FIT3170 Technical Report
Covid 9-Team
Microservice Communication

Kenny Huynh - 27807169
Priyanka Saxena - 28417321
Quinn Roberts - 28800737
Joshua De Luca - 28758323
Chee Chin Chan - 28730151

Table of Content

Glossary

VM: A virtual machine is an emulation of a computer system: it does not have physical features, except being run on a physical machine.

EC2: EC2 stands for Amazon's Elastic Cloud Compute service, which provides cloud-based instances or servers for customers to use. These are much like virtual machines.

Dead letter queue: It is an implementation within a message queue system to store messages that meet one or more of these criteria:

1. Message that is sent to a queue that does not exist.
2. Queue length limit exceeded.
3. Message length limit exceeded.
4. Message is rejected by another queue exchange.
5. Message reaches a threshold read counter number, because it is not consumed.

Test Double: A test double is a piece of code used to simulate another component to replace the dependency while testing. This allows a component to be isolated from other components and tested independently.

High/Low Level Module: The highest level module is the interface of the software and all the modules it can call are the next level lower. This forms a tree structure and the lowest level modules are the leaves at the bottom of the tree that do not call any other modules.

Introduction

This technical report covers different topics under microservice communications, evaluating and recommending certain methodologies, patterns, strategies, or technologies for potential use in the Student Project Management Dashboard project. These recommendations are intended to explain the problem being faced, detail solutions to it and recommend a way to solve this. These recommendations are chosen based on a number of factors and represent the best option according to the authors.

Firstly, the methods, types and patterns utilized to facilitate microservice communications is discussed. This concerns the primary method that services shall use to share information with each other and issue requests. Next, are methods to handle and mitigate microservice errors and faults. These explain ways to isolate services and handle errors in reasonable manners. Thirdly, ways to perform testing of the services and the program without having access to all the composite microservices. Lastly, ways to monitor our microservice system in order to understand the program as a whole are considered.

Each of these sections explains the main ways to solve the problems and the technologies that are used to implement these patterns. The recommendations are intended to balance different factors and as such depending upon the factors

How Microservices Communicate

Synchronous vs Asynchronous communications

In a microservices architecture, it is important to design each individual microservice in a suitable way to allow for reliable and efficient interservice communications. There are two types of interservice communications to consider: synchronous and asynchronous communications.

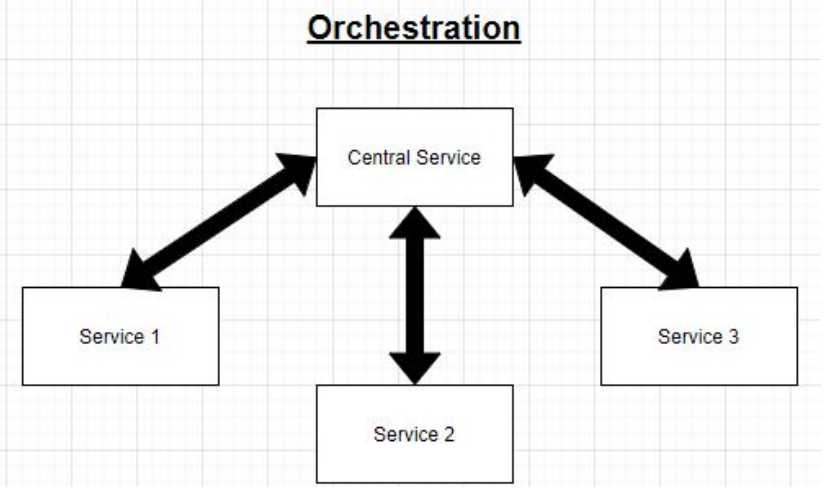
Synchronous communications describe the scenario when a service interacts with another service and its request requires a response. If the service fails to respond within a timeout period, or does not respond at all, the calling service will also fail as it has not received its required response. This can result in an upstream chain of call failures. Another issue with synchronous calls is that while the request is being made, the service is blocked until a response is returned, causing the service to become slow and unresponsive. An advantage of synchronous communications is that the service will receive an acknowledgement from the responding service, indicating that the communication was successful.

Asynchronous communications describe the scenario much like in synchronous communications, except that it does not wait for a response from the called service. By doing this, the calling service is not blocked and therefore is not dependent on any called services: if a called service fails to respond, the calling service will not fail and will be able to continue operating. Since calling and called services are no longer highly dependent, the calling service is able to communicate asynchronously with many other services, increasing its flexibility and efficiency.

In some cases, where the availability of services is required, synchronous communications should be used to ensure that the action or process occurs in a reasonable time after commencement. However, for this project, asynchronous communication is preferred to enhance user experience, as well as to facilitate our many microservices.

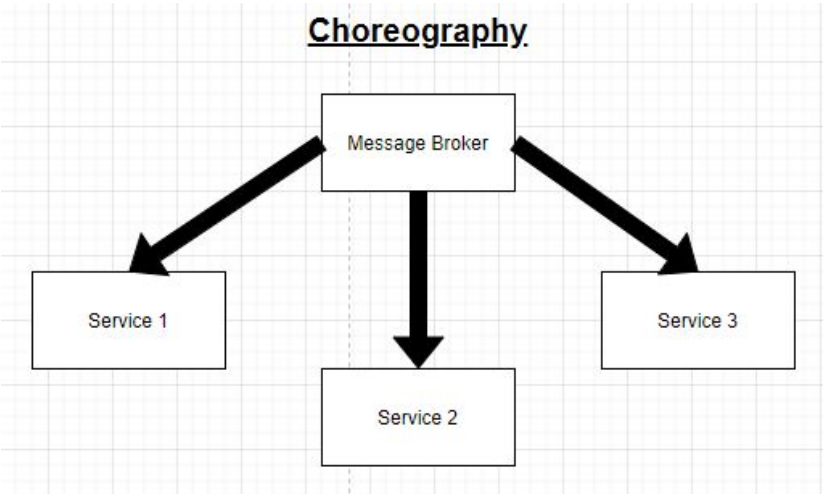
Orchestration vs Choreography

A microservices architecture defines a system of services that are modular, lightweight, and reusable that collaborate and produce an overarching service that is flexible and scalable. When deciding upon how services or microservices communicate and interact with one another, two design paradigms are to be considered: orchestration and choreography.



Orchestration is when there is a central service that contains most of the logic, and where the interservice communication is conducted. The central service will communicate with each service in order, ensuring a response that the next service

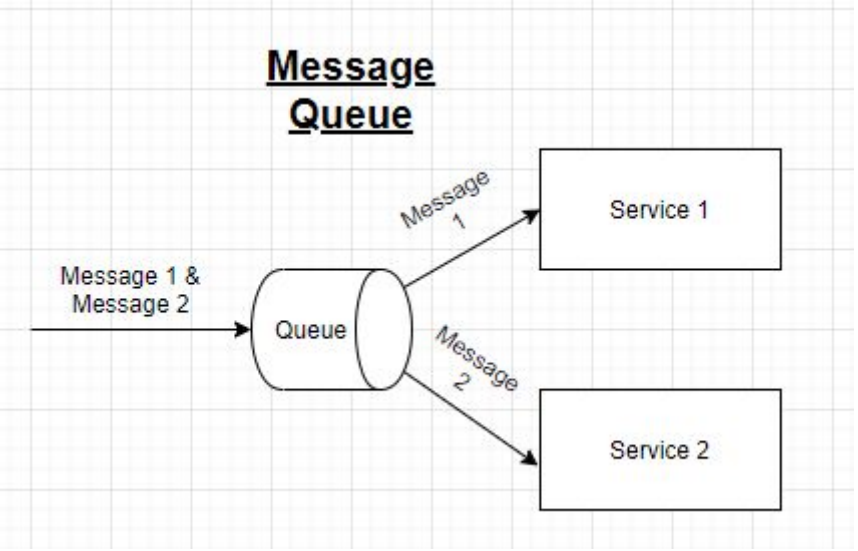
requires is received. Therefore, there is high coupling between services as each service needs to interact with the central service.



Choreography defines a highly modular system with independent services that each contain their own logic to function. Each service only needs to communicate with services that it requires and does not need to communicate with an intermediary central service like in orchestration, resulting in a more loosely coupled system. In order to further reduce coupling, a message-based communication approach can be used where a message broker is used to distribute messages to the required services.

Types of message-based communication patterns

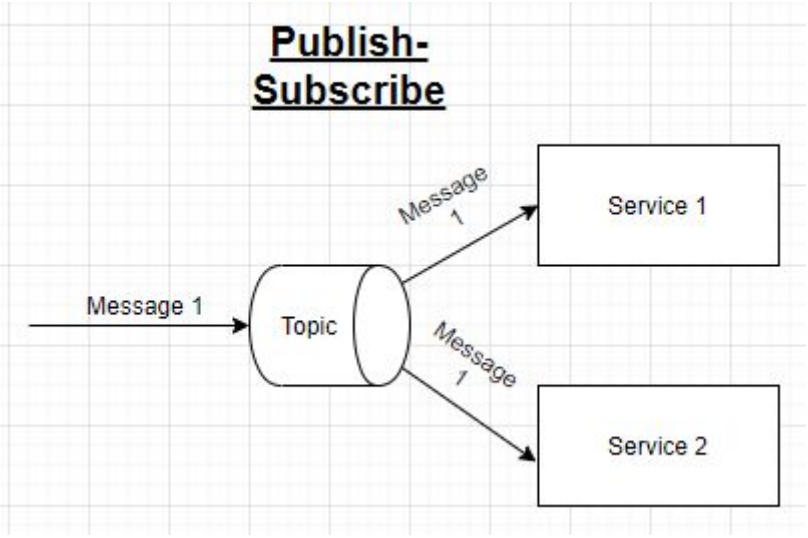
To support the notion of asynchronous communications and choreography, two message-based communication patterns are discussed: the message queue pattern and the publish-subscribe pattern. These patterns help to decouple microservices within a system, improving performance, scalability, and reliability.



The message queue pattern involves a queue in which messages are published and pushed. From this queue, a single consumer can retrieve a particular message which will be removed from the queue after consumption. Therefore, multiple consumers are not able to retrieve the same message. This represents a one-to-one relationship between the producer and the consumer. Being able to store messages in a queue allows messages to persist beyond initial communications as is in synchronous communications and is useful for managing traffic. An advantage of message queues is its ability to distribute processing across multiple services to handle specific loads. However, this functionality is most likely not required in the project.

The publish-subscribe pattern describes a scenario much like the message queue pattern, but instead of a queue, there is a channel which one or many subscribers will subscribe to. These subscribers are subscribing to any messages sent from the publisher. Therefore, a message can be received by multiple subscribers, representing a one-many relationship between the publisher and the subscribers. This pattern is useful when multiple actions need to occur for each event or message published.

As the project develops, the usage of publish-subscribe communication between microservices may be required for multiple subscribers to receive a single published message, and consequently have an



intended set of microservices consume the message as opposed to a single microservice. However, based on the scale of the project, it may be more feasible to have select cases of duplicated published messages, that are passed onto a set of queues for consumption.

Tools to support message-based communications

To facilitate asynchronous communications between services, there are many message broker technologies to consider:

Kafka

Kafka is the most well-known open source distributed publish-subscribe based streaming platform that has powerful message processing capabilities. It is primarily used to perform publish-subscribe functionalities to streams of records, store these streams of records in a fault tolerant manner, as well as process them in real time. Kafka's pull based methodology allows users to consume large batches of messages efficiently and at a higher throughput.

Kafka can be managed and run on different cloud platforms like Amazon Web Services (AWS) as Amazon Managed Streaming for Apache Kafka (Amazon MSK) or Confluent which provides different hosting platforms under two tiers of pricing. [1] [2]

[1] "*Amazon MSK Pricing*". AWS. https://aws.amazon.com/msk/pricing/ (accessed 21 May 2020).

Amazon MSK offers their service at a price which includes:

- Broker instance pricing $0.0578 - $0.525 per hour depending on appropriate instance size
- Broker storage pricing of $0.12 per gb per month
- Data transfer fees based on AWS EC2 instance data transfer fees

Confluent offers their service at two pricing levels:

- Basic
- Standard

Which both include:

- A per hour hosting cost
- Data input and output cost
- Data storage costs

RabbitMQ

RabbitMQ is a message brokering service that implements both a message queue and a publish-subscribe system. However, instead of sending a message to a single queue, the message published by the producer is sent to an exchange. Based on the type of exchange, the message can be sent to a single queue or multiple queues.

RabbitMQ can be serviced at $500 per core per year as VMware RabbitMQ on VMware Tanzu or at a rate of $0.012 - $1 per hour depending on the appropriate instance size as Bitnami RabbitMQ.[3] [4]

ActiveMQ

ActiveMQ is a Java-based open source project which makes use of the Java Message Service API and implements both a message queue and a publish-subscribe system. Published messages are either sent to a queue for singular consumption or to a topic for dispersion to consumers. Topics in this system operate as usual. However, the queue broker acts as a load balancer, providing a round robin message routing functionality for each of its consumers.

ActiveMQ can be managed and run on AWS as Amazon MQ with the following pricing [5]:

- Broker instance rate from $0.038 - $0.72 per hour depending on appropriate instance size
- Storage type pricing at a unit of price per GB per month which will incur greater costs at scale

---

[2] "*Apache Kafka® Re-engineered for the Cloud*". Confluent. https://www.confluent.io/confluent-cloud/ (accessed 21 May 2020).
[3] "*VMware RabbitMQ*". VMware Tanzu. https://tanzu.vmware.com/rabbitmq (accessed 21 May 2020).
[4] "*RabbitMQ Certified by Bitnami*". AWS Marketplace. https://aws.amazon.com/marketplace/pp/Bitnami-RabbitMQ-Certified-by-Bitnami/B01M1DZVS6 (accessed 21 May 2020).
[5] "*Amazon MQ Pricing*". AWS. https://aws.amazon.com/amazon-mq/pricing/ (accessed 21 May 2020).

- Data transfer fees across availability zones in the same region

Amazon SQS & Amazon SNS

Amazon Simple Queue Service (Amazon SQS) is an AWS, cloud-based message queueing system with quick and straightforward setup. As the name states, Amazon SQS implements the message queueing functionality. Furthermore, it offers a feature of in-flight messages, which means that the message is pulled off the queue but is never deleted until a command is sent to directly delete the message, allowing for message persistence.

Since it is a Software-as-a-service (SaaS), its overall cost is lower as there are no infrastructure costs. The pricing is based on data ingestion and delivery, and number of requests.[6]

Amazon Simple Notification Service (Amazon SNS) is much like Amazon SQS, except that it implements the publish-subscribe communication type. However, it also acts as a serverless topic, only being available whenever it is required which can result in subscribers unable to receive the published message from the topic broker. This problem can be circumvented by integrating Amazon SNS with Amazon SQS for example, to allow storage and persistence of the message.

Much like Amazon SQS, Amazon SNS is also a SaaS and thus its overall cost is lower as well. The pricing is based on a pay-as-you-go basis consisting of the number of notifications published and delivered as well as any additional API calls for managing topics and subscriptions.[7]

Google Cloud Pub/Sub

Google Cloud Pub/Sub is a Google Cloud Platform (GCP) fully managed real-time message service that allows the communication of messages between microservices. Google Cloud Pub/Sub publishes messages to the Cloud Pub/Sub service which are then retained on behalf of subscriptions, incurring a cost for the ability to re-process these messages. The service then either pushes messages to subscribers, or has its messages pulled from by subscribers, and thereafter acknowledged. Being a part of GCP allows for simple communications and interactions with other services and resources on the same platform.

The pricing for Google's Cloud Pub/Sub is based on message ingestion and delivery as well as seek-related message storage which means retaining acknowledged messages and storing a snapshot of a subscription's unacknowledged messages.[8]

Azure Service Bus

Azure Service Bus is a Microsoft Azure based, fully managed enterprise cloud messaging broker service that utilizes both message queue and publish-subscribe functionality. It boasts many additional features such as dead-letter queues, scheduled delivery, batching, duplicate detection and more.

---

[6] "*Amazon SQS Pricing*". AWS. https://aws.amazon.com/sqs/pricing/ (accessed 21 May 2020).
[7] "*Amazon SNS Pricing*". AWS. https://aws.amazon.com/sns/pricing/ (accessed 21 May 2020).
[8] "*Pricing*". Google Cloud. https://cloud.google.com/pubsub/pricing (accessed 21 May 2020).

Utilizing Azure Service bus requires an Azure subscription, and is priced at three different tiers[9]:

- Basic
    - The basic tier has a low price point but does not provide access to the use of topics. Furthermore, only 100 brokered connections can be used, which may become a problem in the future if the application is extended.
- Standard
    - The standard tier includes basic tier benefits primarily topics, as well as sessions, de-duplication, and an increased message size to name a few. Hosting costs are approximately $0.45 per day.
- Premium
    - The premium tier includes standard tier benefits, as well as providing a premium form of data safety and persistence with resource isolation and Geo-Disaster Recovery. The hosting costs are much greater at this tier, at approximately $30.58 per day.

Comparisons

Regarding the Student Project Management Dashboard project, the throughput and message processing on Kafka is excessive whereas RabbitMQ is more suitable. Additionally, due to the presumed low intensity and density of data transfer within the project, RabbitMQ better fits the requirements. It is also important to note that Kafka does not provide message acknowledgments whereas each of RabbitMQ, ActiveMQ, Amazon SQS & SNS, Google Cloud Pub/Sub and Azure Service Bus provide their own form of message acknowledgment. [10] [11] [12]

RabbitMQ and ActiveMQ provide the required functionalities but ActiveMQ comes at a much steeper price point under AWS as Amazon MQ; RabbitMQ is the favoured option.

Amazon SQS and Amazon SNS both provide cheaper and more reliable services, as well as being based on a pay-as-you-go basis. If required, initial usage of Amazon SQS for message queue functionality can be enveloped with Amazon SNS to incorporate the publish-subscribe paradigm, providing flexibility for the project. Utilizing these Amazon services also provides the benefit of a single platform, as other planned tools for the project are from the AWS suite.

Google Cloud Pub/Sub employs a mix of the message queue and publish-subscribe communication model which fits the project's requirements. However, the extensiveness of the service as well as its pricing does not place it as a prime candidate.

---

[9] "*Service Bus Pricing*". Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/service-bus/ (accessed 21 May 2020).
[10] A. Mohammed. "*Asynchronous communication in Microservices*". Medium. https://medium.com/@aamermail/asynchronous-communication-in-microservices-14d301b9016 (accessed 20 May 2020).
[11] "*Acknowledging a Message in Active MQ*". pmicheals. https://www.pmichaels.net/2016/10/13/acknowledging-a-message-in-active-mq/ (accessed 21 May 2020).
[12] "*Message transfers, locks, and settlement*". Microsoft Build. https://docs.microsoft.com/en-us/azure/service-bus-messaging/message-transfers-locks-settlement (accessed 21 May 2020).

Although Microsoft's Azure Service Bus provides the functionality necessary for reliable microservice communications, this is only relevant at the Standard tier and above. At these tiers, the time-based costs may be a problem. Furthermore, the increased messaging size, resource isolation, and Geo-Disaster Recovery features of the Azure Service Bus's premium tier is most likely not necessary for this project.

Technology	Message queue functionality	Publish-subscribe functionality	Cost	Fits requirements (Y/N)
Kafka	Consumer Groups	Topics	Moderate with many cost sources	Y
RabbitMQ	Work Queues	Publish/Subscribe	Low	Y
Active MQ	Message Groups	Broadcast	Moderate with many cost sources	Y
Amazon SQS & SNS	Amazon SQS	Amazon SNS	Pay-as-you-go	Y
Google Cloud Pub/Sub	Pull	Publisher-Subscriber	Moderate with few cost sources	Y
Azure Service Bus	Queues	Topics and Subscriptions	High	Y

Table 1 detailing and comparing different message broker technologies

Recommendations

Based on careful consideration of microservice communication design choice, analysis and comparison of message brokerage technologies, Amazon Simple Queue Service (SQS), Amazon Simple Notification Service (SNS) or RabbitMQ should be used to facilitate choreographed microservice communications in the Student Project Management Dashboard.

Error and Failure handling

The core principle of the microservice style is to have each conceptual unit as its own service. This means that the program as a whole consists of multiple individual services, which are all communicating together. These services should be isolated from each other, and only communicate through clearly defined api channels. One of the potential problems that arises from this is how to handle errors in the program. This applies to both errors inside of a single microservice, as well as larger errors such as a whole service being unusable.

Exception in a service

Firstly, let's consider the situation of an internal error in a microservice. If a request to a microservice results in an exception occurring, then this has a chance of propagating into the service that initiated the request. If this happens then there is a chance for an error to propagate through the program, and thus cause a large scale fault or failure.

There are two main approaches to this, which can be split into internal and external approaches. The internal approach is on the level of each individual service, handling the errors inside of the service wholly and not letting them reach the event channel. This is usually achieved through validation of input, catching errors in code and similar common approaches. None of these are unique to microservices, and thus are common in any large program, regardless of architecture. The other approach is to handle the errors at the event channel. This is an external approach because it is handling the errors at the request level, outside of each individual microservice. This method primarily works through the api contract that each service adheres to in it’s communication. These two approaches mostly differ in where they are applied, although the internal approach is also much more varied in its implementation. This is because the internal approach will be dependent upon the design of that particular service, which may differ from other services in the program due to the microservice architecture in use. The external approach however, is much more uniform in it’s application as it is applied to the api contract of all microservices. The external approach is also a more abstract method, as it primarily manifests through the api contract which is then implemented by services.

Internal Approach

The internal approach has two parts, handling an error created within this service itself, and validating responses from an external service. This approach assumes that no other service will handle errors, and that it shouldn’t let its errors leave the service. What this means in practice, is aggressive validation of results from outgoing requests, and never providing undefined behaviour to incoming requests. The specific methods that are used to do these are dependent on the internal architecture of the service. This is because the microservice architecture doesn’t impose any restrictions or requirements on the design of the servicesd, just the design of the overall program. This approach has the benefit of being fairly robust, as an error should be checked and solved in two places. As such, a malformed response would need to pass two levels of checking before it was treated as normal. This is also the downside of this approach as

it has severe redundancy, as checks are performed twice. In practice, only one of these checks is likely to be needed for almost all cases.

External Approach

The external approach in contrast is a more abstract method of handling errors. It uses the api contract to handle errors, by including those errors into it. This means that the api contract that one service produces, also includes details and specifications for what it will do in the case of the service erroring, or the inputs being invalid. These are then implemented with the rest of the event channel for the services. As an example, for a program that uses a RESTful communication method, the HTTP error codes could be used to signify internal faults, invalid input and more. This has the advantage of allowing errors to be consistent between services, as the error response would be the same for all services. It also ensures that the services handle errors in an expected and predictable manner, which makes bug hunting much easier. Unfortunately however it is not foolproof as it doesn't have any capability to handle errors in the implementation of this communication itself. This means that although the correct response in the case of an invalid input is clearly defined, the service may not actually provide it due to a bug or incorrect implementation.

Recommendation

Both of these approaches are very simple approaches, and are fairly logical as the problem of handling errors is not unique to microservices. As an example, the external approach is the same method that an application's frontend might use when communicating with a remote server. In this way there is no real technology that implements them as they are more design solutions to the problem. Additionally, this problem is actually fairly minor and the cases where an error is likely to cascade between different services is rare at best. In practice, the best solution is by far to implement both of these. Both of the approaches can occur simultaneously, and are, in part, co-dependent. The external approach should be the primary method used, as it allows for behaviour to be predictable which reduces further bugs and complexity. Complementing it however, an internal approach of not blindly accepting a request is also advisable, to cover the situations where a service is not following the api contract.

Microservice communication failure

Aside from errors in attempting to fulfill a request, a service could become partially or completely unresponsive. This means that a different server that is requesting some action from it will experience significant delays, if it gets a reply at all. These are more serious than the prior section as these can be caused by a breakdown in the communication or the service. As an example, if the network is severely degraded, then communications using a RESTful api are going to talk significantly longer, or not be transmitted at all. Similarly, if a service goes down or is unresponsive for whatever reason, then other parts of the program will not be able to communicate it. This could be due to a critical bug in the latest version which is causing it to be unable to start up, or could be planned maintenance or deployment of a new version. All of these situations require a method to describe how to act in these situations and provide a method to respond. There are a couple of main methods of handling this[13] with three prominent

[13] "*Designing a Microservices Architecture for Failure*" https://blog.risingstack.com/designing-microservices-architecture-for-failure/ (accessed 22 May 2020)

choices, the Bulkhead, Circuit Breaker[14] and Retry patterns. These patterns function easily well with network issues and overloading as they do with internal errors and complete service outages.

Bulkhead Pattern[15]

The Bulkhead pattern is named for a similar principle in ship design. Here, a bulkhead splits off a section of the ship from the others by a physical wall, segmenting it into compartments. If one of those compartments fails, the flooding will be limited to that compartment and won't sink the whole ship. The bulkhead pattern functions by splitting up the services in the same manner, focusing on isolating them from each other as much as possible. This gives rise to two main principles, limiting what is shared, and being asynchronous.The former principal, limits the effects that one service can have others through the infrastructure and the latter limits the effects of requests on other services. For example, by applying the bulkhead pattern, each service should have its own database, be on it's own server instance and use independent load balancers.

Circuit Breaker[16]

Continuing the trend of patterns drawn from other disciplines, the Circuit Breaker functions much like those in electrical wiring. It's primary purpose is to prevent a service from repeating a request that is likely to continue to fail, by directly returning an exception to the request. The circuit breaker has three states that, mostly, mimic that of the electrical inspiration. When the breaker is *open*, all requests made to the service immediately fail, returning an appropriate error. In contrast, when it is *closed* all requests are routed to the service as normal. However, as requests are routed the breaker also keeps track of the errors being produced by the service. If a threshold is breached, then the breaker switches to closed, aiming to protect the service which is producing abnormal numbers of errors. In some implementations there is also a *half-open* state, which the breaker shifts to after some time being open. In this state, it will selectively allow a few requests through to the service, and use those to gauge the status of the service. If any of those requests fail, then the breaker will trip back into open, but if they all pass it will revert back to being closed.

Retry[17]

The final common pattern, retry, can be thought of as similar to the circuit breaker pattern but more suited towards shorter lived errors. When a service receives an error when it sends a request to another, it's behaviour is determined by the type of error received. When the error directly indicates that the service is going to continue to error for a while, e.g. if it has an open circuit breaker; or from an authentication error, then the request is just cancelled. The issuing service should then handle this cancellation. If however, the error is rare, unusual or otherwise indicates that there are unusual circumstances, this can mean that the failure was caused by some unique malfunction. For instance, a packet becoming malformed in

[14] "*Microservice Architecture Pattern: Circuit Breaker*" https://microservices.io/patterns/reliability/circuit-breaker.html. (Accessed 22 May 2020)
[15] "*Bulkhead pattern - Cloud Design Patterns*" https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead (accessed 23 May 2020)
[16] "*Circuit Breaker pattern - Cloud Design Patterns*" https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker (accessed 23 May 2020)
[17] "*Retry pattern - Cloud Design Patterns*" https://docs.microsoft.com/en-us/azure/architecture/patterns/retry (accessed 23 May 2020)

transit. This means that a subsequent request is not likely to fail, and thus a second identical request is sent immediately after the exception is received. FInally, if the error is a common congestion/network error, timeout or similar then a second identical request is issued after a delay. The delay is intended to give the service and the network time to get through a backlog, or to free (or produce) more resources with which to handle the request.

Technologies

NOTE: Hystrix and resilience4j are referred to as being examples of the bulkhead pattern by numerous sources (including their own) however it appears to be distinct from the descriptions of the bulkhead approach these same sources describe. Instead of focusing on isolating components, they focus on avoiding overloading a single service. Nevertheless the industry considers them to be implementations of this pattern

These patterns are commonly bundled together and used to dynamically provide both fault and error handling, as well as load management. This means that the same set of libraries are used to implement all three of these patterns, Netlifx's Hystrix library is an example of this[18]. Although it is in maintenance mode and no longer actively developed, it served as the genesis for resilience4j, which provides much the same features. For .net there is also Polly, developed by the .NET Foundation[19]. These are all referred to as *fault tolerance* libraries, and they often implement more patterns than are described in this report such as Timeout and Cache.

As all of these libraries implement all three patterns, the choice of library is partially disconnected from the choice of patterns. The technology stack for this project is going to be java through spring boot, thus only java libraries will be considered. The main libraries used are Hystrix, resilience4j and Sentinel. Hystrix is extremely popular, with the vast majority of resources recommending it, including frameworks like spring boot itself. However, it is notably not in development as Netflix have shifted to using other methods of fault tolerance and latency/load management. This means that a new project deciding to use Hystrix should do so carefully. It's spiritual successor is resilience4j[20], which is recommended by even the Hystrix repo readme. As well as providing the same core set of features as Hystrix, it has additional features, and uses a more functional api interface through java 8[21]. Finally, Sentinel is another java library [22] which aims to handle a wider range of things beyond just fault tolerance, encompassing flow shaping and load shifting. All three libraries are open source, used in a number of high profile systems, and implement all of the patterns described above.

Bulkhead also has additional methods of implementation through the use of containers. Services such as Docker are able to provide simplified and isolated virtual machines, which when combined with features like AWS Fargate and AWS EC2, allow for the services to be isolated from each other.

---

[18] "*Netflix Hystrix repository*" https://github.com/Netflix/Hystrix (accessed 22 May 2020)
[19] "*Polly repository*" https://github.com/App-vNext/Polly (accessed 22 May 2020)
[20] "*Resillence4j repository*" https://github.com/resilience4j/resilience4j (accessed 22 May 2020)
[21] " *Resilience4j and Sentinel: Two Open-Source Alternatives to Netflix Hystrix"* https://medium.com/@alitech_2017/resilience4j-and-sentinel-two-open-source-alternatives-to-netflix-hystrix-d75bc89f3b03 (accessed 22 May 2020)
[22] "*Sentinel repository*" https://github.com/alibaba/Sentinel (accessed 22 May 2020)

Due to the lack of development on Hystrix, it should not be a primary choice. Sentinel performs more functions than is needed and has sparse documentation and guidance. For these reasons, as well as the very active development, resilience4j is the recommended choice. For the bulkhead pattern, in addition to the features offered by the library, a container system for AWS should be used, ideally through AWS Fargate and AWS Kubernetes.

Recommendation

Overall, like the prior section, none of these patterns are incompatible. As demonstrated by their presence in all the libraries, the current industry standard is to implement all of these patterns. There are however some minor differences in the use cases for the Retry and Circuit Breaker patterns. Retry is suited better for very short transient faults, often caused by elements outside of the service itself. This is things like network corruption, or short lived congestion. If a fault is longer lived, such as a critical failure of the whole service itself, then the Circuit breaker pattern functions much better. As it stops all requests for a set period of time, this lets the service recover from the fault or overloading. Transient errors however do not need this length of time to let the service recover, meaning that both Retry and Circuit Breaker should be used in tandem to complement each other. As such the recommendation is to prioritise the Retry and Circuit Breaker patterns, and to utilise the Bulkhead pattern in the overall service design through systems like Docker and AWS Buckets

Testing without all components

It is important in software development for testing to be integrated into the development process therefore as microservices are developed they will need to be tested. This can be problematic however as other components that these services have dependencies on might not be developed yet which would make testing them difficult. The following testing methods were explored to see how and if they can be used to test the software without all the components completed.

Unit Testing

Unit testing is a form of testing where a small piece of code is checked to see if it behaves as it is expected to. There are two forms of unit testing, sociable unit testing and solitary unit testing.

Sociable unit testing views the code as a 'black box' where the code itself doesn't matter, all that matters is the inputs and outputs. When the code is run if the input produces the expected output then the test is passed. Because of the black box testing approach a component that relies on a component that the tester does not access cannot be tested this way.

Solitary unit testing is a form of unit testing which explores how the component interacts with other components by replacing those components with test doubles. Because of this a component can be tested to check if it is working as intended without having access to the components it is dependent on.

Integration Testing

Integration testing is a form of testing where components are added together and are all tested together to see if they create the expected results. There are a few approaches to integration testing including big bang, top down and bottom up.

The big bang approach to integration testing involves combining all the components at once and testing the software. While this can be used for small projects in large projects this form of testing is highly inefficient as the tester is given no means by which to tell the component where the fault lies. This means while it is easy to find out there is a fault it is quite time consuming to find the fault. Furthermore this approach cannot be used without access to every single component in the software.

The bottom up approach involves first adding the lowest level modules and then adding the next level and testing them and repeating until the entire project is integrated and tested. The higher level modules that are not yet integrated are simulated by programs called drivers that call the lower level functions. An advantage of bottom up testing is that as modules are added one by one it can be easy to see which modules are the cause of the failure. This also allows the integration to begin before all the components are available if the lower level components are developed first.

The top down approach starts with the highest level module and then integrates lower levels step by step. The lower level modules that would be called by the higher level modules are simulated by stubs which can be called by the higher level modules and return values. Top down integration has the same

strengths as the bottom down approach in regards to its ability to more easily identify the location of errors in the software and allowing integration testing to occur without access to all the components.

Component Testing

Component testing is a form of testing where the larger, well encapsulated part of the code is isolated and individually tested. In microservice architecture each service can be considered to be a component. Any dependencies on other services are replaced by a test double meaning component testing can be done without access to any component except the component being tested. These tests are done to check the inner workings of the components and also to ensure the outputs are correct.

End to End Testing

End to end testing is testing the program in the way it is going to be used from start to finish simulating real life use of the software. End to end testing is used to test the dependencies of each of the microservices and verify that the entire system works as intended in the hands of a user. Due to the entire software needing to be tested end to end testing cannot be used unless all of the microservices have been completed.

Recommendation

Both methods of unit testing described above should be used wherever appropriate. Either bottom up or top down integration testing should be used as well depending on whether high or low level modules are being developed first. Component testing should be used once full microservices are completed to ensure those work as intended. End to end testing should be used at the end of the project development to verify that the software as a whole works as required by the user.

Communication Security

In order to communicate amongst each other and to be able to synchronise their actions, microservices are required to use inter-process communication. Inter-process communication is a mechanism which allows the individual services to have communication with each other and with the client by sharing memory and information. However, sharing data with other services in a complex architecture has its own share of problems, one of which is security concerns over communication in microservices.

Security problems could especially arise in communication in scenarios such as:

1. If user login details are required multiple times throughout the application-run to access a resource, the details would have to be stored in a shared memory space so that the application does not ask for verification every time it requires login data. Asking for the same details by different microservices would result in making the code redundant. But the user details might not be secure in the shared memory and also could be accessed by a third party system which is interacting with the application. [23]
2. The next problem which could be faced while working with a microservices architecture is the security of all individual microservices. Since in this architecture, simultaneous sharing of data is occurring among all the microservices, along with third party systems, it is important to make sure that the confidential data of the microservices is not shared with any individual which could exploit them.

As such, it is important to ensure that the inter-process communication is secured.

Recommendations

Some of the different security mechanisms that can be applied for the inter-service communication in a microservice architecture are:

1. **Applying Defence in depth mechanism**

   This is a technique to apply a number of security layers on the sensitive information. By applying multiple layers, it ensures that an attacker who is successful in bypassing one layer might not be able to bypass another.

2. **Applying API Gateway Pattern**

   API Gateway pattern is a technique in which a single point of entry is provided for all client API requests. This means that the client does not have any direct access to the microservices and hence, they cannot exploit any confidential data. [24]

[23] "*Microservices Security How To Secure Your Microservice Infrastructure?*", edureka!. https://www.edureka.co/blog/microservices-security (accessed 21 May 2020)
[24] "*The API Gateway Pattern*", Manning Free Content Centre. https://freecontent.manning.com/the-api-gateway-pattern/ (accessed 22 May 2020)

3. **Using OAuth for identifying user**

   OAuth uses authorization tokens to store credentials of the user in the form of cookies. Utilising this, the server is able to determine if the user has been granted access to the resource they requested.

   However, in order to protect the data from any third party applications, it is required to encrypt the tokens.

4. **Using Security Scanners and Monitoring Tools**

   Applying periodic security scanning for vulnerabilities is a good security practice. It is an automated process which scans all the elements of the application to ensure that the information is secure.[25]

5. **Using Multi-Factor Authentication**

   Multi-Factor authentication requires another form of verification of identity, along with the id and password from the user. This would make it difficult for an attacker to bypass the system to gain access and would result in better security of the overall application.

[25] "*8 best practices for microservices app sec*", Techbeacon. https://techbeacon.com/app-dev-testing/8-best-practices-microservices-app-sec (accessed 22 May 2020)

Monitoring Systems

To facilitate the proper working of a microservice, proper monitoring is required. As a microservice increases in complexity, more points of failure will appear in the system. When these services do fail or get degraded, we need to be able to notice it, and find out the details of what occurred. This is an integral part of SAFe's CALMR approach to DevOps, and it's important to find the right tools to help show these metrics. While we could simply write logs into text files, it does not allow us to quickly identify issues the same way an updating chart will.

Given the current scope of our project, we are looking for free monitoring systems that can easily be integrated with our project's EC2 implementation. We find that SpringBoot has an Actuator class[26], with many production ready features. This includes dependency management and auto-configuration for Micrometer, an application metrics façade. By using Micrometer, we can change which monitoring system we use, without significantly changing any of our source code[27]. This means that should our project scope change, we are able to easily change to a more suitable monitoring system.

Given this information, we can narrow down the choice of monitoring system to the list of Micrometer supported monitoring systems. Many of these systems are "Sold as A Service" (SAAS), meaning that we can send metrics through their API, where they will analyze and display metrics on their own dashboards. However, after going through the list of supported systems, many of these are not free, with a few exceptions. Hence, for the majority of these, we will not be considering them for our project.

The rest of these systems are standalone systems, with built in time-series databases. They can receive time-series data in real time via the Micrometer implementation. Compared to SAAS monitoring systems, we would need to dedicate space in the cloud to run these databases, which can take up resources and increase the number of instances in our EC2 cloud. However, many of these systems are free and open source, meaning there is potential for community support and extension. Compared to SAAS, we don't need to rely on a third party for our monitoring, but we do need to configure it ourselves. It is also shown that running our own standalone system will cost less on average compared to other SAAS, even with the server cost included[28].

After removing all the paid systems and systems not relevant to our project, we are left with a list of 4 monitoring systems. 3 of them are standalone, Netflix/Atlas, Graphite, and Prometheus. 1 of them is an SAAS but with a free tier, AWS CloudWatch. All of these are usable for our project, but we can further discuss the strengths and weaknesses between them.

---

[26] "Spring Boot Actuator: Production-ready Features," https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-features.html. (accessed 23 May 2020)

[27] "Micrometer Application Monitoring," https://micrometer.io/docs. (accessed 23 May 2020).

[28] "Prometheus vs. CloudWatch for Cloud Native Applications," https://www.infracloud.io/prometheus-vs-cloudwatch/. (accessed 23 May 2020).

Starting with the standalone systems, Netflix/Atlas is an in-memory database, meaning it stores its data in memory while periodically saving to storage. This was done to allow for the gathering and reporting of a very large number of metrics. It was developed when their system was unable to handle the 1.2 billion metrics it needed[29]. While this system is great at handling large amounts of data, and can work for our project, the fact that it uses an in-memory database makes it unsuitable. Our EC2 plan only allows for so many instances and using memory instead of disk space to log makes this inefficient to our small-scale project. We are not likely to ever need that many metrics, so as of now Atlas is not going to be considered.

The next standalone system is Prometheus, a time-series database management system and graphing tool. It has a complete implementation for monitoring with a powerful query language and good community support. Implementation is simple, with binaries all setup to perform out of the box, with many libraries available to quickly extend our monitoring system.[30] The main downside is its difficulty in extending it, as its query language has a high learning curve. Given our project, a simple to start monitoring solution is great, as no other tools are required, and we don't require any complex metrics to use.

The next system is Graphite, a time-series database and graphing tool. The data it stores and uses is simplistic in nature, meaning it's great for storing long term data compared to more robust data formats. Being simple and older also means it can greatly benefit from other graphite compatible tools. It does have a few issues when compared to Prometheus though. It requires the use of these other tools to function to the same capacity as Prometheus, increasing the configuration complexity.[20] Overall Prometheus is more well suited for this project overall if we were to pick a standalone system.

Finally we get to AWS Cloudwatch, the SAAS monitoring system provided by AWS. It has a free tier that allows for unlimited basic metrics, 10 detailed metrics and up to 1000000 API requests[31]. Since we are using AWS EC2, Cloudwatch implementation is automatically done for us. Similar to Prometheus, it is able to receive, analyze and report metrics without much setup. With that being said, it is not as extendable as Prometheus, and past the free tier, may cost more than a server set up with Prometheus.

Recommendation

Given the above information, Cloudwatch will be good for our project, so long as the number of custom metrics is relatively low. It's easy to use and requires no server resources, which is great for our current project scope.

However, should we need to extend the project much greater, past Cloudwatch's free tier limitations, moving our monitoring system to Prometheus is also a valid option. Thanks to the micrometer façade,

---

[29] "*Netflix/atlas Wiki*" https://github.com/Netflix/atlas/wiki. (accessed 23 May 2020).

[30] "*Prometheus vs. Graphite: Which Should You Choose for Time Series or Monitoring?,*" https://logz.io/blog/prometheus-vs-graphite/ (accessed 23 May 2020).

[31] "*Amazon CloudWatch - Application and Infrastructure Monitoring,*" https://aws.amazon.com/cloudwatch/. (accessed 23 May 2020).

changing systems will be simple, and the cost of hosting a server for Prometheus is cheaper than sending those metrics to Cloudwatch.

References

K. V. D. Hallen. "*Microservices: The most occurring obstacles.*" ToThePoint. https://tothepoint.group/microservices-the-most-occurring-obstacles/ (accessed 20 May 2020).

J. Bonér. "*Reactive Microservices: Why Asynchronous Communication Matters*". Lightbend. https://www.lightbend.com/microservices/reactive-microservices-why-asynchronous-communication (accessed 20 May 2020).

J. Schabowsky. "*Microservices Choreography vs Orchestration: The Benefits of Choreography.*" Solace. https://solace.com/blog/microservices-choreography-vs-orchestration/ (accessed 21 May 2020).

T. Reeder. "*Asynchronous Messaging Patterns.*" MuleSoft. https://blogs.mulesoft.com/dev/design-dev/asynchronous-messaging-patterns/ (accessed 22 May 2020).

T. Clemson *"Testing Strategies in a Microservice Architecture"*, Martin Fowler. https://martinfowler.com/articles/microservice-testing/#testing-unit-introduction (Accessed 22 May 2020)

J. Purcell *"Principles of solitary unit testing"*, Drupal. https://events.drupal.org/barcelona2015/sessions/principles-solitary-unit-testing (Accessed 22 May 2020)

"What is Integration Testing ", Software Testing Help. https://www.softwaretestinghelp.com/what-is-integration-testing/ (Accessed 22 May 2020)

"What Is End to End Testing", Software Testing Help. https://www.softwaretestinghelp.com/what-is-end-to-end-testing/ (Accessed 22 May 2020)

"Communication Between Microservices: How to Avoid Common Problems", Stackify. https://stackify.com/communication-microservices-avoid-common-problems/ (Accessed 21 May 2020)

"*Dead letter queue*". Wikipedia. https://en.wikipedia.org/wiki/Dead_letter_queue (accessed 23 May 2020)