Implementing Exponential Backoff and Jitter with dockerized Django application with Redis, celery, and Postgres to handle asynchronous tasks

Exponential backoff is a standard error handling strategy for network applications in which a client periodically retries a failed request with increasing delays between requests. Clients should use exponential backoff for all requests to Memorystore for Redis that return HTTP 5xx and 429 response code errors.

The fictious company that has the payment gateway is **Summit Ecommerce** platform multiple clients that send messages that collide. They all decide to back off. If they use the same deterministic algorithm to decide how long to wait, they will all retry at the same time -- resulting in another collision. Adding a random factor separates the retries. implementation of retry will depend a lot on the application needs.

The issue

The problem here is that N clients compete in the first round, N-1 in the second round, and so on. Having every client compete in every round is wasteful. Slowing clients down may help, and the classic way to slow clients down is capped exponential backoff. Capped exponential backoff means that clients multiply their backoff by a constant after each attempt, up to some maximum value. This leads to short-lived spikes in contention, and other correlated effects on the network. the is to add randomness, which breaks the loop that creates synchronization.

To understand the problem, let's imagine a scenario where the transient fault is happening because the service is overloaded or some throttling is implemented at the service end. This service is rejecting new calls. This is a transient fault as if we call the service after some time, our call could succeed. There could also be a possibility that our retry requests are further adding to the overload of the busy service, which would mean that if there are multiple instances of our applications retrying for the same service, the service will be in the overloaded state longer and will take longer to recover from this state.

The thundering herd problem occurs when a large number of processes or threads waiting for an event are awoken when that event occurs, but only one process is able to

handle the event. When the processes wake up, they will each try to handle the event, but only one will win. All processes will compete for resources, possibly freezing the computer, until the herd is calmed down again. Consider a call path experiencing hundreds of calls per second where the underlying system suddenly develops a problem. A fixed-progression exponential backoff can still generate peaks of load.

A well-known retry strategy is exponential backoff, allowing retries to be made initially quickly, but then at progressively longer intervals: for example, after 2, 4, 8, 15, then 30 seconds. In high-throughput scenarios, it can also be beneficial to add jitter to wait-and-retry strategies, to prevent retries bunching into further spikes of load. Jitter is a decorrelation strategy which adds randomness to retry intervals to spread out load and avoid spikes.

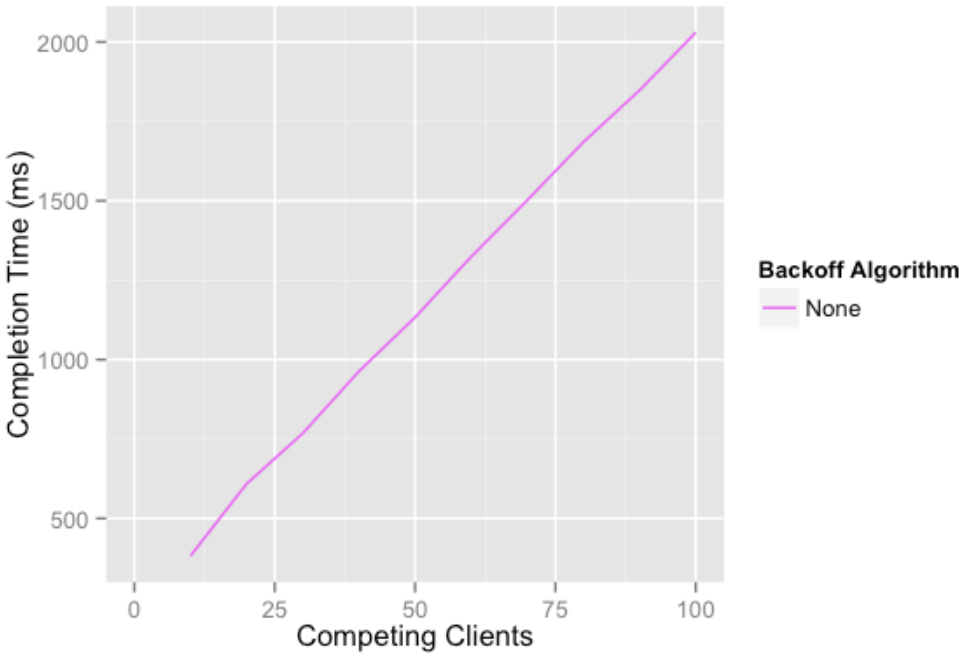
Transient Failures

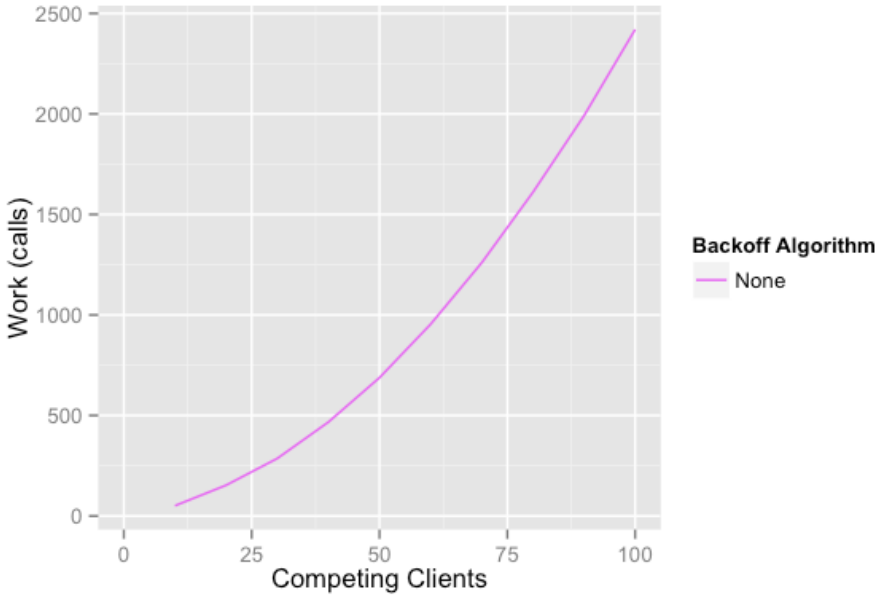
Transient failures are the failures that occur while communicating to the external component or service and that external service is not available. This unavailability or failure to connect is not due to any problem in the service, but due to some reasons like network failure or server overload. The first task should be to identify the transient faults.

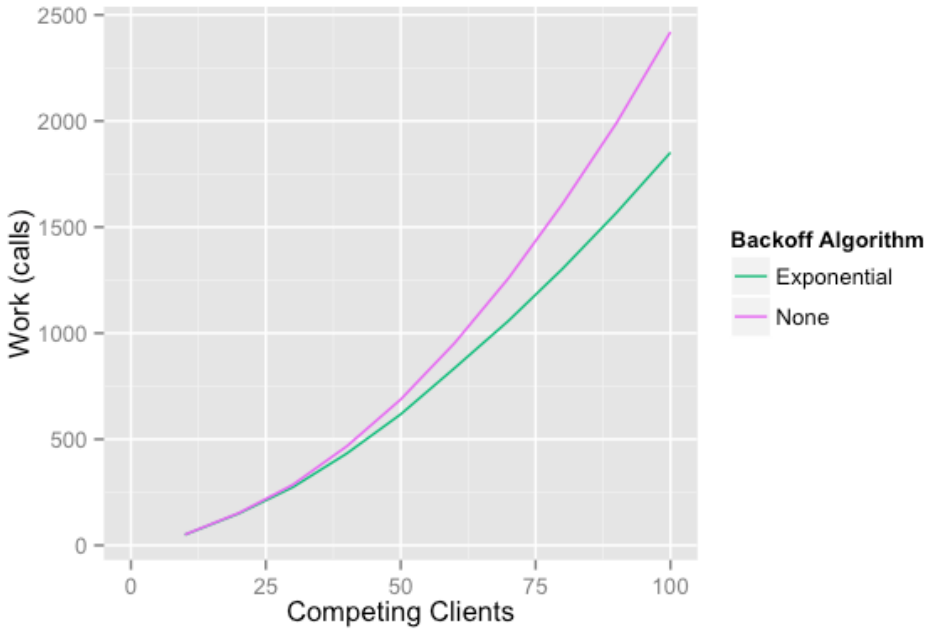
In the new cloud world, the chances of getting such errors has increased since our application itself might also have some elements and components running in the cloud. It might be possible that different parts of our applications are hosted separately on the cloud. Failures like momentary loss of network, service unavailability, and timeouts have become more prominent (comparatively).

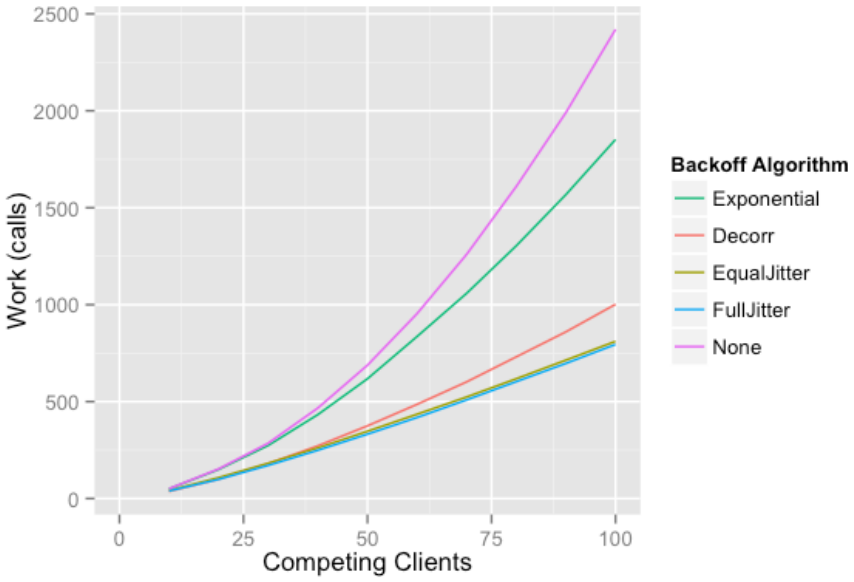
Architectural Design Pattern

In a distributed systems that provides resilience and transient-fault handling capabilities. You can implement those capabilities by applying Polly policies such as Retry, Circuit Breaker, Bulkhead Isolation, Timeout, and Fallback.


2000
1500
1000
500
Completion Time (ms)
0
25
50
75
100
Competing Clients
Backoff Algorithm
None

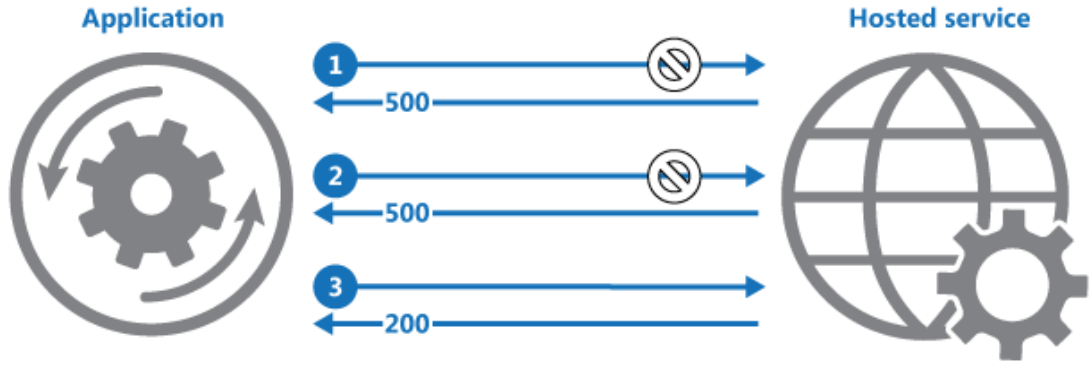

2500
2000
1500
1000
500
0
Work (calls)
0
25
50
75
100
Competing Clients
Backoff Algorithm
None





Retry policy with the number of retries, the exponential backoff configuration, and the actions to take when there's an HTTP exception, such as logging the error

Web applications usually start out simple but can become quite complex, and most of them quickly exceed the responsibility of only responding to HTTP requests.



1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

Case Study

The client can subscribe to have its own store in minutes

All the platform will be multi-tenant, so, it will not be duplicated to every customer. Every customer will see only its own data

The customer will have a temporary domain and will be able to specify a main domain when the store goes up

The customer will have a admin panel to manage its products

The customer will be able to see the requested products

By default the store will use pagar.me as the payment gateway, but the system will be able to accept another gateways

The store will have a search area

The store will have a custom logo

The store will not have a customer area, neither an authentication area, only a checkout area

Microservices

They are divided in product contexts (so it is recommended to implement them if you have business issues, not technical issues), not in modules. Some examples of contexts that a product can have: Marketing, Law, Selling. It is recommended to have a single database per microservice (in order to keep it decoupled as you can).

The microservice must have a fallback rule, it must be written to fail and to deal with fail (without making another microservice fail if there's communication between them)

The good points are:

If some context goes down, the others stay up;

A team can manage its own context without accessing another;

You can scale a unique context;

You have specialized teams by contexts;

You can use different technologies by contexts.

The pain points are:

Since everything is not coupled, you've to deal with distributed communication.

 Distributed Message Queue System

That's generally a queue which you can use to put some messages inside it and read some messages from it as well. Usually you have the following stuff working around it:

Publisher: is able to put/create messages on the queue.

Consumer: is able to get/read messages from the queue.

Queue: is able to store messages inside it for some time till some service reads or deletes the message from it.

Dead Letter Queue: is able to retain the message that has not been processed after some attempts.

Direct Exchange: is able to send a message direct to a single queue consumer that is listening to it.

Fanout Exchange: is able to send a message for all the consumers that are listening to a queue.

Transactions with Pending Status

All payments start out with the status "Pending" and are only updated to "Complete" when payment is successfully received. A transaction is considered pending when it received an approved response from the gateway but the payment application was not able to create a payment record for the transaction. This usually occurs when the invoice being processed against was not in a fully saved state.

Transaction Statuses:

Completed

Pending Offline

Declined

Voided

Authorized

Processing

Gateway Error

Error

Distributed Rate Limiting using Redis and Celery

if API calls are made from multiple servers, adhering to the rate limit is a difficult problem. We have many servers, with each server running multiple running Celery worker processes making API calls. All the Celery workers feed from the same Redis queue to maintain consistency.

Operations in a web application can be classified as critical or request-time operations and background tasks, the ones that happen outside request time. These map to the ones described above:

needs to happen instantly: request-time operations

needs to happen eventually: background tasks

in a Producer Consumer Architecture. this architecture can be described like this:

Producers create data or tasks.

Tasks are put into a queue that is referred to as the task queue.

Consumers are responsible for consuming the data or running the tasks.

Retry Pattern With Exponential Back-Off

Making Tasks More Reliable

Tasks are often used to perform unreliable operations, operations that depend on external resources or that can easily fail due to various reasons. Here's a guideline for making them more reliable:

Make tasks idempotent. An idempotent task is a task that, if stopped midway, doesn't change the state of the system in any way. The task either makes full changes to the system or none at all.

Retry the tasks. If the task fails, it's a good idea to try it again and again until it's executed successfully. You can do this in Celery with Celery Retry. One other interesting thing to look at is the Exponential Backoff algorithm. This could come in handy when thinking about limiting unnecessary load on the server from retried tasks.

It smooths traffic on the resource being requested.

If your request fails at a particular time, there's a good chance other requests are failing at almost exactly the same time. If all of these requests follow the same deterministic back-off strategy (say, retrying after 1, 2, 4, 8, 16... seconds), then everyone who failed the first time will retry at almost exactly the same time, and there's a good chance there will be more simultaneous requests than the service can handle, resulting in more failures. This same cluster of simultaneous requests can recur repeatedly, and likely fail repeatedly, even if the overall level of load on the service outside of those retry spikes is small.

By introducing jitter, the initial group of failing requests may be clustered in a very small window, say 100ms, but with each retry cycle, the cluster of requests spreads into a larger and larger time window, reducing the size of the spike at a given time. The service is likely to be able to handle the requests when spread over a sufficiently large window.

Once we have identified the fault as a transient fault, we need to put some retry logic so that the issue will get resolved simply by calling the service again. The typical way to implement the retry is as follows:

Identify if the fault is a transient fault.

Define the maximum retry count.

Retry the service call and increment the retry count.

If the calls succeed, return the result to the caller.

If we are still getting the same fault, keep retrying until the maximum retry count is hit.

If the call is failing even after maximum retries, let the caller module know that the target service is unavailable.

Code Architecture

• Instantiate Jitter objects directly within JavaScript and create function chains of Jitter processes within procedural code.

• Create Jitter matrices with JavaScript and access and set the values and parameters of Jitter matrices from within JavaScript functions.

• Use Jitter library operations (e.g. jit.op operators and jit.bfg basis functions) to do fast matrix operations on Jitter matrices within JavaScript to create low-level Jitter processing systems.

• Receive callbacks from Jitter objects by listening to them and calling functions based on the results (e.g. triggering a new function whenever a movie loops).

Before beginning this tutorial, you should review the basics of using JavaScript in Max by looking at the JavaScript tutorials starting with Basic JavaScripting and JavaScript Scripting. These tutorials cover the basics of instantiating and controlling a function chain of Jitter objects within js JavaScript code.

Celery is a task queue. It receives tasks from our Django application, and it will run them in the background. Celery needs to be paired with other services that act as brokers.

Making Tasks More Reliable

Tasks are often used to perform unreliable operations, operations that depend on external resources or that can easily fail due to various reasons. Here's a guideline for making them more reliable:

Make tasks idempotent. An idempotent task is a task that, if stopped midway, doesn't change the state of the system in any way. The task either makes full changes to the system or none at all.

Retry the tasks. If the task fails, it's a good idea to try it again and again until it's executed successfully. You can do this in Celery with Celery Retry. One other

interesting thing to look at is the Exponential Backoff algorithm. This could come in handy when thinking about limiting unnecessary load on the server from retried tasks.

Brokers intermediate the sending of messages between the web application and Celery. In this tutorial, we'll be using Redis. Redis is easy to install, and we can easily get started with it without too much fuss. A system can it can still perform quite poorly under high contention. The simplest of these contention cases is when a whole lot of clients start at the same time, and try to update the same database row. With one client guaranteed to succeed every round, the time to complete all the updates grows linearly with contention.

Exponential backoff, retry algorithm will look like following:

Identify if the fault is a transient fault.

Define the maximum retry count.

Retry the service call and increment the retry count.

If the calls succeeds, return the result to the caller.

If we are still getting the same fault, Increase the delay period for next retry.

Keep retrying and keep increasing the delay period until the maximum retry count is hit.

If the call is failing even after maximum retries, let the caller module know that the target service is unavailable.

Production-ready distributed task queue management system with celery

Using Celery but making a production-ready which means Highly efficient, Resilient, transparent, and scalable

Enable an application to handle transient failures when it tries to connect to a service or network resource, by transparently retrying a failed operation. This can improve the stability of the application.

When to use this pattern

Use this pattern when an application could experience transient faults as it interacts with a remote service or accesses a remote resource. These faults are expected to be short lived, and repeating a request that has previously failed could succeed on a subsequent attempt.

This pattern might not be useful:

When a fault is likely to be long lasting, because this can affect the responsiveness of an application. The application might be wasting time and resources trying to repeat a request that's likely to fail.

For handling failures that aren't due to transient faults, such as internal exceptions caused by errors in the business logic of an application.

As an alternative to addressing scalability issues in a system. If an application experiences frequent busy faults, it's often a sign that the service or resource being accessed should be scaled up.

Code Implementation

I created a Docker compose configuration for our project. Docker compose is a tool that allows us to run our Docker image easily from our project location. Basically, it allows us to easily manage the different services that combine with our project.

Django Rest Framework, with Celery and Redis as broker.

As of Celery 4.2 you can configure your tasks to use an exponential backoff automatically: http://docs.celeryproject.org/en/master/userguide/tasks.html#automatic-retry-for-known-exceptions

The main component of a celery enabled program or a celery setup is the celery worker.

A celery worker can run multiple processes parallel

A task is a class that can be created out of any callable. It performs dual roles in that it defines both what happens when a task is called (sends a message), and what happens when a worker receives that message.

A task message is not removed from the queue until that message has been acknowledged by a worker. A worker can reserve many messages in advance and even

if the worker is killed – by power failure or some other reason – the message will be redelivered to another worker.

Conclusions

I hope this has been an interesting tutorial for you and a good introduction to using Celery with Django.

Here are a few conclusions :

It's good practice to keep unreliable and time-consuming tasks outside the request time.

Long-running tasks should be executed in the background by worker processes (or other paradigms).

Background tasks can be used for various tasks that are not critical for the basic functioning of the application.

Celery can also handle periodic tasks using the celery beat service.

Tasks can be more reliable if made idempotent and retried (maybe using exponential backoff).