Universidad Zaragoza

Trabajo Fin de Grado

Distribuidor de carga para SMS GW
Load balancer for SMS GW

Autor
Óscar Palacín Grasa

Director
Fernando Orús Morlans

Ponente
Julián Fernández Navajas

Escuela de Ingeniería y Arquitectura

2023

Índice de contenidos

Índice de figuras

Anexos

Anexo A: Configuración mensaje HTTP

En la figura 25 se muestra la estructura que tienen que tener las peticiones HTTP para ser aceptadas por el IRISGW.

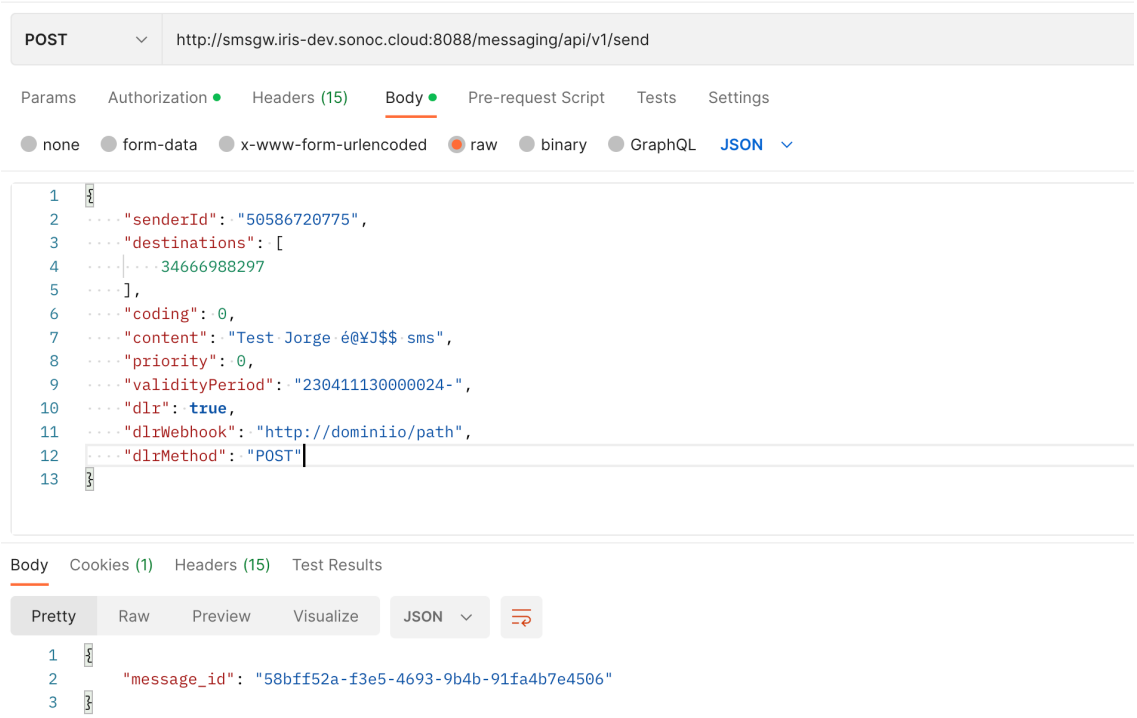

Figura 25. Estructura mensajes HTTP

Anexo B: Configuración conector Jasmin

Para crear un cliente SMPP en Jasmin se debe crear un conector smmpccm. Éste a su vez necesita definir un user, group, y mtrouter. A continuación se muestra la configuración utilizada.

Creamos un conector:

```
smppccm -a
cid DEMO_CONNECTOR
host  192.168.2.175
port 2775
username smpp-tfg
password smpp-tfg
submit_throughput 110
ok
```

Establecemos este conector como ruta por defecto:

```
mtrouter -a
type defaultroute
connector smppc(DEMO_CONNECTOR)
rate 0.00
ok
```

Creamos un usuario para el envío de mensajes:

```
user -a
username tfg-2
password tfg-2
gid foogroup
uid foo
ok
group -a
gid foogroup
ok
```

Este usuario que se ha creado es el que luego se utiliza en la petición para poder conectarnos. En la figura 26 se observa como se ha configurado el conector que hemos creado como ruta por defecto para el envío de los SMS.

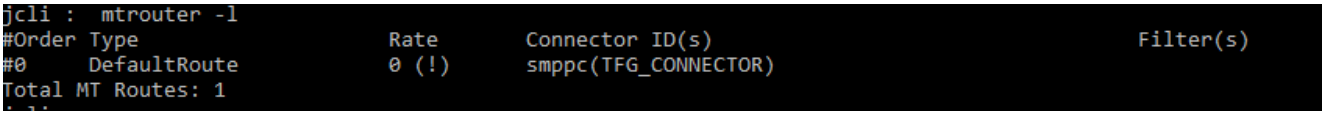

Figura 26. Rutas configuradas en Jasmin

Para que IRIS acepte los mensajes provenientes del conector se debe crear un incoming connection y asociarla a un cliente que se utilizara de prueba. Para este cliente se debe configurar el número de binds que pueden tener abiertos,el tipo de sesión que puede abrir y el número de mensajes máximos por sesión al igual que se haría para cualquier cliente.

Mediante el uso de este script en Python se pueden enviar SMS de prueba a cliente a través de la API de Jasmin.

```
# Python example
# http://jasminsms.com
import urllib.request, urllib.error, urllib.parse
import urllib.request, urllib.parse, urllib.error

params = {'username':'tfg-2', 'password':'tfg-2', 'to':'+620444007', 'content':'Hello',
          'dlr-level':2}
response = urllib.request.urlopen("http://127.0.0.1:1401/send?%s" %
urllib.parse.urlencode(params)).read()
print(response)
```

Anexo C: Generación jar

Se van a explicar los pasos que se han seguido para generar un programa jar utilizando el entorno de desarrollo intelliJ.

1. En la sección "File" seleccionar "Project Structure".
2. Luego, se debe seleccionar "Artifacts" en la sección de la izquierda.
3. Después, se debe añadir un "JAR" > "From modules with dependencies".

4. Posteriormente, se debe seleccionar el módulo de código del proyecto que se va a empaquetar.
5. Seguidamente, se debe especificar el archivo Manifest.mf en la sección de "Output Layout".
6. Luego, se debe especificar la ubicación y el nombre del archivo .jar en la sección de "Output Layout".
7. Posteriormente, se debe hacer clic en "Build" en la barra de menú y seleccionar "Build Artifacts".

Ya se tiene generado un paquete JAR para poder ejecutar nuestra aplicación java en cualquier entorno simplemente ejecutando:

- **java -jar <archivo.jar>**

Anexo D: Dockerfile

Los ficheros de creación de imágenes Docker se llaman Dockerfile, sin ninguna extensión, para ser reconocidos correctamente a la hora de crear imágenes. Los ficheros se leen de arriba abajo sin saltos, con una orden por línea. Para la imagen del proxy se tiene el siguiente dockerfile:

```
FROM openjdk:19-jdk-alpine
WORKDIR /src/app
COPY out/artifacts/proxy_final.jar /src/app
EXPOSE 8057
CMD ["java", "-jar", "proxy_final.jar"]
```

Para la imagen de envio_mensaje:

```
FROM openjdk:19-jdk-alpine
WORKDIR /src/app
COPY /proxy/out/artifacts/enviar_mensaje.jar /src/app
EXPOSE 8056
CMD ["java", "-jar", "enviar_mensaje.jar"]
```

Anexo E: Archivo configuración despliegue kubernetes

El primer archivo YAML configura un deployment con tres réplicas de un conjunto de pods que ejecutan la imagen proxy alojada en el repositorio privado de SONOC. Se indica que la aplicación expone el puerto 8057. Además, crea un servicio nodePort llamado proxy-service que utiliza el selector app: proxy para apuntar al conjunto de réplicas creado por el Deployment. El Service no especifica el puerto en el que escucha el servicio por tanto será un puerto a partir del 30000 e indica que todo el tráfico que se reciba en ese puerto se redirige al puerto 8057 de cualquiera de las réplicas asociadas.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: proxy-deployment
spec:
```

```
  replicas: 3 # Define el número de réplicas que deseas
  selector:
    matchLabels:
      app: proxy
  template:
    metadata:
      labels:
        app: proxy
    spec:
      containers:
        - name: proxy
          image: dockers.sonoc.io/proxy
          ports:
            - containerPort: 8057
---------------------------------------------------------------------
apiVersion: v1
kind: Service
metadata:
  name: proxy-service
spec:
  selector:
    app: proxy
  ports:
  - name: smpp
    port: 8057
    targetPort: 8057
    protocol: TCP
  type: NodePort
```

A continuación, se tiene el archivo de configuración de un deployment que configura 3 réplicas que ejecutan la imagen envio_mensaje. En este caso se expone la aplicación por el puerto 80. Además se configura el servicio que permite la comunicación entre los pods de proxy y los pods de envio_mensaje.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: envia-mensaje-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: envia-mensaje
  template:
    metadata:
      labels:
        app: envia-mensaje
```

```
    spec:
      containers:
      - name: envia-mensaje-container
        image: dockers.sonoc.io/envio_mensaje
        ports:
        - containerPort: 80
---------------------------------------------------------------------
apiVersion: v1
kind: Service
metadata:
  name: mensaje-service
spec:
  selector:
    app: envia-mensaje
  ports:
    - name: socket
      protocol: TCP
      port: 8056
      targetPort: 8056
```

Anexo F: Fichero configuración HPA

En estos archivos YAML se configura un HPA. En el primer caso utiliza la métrica predeterminada de consumo de memoria, en el segundo caso utiliza la métrica personalizada mensajes recibidos por segundo. En ambos casos se configura el número mínimo y máximo de réplicas como 1 y 10 respectivamente.

Configuración que utiliza la métrica estándar consumo de memoria de los pods:

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: mensaje-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: envia-mensaje-deployment
  minReplicas: 1
  maxReplicas: 10
  metrics:
  - type: Resource
    resource:
      name: memory
      target:
        type: Utilization
        averageUtilization: 8
```

Configuración que utiliza la métrica personalizada mensajes recibidos por segundo:

```
apiVersion: autoscaling/v2beta2
kind: HorizontalPodAutoscaler
metadata:
  name: myhpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: envia-mensaje-deployment
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Pods
      pods:
        metric:
          name: myapp_requests_per_second
        target:
          type: AverageValue
          averageValue: 10
```

Anexo G: Configuración Prometheus

Como se puede ver en la figura 27 el HPA consulta de forma recursiva las métricas del objeto al que apunta, en este caso el deployment proxy, para calcular si debe aumentar o disminuir el número de réplicas del deployment envio_mensaje. El HPA consulta las métricas de un registro dentro del cluster llamado metrics registry donde se exponen todas las métricas del sistema. El interfaz de metric registry está compuesto de 3 apis distintas:

- Resource Metrics API: métricas de uso predefinidas como CPU y memoria de los pods.
- Custom Metrics API: métricas personalizadas relacionadas con objetos de kubernetes.
- External Metrics API: métricas personalizadas no relacionadas con objetos de kubernetes.

El primer paso para exponer métricas personalizadas ha sido modificar nuestra aplicación proxy para que a través de un cliente Prometheus exponga el número de mensajes recibidos de los cliente en el endpoint http://ip_pod:8080/metrics. Para comprobar su funcionamiento se creó un servicio nodePort que permitiese consultar a las métricas desde fuera del cluster a través de una petición curl o verlas desde el navegador web realizando un port-forwarding.

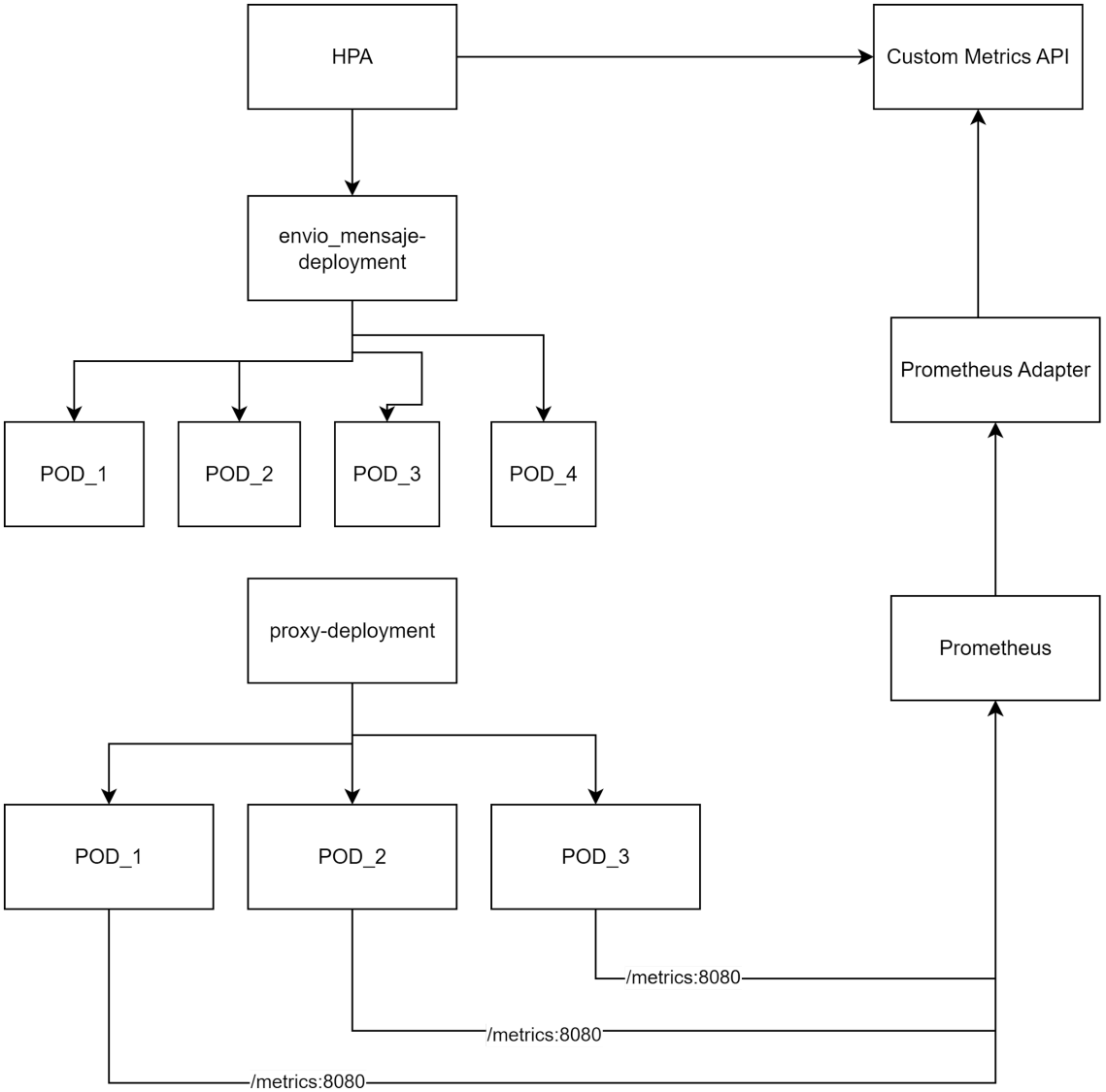

Figura 27. Esquema métricas personalizadas

Posteriormente como se ve en la figura 27 se debe instalar un colector de métricas que recoja las métricas que se exponen en los pods. En nuestro caso se ha utilizado Prometheus ya que ya se tenía instalado previamente en el cluster aunque no con este objetivo. Para que Prometheus recolecta las métricas de nuestro pods es necesario modificar su configuración:

- **kubectl edit configmap <configmap-name>**

En este fichero configmap se debe añadir:

```
scrape_configs:
 - job_name: 'proxy-metrics'
   static_configs:
     - targets: ['proxy-deployment:8080']
   namespace: 'prepod'
```

Para asegurarnos que se ha actualizado correctamente ejecutamos el siguiente comando y deberá aparecer la nueva configuración

- **kubectl describe configmap <configmap-name> -n <namespace>**

Finalmente se necesita instalar un metric API server como Prometheus Adapter que nos permita exponer las métricas que se recogen con Prometheus a través de la Custom Metrics API y de esa forma sea accesible para el HPA. Para instalarlo se ha utilizado el gestor de paquetes de Kubernetes helm y se han seguido los pasos de instalación de la documentación oficial https://github.com/kubernetes-sigs/prometheus-adapter.

Para exponer la métrica personalizada myapp_requests_per_second(número de mensajes por segundo) ha sido necesario modificar la configuración básica de Prometheus Adapter.

- **kubectl edit configmap <configmap-name> -n <namespace>**

Se añade lo siguiente en la sección de reglas

```
rules:
  - seriesQuery: 'myapp_requests_total'
    resources:
      overrides:
        resource: "myapp_requests_per_second"
    name:
      matches: "myapp_requests_total"
      as: "myapp_requests_per_second"
    metricsQuery: 'rate(myapp_requests_total[1s])'
```

Como se observa se pasa de tener una métrica que es el número de mensajes totales recibidos a otra que a el número de mensajes totales recibidos por segundo utilizado la función de consulta rate.

Ahora ya se exponen las métricas a través de la API y son accesibles para el cluster. Se ha modificado el fichero de configuración del HPA para que consulte esta nueva métrica personalizada para el autoescalado.

Anexo H: Prueba de concepto

En la figura 28 se observa el envío del resto de mensajes de la prueba desde el cliente.

```
 DEBUG org.jsmpp.session.PDUProcessTask - Received PDU 00000018800000040000000000000003366330303037373900
jsmpp.session.AbstractSession - submit_sm response with sequence_number 3 received for session c250cd98
xample.examples.SimpleSubmitRegisteredExample - Message submitted, message_id is 6c00779
jsmpp.DefaultPDUSender - Sending PDU 00000052000000040000000000000004434d5400010031363136000100363238313736353034363537000000013233
c250cd98] DEBUG org.jsmpp.session.SMPPSession - Received PDU in session c250cd98 in state BOUND_TRX: 00000018800000040000000000000004
 DEBUG org.jsmpp.session.PDUProcessTask - Received PDU 00000018800000040000000000000004633463303333537 00
jsmpp.session.AbstractSession - submit_sm response with sequence_number 4 received for session c250cd98
xample.examples.SimpleSubmitRegisteredExample - Message submitted, message_id is c4c0357
jsmpp.DefaultPDUSender - Sending PDU 00000052000000040000000000000005434d5400010031363136000100363238313736353034363537000000013233
c250cd98] DEBUG org.jsmpp.session.SMPPSession - Received PDU in session c250cd98 in state BOUND_TRX: 00000018800000040000000000000005
 DEBUG org.jsmpp.session.PDUProcessTask - Received PDU 00000018800000040000000000000005613765383166620 00
jsmpp.session.AbstractSession - submit_sm response with sequence_number 5 received for session c250cd98
```

Figura 28. Prueba Concepto. Logs Cliente. Envío de múltiples mensajes

En la figura 29 se ve como el proxy recibe el segundo y tercer mensaje y le envía un submit_sm_resp para verificar que ha recibido los mensajes.

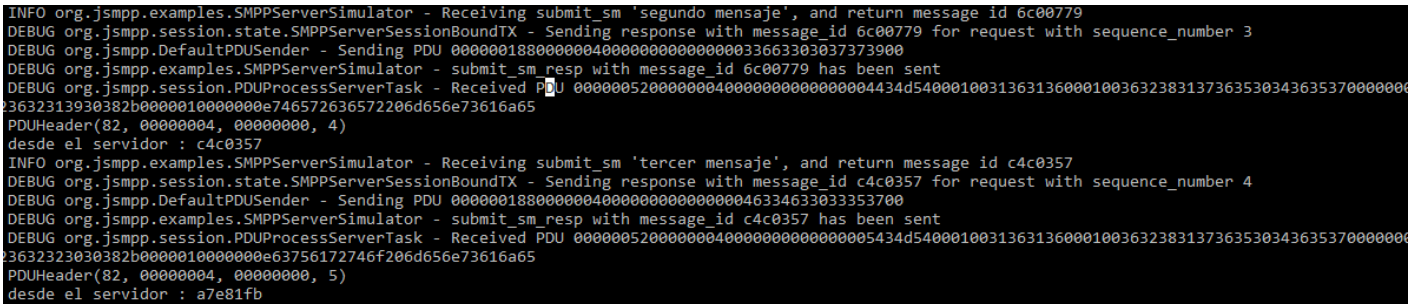

Figura 29. Prueba Concepto. Logs Proxy. Recepción de mensajes

En la figura 30 se aprecia como el proxy recibe la confirmación de que el cliente ha recibido el DLR del primer mensajes y también envía los DLR para los mensajes 2 y 3. En la figura 31 se observa como el cliente recibe estos DLR.

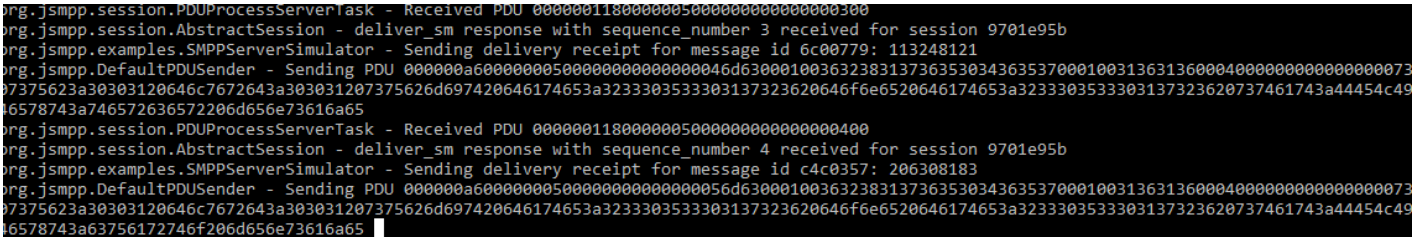

Figura 30. Prueba Concepto. Logs Proxy. Envío de DLR de los mensajes recibidos

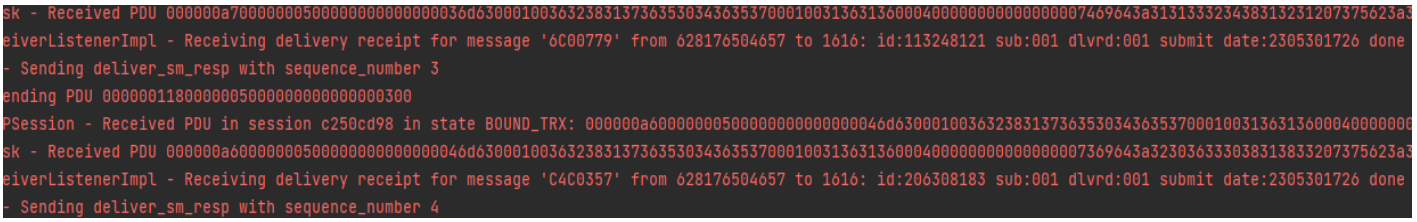

Figura 31. Prueba Concepto. Logs Cliente. Recepción de los DLR

También se ha realizado un caso de prueba donde desde el IRISGW nos indican que no se ha podido procesar el mensaje y nos manda un código de error 500 como se recoge en la figura 32. Esto implica que desde el servicio envio_mensaje no envíe un DLR para el cliente de forma que podrá identificar que ha habido un error en el envío del mensaje.

Figura 32. Prueba Concepto. Error en el envío del mensaje

Anexo I: Prueba de carga

En la figura 33 se observa el envío simultáneo de 10 mensajes y cómo se reciben las confirmaciones desde el proxy de la recepción de los mensajes.

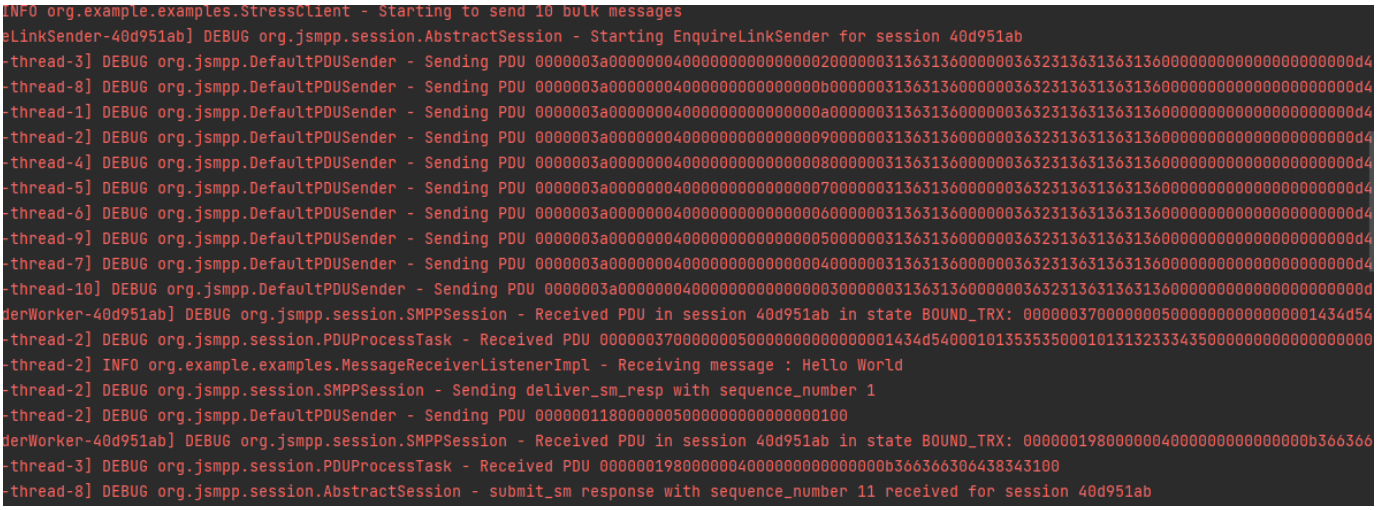

Figura 33. Prueba de carga. Resultados envío 10 mensajes

La figura 34 describe los resultados de la prueba. El tiempo total necesario para el envío de los 10 mensajes ha sido de de 2020 ms y el máximo delay para el envío del mensaje a ha sido de 1135 ms.

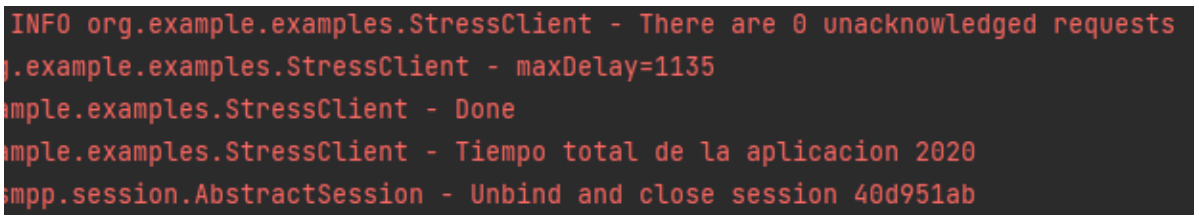

Figura 34. Prueba de carga. Resultados envío 10 mensajes

En la figura 35 se recoge el envío simultáneo de 100 mensajes SMS, esto implica que el sistema necesite las 10 réplicas disponibles del servicio envio_mensaje. Se observa cómo se reciben progresivamente los submit_sm_resp correspondientes a los mensajes enviados.

Figura 35. Prueba de carga. Envío 100 mensajes

La figura 36 describe los resultados de la prueba. El tiempo total necesario para el envío de los 100 mensajes ha sido de de 8070 ms y el máximo delay para el envío del mensaje a ha sido de 7223 ms.

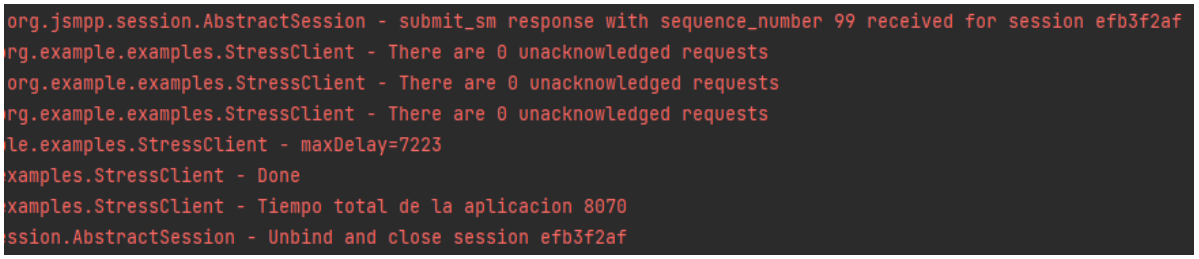

Figura 36. Prueba de carga. Resultados envío 100 mensajes con 10 réplicas

Posteriormente se aumentó el número de réplicas disponibles para el autoescalado de 10 réplicas a 20 réplicas. Se analizó si un aumento el número de réplicas se traduce en una reducción del tiempo innecesario para el envío de los mensajes o reducción en el delay de los mensajes de forma individual. Como se aprecia en la figura 37 se consigue una reducción considerable tanto en el tiempo total requerido como en el máximo delay de los mensajes. La reducción significativa del máximo Delay se debe a que con 20 réplicas ninguno de los mensajes tiene que esperar a que el módulo envíe previamente otro mensajes como sí ocurre cuando se tienen menos replicas.

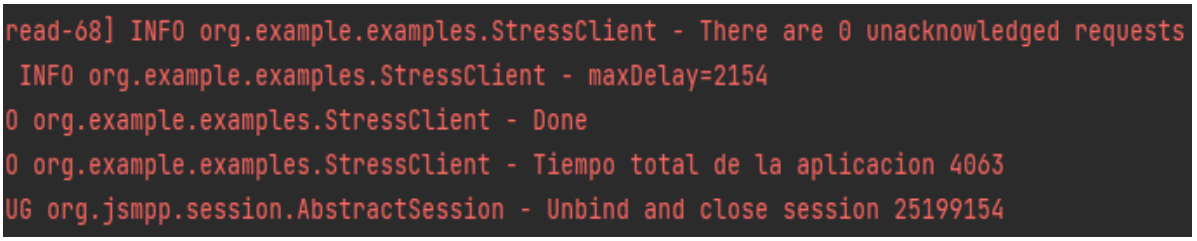

Figura 37. Prueba de carga. Resultados envío 100 mensajes con 20 réplicas

La última prueba consiste en el envío simultáneo de 1000 mensajes, al igual que en la prueba anterior se mantienen las 20 réplicas disponibles. Como se observa en la figura 38

el tiempo total necesario para el envío de los 10 mensajes ha sido de de 2020 ms y el máximo delay para el envío del mensaje a ha sido de 1135 ms.

```
org.example.examples.StressClient - maxDelay=5266
example.examples.StressClient - Done
example.examples.StressClient - Tiempo total de la aplicacion 6065
.jsmpp.session.AbstractSession - Unbind and close session bf0cbaf2
```

Figura 38. Prueba de carga. Resultados envío 1000 mensajes