQ

Conclusions

Dapr offers a robust framework for building microservices with Pub/Sub capabilities, providing significant advantages in terms of abstraction, interoperability, and resilience.

However, its sidecar architecture and the complexities of distributed systems can introduce challenges that developers need to navigate.

Careful consideration of performance metrics and architectural needs is essential when evaluating Dapr for specific use cases.

Thank you!