Improving the resilience of Java API servers:

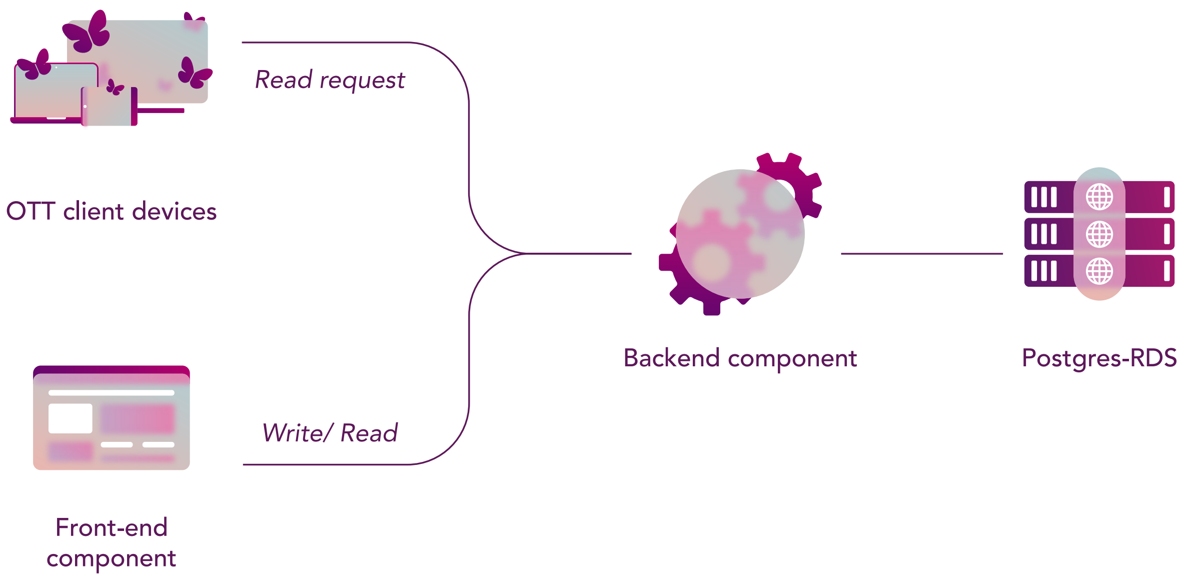
The system at a glance

When servers experience large volumes of users, the CPU prioritizations can struggle to execute requests for licenses and in some cases, will result in error messages being sent to the connecting viewing parties. This eBook will detail the approach of one of the Irdeto Engineers as they explain their approach to solving this issue.

Table of Contents

The system we are working on is a digital rights and rules manager. Digital Rights Management (DRM) is an approach to copyright protection for digital media. A DRM system prevents unauthorized redistribution of digital media and restricts how consumers copy content they've purchased.

Our system generates a key for encryption and stores the encrypted content in the operator's Content Delivery Network (CDN). When a user wants to play this content, they need a license that contains the decryption key for the content and policies. These policies define the quality of the content, either High Definition (HD) or Ultra-High Definition (UHD), set geo-restrictions and time limits, etc. To obtain the license, the user goes back to our service.



What is application resilience?

The resilience of an application refers to its ability to continue functioning even when faced with unexpected situations. Example of these instances include:

- Hardware failures
- Network failures
- Partial system failures
- Increased user traffic handling or spike handling

HARDWARE FAILURES

Hardware failures can occur anytime, even the hard disk has a Mean Time to Failure (MTTF) of 10 to 50 years. A system can crash for many reasons, human error and power outages are amongst the most common causes. During such occurrences, the machine or server can be shut down and the issues are fixed either using master/slave or horizontal scaling approaches.

NETWORK ISSUES

Anything sent over an open network has a greater chance of failing than when using internal communication. Some common types of network issues include:

- **Connectivity problems** occur when there is a physical fault with the connection between devices, such as broken cables or faulty network adapters.
- **Configuration errors** occur when network devices are configured incorrectly, such as routers or switches.
- **Latency and bandwidth issues** refer to slow response times, high latency and insufficient bandwidth.
- **Security issues** which are caused by attacks from malicious actors, such as malware or hacking.

Most of these issues can be fixed by the network administrator, but also by retrying the software for any network hiccups and by having a fallback to serving details from local memory instead of going to the internet.

PARTIAL FAILURES

A partial failure of software describes a situation where an application or system only performs its intended functions partially. This type of failure can be caused by bugs, configuration errors and hardware issues to name a few.

When handling partial failures, remember to:

- Always provide a fallback to each external system that may fail
- Have local highly available caches with the most recent state of the external service to serve, in case of the external service failure.

SPIKE LOAD

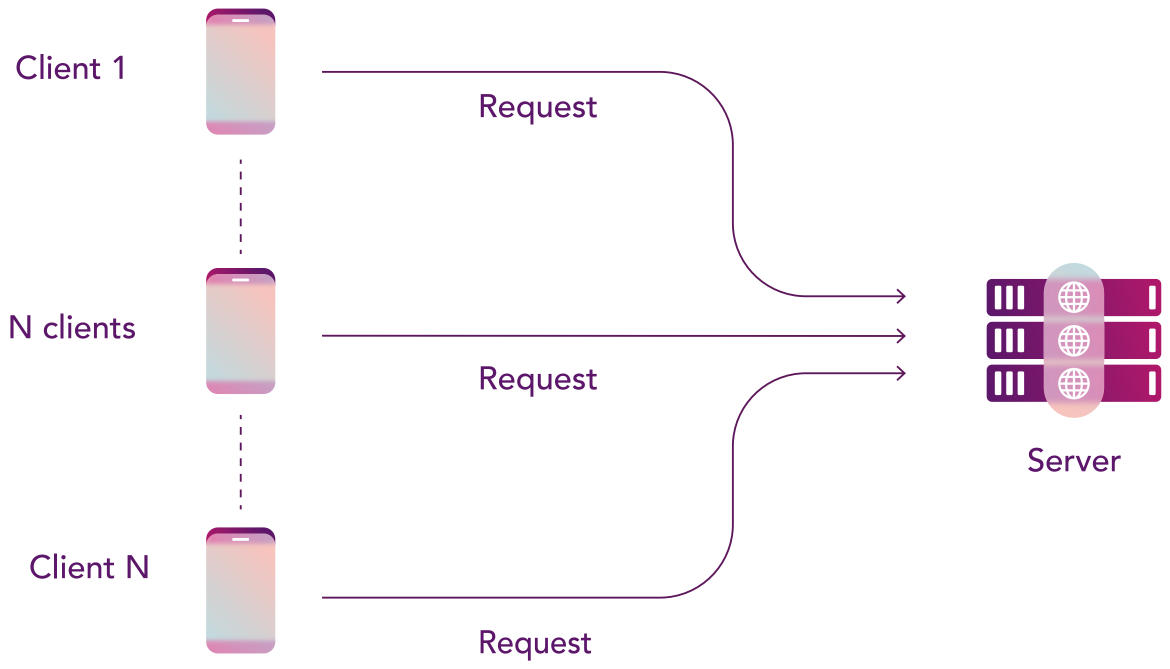
A spike load refers to the surge in the number of users over a very short period of time. An example would be when viewers turn on the stream for a high-profile sports game. What would normally be flat or consistent, then experiences a spike as users tune in. During this process, the system is monitored.

We will walk you through the kind of spike-related issues we faced and how we tackled them. But before we continue, there are two main things to consider in our application when there is a sudden increase in load or a spike:

- Flow control
- Request prioritization

Flow control

Flow control is a way to make sure the API server is not overwhelmed by concurrent client requests.



As shown above, the N-number of clients will send requests to the server, where the server then processes the request and sends the response back to the client.
Every server has limited resources like:

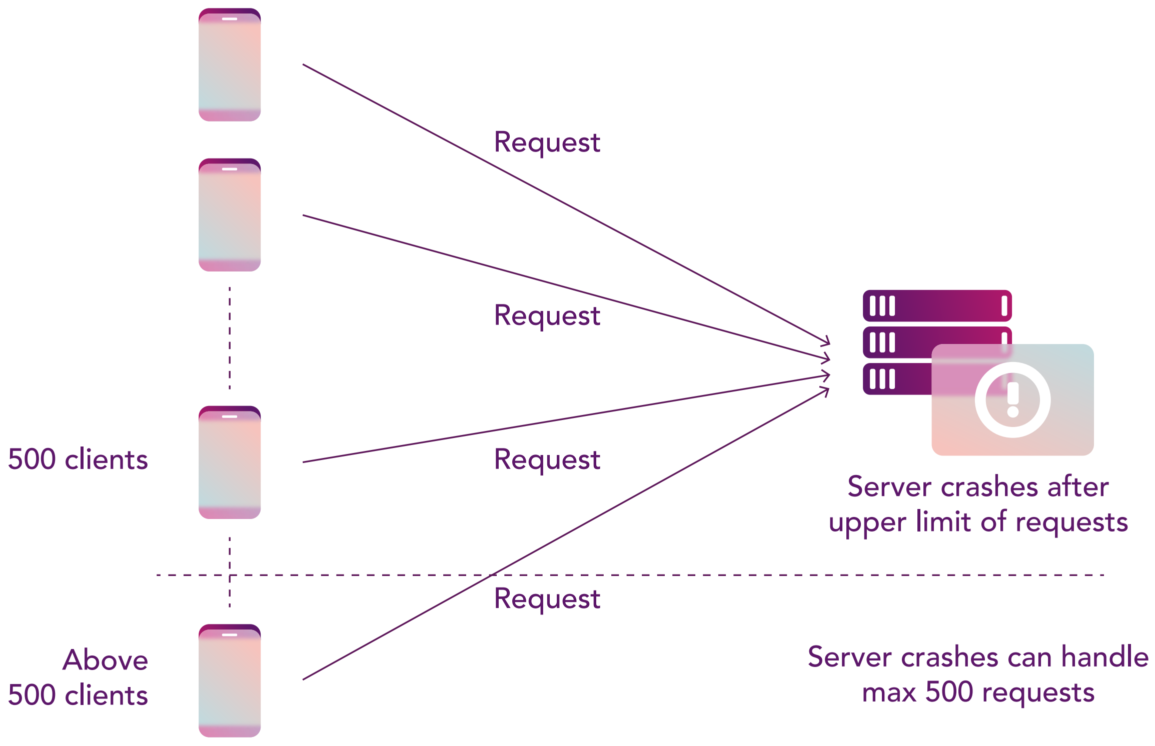
- Limited storage resources (Memory)
- Limited computational resources (CPU)

Flow control and memory

Every client who wants to send a request to our API server needs to have an HTTP connection, as well as a means of allocating resources such as buffers, to hold request data on the server. As the number of connections/requests increases, the memory utilization of the server also increases.

All our servers run with limited (memory) resources, so after some time if the incoming requests continue to rise, then our server will run out of memory and shut itself down with the "OutOfMemory" error. This in turn means all client requests will start failing with a "Server not available" error and the whole client base will be impacted.

The diagram below shows a server that has a memory of 500 concurrent requests and when it reaches 501 requests, the server starts displaying an "OutOfMemory" error and eventually will terminate itself.


Request
Request
Request
Request
500 clients
Above 500 clients
Server crashes after upper limit of requests
Server crashes can handle max 500 requests

If there are already N concurrent requests at the server, then any requests after N should fail with the error message "Server Busy".

Flow control and CPU

To be able to handle more requests concurrently, we need to increase the number of threads (workers), that execute the application code for the request and return the response. Each thread is also a stack of memory, meaning that more threads contribute to more memory.

As the number of threads increases, there are more executors than the number of physical CPUs available. This means that the operating system schedule needs to allocate more CPU cycles to each of these threads, and as a result, more task switching occurs in the operating system.

An increase in the number of task switches can lead to lower performance because it introduces overhead and consumes CPU time. The more frequent the switching of tasks, the more time the operating system must spend on saving and restoring process states, leaving less time for the actual processing. This results in reduced CPU utilization, lower throughput, and longer response times for user requests.

Moreover, task switching also causes cache misses and memory page faults, which can further increase the overhead. These events require additional CPU time to retrieve data from slow memory, further reducing performance.

In general, it is important to minimize the number of task switches to ensure the efficient and optimal performance of the system. This can be achieved by optimizing the scheduling algorithms, reducing the number of processes and threads and optimizing the memory management system.

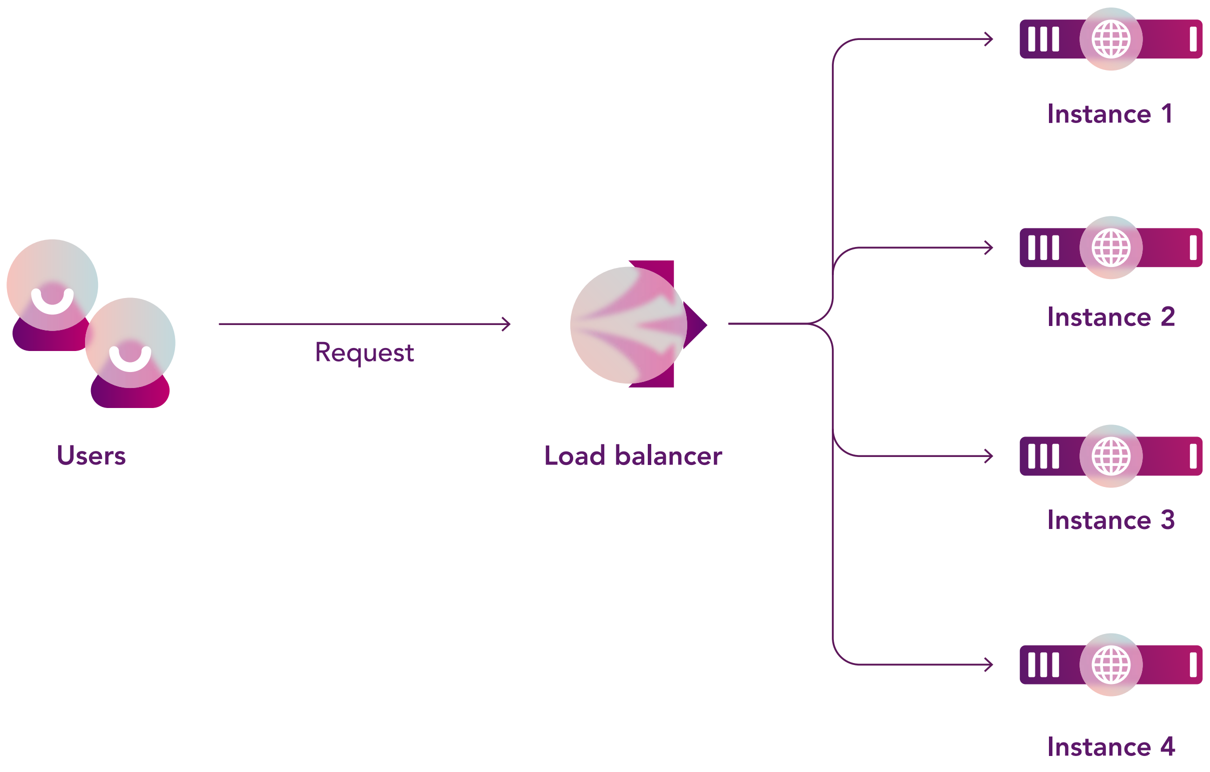
Having a huge number of threads to handle more concurrent requests will instead create more problems.

Request prioritization

HTTP servers normally open a TCP port where users can send requests which are put into a queue where a pool of workers then will pick up each request to for processing and return the response.

Each application will have multiple endpoints to serve the requests, for example in our case we have one endpoint to serve the license (needed to play the video) and along with that, we have another endpoint "/health/ready" to inform the LoadBalancer (client) whether the application is healthy to serve the request or not.

This health endpoint plays a crucial role in horizontally scaled applications where you have multiple instances of your service, with a load-balancer in front of it that distributes the incoming load to each of these instances in a round-robin fashion.



For the load-balancer to send a request to an instance, it needs to know whether it is ready to serve the request and so it uses the "/health/ready" endpoint to check the readiness of the instance. The load-balancer uses this endpoint periodically (let's say, every 5 seconds) to check whether the instance is still ready to serve the request.

In our application at this "/health/ready" we check that all the things required for the application are present: configuration, connection to database and connections to Kafka and then we return READY(200). If any of these are not available, we then return UNAVAILABLE(non-200).

If there are a huge number of requests from the user for a license and at the same time the LoadBalancer also sends a health check request, this health request will either be a queue waiting to be processed, or it might return 503 if the queue is full.

If it is in the queue and takes more time to get a READY response, or the queue is full and a 503 is thrown, the LoadBalancer will consider that the application is not ready to process the requests. The instance is then considered dead, and traffic will not be forwarded to it. This can happen in all the instances of the application and eventually, the service will be unavailable.

We might say we can horizontally scale the application when we have this huge spike load, but this load varies from some hundreds to tens of thousands, or even millions in a matter of seconds. For example, if there is a football match all the users will tune in just minutes before the game, resulting in the huge number of license requests at the same time to our service, resulting in substantial spike loads similar to a DDoS. None of the horizontal scaling methods are able to scale that fast.

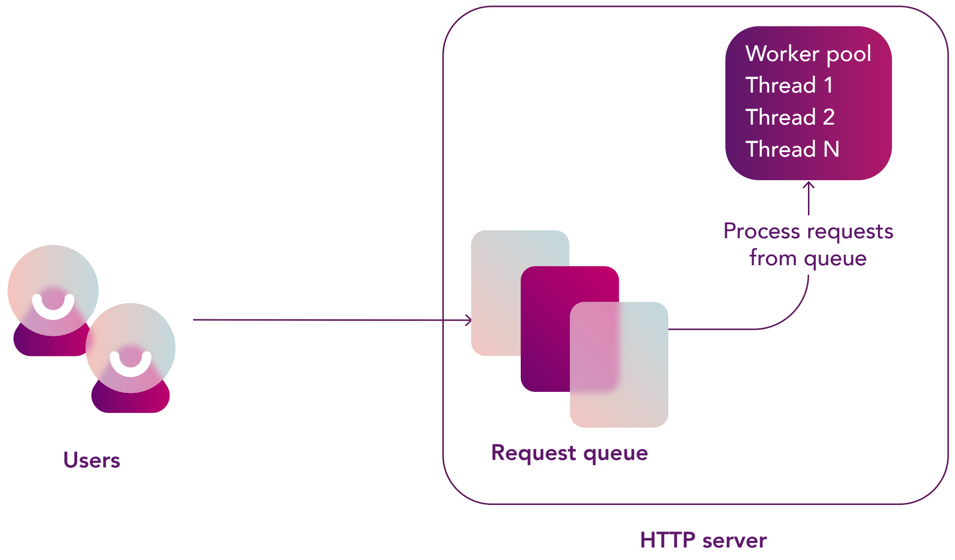
How do current HTTP servers handle spike loads?

Let's see how HTTP servers handle these spike loads instead and what solutions they provide, based on two frameworks that we use to build our HTTP applications:

- Spring Boot
- Vertx

Spring Boot and Vertx both accept HTTP requests processing them with the application logic and returning the response.

Each HTTP server queues the incoming requests before giving them to the worker threads to process them. This approach provides flow control, as an excess number of requests will be waiting in the queue to be processed. The application server will always process N-requests (number threads) at any given movement of time.



The request queue can grow quickly, especially when the number of concurrent users accessing the system increases, resulting in the application running out of memory and crashing. To avoid this situation, each of the HTTP servers provides a configuration for the allowed max queue size. This is so that any new request after the queue is full will not be accepted by the server, resulting in a "server busy response" for the user.

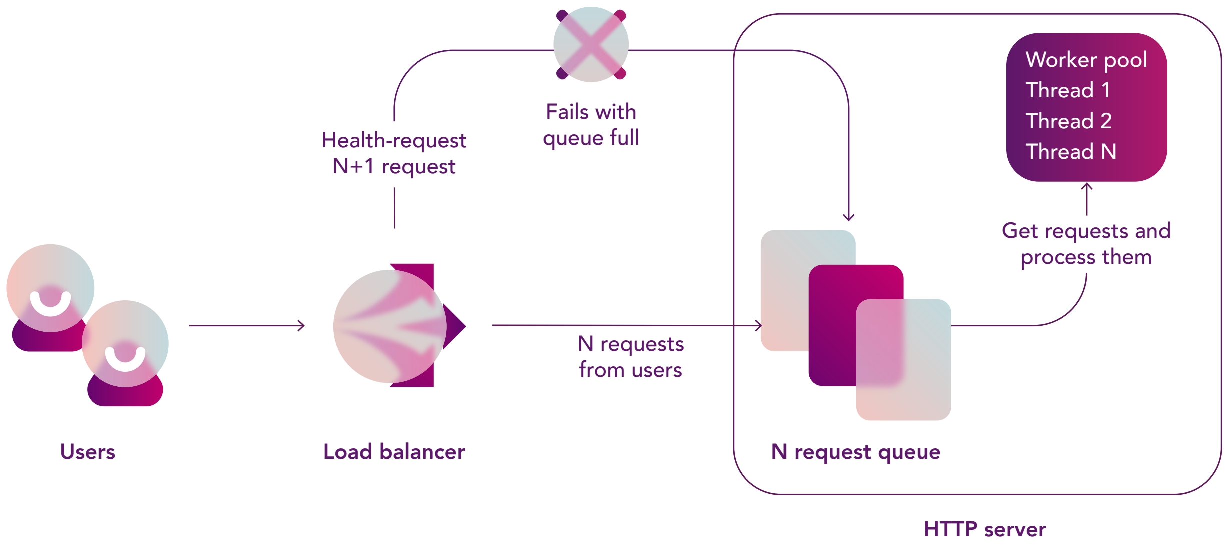
This configuration makes sure that the application server can work without crashing in spike load scenarios, providing flow control for the application.

What are the issues with using HTTP servers?

As shown above, current HTTP servers provide flow control at the container level which treats all the resources in our application at the same level of priority. If users send huge requests for one of the resources, these requests will pile up in the queue and any other resource requests will either fail, timeout, or get processed later on.

In our application, when there are a huge number of license requests – like during the high-profile football matches – any request to the server when the queue is full will be suspended with "SERVER_UNAVAILABLE".

As the number of license requests to our service spikes (due to the high simultaneous viewings), all health check requests start failing either because the request queue in the application server is full, or the requests timeout. This results in all our instances of an application going down, displaying "Service was unavailable" during the most important moments of the stream.

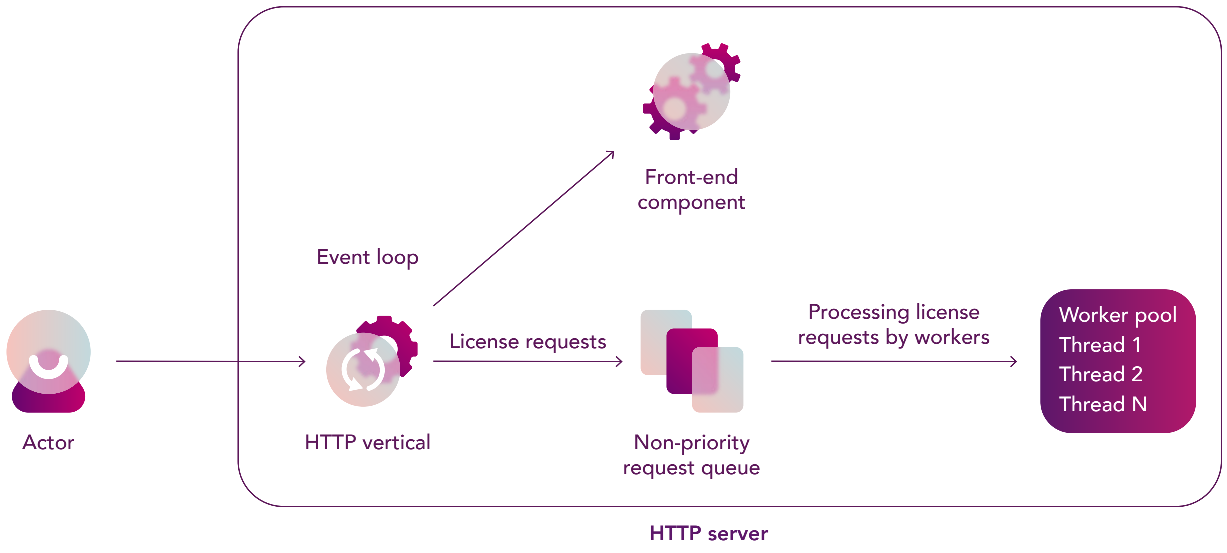


How did we fix HTTP server issues?

To fix the above issues we need to have logic to handle flow control and also provide prioritization of requests so that a huge request for one resource will not starve the other priority resource requests.

We choose the approach of having flow control per resource, so that when requests for licenses spike, we only discard new requests for a license, rather than other resources like health. This approach also provides us with a way to prioritize the requests for different resources as they will be flow controlled separately.

In the case of the Vertx framework, we added an HTTP vertical which is an always-running event loop. It accepts requests, checks what kind of request it is and if it is a priority, it hands it to priority handler threads. If not, it hands it to the non-priority processing worker pool.



In both cases it checks whether individual requests are not beyond the allowed maximum number of concurrent requests. This way there are always N number of active concurrent requests, and the memory of the application is always X amount as the number of concurrent requests is fixed.

For Spring Boot, we cannot use this approach as it follows the thread per request model, meaning that when the Spring Boot container (Jetty) accepts a request, it will hand over it to the container thread. This thread is then responsible for processing the request and generating a response. It's worth noting that it is not possible to switch the processing from this container thread to another thread.

But Spring Boot also provides a way to process the requests asynchronously using *DeferredResult.* With this approach we can accept the request at one thread similar to Vertx and hand over the non-priority requests to second thread pool for processing and priority requests to a third.

We can apply flow control per resource as we are dispatching the request to a separate worker pool.

Right now, we are handling the priority requests like "health" on the event-loop thread itself, but even this isn't a perfect system. A couple things to consider:

- When handling priority requests at the event loop, no other requests will be accepted during the processing of priority requests. If the priority requests are very frequent, then the event loop will always be busy handling these priority requests and others will never be handled.
- Handling priority requests using a separate worker's pool might mean that if these priority requests do not arrive frequently, then these extra worker pools will be underutilized.

Now whenever there is a huge load on our services for license requests, other priority requests (like health) will be handled without failing them, so the application will continue processing all requests at its own pace without crashing.

If you would like to see a working demo application using the Vertx framework for the above solution, click below to be redirected:

GitHub Demo

After fixing the resilience of the application, we noticed when the number of requests spikes, our application handles them gracefully and adds in new instances of the service as part of auto-scaling.

Want to learn more?

When the resilience of your application is braced against the increase in the number of users, the benefit for the user means assured and immediate access to the viewing content, as well as an uninterrupted viewing experience. For the customer, they will be able to provide an impeccable experience to their customers without complaints, negative reviews or bad press.

To learn more about how we approach solving problems in the video entertainment industry, follow us on LinkedIn.

irdeto

Protect. Renew. Empower.