Interservice Communication

Unlike a monolith architecture, a microservice architecture involves a great number of small services interacting with each other’s in order to complete a single task. This raises several challenges that are to be considered.

# Challenges

## Resilience

Considering that there can be many services interacting with each other’s to complete a single task and each of these services can exist in dozens or more instances, the possibility of failure can be significant. The reason can be both hardware or software: VM reboot, crashes, network saturation, etc. The following will provide a brief overview of potential solution.

### [Retry](https://docs.microsoft.com/en-us/azure/architecture/patterns/retry)

In the case of transient fault (i.e. an error that fixes itself), a simple retry mechanism might suffice. However, we have to make sure that the operation is idempotent such that unintended side effects are avoided.

### [Circuit Breaker](https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker)

In the case where the call fails repeatedly, it is advisable not to continuously float the backend with additional calls since these requests might consume valuable resources and cause additional failures. In this case, the caller would immediately receive an error and handle it gracefully. The circuit breaker would allow a certain number of calls to the service through in order to detect resolution.

## Load Balancing

Since there can be multiple instances of the same service, we would need to select the best available candidate to fulfill the call based on predefined metrics. A service mesh can be helpful in this scenario.

## Distributed Tracing

Considering that a single transaction can span multiple services, we would need to combine the logs and metrics of many services in order to obtain a complete image to monitor the overall performance and health of the system.

## Service Versioning

Since each service can be deployed independently (and they should be, otherwise, it could be a sign of high coupling) and it is critical to avoid breaking other services or clients, we might be required to run multiple versions of a service side-by-side and route requests to a particular version.

## Encryption

For security purposes, traffic between services should be encrypted. Often these services will be deployed on third party cloud provider network and they could already be [breached](https://www.troyhunt.com/your-corporate-network-is-already/).

# Messaging Patterns

In microservices, there are two basic messaging patterns used to communicate between services.

## Synchronous

This is the traditional request-response pattern where a caller issues a request and waits for a response from the receiver.

## Asynchronous

In this pattern, the caller sends a message without waiting for a response. Even if it is less trivial than the request-response pattern, it does have some advantages.

* Reduced coupling: The sender does not need to know about the consumer. A layer of abstraction can exist between them.
* Multiple subscribers: In a pub/sub model, multiple consumers can receive the same events.
* Failure isolation: If a consumer fails, the sender can still send the message and have it process with the consumer recovers. In a microservice architecture where each service has its own lifecycle so we must be resilient to intermittent downtime.
* Responsiveness: A service can issue a response immediately if it doesn’t need to wait for the services it depends upon.
* Load leveling: Queue can provide the means for services to process the data at its own rate
* Workflows: Queues can manage workflow

Despite these advantages, there are some serious drawbacks.

* Coupling with messaging infrastructure: There could be a tight coupling between the services in the underlying infrastructure supporting the communication. This could prevent us from switching to another infrastructure.
* Latency: If the message queues fill up, this can add significant latency.
* Cost: When confronted with high volume of throughputs, the cost of the messaging infrastructure is significant.
* Complexity: Asynchronous messaging is far less trivial than its counterpart. For example, we need to handle the possibility of duplicate message (via de-duplication or idempotent operations).
* Throughput: If there is a queue present, the queue can become a bottleneck for the system.

Consequently, it is a matter of analyzing the interaction between the service, look at the strengths and weaknesses of either pattern. For example, if the caller absolutely needs a timely response from the receiver, we should consider a synchronous pattern.

# Service Mesh

A service mesh is made to handle service-to-service communication in a microservice style architecture. It is designed to address many of the concerns outlined above and is similar to the [ambassador pattern](https://docs.microsoft.com/en-us/azure/architecture/patterns/ambassador), which is a helper that sends network request for the application but handles all the quirks for service to service communication such as retry, monitoring, security, etc. These concerns are therefore moved out of the individual service and into a dedicated layer that can be reused by other services.

# References

[Interservice Communication](https://docs.microsoft.com/en-us/azure/architecture/microservices/interservice-communication)