Asynchronous operations at scale

BRAINLY

Introduction

What is communication? The simplest definition is that it's a transfer of information from one entity to another. In application development, when we want to communicate with external services like a database, payment provider, or anything else, we can almost always choose to do it synchronously or asynchronously. What's the difference?

Synchronous means we stop whatever we're doing, focus on delivering information and retrieving a response, and only then can we go further with our task. The most common example is simple HTTP request execution:

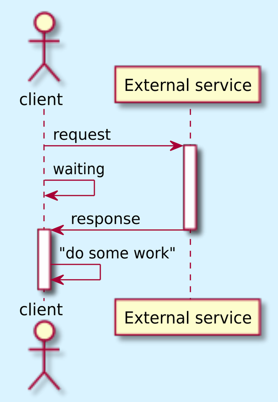


Figure 1: Sequence diagram of synchronous communication

```python
# sync.py
import requests


def sync_request():
    status = requests.get("http://example.com/").status_code
    print(status)


sync_request()
print("doing something else")

# 200
# doing something else
```

Figure 2: Example in Python of sequential HTTP communication

Asynchronous means we start communication, send a message, continue with whatever we we doing, and react when a response comes back:

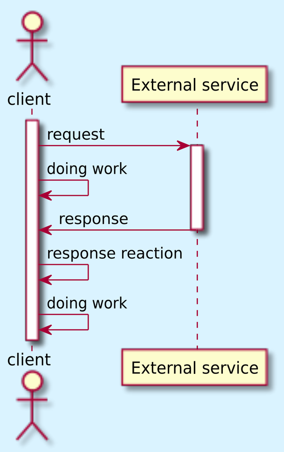


Figure 3: Sequence diagram of synchronous communication

```python
# async.py
from concurrent.futures import ThreadPoolExecutor

import requests


def request():
    response = requests.get("http://example.com/")
    print(response.status_code)


def async_print():
    print("status")


executor = ThreadPoolExecutor()
executor.submit(request)
executor.submit(async_print)
executor.shutdown(wait=True)

# status
# 200
```

Figure 4: Example in Python of asynchronous HTTP communication

In real life we can compare synchronous communication to phone calls, and asynchronous to texting

Comparison

Synchronous	Asynchronous
Simple implementation, usually default approach	Trickier implementation, have to take more into account
Resources are unused during waiting	There is possibility of making useful work during waiting time
Code is more readable	You have to jump through the code to find callbacks and other execution places

Figure 5: Comparison of communication paradigms

Queue systems

A queue system is a very useful tool in asynchronous communication. Brief definition:

> *A queue is a line of things waiting to be handled - in sequential order starting at the beginning of the line. A message queue is a queue of messages sent between applications. It includes a sequence of work objects that are waiting to be processed.*

https://www.cloudamqp.com/blog/2014-12-03-what-is-message-queuing.html

Using queues makes it easy to create decoupled components, which can scale, you can have much more consumers than producers. Right section shows detailed implementation of pub-sub. Another popular library for async tasks in Python is celery which can use different transport layers, like RabbitMQ, Redis, etc.

Case study - what happens when you answer a question

Every day we get a lot of questions answered. To enable our **100 million monthly unique users** to help each other, as our user base grew, we needed to grow our infrastructure too. First implementation of the feed (list of newest questions, filterable by grade, subject, or is question answered) was simple select on a SQL database. At first, it was good solution. **But with great traffic comes great responsibility.** When for some reason we get a lot of requests for feed and its database is under heavy load, we should be able to sill serve basic information about the question. Assuming we have microservice that allows us to collect questions and it's answers.

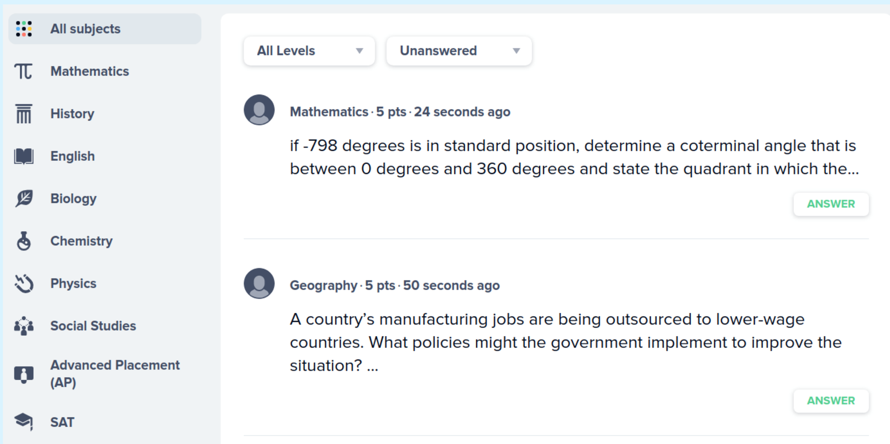


Figure 6: Fragment of the question feed on right, on the left, is the sidebar with the list of subjects.

We could approach implementing new feed in the following way:
- Start the transaction in questions database
- Add answer there
- Notify by HTTP call our feed service that there is a new answer
- When everything went fine commit transaction
- Rollback otherwise

Looks fine, **but**:
- HTTP calls can take a long time to execute (even because of the network latency)
- Adding answer now depends on external service
- Adding new service that wants to collect answers will create a new dependency

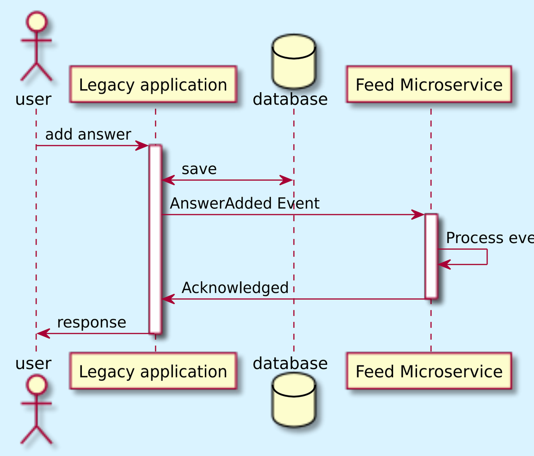


Figure 7: Sequence diagram presenting possible synchronous flow with notifying feed microservice about new questions.

How we implemented new feed:
- Every time answer is added message is published to RabbitMQ
- Microservice listens on this queue and processes it

What are the **benefits**:
- No matter how many microservices subscribe for given topic the will be only overhead of pushing one message to RabbitMQ
- Even when for some reason feed service is down, or takes a long time to respond we are able to properly handle answer adding

What are the **drawbacks**:
- The feed will always be a little in the past. Answered questions will not appear there instantly, usually, this takes a few milliseconds - We can't (easily) rollback a transaction if there is an error, more on that on the right also lookup saga pattern for transactions in distributed systems.

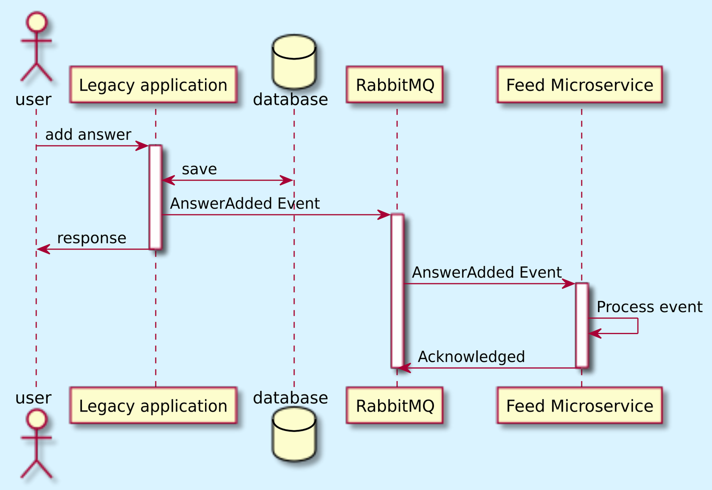


Figure 8: Sequence diagram presenting possible asynchronous flow with notifying feed microservice about new questions.

Handling errors

To handle the async communication we're using RabbitMQ - a queue system. It comes with quite a few advantages:
- The consumer decides how many messages it can handle at once, if we need more processing power usually we can just add more consumer instances
- When there is sudden traffic hit, instances of consumers will not be overwhelmed, they will just process as much as they declared. When we add **auto scaling** by queue size, it will quickly adapt to changes and process messages fast when needed, and keep only a few instances when traffic is low. However, this approach comes with a downside - we will have a delay in processing messages if they pile up.

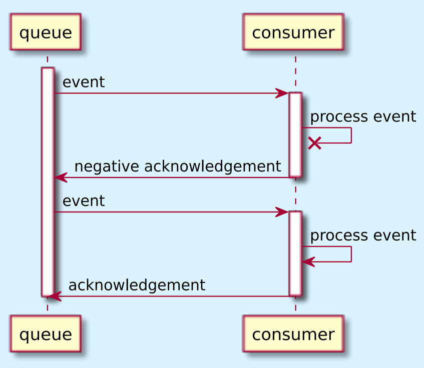


Figure 9: Sequence diagram presents redelivery of the event when consumer failed processing it once

Not every delivery to consumer and processing will be successful. If we're using at most once delivery, we have already we have already come to terms with that, but what if we have at least one delivery? There are two cases possible, their sequence diagrams are presented on figure 9 and 10.

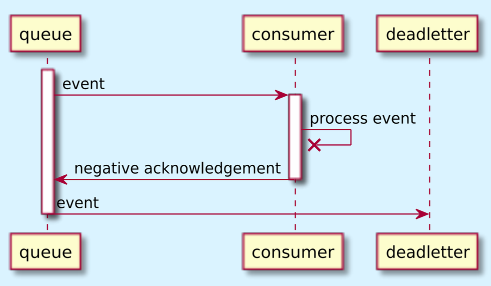


Figure 10: Sequence diagram presents moving the to deadletter queue when consumer failed processing it once

Solution 1: **redeliver** messages on processing failure.
The good approach when there was an error, for example on DB, or connection to external service, but what will happen when the error occurs every time?
Consider the example: The messages body is JSON, it may be malformed. Then messages then will get redelivered, again and again, few hundred times per second, consumers CPU will rise to the max, and we will get thousands of error logs (yes, it has happened).

Solution 2: **Dead-letter** queue.
One of the possible solutions to this problem is when an error happens because of the external service we delay redelivery, other way is that we use the second approach:
- Dead-letter - whenever queue system receives negative acknowledgment signal it sends message to another - dead-letter queue. Then we're notified about an error and can manually investigate why it has happened and eventually send messages to the main queue.

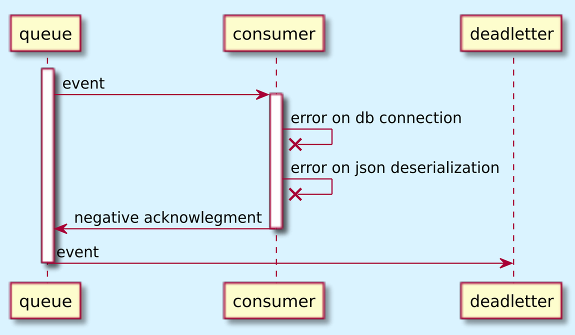


Figure 11: Sequence diagram presents moving the to deadletter queue when consumer failed processing it on non retriable error

Solution 3: **Composition** of both. When we know that an error is not deterministic, we will try to process the message again. When an error will happen every time or is unknown it's better to send it to dead-letter.

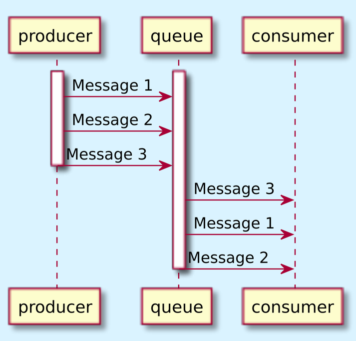


Figure 12: Sequence diagram illustrating one of the possible orders of messages.

Messages order

Our queue system (RabbitMQ) and many others do not guarantee you that when the publisher send messages in a given order, a subscribers will get them in the same order. In some cases, like counting views on a page, you will not care, which user viewed the page first. In cases, when we need to know what exactly happened first - there is only one product left in e-commerce and we have to know who bought it first we can add a timestamp to event and trust publishers that they will do it lawfully. Another way may include architectural solutions, like messages sourcing which was well described by my colleague in his article: **CQRS and Event Sourcing applied in event-based microservices** https://goo.gl/Jec5VE or using conflict-free replicated data types **(CRDT)**.

link to CQRS and Event Sourcing applied in event-based microservices

Our general approach

To abstract all async communication we have created our own library which uses RabbitMQ as transport, but most important are it's base ideas which can be transferred to any implementation.
I will use the following terms:
- Publisher - puts a message into the bus
- Subscriber - listen to the messages on a specific topic
- Message - is a information sent to the bus, has fields a:
  - Header - contains topic - name identifying message, its role and message id
  - Body - list of bytes, message content
- µservice - a place where a work is done

Types of services

Publisher
Only puts messages to the bus. Example service is a CRON like scheduler, every minute it publishes a message to the bus, or legacy application which emits an event when a question, answer etc is added. Concept is presented on figure 13.

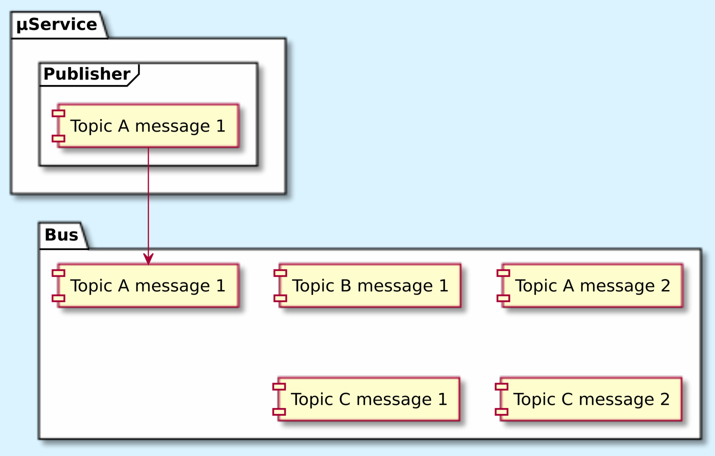


Figure 13: A publisher sends messages to the bus when other types of messages are present.

Consumer
Consume messages from a topic(s) and do some work with it. Example service is the feed which consumes events about asked questions, answers added.

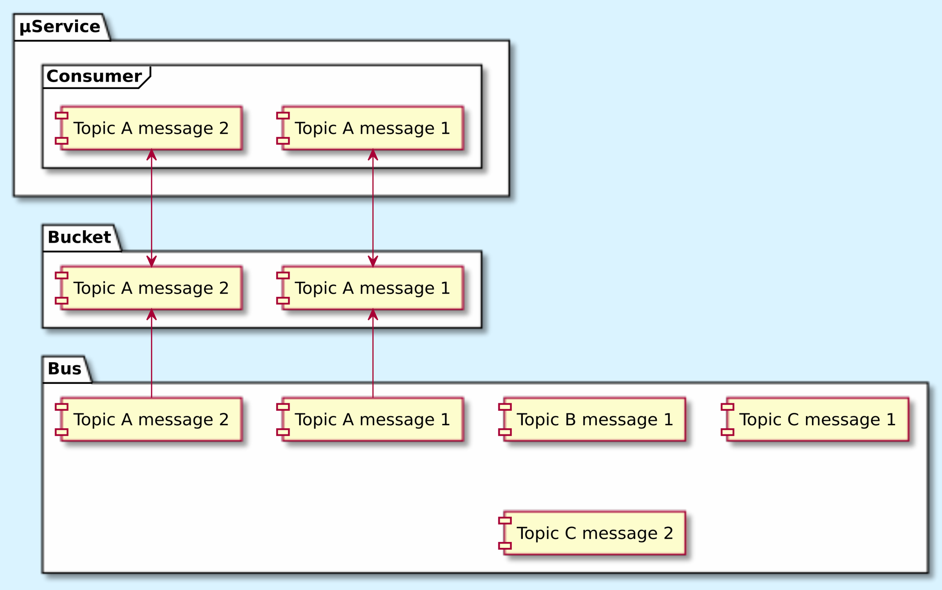


Figure 14: Consumer listens only on topic(s) it's interested in.

Many µservices will be instances of both. For example, we have service for asynchronous content validation - it listens on the topic "`QuestionNeedsValidation`" and when a message arrives it evaluation content, and then publish event "`QuestionValidated`" with evaluation result.

Delivery strategies

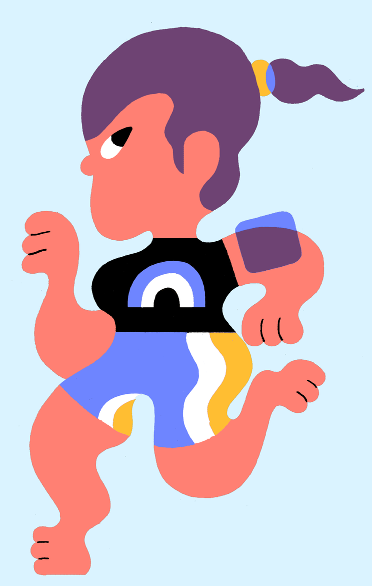
Our world isn't perfect. Sometime messages **won't reach a consumer**, most often it will be a failure on the network. Messaging system, whether we use something ready, or implement our own will always provide us with one of the following solutions:
- **At most once delivery** - we send the message to a consumer, and hope it will get it

This approach is usually good when we have a lot of events, and we care more for speed than accuracy. A good example of that would be counting of user views on each page or requests per endpoint.

- **At least once delivery** - we send the message to a consumer, and wait for the confirmation that it processed it, if we didn't get it, or get error response, we retry sending it.

Why not exactly once delivery? Failure can occur in 3 possible points:
- On sending a message to a client - then it will be redelivered and processed only once
- On processing a message - same as above
- On sending an acknowledgment (ack) back - it will be delivered second time

The third case is worst, but our client can implement countermeasures. Easiest one is storing identifiers of successfully processed messages in a database with a unique index.



Key take always (tl;dr)

- everything is a tradeoff
- synchronous communication easy to write
- asynchronous communication easy to scale

author: Hubert Bryłkowski - Backend Engineer @ Brainly

@hubbry

hubert@brylkowski.com