



Máster en Cloud Apps
Desarrollo y despliegue de aplicaciones en la nube

Curso académico 2021/2022

Trabajo de Fin de Máster

Escalabilidad y tolerancia a fallos en Kubernetes

Autores: Isaac Huertas y Virginia Martín
Tutor: Micael Gallego

Índice

Resumen	4
Introducción y objetivos	5
¿Por qué Kubernetes?	5
I. Web stateless	6
Tolerancia a fallos	6
Pruebas de carga	6
Resultados	7
Análisis de los resultados	7
¿Cómo podríamos mejorar estos resultados?	7
Escalabilidad	8
Conclusiones	9
II. Web JCache	10
Tolerancia a fallos	11
Pruebas de carga	11
Resultados	13
Análisis de los resultados	13
¿Cómo podríamos mejorar estos resultados?	13
Conclusiones	14
III. Web MongoRedis	15
Tolerancia a fallos	15
Pruebas de carga	15
Resultados	16
Análisis de los resultados	16
¿Cómo podríamos mejorar estos resultados?	16
Conclusiones	16
IV. Web MongoDB	17
Web MongoDB standalone	17
Tolerancia a fallos	17
Pruebas de carga	17
Resultados	18
Análisis de los resultados	18
¿Cómo podríamos mejorar estos resultados?	18
Conclusiones	18
Web MongoDB replicaset	19
Cambios en el código	19
Tolerancia a fallos	20
Pruebas de carga	20
Resultados	20
Análisis de los resultados	20
¿Cómo podríamos mejorar estos resultados?	20
Conclusiones	20
V. Web MySQL	21
Web MySQL standalone	21
Tolerancia a fallos	21
Pruebas de carga	21
Resultados	22
Análisis de los resultados	22
¿Cómo podríamos mejorar estos resultados?	22
Conclusiones	22
Web MySQL replicaset	23
Cambios en el código	23
Tolerancia a fallos	24
Pruebas de carga	24
Resultados	25
Análisis de los resultados	25
¿Cómo podríamos mejorar estos resultados?	25

Conclusiones	25
VI. RabbitMQ	26
Configuración	26
Escalabilidad	28
HPA	28
KEDA	30
Conclusiones	32
CONCLUSIONES	33
TRABAJOS FUTUROS	33
BIBLIOGRAFÍA	33

Resumen

Este trabajo de fin de Máster contiene un estudio sobre la escalabilidad y tolerancia a fallos de distintas aplicaciones desplegadas en un ecosistema Kubernetes.

En la actualidad todos somos conscientes que nuestras aplicaciones tienen que estar siempre disponibles y con un rendimiento aceptable, aunque tengan una alta carga, ya que, de lo contrario, podemos perder clientes o nos puede ocasionar pérdidas millonarias porque nuestros servicios no están disponibles o porque no cumplimos con la SLA.

Para que esto no ocurra, es importante estudiar y entender todo lo que podemos hacer para intentar minimizar todos estos posibles riesgos.

Las aplicaciones objeto de estudio sobre las que versa el trabajo son aquellas más utilizadas en entornos productivos: web Stateless, web con JCache, web con persistencia de datos relacional (MySQL), web con persistencia de datos no relacional (MongoDB) y sistemas de colas de mensajería RabbitMQ.

Esta memoria se compone de las siguientes secciones: Introducción y objetivos, donde se describen las metas perseguidas; una sección principal, compuesta por cinco capítulos en los que estudiamos las diferentes aplicaciones arriba mencionadas, las ventajas y desventajas de cada una de ellas, y posibles mejoras y conclusiones de cada modelo. A continuación, podemos encontrar las conclusiones generales a las que llegamos una vez finalizado el estudio y los trabajos futuros que se derivan del mismo. La última sección contiene la bibliografía en la que nos hemos apoyado.

Introducción y objetivos

Con el aumento de la complejidad en las aplicaciones, nacen diferentes plataformas que proveen de una serie de elementos software para acelerar el tiempo de desarrollo, haciendo más fácil el desplegar, escalar y administrar las propias aplicaciones.

Dentro de estas plataformas, la que hemos elegido para llevar a cabo el desarrollo de nuestro TFM es Kubernetes.

¿Por qué Kubernetes?

Porque Kubernetes es un sistema de código abierto que permite desplegar, escalar y gestionar aplicaciones en contenedores en cualquier lugar. Automatiza las tareas operativas de la gestión de los contenedores e integra comandos para desplegar aplicaciones, aplicarles cambios, ampliarlas o reducirlas según las necesidades del momento, monitorizarlas y realizar muchas más acciones. En definitiva, hace que resulte más fácil gestionar las aplicaciones.

Esto ofrece grandes ventajas, como son:

- Operaciones automáticas
- Abstracción de infraestructura
- Monitorización del estado de los servicios

Teniendo en cuenta que Kubernetes nos permite crear aplicaciones fáciles de gestionar y desplegar en cualquier lugar, se ha querido llevar a cabo un estudio de forma empírica, de varios de los elementos software que provee para analizar en detalle la escalabilidad y tolerancia a fallos que ofrecen cada uno de ellos.

Por lo tanto, los objetivos principales de este TFM son los siguientes:

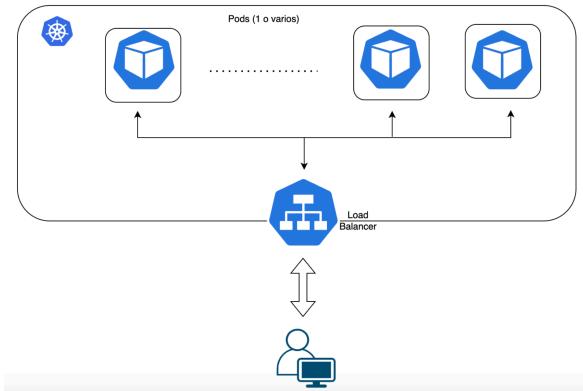
- Diseño de estrategias de chaos testing para introducir fallos en las aplicaciones
- Medir el impacto que tienen dichos fallos en el rendimiento de las aplicaciones
- Ser capaces de proponer mecanismos de mejora para aumentar la tolerancia a fallos de las aplicaciones
- Estudiar mecanismos de autoescalado para intentar reducir el impacto de los fallos de manera automática en base a ciertas reglas y/o eventos

Para cubrir estos objetivos, a continuación, vamos a describir en detalle cada uno de las arquitecturas sobre los que hemos estudiado la escalabilidad y tolerancia a fallos.

I. Web stateless

El primer tipo de aplicación sometida a estudio es la web stateless. Hemos tomado como ejemplo, la *web de gatitos* que hemos visto durante el desarrollo de la asignatura.

La arquitectura de esta aplicación es la siguiente:



Partimos de un escenario inicial en el que tenemos una única réplica de la aplicación. A partir de aquí, hemos analizado:

- Tolerancia a fallos
 - Medir el rendimiento con una única réplica, simulando carga de procesamiento con y sin chaos-monkey.
 - Escalar a dos réplicas y repetir las pruebas anteriores.
- Escalabilidad
 - Aplicar un HPA (Horizontal Pod Autoescaler) en función de la carga de la CPU.

Tolerancia a fallos

Pruebas de carga

Para estudiar la tolerancia a fallos de la aplicación, hemos realizado pruebas de carga con la herramienta JMeter. Para ello, hemos diseñado dos test plans diferentes:

- Test plan con 500 usuarios y Ramp-up de 3 minutos
- Test plan con 1000 usuarios y Ramp-up de 3 minutos

Cada uno de los test plans los hemos ejecutado con y sin chaos-monkey, primero con una única réplica de la aplicación y, posteriormente, escalando el número de réplicas a dos.

Resultados

A continuación, se muestra una comparativa del resultado de las pruebas:



Análisis de los resultados

- Como se puede observar, cuando no hay caos ninguna de las peticiones falla ni aunque se duplique el tráfico, pasando de 500 a 1000 usuarios.
- Al introducir chaos-monkey las peticiones empiezan a fallar. El porcentaje de fallo es muy pequeño, dado que la aplicación es stateless, no maneja estados, no tiene persistencia de datos y tarda muy poco en levantar un nuevo pod cuando este se mata.
- El porcentaje de fallo al introducir una segunda réplica disminuye en comparación con el que se obtiene haciendo uso de una sola réplica.

¿Cómo podríamos mejorar estos resultados?

- Aumentar el número de réplicas la aplicación, puesto que cuanto mayor sea el número de pods, menor será el número de errores que puedan producirse.
- Introducir algún service mesh, permitiendo reintento de peticiones fallidas.
- Implementar un patrón de Circuit Breaker para dejar de enviar peticiones a un servicio que esté fallando.

Escalabilidad

Para estudiar la escalabilidad, vamos a aplicar un HPA (Horizontal Pod Autoescaler), que es uno de los mecanismos de autoescalado que nos ofrece Kubernetes. En este caso, aplicaremos un HPA que añada réplicas en función de la carga de la CPU del pod.

Partimos de un solo pod:

```
Context: default
Cluster: default
User: default
K9s Rev: v0.24.15 → v0.26.6
K8s Rev: v1.24.6+k3s1
CPU: 3%↑
MEM: 71%↑

<0> all    <a>    Attach    <l> ...
<1> default <ctrl-d> Delete    <p>
<d>    Describe  <shift> ...
<e>    Edit     <s>
<?>   Help     <f>
<ctrl-k> Kill    <y>
<ctrl-s> Save   <ctrl-z> Exit
                                     Warning Memory level!

Pods(default)[1]
NAME      PF  READY  RESTARTS  STATUS    CPU  MEM  %CPU/R  %CPU/L  %MEM/R  %MEM/L  IP          NODE
webgatos-dd5c6b4d4-c55ww  •  1/1    0  Running  0  0  0  0  0  0  0  10.42.1.89  k8s-agent-sm
```

Simulamos carga:

```
kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://167.235.217.41:5000; done"
```

Vemos como el número de réplicas va aumentando según la aplicación va recibiendo más carga, hasta llegar a 10 réplicas, que es el máximo que hemos definido en el HPA:

```
Context: default
Cluster: default
User: default
K9s Rev: v0.24.15 → v0.26.6
K8s Rev: v1.24.6+k3s1
CPU: 5%
MEM: 71%

<0> all    <a>    Attach    <l> ...
<1> default <ctrl-d> Delete    <p>
<d>    Describe  <shift> ...
<e>    Edit     <s>
<?>   Help     <f>
<ctrl-k> Kill    <y>
<ctrl-s> Save   <ctrl-z> Exit
                                     Warning Memory level!

Pods(default)[5]
NAME      PF  READY  RESTARTS  STATUS    CPU  MEM  %CPU/R  %CPU/L  %MEM/R  %MEM/L  IP          NODE
load-generator  •  1/1    0  Running  0  n/a  n/a  n/a  n/a  n/a  10.42.1.47  k8s-agent
webgatos-dd5c6b4d4-5xhhm  •  1/1    0  Running  0  0  0  0  0  0  0  10.42.1.227  k8s-agent
webgatos-dd5c6b4d4-c55ww  •  1/1    0  Running  50  16  25  25  25  25  12  10.42.1.89  k8s-agent
webgatos-dd5c6b4d4-kdhnx  •  1/1    0  Running  0  0  0  0  0  0  0  10.42.1.247  k8s-agent
webgatos-dd5c6b4d4-vlrt6  •  1/1    0  Running  0  0  0  0  0  0  0  10.42.0.132  k8s-agent

C:\Users\Virginia>kubectl get hpa webgatos --watch
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
webgatos  Deployment/webgatos  <unknown>/10%  1           10          1           30s
webgatos  Deployment/webgatos  1%/10%       1           10          1           45s
webgatos  Deployment/webgatos  50%/10%      1           10          1           75s
webgatos  Deployment/webgatos  50%/10%      1           10          4           90s

Context: default
Cluster: default
User: default
K9s Rev: v0.24.15 → v0.26.6
K8s Rev: v1.24.6+k3s1
CPU: 13%
MEM: 75%

<0> all    <a>    Attach    <l> ...
<1> default <ctrl-d> Delete    <p>
<d>    Describe  <shift> ...
<e>    Edit     <s>
<?>   Help     <f>
<ctrl-k> Kill    <y>
<ctrl-s> Save   <ctrl-z> Exit
                                     Warning Memory level!

Pods(default)[11]
NAME      PF  READY  RESTARTS  STATUS    CPU  MEM  %CPU/R  %CPU/L  %MEM/R  %MEM/L  IP          NODE
load-generator  •  1/1    0  Running  80  n/a  n/a  n/a  n/a  n/a  10.42.1.47  k8s-agent
webgatos-dd5c6b4d4-9zg2m  •  1/1    0  Running  19  16  19  9  25  12  10.42.1.227  k8s-agent
webgatos-dd5c6b4d4-9bims  •  1/1    0  Running  23  16  23  11  25  12  10.42.0.41  k8s-agent
webgatos-dd5c6b4d4-9zg2f  •  1/1    0  Running  0  0  0  0  0  0  0  10.42.0.139  k8s-agent
webgatos-dd5c6b4d4-c55ww  •  1/1    0  Running  15  16  15  7  25  12  10.42.1.89  k8s-agent
webgatos-dd5c6b4d4-gk56f  •  1/1    0  Running  0  0  0  0  0  0  0  10.42.0.244  k8s-agent
webgatos-dd5c6b4d4-kdhnx  •  1/1    0  Running  20  16  20  10  25  12  10.42.1.247  k8s-agent
webgatos-dd5c6b4d4-t646m  •  1/1    0  Running  0  0  0  0  0  0  0  10.42.1.140  k8s-agent
webgatos-dd5c6b4d4-vlrt6  •  1/1    0  Running  17  16  17  8  25  12  10.42.0.132  k8s-agent
webgatos-dd5c6b4d4-wkdrp  •  1/1    0  Running  0  0  0  0  0  0  0  10.42.1.130  k8s-agent
webgatos-dd5c6b4d4-z95pj  •  1/1    0  Running  0  0  0  0  0  0  0  10.42.0.118  k8s-agent

C:\Users\Virginia>kubectl get hpa webgatos --watch
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
webgatos  Deployment/webgatos  <unknown>/10%  1           10          1           30s
webgatos  Deployment/webgatos  1%/10%       1           10          1           45s
webgatos  Deployment/webgatos  50%/10%      1           10          1           75s
webgatos  Deployment/webgatos  50%/10%      1           10          4           90s
webgatos  Deployment/webgatos  78%/10%      1           10          5           105s
webgatos  Deployment/webgatos  34%/10%      1           10          10          2m
webgatos  Deployment/webgatos  18%/10%      1           10          10          2m15s
webgatos  Deployment/webgatos  17%/10%      1           10          10          2m30s
webgatos  Deployment/webgatos  17%/10%      1           10          10          2m45s
```

Una vez paramos de simular carga, vemos cómo comienza a desescalar hasta quedarse en 1 réplica, que es el mínimo que tenemos definido en el HPA:

```
webgatos-dd5c6b4d4-5xhhm • 1/1     0 Terminating 1 16   1   0   25   12 10.42.1.227 k8s-
webgatos-dd5c6b4d4-9bjms • 1/1     0 Terminating 1 16   1   0   25   12 10.42.0.41 k8s-
webgatos-dd5c6b4d4-9zg2f • 1/1     0 Terminating 1 16   1   0   25   12 10.42.0.139 k8s-
webgatos-dd5c6b4d4-c55ww • 1/1     0 Terminating 1 16   1   0   25   12 10.42.1.89 k8s-
webgatos-dd5c6b4d4-gk56f • 1/1     0 Terminating 1 16   1   0   25   12 10.42.0.244 k8s-
webgatos-dd5c6b4d4-kdhnx • 1/1     0 Terminating 1 16   1   0   25   12 10.42.1.247 k8s-
webgatos-dd5c6b4d4-t646m • 1/1     0 Terminating 1 16   1   0   25   12 10.42.1.140 k8s-
webgatos-dd5c6b4d4-vlrt6 • 1/1     0 Terminating 1 16   1   0   25   12 10.42.0.132 k8s-
webgatos-dd5c6b4d4-wkdrp • 1/1     0 Running    1 16   1   0   25   12 10.42.1.130 k8s-
webgatos-dd5c6b4d4-z95pj • 1/1     0 Terminating 1 16   1   0   25   12 10.42.0.118 k8s-
```



```
C:\Users\Virginia>kubectl get hpa webgatos --watch
NAME      REFERENCE      TARGETS      MINPODS      MAXPODS      REPLICAS      AGE
webgatos  Deployment/webgatos  1%/10%       1           10          10          4m38s
webgatos  Deployment/webgatos  1%/10%       1           10          10          9m1s
webgatos  Deployment/webgatos  1%/10%       1           10          1           9m16s
```

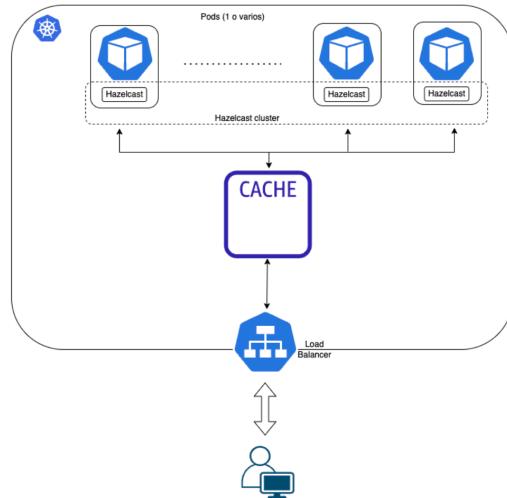
Conclusiones

- Es una de las soluciones más sencillas de implementar.
- Es fácilmente escalable al no gestionar estados.
- Al escalar por métricas propias de Kubernetes es importante establecer límites de recursos en los pods..
- La tolerancia a fallos es mayor cuantas más réplicas de nuestra aplicación tengamos.

II. Web JCache

En el siguiente ejemplo vamos a estudiar la escalabilidad y tolerancia a fallos de una web con caché, utilizando JCache con Spring y un clúster de Hazelcast para conseguir que la caché sea compartida entre todos los nodos del clúster.

La arquitectura de la aplicación es la siguiente:



Partimos de un escenario inicial en el que tenemos una única réplica de la aplicación. A partir de aquí, hemos analizado:

- Tolerancia a fallos
 - Medir el rendimiento con una única réplica, simulando carga de procesamiento con y sin chaos-monkey.
 - Escalar a dos réplicas, montando un clúster de Hazelcast para que los pods puedan autodescubrirse entre sí, consiguiendo una caché compartida, y repetir las pruebas anteriores.

Una vez tenemos instalada la aplicación, antes de iniciar las pruebas de carga, comprobamos el correcto funcionamiento de la caché. Para ello, realizamos mediante Postman una petición tipo GET, obteniendo el siguiente tiempo de respuesta:

KEY	VALUE	DESCRIPTION
Key	Value	Description

```
1 [ { "id": 1, "user": "Pepe", "title": "Vendo moto", "text": "Barata, barata" }, { "id": 2, "user": "Juan", "title": "Compro coche", "text": "Pago bien" }, { "id": 3, "user": "Luis", "title": "Necesito coche", "text": "Por favor" } ]
```

Volvemos a realizar la misma petición GET y vemos que el tiempo de respuesta mejora sustancialmente:

The screenshot shows the Postman interface with a successful GET request to `http://localhost:51921/posts/`. The response status is 200 OK, time is 60 ms, and size is 1.05 KB. The response body is a JSON array containing three posts:

```
[{"id": 1, "user": "Pope", "title": "Vendo moto", "text": "Barata, barata"}, {"id": 2, "user": "Juan", "title": "Compro coche", "text": "Pago bien"}, {"id": 3, "user": "Luis", "text": ""}]
```

Repetimos peticiones GET, observando tiempos de respuesta similares al de la segunda petición. Y si miramos los logs de la aplicación, vemos una única traza de la primera vez que ha tenido que ir a la MySQL a consultar los datos, pues las siguientes peticiones han devuelto los datos desde la caché:

```
2023-05-15 18:22:30.316 TRACE 1 --- [main] o.h.type.descriptor.sql.BasicBinder : binding parameter [1] as [VARCHAR] - [Montaña]
2023-05-15 18:22:30.316 TRACE 1 --- [main] o.h.type.descriptor.sql.BasicBinder : binding parameter [2] as [VARCHAR] - [Busco bici]
2023-05-15 18:22:30.316 TRACE 1 --- [main] o.h.type.descriptor.sql.BasicBinder : binding parameter [3] as [VARCHAR] - [Edu]
2023-05-15 18:22:30.316 TRACE 1 --- [main] o.h.type.descriptor.sql.BasicBinder : binding parameter [4] as [BIGINT] - [10]
2023-05-15 18:22:30.316 TRACE 1 --- [main] org.hibernate.SQL : select posts0_.id as id1_, posts0_.user as user2_, posts0_.title as title3_0_, posts0_.text as user4_0_
2023-05-15 18:22:30.316 TRACE 1 --- [main] org.hibernate.SQL : from post post0
2023-05-15 18:22:30.758 INFO 1 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context path ''
2023-05-15 18:22:30.767 INFO 1 --- [main] e.s.codeurj.c.board.Application : Started Application in 10.022 seconds (JVM running for 10.664)
2023-05-15 18:22:30.831 INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Spring DispatcherServlet 'dispatcherServlet'
2023-05-15 18:22:30.831 INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms
2023-05-15 18:22:35.437 INFO 1 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : [10.42.1.164]:5701 [dev] [4.2.7] Added cache config: CacheConfig{name='posts', managerPrefix='/hz', inMemoryFormat=BINARY, backupCount=1, hibernateCacheMode=HARD_SYNC, maxEntries=1000, sync=true}
2023-05-15 18:22:35.437 INFO 1 --- [nio-8080-exec-1] com.hazelcast.cache.impl.CacheService : [10.42.1.164]:5701 [dev] [4.2.7] Initializing cluster partition table arrangement...
2023-05-15 18:24:00.428 INFO 1 --- [nio-8080-exec-3] com.hazelcast.cache.impl.CacheService : [10.42.1.164]:5701 [dev] [4.2.7] Initializing cluster partition table arrangement...
2023-05-15 18:24:00.682 DEBUG 1 --- [nio-8080-exec-3] org.hibernate.SQL : select
2023-05-15 18:24:00.682 DEBUG 1 --- [nio-8080-exec-3] org.hibernate.SQL : posts0_.id as id1_, posts0_.user as user2_, posts0_.title as title3_0_, posts0_.text as user4_0_
2023-05-15 18:24:00.682 DEBUG 1 --- [nio-8080-exec-3] org.hibernate.SQL : from post post0
2023-05-15 18:24:00.682 DEBUG 1 --- [nio-8080-exec-3] org.hibernate.SQL : where posts0_.id = ?
2023-05-15 18:24:00.682 DEBUG 1 --- [nio-8080-exec-3] org.hibernate.SQL : order by posts0_.id asc
2023-05-15 18:24:00.682 DEBUG 1 --- [nio-8080-exec-3] org.hibernate.SQL : limit ?
```

Tolerancia a fallos

Pruebas de carga

Para estudiar la tolerancia a fallos de la aplicación, hemos realizado pruebas de carga con la herramienta JMeter. Para ello, hemos diseñado cuatro test plans diferentes:

- Test plan con 500 usuarios y Ramp-up de 3 minutos sin caos
- Test plan con 1000 usuarios y Ramp-up de 3 minutos sin caos
- Test plan con 500 usuarios y Ramp-up de 5 minutos con caos
- Test plan con 1000 usuarios y Ramp-up de 5 minutos con caos

NOTA: En las pruebas con chaos, hemos extendido el tiempo de Ramp-up al observarse que el tiempo de recuperación de la MySQL es elevado.

Para comprobar el correcto comportamiento de la caché compartida tras el escalado a dos réplicas, hemos revisado los logs del primer pod, comprobando que descubre a los dos miembros del clúster de Hazelcast:

```
Members {size:2, ver:2} [
    Member [10.42.1.114]:5701 - 3ecd8f94-cb46-41d0-bc36-911ccda46408
    Member [10.42.0.45]:5701 - ad24b4bf-f9e3-43f3-8e3d-b5cb0b782253 this
]
```

Y posteriormente, los logs del segundo pod para comprobar lo mismo que en el caso anterior:

```
Members {size:2, ver:2} [
    Member [10.42.1.114]:5701 - 3ecd8f94-cb46-41d0-bc36-911ccda46408 this
    Member [10.42.0.45]:5701 - ad24b4bf-f9e3-43f3-8e3d-b5cb0b782253
]
```

Resultados

A continuación se muestra una comparativa del resultado de las pruebas:



Análisis de los resultados

- Como se puede observar, cuando no hay caos ninguna de las peticiones falla ni aunque se duplique el tráfico, pasando de 500 a 1000 usuarios.
- El porcentaje de fallo al introducir una segunda réplica disminuye en comparación con el que se obtiene haciendo uso de una sola réplica.
- En cuanto a tiempos de respuesta de las peticiones a lo largo del test, se nota una mejora de los mismo por el uso de una caché compartida.

¿Cómo podríamos mejorar estos resultados?

- Aumentar el número de réplicas la aplicación, puesto cuanto mayor sea el número de pods, menor será el número de errores que puedan producirse.
- Introducir algún service mesh, permitiendo reintento de peticiones fallidas.
- Implementar un patrón de Circuit Breaker para dejar de enviar peticiones a un servicio que esté fallando.

Además, en JMeter también podemos comprobar que la caché funciona como se esperaba, al disminuirse el tiempo que tardan en responder las peticiones a partir de la segunda vez que se solicitan:

Muestra # ▾	Tiempo de comienzo	Nombre del hilo	Etiqueta	Tiempo de Muestra (ms)	Estado	Bytes	Sent Bytes	Latency	Connect Time(ms)
1	20:50:55.533	Usuarios web-jacache	Get posts	572	✓	1082	124	572	1
2	20:50:56.105	Usuarios web-jacache	Get post 1	98	✓	261	125	98	0
3	20:50:56.203	Usuarios web-jacache	Get post 2	62	✓	258	125	62	0
4	20:50:56.265	Usuarios web-jacache	Get post 3	62	✓	261	125	62	0
5	20:50:56.327	Usuarios web-jacache	Get post 4	64	✓	258	125	63	0
6	20:50:56.381	Usuarios web-jacache	Get post 5	64	✓	263	125	63	0
7	20:50:56.455	Usuarios web-jacache	Get post 6	63	✓	263	125	63	0
8	20:50:56.510	Usuarios web-jacache	Get post 7	67	✓	267	125	67	0
9	20:50:56.585	Usuarios web-jacache	Get post 8	57	✓	271	125	57	0
10	20:50:56.642	Usuarios web-jacache	Get post 9	63	✓	259	125	63	0
11	20:50:56.643	Usuarios web-jacache	Get posts	162	✓	1082	124	162	1
12	20:50:56.705	Usuarios web-jacache	Get post 1	56	✓	261	125	55	0
13	20:50:56.705	Usuarios web-jacache	Get post 10	71	✓	256	125	71	0
14	20:50:56.761	Usuarios web-jacache	Get post 2	53	✓	258	125	52	0
15	20:50:56.813	Usuarios web-jacache	Get post 3	52	✓	261	125	52	0
16	20:50:56.865	Usuarios web-jacache	Get post 4	53	✓	258	125	53	0
17	20:50:56.910	Usuarios web-jacache	Get post 5	53	✓	263	125	53	0
18	20:50:56.971	Usuarios web-jacache	Get post 6	53	✓	263	125	53	0
19	20:50:57.032	Usuarios web-jacache	Get post 7	54	✓	267	125	54	0
20	20:50:57.076	Usuarios web-jacache	Get post 8	51	✓	271	125	55	0
21	20:50:57.134	Usuarios web-jacache	Get post 9	52	✓	259	125	52	0
22	20:50:57.186	Usuarios web-jacache	Get post 10	59	✓	255	126	59	0
23	20:50:57.532	Usuarios web-jacache	Get posts	146	✓	1082	124	146	1

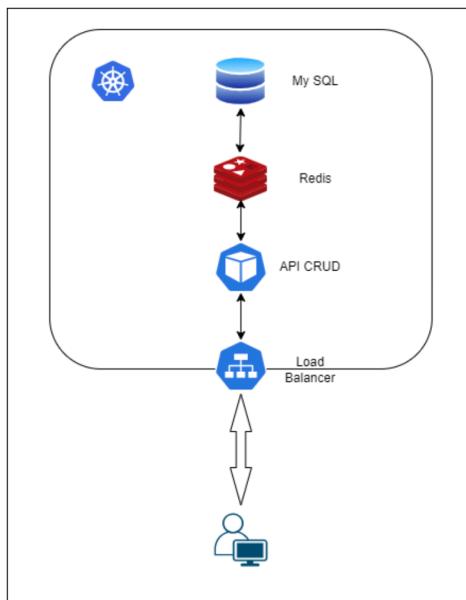
Conclusiones

- El uso de una caché compartida hace que el tiempo de respuesta de las peticiones GET disminuya notablemente, lo que mejora la performance de la aplicación.
- La escalabilidad es más compleja de implementar que en una web stateless por el uso de la caché compartida.
- La tolerancia a fallos es mayor cuantas más réplicas de nuestra aplicación tengamos.

III. Web MongoRedis

En el siguiente ejemplo vamos a estudiar la escalabilidad y tolerancia a fallos de una API cuya persistencia de datos se hace en una base de datos no relacional y por delante le ponemos una caché redis. Como base de datos no relacional hemos elegido MongoDB.

La arquitectura de la aplicación es la siguiente:



Partimos de un escenario inicial en el que tenemos una única réplica de la aplicación. A partir de aquí, hemos analizado:

- Tolerancia a fallos
 - Medir el rendimiento con una única réplica, simulando carga de procesamiento con y sin chaos-monkey

Tolerancia a fallos

Pruebas de carga

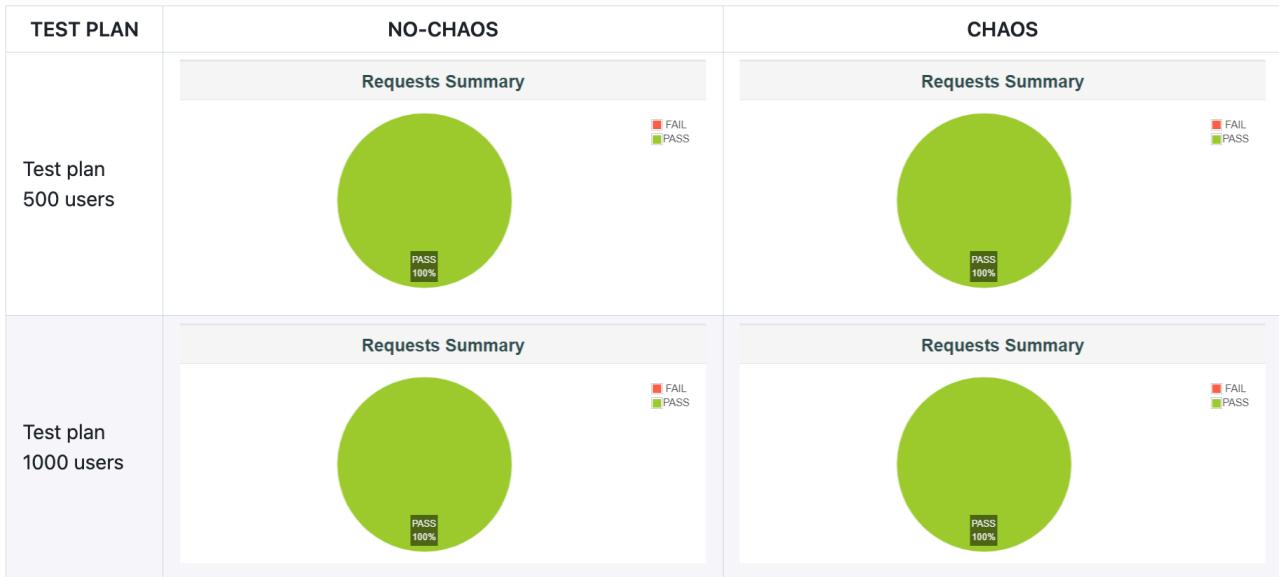
Para estudiar la tolerancia a fallos de la aplicación, hemos realizado pruebas de carga con la herramienta JMeter. Para ello, hemos diseñado dos test plans diferentes:

- Test plan con 500 usuarios y Ramp-up de 5 minutos
- Test plan con 1000 usuarios y Ramp-up de 5 minutos

Cada uno de los test plans los hemos ejecutado con y sin chaos-monkey.

Resultados

A continuación se muestra una comparativa del resultado de las pruebas:



Análisis de los resultados

- Como se puede observar, cuando no hay caos el despliegue no falla aunque haya 1000 usuarios concurrentes.
- Al habilitar chaos monkey tampoco conseguimos hacer la aplicación pero si nos fijamos en detalle en los resultado de los tests vemos que el tiempo medio de respuesta es muy alto.

Requests	Executions				Response Times (ms)							Throughput		Network (KB/sec)	
	Label	#Samples	FAIL	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent	
Total	2000	0	0.00%	716.67	42	12447	248.00	369.10	5750.70	10407.73	6.64	1306.70	1.43		
Get posts	1000	0	0.00%	1379.67	245	12447	280.00	5750.40	8721.60	11023.64	3.32	1305.70	0.42		
Post new post	1000	0	0.00%	53.66	42	599	46.00	52.00	80.00	222.95	3.32	1.21	1.01		

¿Cómo podríamos mejorar estos resultados?

- Para reducir la latencia de las peticiones tendríamos que montar la replicación de la caché de Redis.

NOTA: No hemos podido realizar este estudio ya que el sentinel de Redis requiere 4 GB de RAM y nuestras máquinas del clúster no disponen de dicha memoria.

Conclusiones

- Los resultados de las pruebas mejoraríaan al introducir un mecanismo de replicación en la caché, aun así, con una única réplica notamos una mejora en los tiempos de respuesta de las peticiones GET respecto a las pruebas realizadas sólo contra la base de datos.

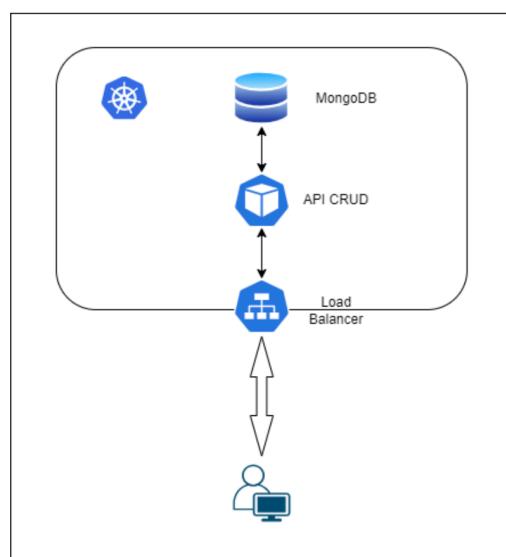
IV. Web MongoDB

En el caso de las webs con persistencia de datos basadas en el modelo no relacional, hemos estudiado dos escenarios con MongoDB: uno con una arquitectura standalone, con una sola réplica de base de datos y otro con un mecanismo de replicación.

Web MongoDB standalone

En este ejemplo vamos a estudiar la tolerancia a fallos de una API cuya persistencia de datos se hace en una base de datos no relacional sin replicación.

La arquitectura de la aplicación sometida a estudio es la siguiente:



Partimos de un escenario inicial en el que tenemos una única réplica de base de datos. A partir de aquí, hemos analizado:

- Tolerancia a fallos
 - Medir el rendimiento con una única réplica, simulando carga de procesamiento con y sin chaos-monkey.

Tolerancia a fallos

Pruebas de carga

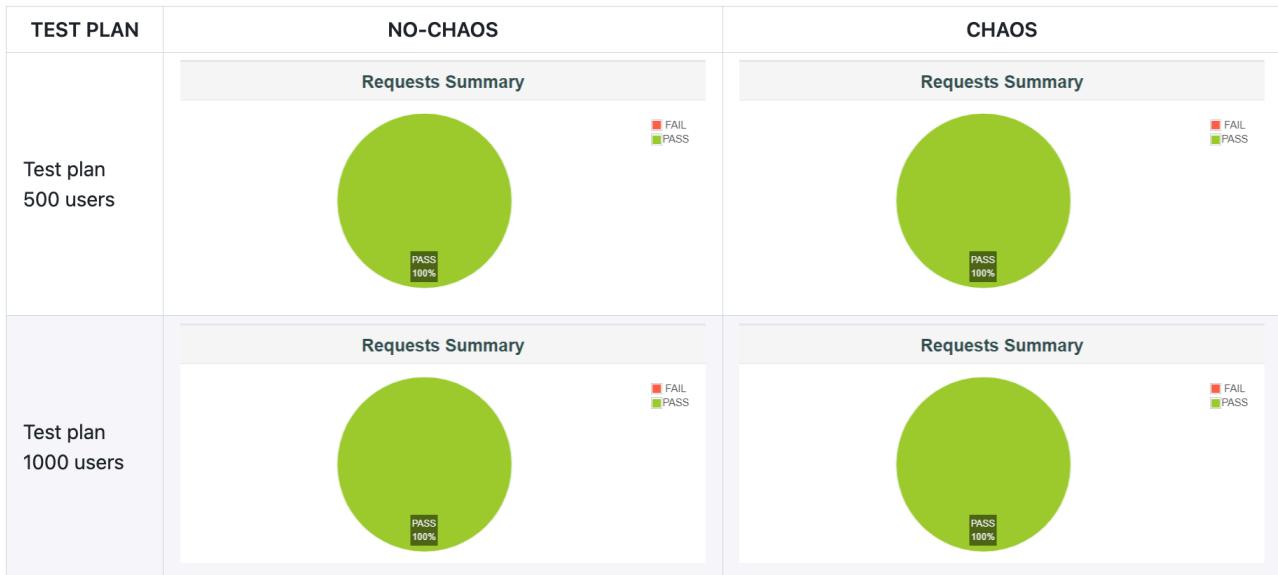
Para estudiar la tolerancia a fallos de la aplicación, hemos realizado pruebas de carga con la herramienta JMeter. Para ello, hemos diseñado dos test plans diferentes:

- Test plan con 500 usuarios y Ramp-up de 10 minutos
- Test plan con 1000 usuarios y Ramp-up de 10 minutos

Cada uno de los test plans los hemos ejecutado con y sin chaos-monkey.

Resultados

A continuación, vamos a comparar el resultado de las diferentes pruebas realizadas:



Análisis de los resultados

- En ninguna de las pruebas provocamos fallos. Esto se debe a que nuestra base de datos MongoDB es muy pequeña y se vuelve a recuperar en tiempos muy bajos.

¿Cómo podríamos mejorar estos resultados?

- Aunque los resultados ya son muy buenos, no es despliegue apto para producción porque no tiene tolerancia a fallos. Por lo que deberíamos añadir a este despliegue la replicación de la base de datos.

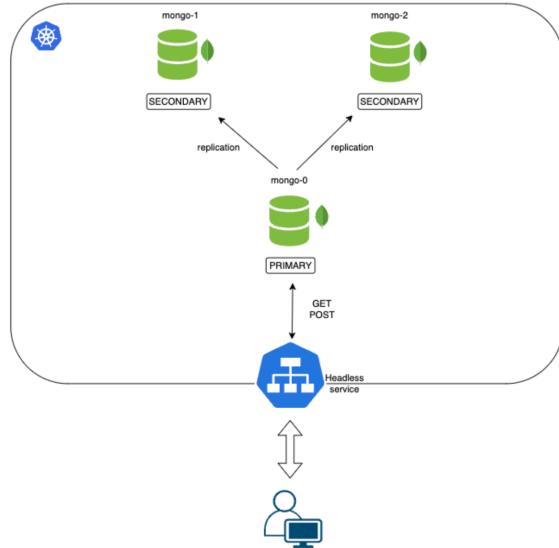
Conclusiones

- Las bases de datos MongoDB tienen un tiempo de recuperación mínimo por lo que es difícil hacerlas fallar.
- No es una arquitectura apta para producción, ya que carece de replicación y no es tolerante a fallos.

Web MongoDB replicaset

En el siguiente ejemplo, vamos a someter a estudio una aplicación que tiene persistencia de datos no relacional, mediante MongoDB. Una vez hemos estudiado las limitaciones de tener una sola réplica, vamos a estudiar una mejora que nos permite una mayor tolerancia a fallos usando el recurso StatefulSet de Kubernetes.

La arquitectura de la aplicación sometida a estudio es la siguiente:



Cambios en el código

Para hacer este cambio de arquitectura, tenemos que adaptar nuestro código para que la aplicación pueda conectarse a todos los miembros del clúster de MongoDB.

En el application.properties de nuestra aplicación, haremos uso de una nueva propiedad de SpringBoot (spring.data.mongodb.uri), en la cual le indicaremos la cadena de conexión que especifica todos los miembros del clúster de MongoDB.

```
spring.data.mongodb.uri=${SPRING_DATA_MONGODB_URI}
```

Para pasarle el valor a esta propiedad, lo haremos mediante una variable de entorno en el manifiesto de despliegue que contenga la cadena de conexión:

```
env:  
- name: SPRING_DATA_MONGODB_URI  
  value: mongodb://mongo-0.mongodb-svc,mongo-1.mongodb-svc,mongo-2.mongodb-svc?replicaSet=ns0/posts
```

Una vez hemos montado el escenario con replicación, hemos analizado:

- Tolerancia a fallos
 - Medir el rendimiento con un replicaset (3 réplicas de BD), simulando carga de procesamiento con y sin chaos-monkey.

Tolerancia a fallos

Pruebas de carga

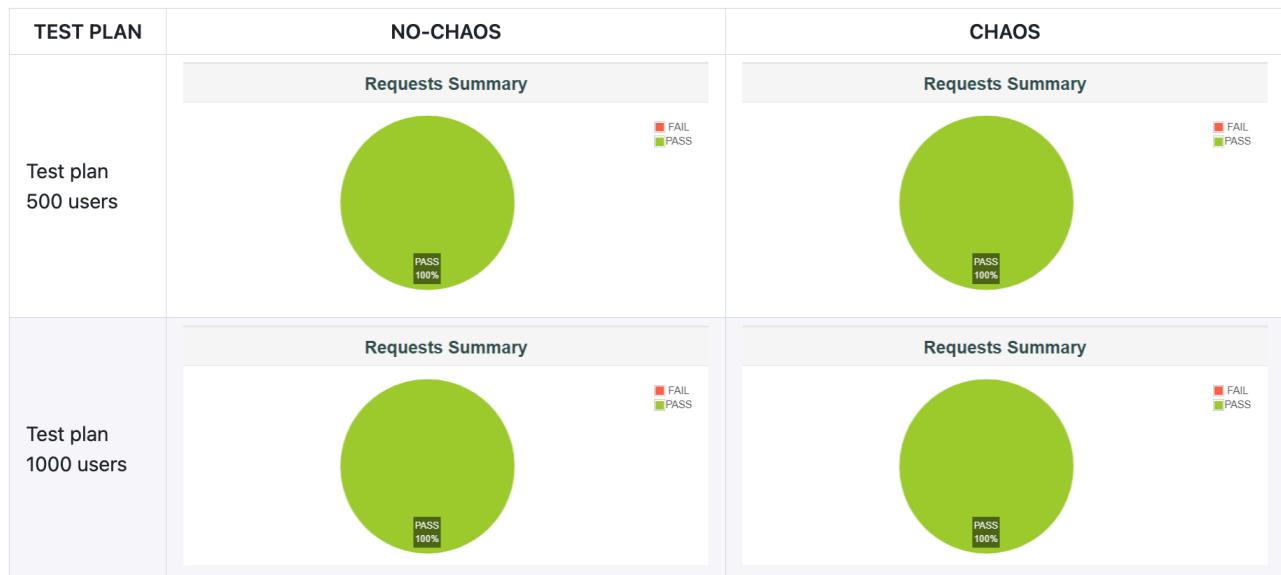
Para estudiar la tolerancia a fallos de la aplicación, hemos realizado pruebas de carga con la herramienta JMeter. Para ello, hemos diseñado dos test plans diferentes:

- Test plan con 500 usuarios y Ramp-up de 10 minutos
- Test plan con 1000 usuarios y Ramp-up de 10 minutos

Cada uno de los test plans los hemos ejecutado con y sin chaos-monkey.

Resultados

A continuación, vamos a comparar el resultado de las diferentes pruebas realizadas:



Análisis de los resultados

- Como se puede observar, no hay fallo de peticiones ni en los escenarios sin caos ni en los que introducen caos. Esto es porque al matar a uno de los nodos, otro se vuelve el primario de una manera muy rápida.
- Es altamente tolerante a fallos.

¿Cómo podríamos mejorar estos resultados?

- Se podría aumentar el número de réplicas para reducir las probabilidades de fallo al introducir cargas muy altas. Intuimos que, si aumentamos mucho la carga, de tal manera que metamos gran número de usuarios en poco espacio de tiempo (cosa que con nuestras máquinas no hemos podido probar, pues se nos cuelga antes el JMeter), podría producirse algún error, pero el porcentaje de fallo sería muy pequeño, pues la MongoDB es muy rápida en designar a un nuevo nodo primario.

Conclusiones

- Es una de las soluciones más tolerante a fallos de las que hemos estudiado.
- La escalabilidad es algo compleja al introducir el replicaset.

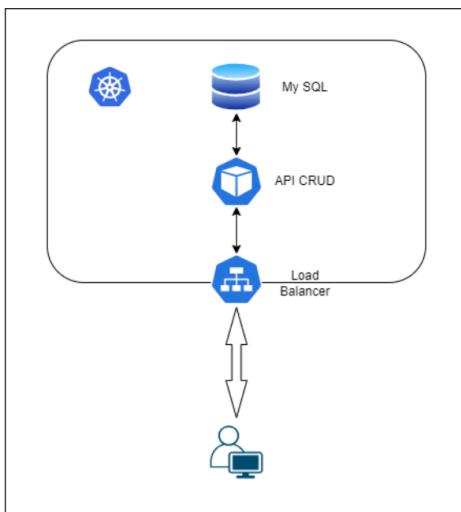
V. Web MySQL

En el caso de las webs con persistencia de datos basadas en el modelo relacional, hemos estudiado dos escenarios con MySQL: uno con una arquitectura standalone, con una sola réplica de base de datos y, viendo las limitaciones del primero, otro con un mecanismo de replicación.

Web MySQL standalone

En este ejemplo vamos a estudiar la tolerancia a fallos de una API cuya persistencia de datos se hace en una base de datos relacional sin replicación.

La arquitectura de la aplicación sometida a estudio es la siguiente:



Partimos de un escenario inicial en el que tenemos una única réplica de base de datos. A partir de aquí, hemos analizado:

- Tolerancia a fallos
 - Medir el rendimiento con una única réplica, simulando carga de procesamiento con y sin chaos-monkey.

Tolerancia a fallos

Pruebas de carga

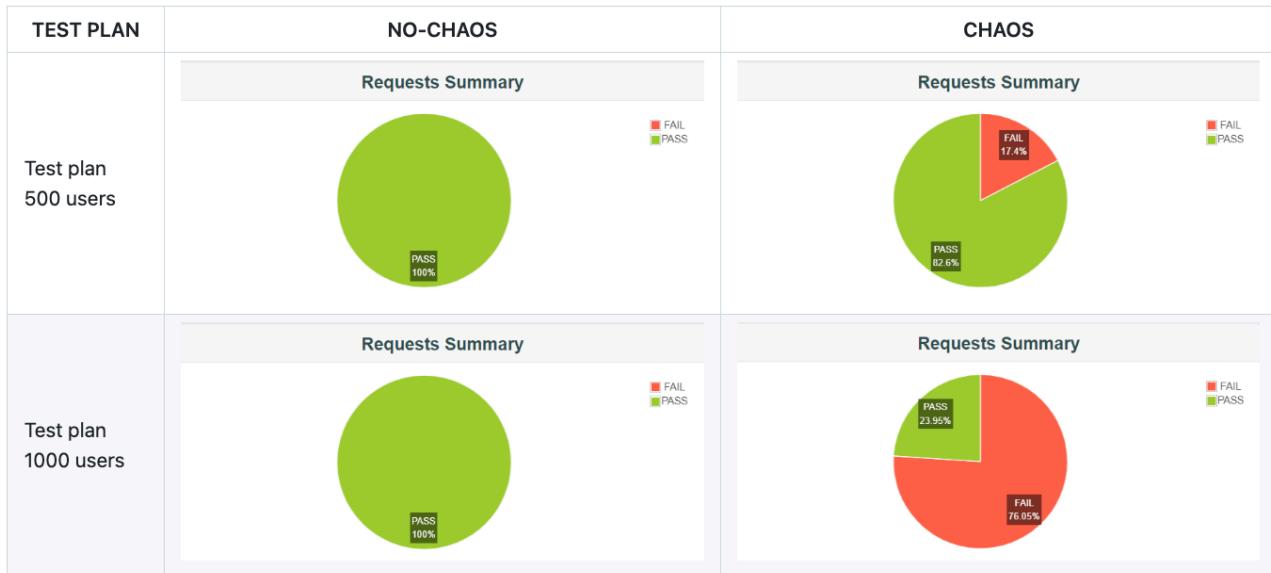
Para estudiar la tolerancia a fallos de la aplicación, hemos realizado pruebas de carga con la herramienta JMeter. Para ello, hemos diseñado dos test plans diferentes:

- Test plan con 500 usuarios y Ramp-up de 5 minutos
- Test plan con 1000 usuarios y Ramp-up de 5 minutos

Cada uno de los test plans los hemos ejecutado con y sin chaos-monkey.

Resultados

A continuación, vamos a comparar el resultado de las diferentes pruebas realizadas:



Análisis de los resultados

- Como se puede observar, cuando no hay caos el despliegue no falla aunque haya 1000 usuarios concurrentes.
- Como es obvio, al habilitar el chaos-monkey empezamos a encontrar errores tanto en las peticiones GET como en las POST. A mayor número de usuarios concurrentes, mayor es el número de fallos que se producen.

¿Cómo podríamos mejorar estos resultados?

- Aumentar la capacidad de las máquinas del clúster.
 - En nuestro caso, el clúster está formado por máquinas con 4GB de RAM, esto hace que la base de datos MySQL tarde bastante en volver a estar disponible tras un reinicio.
- Cuando no podemos aumentar la capacidad de nuestro clúster, podemos optar por la replicación de la base de datos.
- Para aumentar la tolerancia al fallo también podemos añadir a nuestro despliegue una caché, (Redis, Hazelcast, etc). Los métodos POST seguirían fallando, pero las peticiones GET no darían error.

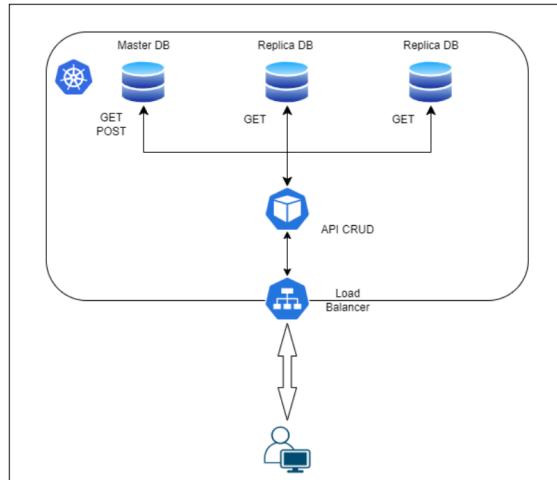
Conclusiones

- Las bases de datos MySQL consumen una gran cantidad de recursos por lo que es necesario tener máquinas potentes.
- Como hemos podido demostrar con las pruebas no es una arquitectura tolerante a fallos, por lo que no es recomendable en entornos de producción.

Web MySQL replicaset

Visto las limitaciones encontradas en el modelo de base de datos relacional con una sola réplica, procedemos a realizar el estudio con un mecanismo de replicación.

La arquitectura de la aplicación que se ha sometido a estudio es la siguiente:



En esta arquitectura las peticiones POST, PUT y DELETE se realizarán **sólo** en la base de datos master y las peticiones GET se realizarán en las réplicas.

Cambios en el código

Para hacer este cambio de arquitectura tenemos que adaptar nuestro código ya que por defecto JPA no acepta dos conexiones a bases de datos.

--	--	--

Las clases más importantes en esta transformación son:

- [TransactionReadonlyAspect.java](#). Clase encargada de cambiar entre la base de datos master o las réplicas según este indicado en la propiedad `@Transactional` indicada en la clase PostService.java.
- [DataSourceConfiguration.java](#) Es la clase encargada de leer los nuevos parámetros del properties y cargar esos valores en un hashmap.

- [PostService.java](#). Debemos anotar las peticiones POST con la anotación **@Transactional(readOnly = false)** y las peticiones GET con **@Transactional(readOnly = true)**.

```
import es.codeurjc.board.repository.PostRepository;
@Service
public class PostService {

    @Autowired
    private PostRepository posts;
    public void save(Post post) {
        posts.save(post);
    }
    public List<Post> findAll() {
        return posts.findAll();
    }
    public Optional<Post> findById(long id) {
        return posts.findById(id);
    }
    public void replace(Post updatedPost) {
        posts.findById(updatedPost.getId()).orElseThrow();
        posts.save(updatedPost);
    }
    public void deleteById(long id) {
        posts.deleteById(id);
    }
}
```



```
import es.codeurjc.board.repository.PostRepository;
import org.springframework.transaction.annotation.Transactional;
@Service
public class PostService {

    @Autowired
    private PostRepository posts;
    @Transactional(readOnly = false)
    public void save(Post post) {
        posts.save(post);
    }
    @Transactional(readOnly = true)
    public List<Post> findAll() {
        return posts.findAll();
    }
    @Transactional(readOnly = true)
    public Optional<Post> findById(long id) {
        return posts.findById(id);
    }
    @Transactional(readOnly = false)
    public void replace(Post updatedPost) {
        posts.findById(updatedPost.getId()).orElseThrow();
        posts.save(updatedPost);
    }
    @Transactional(readOnly = false)
    public void deleteById(long id) {
        posts.deleteById(id);
    }
}
```

Tolerancia a fallos

Pruebas de carga

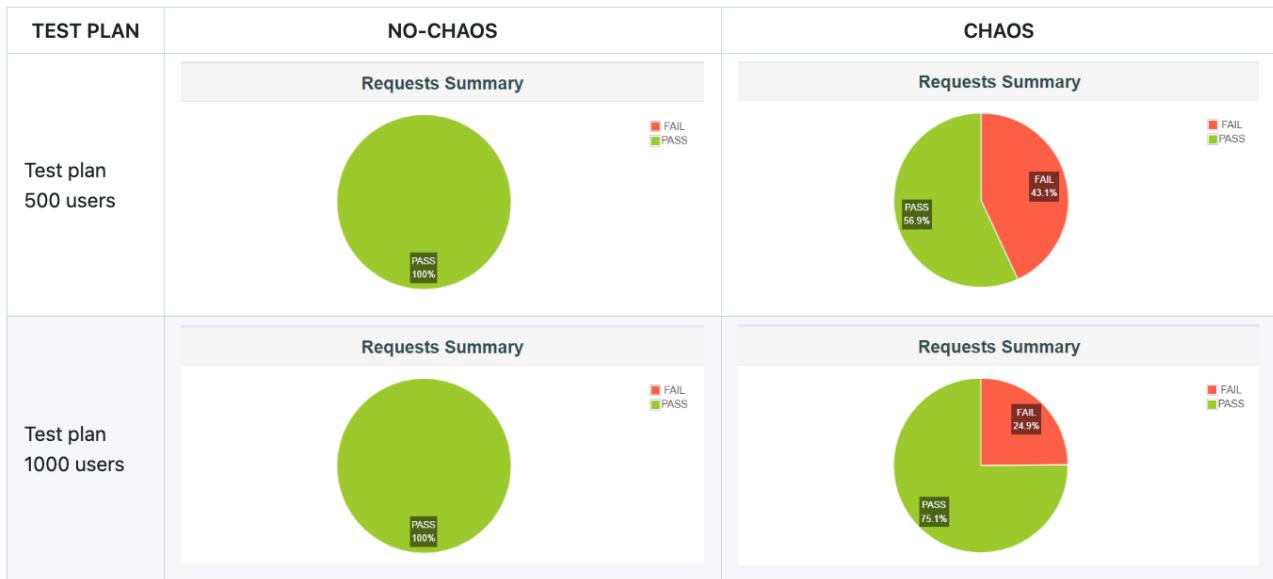
Para estudiar la tolerancia a fallos de la aplicación, hemos realizado pruebas de carga con la herramienta JMeter. Para ello, hemos diseñado dos test plans diferentes:

- Test plan con 500 usuarios y Ramp-up de 10 minutos
- Test plan con 1000 usuarios y Ramp-up de 10 minutos

Cada uno de los test plans los hemos ejecutado con y sin chaos-monkey.

Resultados

A continuación, vamos a comparar el resultado de las diferentes pruebas realizadas:



Análisis de los resultados

- Como se puede observar cuando no hay caos el despliegue no falla en las peticiones GET y POST aunque haya 1000 usuarios concurrentes.
- Al habilitar caos empezamos a tener errores. Aunque las gráficas nos pueden llevar a engaños, si observamos más detenidamente los datos vemos que el número de fallos con 500 y 1000 usuarios es similar en ambos casos.

¿Cómo podríamos mejorar estos resultados?

- Con las pruebas se ha demostrado que los errores se producen principalmente en los métodos POST ya que el nodo maestro no tiene replicación, para mejorar la tolerancia al fallo podríamos estudiar la georedundancia de la infraestructura.
- Aunque los errores en las peticiones GET son mínimos podríamos mejorar aún más los resultados si añadimos algún tipo de caché (JCache, Redis) a nuestra aplicación.

Conclusiones

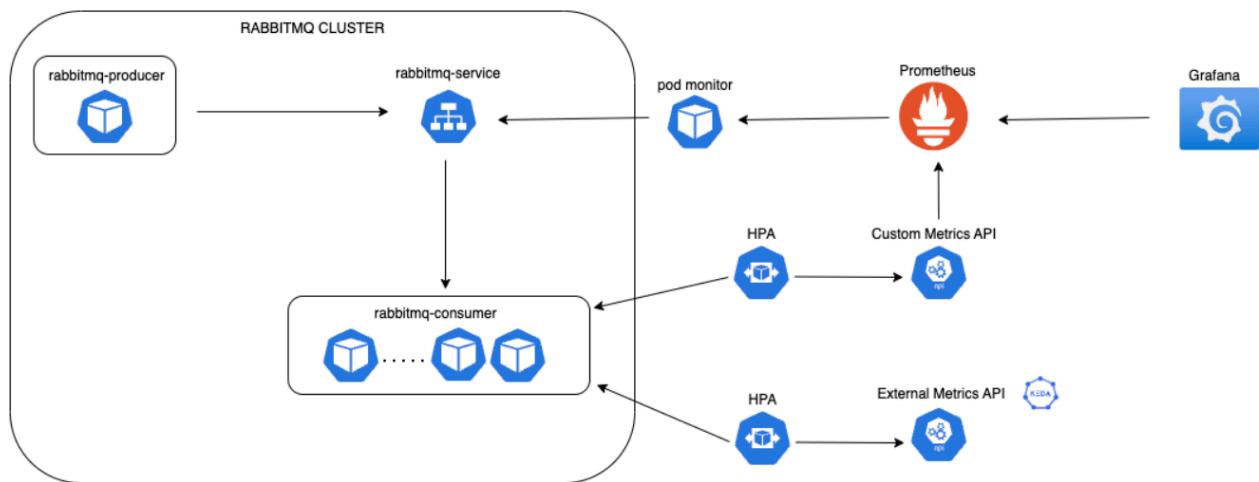
- Como hemos podido demostrar con las pruebas, esta arquitectura es mucho más tolerante al fallo que la standalone.
- Es una arquitectura más acorde a un entorno de producción.
- Esta solución es más "económica", ya que con las mismas máquinas se producen menos fallos.
- Aunque las bases de datos MySQL se pueden instalar en Kubernetes nuestra experiencia nos dice que de momento son bastante complicadas de gestionar.

VI. RabbitMQ

En el siguiente ejemplo vamos a estudiar diferentes mecanismos de autoescalamiento en un clúster de RabbitMQ, concretamente los siguientes:

- HPA basado en custom metrics (con instalación de Prometheus para la recogida de métricas y visualización en Grafana).
- KEDA:
 - Scaler RabbitMQ Queue basado en external metrics.
 - Scaler Cron.

La arquitectura de la aplicación es la siguiente:



Configuración

Para montar el escenario de pruebas, hemos seguido los siguientes pasos:

1. Despliegue de Prometheus mediante un operador para la recolección de métricas a través de la RabbitMQ.
2. Despliegue de un clúster de RabbitMQ.
3. Habilitar el plugin *rabbitmq_prometheus* en todos los nodos, lo que permite el envío de las métricas de RabbitMQ a Prometheus:

```
rabbitmq@rabbitmq-server-0:/ $ rabbitmq-plugins enable rabbitmq_prometheus
Enabling plugins on node rabbit@rabbitmq-server-0.rabbitmq-nodes.default:
rabbitmq_prometheus
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_peer_discovery_common
  rabbitmq_peer_discovery_k8s
  rabbitmq_prometheus
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@rabbitmq-server-0.rabbitmq-nodes.default...
Plugin configuration unchanged.
```

4. Desplegar un pod monitor para enviar las métricas de RabbitMQ a Prometheus.

Para verificar que el pod monitor se ha instalado correctamente y Prometheus está recolectando las métricas de la RabbitMQ, podemos acceder a la UI de Prometheus (servicio expuesto en el puerto 9090) y verificar que en la pestaña Targets todos los endpoints están **Up**:

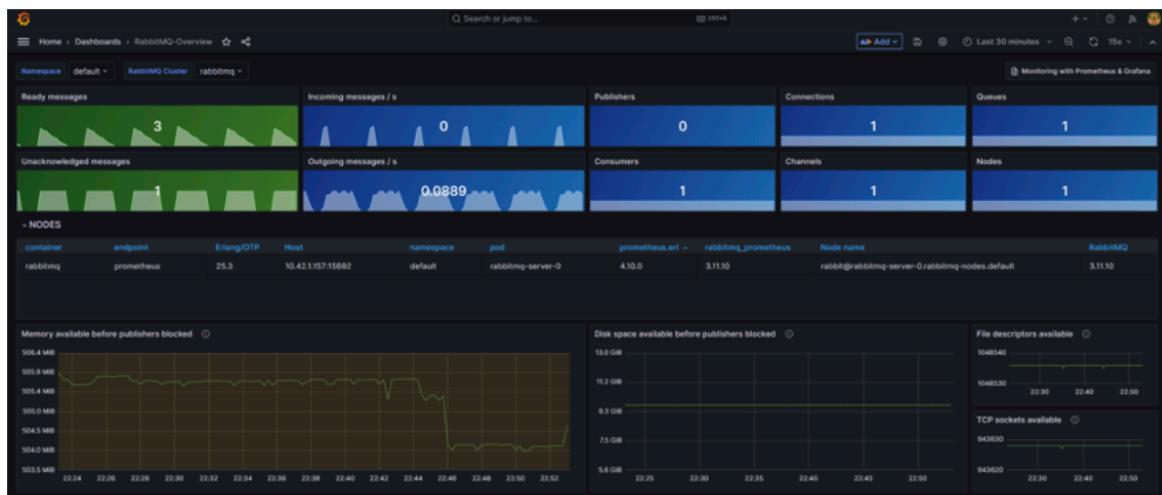
The screenshot shows the Prometheus Targets page. At the top, there are tabs for 'All', 'Unhealthy', and 'Collapse All'. A search bar is followed by a filter for 'endpoint or labels'. Below this, two targets are listed:

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://10.42.2.9:9090/metrics	Up	containers:rabbitmq; endpoint:web; instance="10.42.2.9:9090"; job:rabbitmq; namespace:rabbitmq; pod:rabbitmq_provisioner; service:rabbitmq	4.865s ago	4.398ms	
http://10.42.3.156:9090/metrics	Up	containers:rabbitmq; endpoint:web; instance="10.42.3.156:9090"; job:rabbitmq; namespace:rabbitmq; pod:rabbitmq_provisioner; service:rabbitmq	17.237s ago	5.954ms	

5. Desplegar un productor de mensajes mediante un Cronjob, que publicará 20 mensajes en la RabbitMQ cada 5 minutos.

6. Desplegar un consumidor suscrito a la cola de la RabbitMQ que consumirá un mensaje cada 10 segundos.

7. Despliegue de Grafana para la visualización de las métricas. Accedemos a la UI de Grafana, a través del servicio expuesto en el puerto 3000 (username: admin, password: admin), e importamos un dashboard que trae Grafana por defecto, *RabbitMQ Overview* para visualizar los datos:



8. Despliegue de un adaptador de Prometheus para la recogida de métricas custom.

Escalabilidad

Para estudiar la escalabilidad vamos a usar dos técnicas: un HPA basado en métricas custom, y KEDA (scaler basado en métricas externas y scaler Cron).

HPA

Una vez tenemos habilitado el envío de las métricas custom desde nuestro cluster de RabbitMQ a Prometheus, y la visualización de los datos mediante el dashboard de Grafana, vamos a escalar el clúster de RabbitMQ por una de estas métricas custom, en nuestro caso, la métrica `rabbitmq_queue_messages`.

Accedemos a la UI de Prometheus para verificar que se están recolectando datos sobre la métrica:

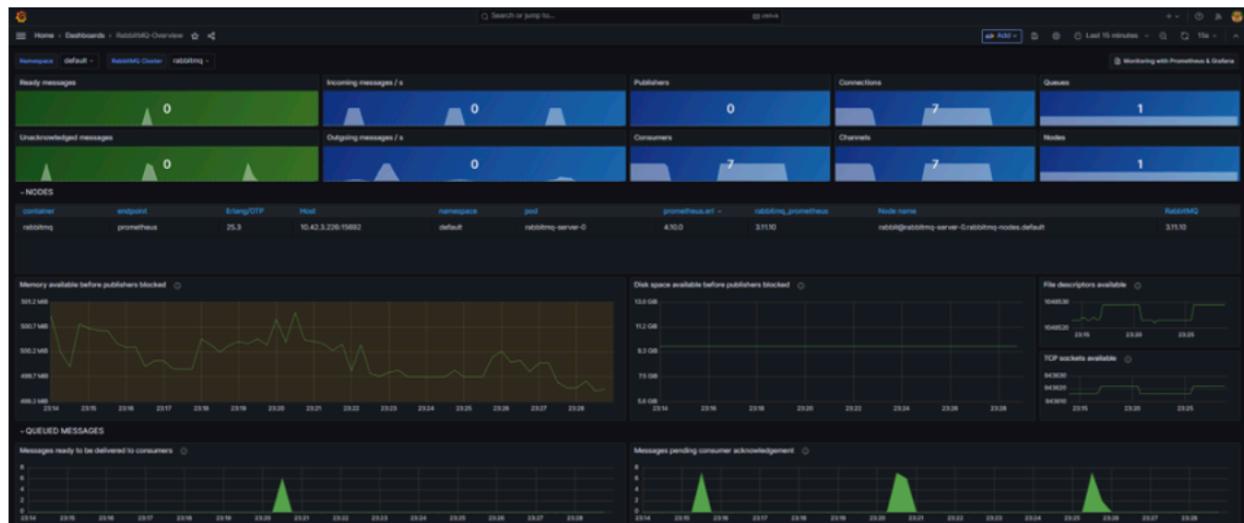


Deplegamos el HPA que autoescala el número de pods del consumidor en función del valor de la métrica custom `rabbitmq_queue_messages`.

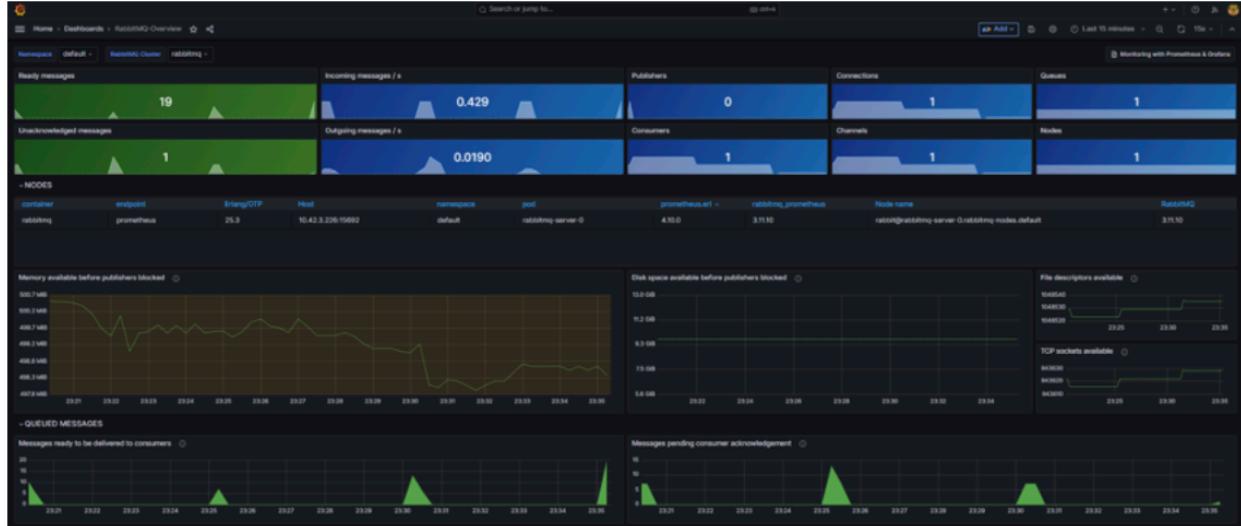
Podemos ver cómo, según se van produciendo mensajes, el número de pods del consumidor va aumentando (hasta un máximo de 10 réplicas, que hemos definido en el HPA); mientras que, una vez que los mensajes son consumidos, el número de pods del consumidor va disminuyendo (hasta mínimo de 1 réplica, que hemos definido en el HPA):

NAME	REFERENCE	TARGETS	MINPODS	MAXPODS	REPLICAS	AGE
rabbitmq-consumer	Deployment/rabbitmq-consumer	3/1 (avg)	1	20	8	45s
rabbitmq-consumer	Deployment/rabbitmq-consumer	625m/1 (avg)	1	20	13	60s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	75s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	90s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	105s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	2m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	2m15s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	2m30s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	2m45s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	3m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	3m15s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	3m30s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	3m45s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	4m
rabbitmq-consumer	Deployment/rabbitmq-consumer	539m/1 (avg)	1	20	13	4m15s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	4m31s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	4m46s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	5m1s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	5m16s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	12	5m31s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	5m46s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	6m1s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	6m16s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	6m31s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	6m46s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	7m1s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	7m16s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	7m31s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	7m46s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	8m1s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	8m16s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	8m31s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	8m46s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	7	9m1s
rabbitmq-consumer	Deployment/rabbitmq-consumer	1858m/1 (avg)	1	20	7	9m16s
rabbitmq-consumer	Deployment/rabbitmq-consumer	462m/1 (avg)	1	20	13	9m31s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	9m46s
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	10m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	10m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	10m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	11m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	11m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	11m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	11m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	12m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	12m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	12m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	12m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	13m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	13m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	13m
rabbitmq-consumer	Deployment/rabbitmq-consumer	0/1 (avg)	1	20	13	14m

Si nos fijamos en el dashboard de Grafana, también vemos este aumento de réplicas del consumidor reflejado en las gráficas:

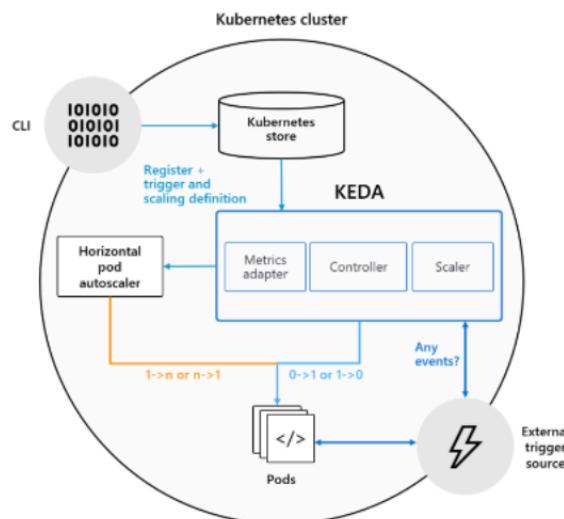


Y la reducción a una sola réplica cuando todos los mensajes son consumidos:



KEDA

Estudiamos el escalado mediante KEDA, mecanismo que permite el autoescalamiento mediante la creación de un HPA en función de determinados eventos. Cuando se produce el evento que hayamos definido, se lanza un trigger que lleva a cabo el autoescalamiento entre los valores mínimos y máximos que le hayamos definido. La arquitectura general de KEDA es la siguiente:



- Scaler RabbitMQ

Autoescala las réplicas del consumidor por una métrica externa, en este caso usamos para el escalado la métrica `QueueLength`. Cuando la longitud de la cola alcanza el valor especificado en este parámetro, las réplicas aumentan hasta el máximo que le hemos definido:

Context: k3s Cluster: k3s User: k3s K9s Rev: v0.27.4 K8s Rev: v1.26.5+k3s1 CPU: 5% MEM: 55%												
<0> all <a> Attach <1> Logs <1> default <ctrl-d> Delete <p> Logs Prev /--> /--> <d> Describe <shift-f> Port-Forward /--> <e> Edit <s> Shell /--> <?> Help <n> Show Node /--> <ctrl-k> Kill <f> Show PortF /-->												
Pods(default)[1]												
NAME†	PF	READY	RESTARTS	STATUS	CPU	MEM	%CPU/R	%CPU/L	%MEM/R	%MEM/L	IP	NODE
rabbitmq-consumer-6bdbcb97b64-6z4kj	●	1/1	0	Running	1	51	1	1	80	40	10.42.2.139	k3s-agent-lar
rabbitmq-consumer-6bdbcb97b64-9cdrp	●	1/1	0	Running	0	0	0	0	0	0	10.42.1.204	k3s-agent-lar
rabbitmq-consumer-6bdbcb97b64-s2fpd	●	1/1	0	Running	0	0	0	0	0	0	10.42.1.203	k3s-agent-lar
rabbitmq-consumer-6bdbcb97b64-s6ldq	●	1/1	0	Running	0	0	0	0	0	0	10.42.1.205	k3s-agent-lar
rabbitmq-consumer-6bdbcb97b64-wtqej	●	1/1	0	Running	0	0	0	0	0	0	10.42.3.182	k3s-agent-lar
rabbitmq-producer-28102635-x6bvh	●	0/1	0	Completed	0	0	n/a	n/a	n/a	n/a	10.42.1.196	k3s-agent-lar
rabbitmq-server-0	●	1/1	0	Running	5	157	0	0	7	7	10.42.2.192	k3s-agent-lar

Mientras que, cuando la longitud de la cola disminuye, las réplicas disminuyen también hasta el mínimo que le hemos indicado:

Context: k3s Cluster: k3s User: k3s K9s Rev: v0.27.4 K8s Rev: v1.26.5+k3s1 CPU: 1% MEM: 55%												
<0> all <a> Attach <1> Logs <1> default <ctrl-d> Delete <p> Logs Prev /--> /--> <d> Describe <shift-f> Port-Forward /--> <e> Edit <s> Shell /--> <?> Help <n> Show Node /--> <ctrl-k> Kill <f> Show PortF /-->												
Pods(default)[7]												
NAME†	PF	READY	RESTARTS	STATUS	CPU	MEM	%CPU/R	%CPU/L	%MEM/R	%MEM/L	IP	NODE
rabbitmq-consumer-6bdbcb97b64-6z4kj	●	1/1	0	Running	1	51	1	1	80	40	10.42.2.139	k3s-agent-1
rabbitmq-consumer-6bdbcb97b64-8j28w	●	1/1	0	TerminatingΔ	0	0	0	0	0	0	10.42.1.214	k3s-agent-1
rabbitmq-consumer-6bdbcb97b64-2214g	●	1/1	0	TerminatingΔ	0	0	0	0	0	0	10.42.3.183	k3s-agent-1
rabbitmq-consumer-6bdbcb97b64-nnscd	●	1/1	0	TerminatingΔ	0	0	0	0	0	0	10.42.1.213	k3s-agent-1
rabbitmq-consumer-6bdbcb97b64-p587j	●	1/1	0	TerminatingΔ	0	0	0	0	0	0	10.42.1.212	k3s-agent-1
rabbitmq-producer-28102645-zpqzq	●	0/1	0	Completed	0	0	n/a	n/a	n/a	n/a	10.42.1.211	k3s-agent-1
rabbitmq-server-0	●	1/1	0	Running	3	157	0	0	7	7	10.42.2.192	k3s-agent-1

- Scaler Cron

Autoescala el número de réplicas por un periodo de tiempo determinado, indicado entre un start y un stop. De tal manera que, cuando se alcanza la hora definida en el start, las réplicas del consumidor aumentan hasta el valor máximo que le hemos indicado:

Context: k3s Cluster: k3s User: k3s K9s Rev: v0.27.4 K8s Rev: v1.26.5+k3s1 CPU: 1% MEM: 55%												
<0> all <a> Attach <1> Logs <1> default <ctrl-d> Delete <p> Logs Prev /--> /--> <d> Describe <shift-f> Port-Forward /--> <e> Edit <s> Shell /--> <?> Help <n> Show Node /--> <ctrl-k> Kill <f> Show PortF /-->												
Pods(default)[12]												
NAME†	PF	READY	RESTARTS	STATUS	CPU	MEM	%CPU/R	%CPU/L	%MEM/R	%MEM/L	IP	NODE
rabbitmq-consumer-6bdbcb97b64-6z4kj	●	1/1	0	Running	1	51	1	1	80	40	10.42.2.139	k3s-agent-1ar
rabbitmq-consumer-6bdbcb97b64-8rwfn	●	1/1	0	Running	0	0	0	0	0	0	10.42.1.220	k3s-agent-1ar
rabbitmq-consumer-6bdbcb97b64-294zm	●	1/1	0	Running	0	0	0	0	0	0	10.42.1.216	k3s-agent-1ar
rabbitmq-consumer-6bdbcb97b64-vxpxl	●	1/1	0	Running	0	0	0	0	0	0	10.42.1.218	k3s-agent-1ar
rabbitmq-consumer-6bdbcb97b64-gpwf6	●	1/1	0	Running	0	0	0	0	0	0	10.42.1.219	k3s-agent-1ar
rabbitmq-consumer-6bdbcb97b64-ktbzv	●	1/1	0	Running	0	0	0	0	0	0	10.42.1.221	k3s-agent-1ar
rabbitmq-consumer-6bdbcb97b64-ktnqg	●	1/1	0	Running	0	0	0	0	0	0	10.42.2.254	k3s-agent-1ar
rabbitmq-consumer-6bdbcb97b64-1ff4r	●	1/1	0	Running	0	0	0	0	0	0	10.42.1.217	k3s-agent-1ar
rabbitmq-consumer-6bdbcb97b64-tfffc6	●	1/1	0	Running	0	0	0	0	0	0	10.42.3.185	k3s-agent-1ar
rabbitmq-consumer-6bdbcb97b64-z44vs	●	1/1	0	Running	0	0	0	0	0	0	10.42.3.184	k3s-agent-1ar
rabbitmq-producer-28102650-vrjg9	●	0/1	0	Completed	0	0	n/a	n/a	n/a	n/a	10.42.1.215	k3s-agent-1ar
rabbitmq-server-0	●	1/1	0	Running	3	157	0	0	7	7	10.42.2.192	k3s-agent-1ar

Del mismo modo, cuando se alcanza la hora indicada en el campo stop, las réplicas del consumidor disminuyen hasta el mínimo que le hemos definido:

Context: k3s	<0> all	<a>	Attach	<l>	Logs	---	---					
Cluster: k3s	<1> default	<ctrl-d>	Delete	<p>	Logs Prev/	/	\					
User: k3s		<d>	Describe	<shift-f>	Port-Forward	<\	/					
K9s Rev: v0.27.4		<e>	Edit	<s>	Shell	</\	/					
K8s Rev: v1.26.5+k3s1		<?>	Help	<n>	Show Node	/	\					
CPU: 1%		<ctrl-k>	Kill	<f>	Show PortF	V	V					
MEM: 58%↑												
NAME†	PF	READY	RESTARTS	STATUS	CPU	MEM	%CPU/R.	%CPU/L.	%MEM/R.	%MEM/L.	IP	NODE
rabbitmq-consumer-6bdcb97b64-6z4kj	●	1/1	0	Running	1	51	1	1	80	40	10.42.2.139	k3s-agent-1
rabbitmq-consumer-6bdcb97b64-9kbbn	●	1/1	0	Terminating	1	28	1	1	44	22	10.42.1.224	k3s-agent-1
rabbitmq-consumer-6bdcb97b64-dcm6p	●	1/1	0	Terminating	1	28	1	1	44	22	10.42.1.228	k3s-agent-1
rabbitmq-consumer-6bdcb97b64-h8cx8	●	1/1	0	Terminating	1	28	1	1	44	22	10.42.1.227	k3s-agent-1
rabbitmq-consumer-6bdcb97b64-jbhsn	●	1/1	0	Terminating	1	33	1	1	51	25	10.42.3.186	k3s-agent-1
rabbitmq-consumer-6bdcb97b64-jhkoh	●	1/1	0	Terminating	1	30	1	1	47	23	10.42.1.226	k3s-agent-1
rabbitmq-consumer-6bdcb97b64-jkf4w	●	1/1	0	Terminating	1	30	1	1	47	23	10.42.2.2	k3s-agent-1
rabbitmq-consumer-6bdcb97b64-kbt49	●	1/1	0	Terminating	1	30	1	1	47	23	10.42.1.225	k3s-agent-1
rabbitmq-consumer-6bdcb97b64-sdxbc	●	1/1	0	Terminating	1	28	1	1	45	22	10.42.3.187	k3s-agent-1
rabbitmq-consumer-6bdcb97b64-wlmch	●	1/1	0	Terminating	1	29	1	1	45	22	10.42.1.223	k3s-agent-1
rabbitmq-producer-28102655-29gvc	●	0/1	0	Completed	0	0	n/a	n/a	n/a	n/a	10.42.1.222	k3s-agent-1
rabbitmq-server-0	●	1/1	0	Running	3	157	0	0	7	7	10.42.2.192	k3s-agent-1

Conclusiones

- El escalado tanto por métricas custom como por métricas externas ofrece una gran flexibilidad, aunque es más complejo de implementar que el que se basa en métricas propias de Kubernetes.
- KEDA puede consumir métricas de innumerables fuentes, proporcionando un mecanismo de autoescalado muy potente.

CONCLUSIONES

Después de todos los escenarios probados, hemos llegado a las siguientes conclusiones:

- La tolerancia a fallos es mayor cuantas más réplicas tengamos de la aplicación.
- Es importante limitar los recursos de los pods para poder aplicar los mecanismos de escalabilidad que ofrece Kubernetes.
- En las arquitecturas con persistencia de datos mediante el modelo relacional, la escalabilidad es más compleja que en aquellas que implementan el modelo no relacional.
- Las web stateless son las más tolerantes a fallos y las que mejor se prestan a la escalabilidad, pero las más limitadas, puesto que la complejidad de aplicación que requiere un entorno productivo es bastante mayor.

TRABAJOS FUTUROS

- Implementación de un clúster multirregión para ver si mejoran las latencias dependiendo de dónde esté cada origen.
- Implementación de un service mesh para mejorar la tolerancia a fallos.
- Implementación de un servicio de mensajería escalable y tolerante a fallos con las técnicas estudiadas.
- Implementación de un clúster de Redis caché.

BIBLIOGRAFÍA

Hemos consultado numerosos artículos y documentación en Internet. A continuación, el listado de todos ellos:

<https://kubernetes.io/es/>
<https://nakamasato.medium.com/kubernetes-hpa-with-custom-metrics-rabbitmq-and-prometheus-d65ac7cb7e65>
<https://ranchermanager.docs.rancher.com/v2.0-v2.4/explanations/integrations-in-rancher/cluster-monitoring/custom-metrics>
<https://github.com/stefanprodan/k8s-prom-hpa>
<https://github.com/nakamasato/rabbitmq-producer>
<https://github.com/nakamasato/rabbitmq-consumer>
<https://www.rabbitmq.com/prometheus.html>
<https://keda.sh/>
<https://www.returngis.net/2020/05/aplicaciones-con-estado-en-kubernetes-con-statefulsets/>
<https://www.mongodb.com/docs/manual/tutorial/deploy-replica-set/>
<https://www.mongodb.com/docs/manual/replication/>
<https://www.hetzner.com/>
<https://github.com/hazelcast/hazelcast-code-samples/tree/master/hazelcast-integration/kubernetes/samples>
<https://github.com/hazelcast/hazelcast>
<https://hazelcast.com/resources/how-to-use-embedded-hazelcast-on-kubernetes/>
<https://docs.hazelcast.com/tutorials/caching-springboot-jcache>
<https://jmeter.apache.org/>
<https://kubernetes.io/docs/tasks/run-application/run-replicated-stateful-application/>
<https://redis.io/>

<https://bitnami.com/stack/redis/helm>

<https://artifacthub.io/packages/helm/inspur/redis-cluster>

<https://www.mirantis.com/blog/how-to-use-statefulsets-and-create-a-scalable-mysql-server-on-kubernetes/>

<https://github.com/bitnami/charts/tree/main/bitnami/mysql/#parameters>

