

Redes de Computadoras 2020

# Sistemas distribuidos complejos

---

Alumnos:

Losano Quintana, Juan Cruz ([locxhalosano45@gmail.com](mailto:locxhalosano45@gmail.com))

Piñero, Tomás Santiago ([tom-300@hotmail.com](mailto:tom-300@hotmail.com))

Docentes:

Natasha Tomattis ([natasha.tomattis@mi.unc.edu.ar](mailto:natasha.tomattis@mi.unc.edu.ar))

Ayudantes alumnos:

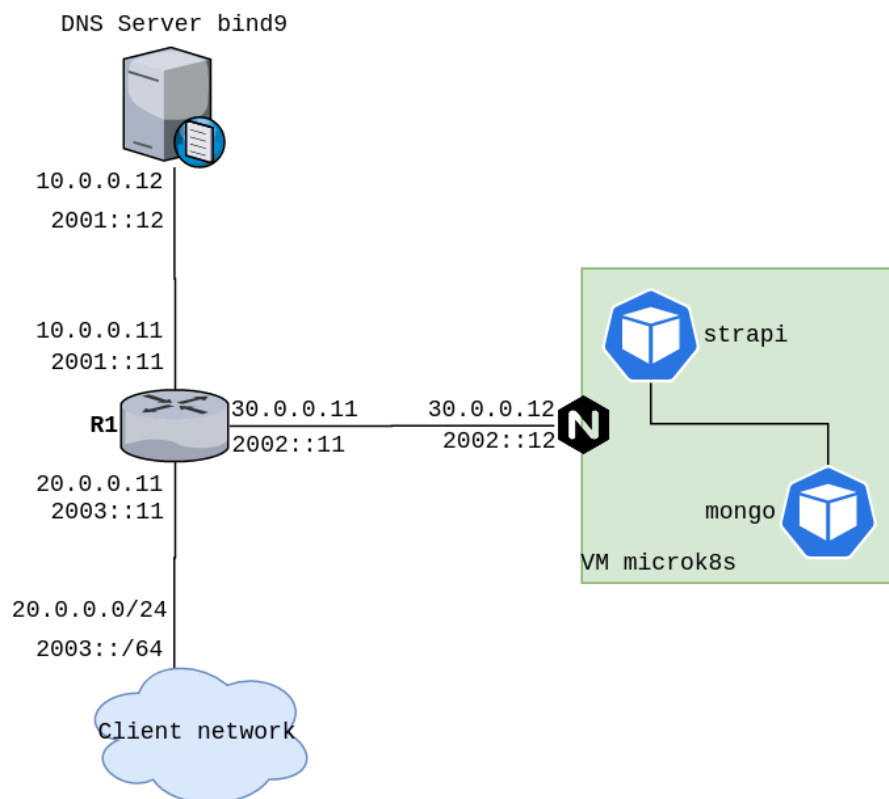
Aguerreberry Matthew, Sulca Sergio, Moral Ramiro, Soriano Juan

Mayo, 2020

## Requisitos

- Computadora por cada 2 personas

## Consigna



## Requerimientos mínimos:

- VM con microk8s instalado, en linux local correr los containers de la red del clientes, quagga router y dns server.
- Registros DNS A, AAAA y PTR para el cluster kubernetes.
- Soporte a clientes IPv6 (no es necesario que dentro del cluster se comuniquen a través de IPv6).
- Consultas a strapi desde client network con http y https.
- Habilitar autenticación en Strapi.
- Informe final con un resumen de cómo funciona el sistema y fuentes de información.

## Microk8s

Para alojar el *cluster* de *kubernetes* usamos una máquina virtual con sistema operativo Linux, *microk8s* y *docker* instalado en el Linux local.

Dentro del *cluster* existe un *nodo*, que contiene los dos *Pods* que se van a utilizar: uno para Strapi y otro para mongo. Strapi es un *headless CMS (Content Management System)* de código abierto basado en *Node.js* para todas las necesidades de API y gestión de contenidos. En este caso utilizamos *mongodb* para la base de datos de Strapi.

Primero creamos los archivos *yaml* para los *Pods*. Especificamos un *deployment* para strapi y un *statefulset* para mongo. Como strapi se demoraba mucho en iniciar cada vez que se eliminaban los *Pods*, decidimos hacerlo *statefulset* para tener la imagen lista en un volumen. Si bien no es correcto, es la idea que se nos ocurrió para evitar esos tiempos largos de espera. Lo ideal hubiera sido crear una imagen de docker con todas las dependencias de strapi listas. Para mongo se utilizó un *statefulset* dado que maneja una base de datos, necesita guardar información y manejar volúmenes. A ambos se les asignó un *VolumeClaim* de 1 gb cada uno. Los puertos utilizados fueron los puertos por defecto de cada aplicación: 1337 para strapi y 27017 para mongo.

```
green@virma:~$ kubectl get pods
NAME          READY   STATUS    RESTARTS   AGE
mongo-0       1/1     Running   6           2d23h
strapi-0      1/1     Running   6           2d23h
```

Fig. 1 - *Pods* existentes en el nodo.

Una vez creados los *Pods*, procedimos a escribir los archivos *yaml* para los servicios. En estos servicios se exponen el *pod* que contiene strapi y el que contiene mongo. Los dos servicios son del tipo *ClusterIP*, que no exponen una dirección IP para ser accedidos fuera del *cluster*. De esta forma se comunican el *pod* de strapi con el servicio de mongo, y el *ingress* con el servicio de strapi. Los servicios usan los puertos mencionados al final del párrafo anterior.

```
green@virma:~$ kubectl get services
NAME          TYPE        CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
kubernetes    ClusterIP   10.152.183.1  <none>       443/TCP    4d21h
m-service     ClusterIP   None          <none>       27017/TCP  2d23h
s-service     ClusterIP   10.152.183.171 <none>       1337/TCP   2d23h
```

Fig. 2 - Servicios disponibles.

Para poder acceder a los servicios que provee el cluster desde fuera del mismo, se puede exponer el servicio o se puede usar un *ingress*. En este caso implementamos un *ingress* que expone el servicio de strapi fuera del *cluster*. Dicho ingress utiliza *nginx* como *proxy reverso*, de esta forma el *proxy* maneja los *requests* de los clientes a un servicio específico dentro del *cluster*, los clientes solo necesitan el nombre de dominio que apunta a un servicio.

```
green@virma:~$ kubectl get ingress
NAME          CLASS    HOSTS             ADDRESS      PORTS      AGE
s-ingress     <none>   strapi.strapi.com 127.0.0.1    80, 443    2d23h
```

Fig. 3 - *Ingress* para acceder al servicio de strapi.

Por último, verificamos que todo estuviera funcionando en el *cluster* probando consumir el strapi desde el navegador del *host* de la máquina virtual. Para esto, modificamos el archivo */etc/hosts* y agregamos la entrada “127.0.0.1 strapi.strapi.com”. En la sección [“Conexión entre microk8s y docker”](#) vamos a explicar cómo configuramos el servidor DNS y HTTPS.

## Docker

Las direcciones IP utilizadas se muestran en la imagen de la consigna, pero mostramos tabla a continuación.

	Interfaz	IPv4	IPv6
R1	eth0	10.0.0.11/24	2001::11/64
	eth1	20.0.0.11/24	2002::11/64
	eth2	30.0.0.11/24	2003::11/64
DNS Server	eth0	10.0.0.12/24	2001::12/64
Cluster	mac0	30.0.0.12/24	2003::12/64
Client network	--	20.0.0.0/24	2002::/64

Tabla 1 - Direcciones IPv4 e IPv6 de la topología.

## Conexión entre microk8s y docker

La conexión entre la máquina virtual con nodo y el docker en el *host* se realizó por medio de una interfaz de tipo *bridge* en modo promiscuo (así puede capturar todo el tráfico que pasa por ella). Para crearla y configurarla, utilizamos los siguientes comandos:

```
~#: ip link add mac0 type bridge
~#: ip link set mac0 up
~#: ip link set mac0 promisc on
```

Tabla 2 - Configuración de interfaz *mac0*.

```
6: mac0: <BROADCAST,MULTICAST,PROMISC,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UNKNOWN group default qlen 1000
    link/ether 9a:d7:5c:c0:3d:b1 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::98d7:5cff:fec0:3db1/64 scope link
        valid lft forever preferred lft forever
```

Figura 4 - Interfaz *mac0*.

Con esta interfaz ya creada, los clientes que se encuentran en la *Client Network* (direcciones *20.0.0.0/24* y *2002::/64*) y el servidor DNS son capaces de llegar al *cluster* por medio del router R1.

En la máquina virtual con el *cluster* de kubernetes configuramos la interfaz en modo *bridge* conectado al *bridge mac0*, asignando las direcciones IP y *gateway*:

```
Link speed 1000 Mb/s
IPv4 Address 30.0.0.12
IPv6 Address 2003::12
Hardware Address 08:00:27:AE:D1:95
Default Route 30.0.0.11
DNS
```

Figura 5 - Interfaz de la VM.

En los archivos de configuración de red de docker, creamos la red que conecta la interfaz del router con la VM con tipo *macvlan*. De esta forma se puede exponer la MAC de la interfaz del router para que la VM la conozca. Se especifica que va a usar la interfaz *mac0*.

Page 10 of 10

---



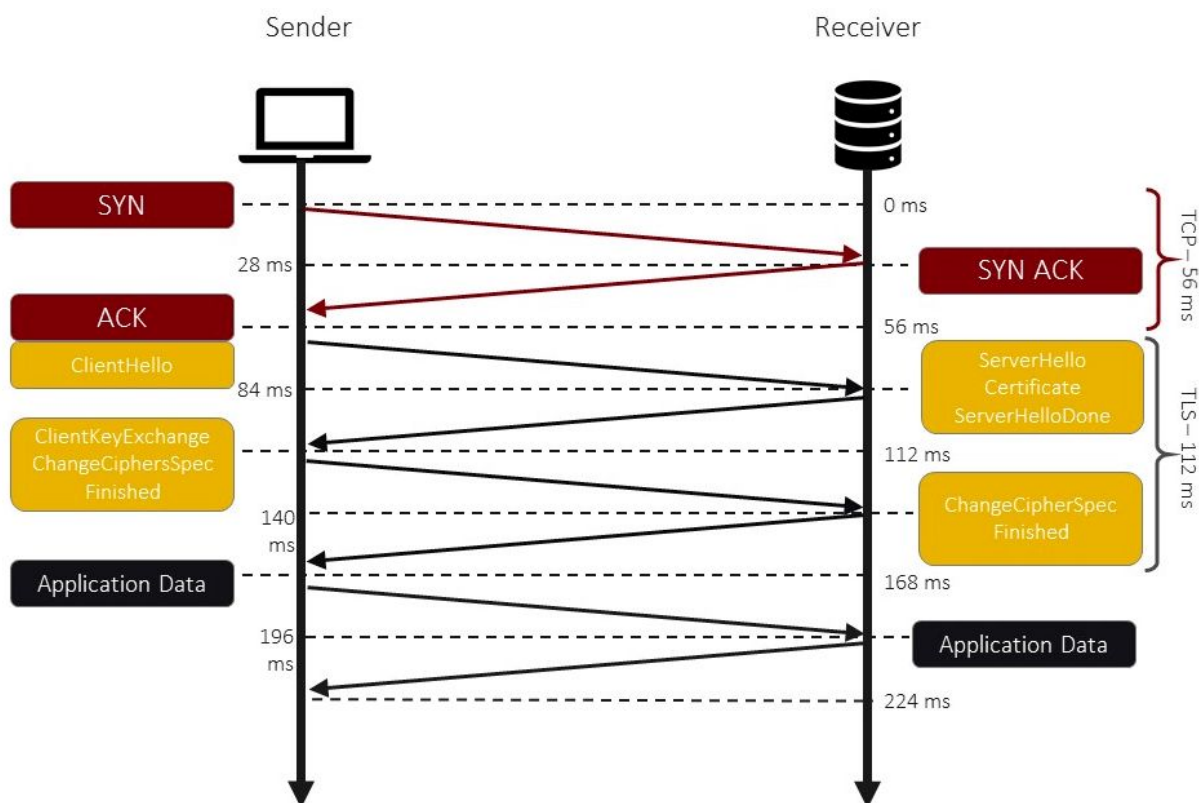
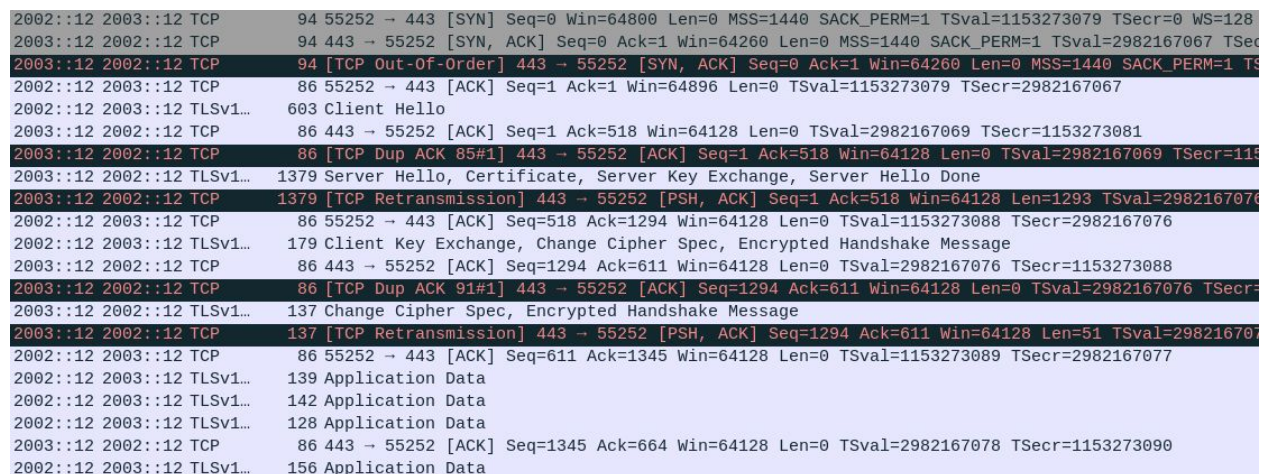


Fig. 8 - Diagrama de secuencia del HTTPS *handshake*.

Durante el *TLS handshake* el cliente y el servidor se ponen de acuerdo en qué versión de TLS y qué conjunto de cifrado utilizar; el cliente realiza la verificación del certificado del servidor y ambos generan las llaves para utilizar encriptación simétrica una vez terminado el *handshake*.

La siguiente figura muestra una captura del *handshake* con *Wireshark*.





```

2002::12 2003::12 TCP 94 55252 → 443 [SYN] Seq=0 Win=64800 Len=0 MSS=1440 SACK_PERM=1 TSval=1153273079 TSecr=0 WS=128
2003::12 2002::12 TCP 94 443 → 55252 [SYN, ACK] Seq=0 Ack=1 Win=64260 Len=0 MSS=1440 SACK_PERM=1 TSval=2982167067 TSecr=0
2003::12 2002::12 TCP 94 [TCP Out-Of-Order] 443 → 55252 [SYN, ACK] Seq=0 Ack=1 Win=64260 Len=0 MSS=1440 SACK_PERM=1 TSval=2982167067 TSecr=0
2002::12 2003::12 TCP 86 55252 → 443 [ACK] Seq=1 Ack=1 Win=64896 Len=0 TSval=1153273079 TSecr=2982167067
2002::12 2003::12 TLSv1... 603 Client Hello
2003::12 2002::12 TCP 86 443 → 55252 [ACK] Seq=1 Ack=518 Win=64128 Len=0 TSval=2982167069 TSecr=1153273081
2003::12 2002::12 TCP 86 [TCP Dup ACK 85#1] 443 → 55252 [ACK] Seq=1 Ack=518 Win=64128 Len=0 TSval=2982167069 TSecr=1153273081
2003::12 2002::12 TLSv1... 1379 Server Hello, Certificate, Server Key Exchange, Server Hello Done
2003::12 2002::12 TCP 1379 [TCP Retransmission] 443 → 55252 [PSH, ACK] Seq=1 Ack=518 Win=64128 Len=1293 TSval=2982167070 TSecr=1153273088
2002::12 2003::12 TCP 86 55252 → 443 [ACK] Seq=518 Ack=1294 Win=64128 Len=0 TSval=1153273088 TSecr=2982167076
2002::12 2003::12 TLSv1... 179 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
2003::12 2002::12 TCP 86 443 → 55252 [ACK] Seq=1294 Ack=611 Win=64128 Len=0 TSval=2982167076 TSecr=1153273088
2003::12 2002::12 TCP 86 [TCP Dup ACK 91#1] 443 → 55252 [ACK] Seq=1294 Ack=611 Win=64128 Len=0 TSval=2982167076 TSecr=1153273088
2003::12 2002::12 TLSv1... 137 Change Cipher Spec, Encrypted Handshake Message
2003::12 2002::12 TCP 137 [TCP Retransmission] 443 → 55252 [PSH, ACK] Seq=1294 Ack=611 Win=64128 Len=51 TSval=2982167077 TSecr=1153273088
2002::12 2003::12 TCP 86 55252 → 443 [ACK] Seq=611 Ack=1345 Win=64128 Len=0 TSval=1153273089 TSecr=2982167077
2002::12 2003::12 TLSv1... 139 Application Data
2002::12 2003::12 TLSv1... 142 Application Data
2002::12 2003::12 TLSv1... 128 Application Data
2003::12 2002::12 TCP 86 443 → 55252 [ACK] Seq=1345 Ack=664 Win=64128 Len=0 TSval=2982167078 TSecr=1153273090
2002::12 2003::12 TLSv1... 156 Application Data

```

Figura 9 - Captura del *handshake* HTTPS.

Para crear el certificado propio y la SK, utilizamos el siguiente comando del *toolkit openssl*, el cual se encarga de crear certificados y llaves privadas utilizando criptografía RSA (Rivest, Shamir y Adleman) de 2048 bits para el dominio indicado por <Dominio.com> con validez de 1 año.

```

~$: openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
<llave-privada.key> -out <certificado.crt> -subj
"/CN=<Dominio.com>/O=<Dominio.com>"

```

Tabla 4 - Comando *openssl* para crear el certificado y las keys.

En el cluster de kubernetes le asignamos la SK y el certificado para que sean usados cuando se acceda al servicio especificado en el *ingress*, mediante un “*secret tls*”. Un secreto es un objeto seguro que almacena datos sensibles. Reducen el riesgo de exposición de datos a usuarios no autorizados. Para crear el secreto usamos:

```

~$ kubectl create secret tls <Nombre-secreto> --key
<llave-privada.key> --cert <certificado.crt>

```

Tabla 5 - Comando para crear el secreto.

Así, en el *ingress* se especifica que para el servicio a donde apunta el dominio de *strapi*, en nuestro caso “*strapi.strapi.com*”, se utilice el certificado y la llave privada que se encuentran en el secreto. De esta forma cuando se accede a *strapi.strapi.com* el *ingress* puede compartir el certificado de *strapi*.



Figura 10 - Certificado.

Autenticacion Strapi

La autenticación en strapi consiste en autenticar usuarios que puedan realizar cambios en determinadas tablas de la base de datos. Nosotros como Admin creamos la tabla “alumnos” con los atributos: Nombre, Email, Matrícula e indicamos que para ejecutar métodos tales como *POST*, *GET*, *DELETE* sobre la tabla alumnos, los debe hacer un usuario autenticado.

**Permissions**

Only actions bound by a route are listed below.

APPLICATION — Define all your project's allowed actions.

Alumnos

<input type="checkbox"/> count	<input checked="" type="checkbox"/> create	<input checked="" type="checkbox"/> delete
<input checked="" type="checkbox"/> find	<input checked="" type="checkbox"/> findone	<input checked="" type="checkbox"/> update

Figura 11 - Permisos para usuarios autenticados.

Para realizar las consultas utilizamos *curl*, un ejemplo simple es el siguiente:

```
~$ curl --cacert strapi.crt https://strapi.strapi.com
```

Tabla 6 - Ejemplo *curl*.

Con la opción `--cacert <certificado.crt>` indicamos a *curl* que verifique el *peer* a *strapi.strapi.com* usando el certificado especificado. De esta forma se puede establecer una conexión segura. De otra manera, en vez de usar esa opción, se podría usar la *flag* `-k`, que indica que verifique cualquier conexión *tls* o *ssl* como segura, similar a cuando se “aceptan los riesgos” utilizando un navegador.

Procedimos a crear un usuario desde un cliente ubicado en la red de *docker*, utilizando el path `/auth/local/register` de *strapi* para registrar nuevos usuarios. Así se obtiene un *token* único para el usuario, permitiendo realizar las operaciones habilitadas anteriormente. La creación se puede observar en la Figura 12.

```
jclq@jclq-Inspiron-7591:~$ docker exec -ti pcl ash
/ # curl --cacert strapi.crt -d "@auth.json" -H "Content-Type: application/json"
-X POST https://strapi.strapi.com/auth/local/register
{"jwt":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjVlZThlZDI2YTMzNGY4MDAwZDFjNDAYZCIsImIhdCI6MTU5MjMyMzNiwiZXhwIjoxNTk0TE1MzY2fQ.j5REsB0W76tcphrivsZGyAx3qt5PYdgpvwWQ AUTJk","user":{"confirmed":true,"blocked":false,"_id":"5ee8ed26a334f8000d1c402d","username":"UsuarioRandom","email":"usuariorandom@strapi.com","provider":"local","createdAt":"2020-06-16T16:02:46.334Z","updatedAt":"2020-06-16T16:02:46.400Z","_v":0,"role":{"_id":"5ede98ccad75ab0088ce81b3","name":"Authenticated","description":"Default role given to authenticated user.","type":"authenticated","createdA
```

Figura 12 - Creación de usuario.

Luego, con ese usuario creamos un alumno con el método *POST*, pasándole un *json* con información necesaria. El *POST* se hace sobre el path `/alumnos` y agregamos un *header* con el *flag* `-H` que contiene el *token* del usuario.

```
jclq@jclq-Inspiron-7591:~$ docker exec -ti pcl ash
/ # curl --cacert strapi.crt -d "@post.json" -H "Content-Type: applicati
on/json" -H "Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJpZCI6IjVlZThlZDI2YTMzNGY4MDAwZDFjNDAYZCI6ImVhdCI6MTU5MjMyMzY2fQ.j
5REsB0W76tcphrivsZGyAx3qt5PYdgpvwWQ AUTJk" -X POST
https://strapi.strapi.com/alumnos
{"_id":"5ee8ef60a334f8000d1c402e","nombre":"Jose de San Martin","matricu
la":12345678,"email":"donjose@donpepito.org","createdAt":"2020-06-16T16:
12:16.434Z","updatedAt":"2020-06-16T16:12:16.434Z","__v":0,"id":"5ee8ef6
0a334f8000d1c402e"}/ #
```

Figura 13 - Agregando un alumno con autenticación

Si no se pasa el *token* en la *request*, strapi nos dice que no tenemos el acceso permitido.

```
jclq@jclq-Inspiron-7591:~$ docker exec -ti pcl ash
/ # curl --cacert strapi.crt https://strapi.strapi.com/alumnos
{"statusCode":403,"error":"Forbidden","message":"Forbidden"}/ #
```

Figura 14 - GET sin *token*.

Nuevamente, con el token se puede hacer el GET sobre /alumnos.

```
jclq@jclq-Inspiron-7591:~$ docker exec -ti pcl ash
/ # curl --cacert strapi.crt -H "Authorization: Bearer eyJhbGciOiJIUzI1N
iIsInR5cCI6IkpXVCJ9.eyJpZCI6IjVlZThlZDI2YTMzNGY4MDAwZDFjNDAYZCI6ImVhdCI6
MTU5MjMyMzY2fQ.j5REsB0W76tcphrivsZGyAx3qt5PYdgpvwWQ AUTJk" https://strapi.strapi.com/alumnos
[{"_id":"5ee8ef60a334f8000d1c402e","nombre":"Jose de San Martin","matric
ula":12345678,"email":"donjose@donpepito.org","createdAt":"2020-06-16T16
:12:16.434Z","updatedAt":"2020-06-16T16:12:16.434Z","__v":0,"id":"5ee8ef
60a334f8000d1c402e"}]/ #
```

Figura 15 - GET con *token*.



## Fuentes

- [1] Strapi, <https://strapi.io/>
- [2] Docker Hub, mongo image, [https://hub.docker.com/\\_/mongo](https://hub.docker.com/_/mongo)
- [3] Docker Hub, strapi image, <https://hub.docker.com/r/strapi/strapi>
- [4] Recording Checkpoint 2 y 3, Redes de computadoras 2020, [https://drive.google.com/file/d/1j4\\_01t5TWHD63xXUUSypQhvJmjiMWMar5/view](https://drive.google.com/file/d/1j4_01t5TWHD63xXUUSypQhvJmjiMWMar5/view)
- [5] Medium, Deploy Strapi on Kubernetes with HTTPS, <https://medium.com/swlh/deploy-strapi-on-kubernetes-with-https-1e3437bf2ca3>
- [6] How HTTPS works, <https://howhttps.works/>
- [7] Love2dev, How HTTPS works, <https://love2dev.com/blog/how-https-works/>
- [8] Kinsa, TLS vs SSL: ¿Cuál es la diferencia? ¿Cuál debería usar?, <https://kinsta.com/es/base-de-conocimiento/tls-vs-ssl/>
- [9] Cloudflare, What happens in a TLS handshake?, <https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/>
- [10] Kubernetes, Documentation, <https://kubernetes.io/es/docs/home/>
- [11] Lorenzo Fissore, Crear una bridge interface en modo promiscuo, Slack de Redes de computadoras 2020, canal 'practico\_6'