TSR - PRÁCTICA 3
CURSO 2020/21

DESPLIEGUE DE SERVICIOS

El laboratorio 3 se desarrollará a lo largo de tres sesiones. Sus objetivos principales son:

> "Que el estudiante comprenda algunos de los retos que conlleva el despliegue de un servicio multi-componente, presentándole un ejemplo de herramientas y aproximaciones que puede emplear para abordar tales retos"

Esta práctica requiere conocimientos asociados al tema 4 (**Despliegue**), y depende de la práctica 2, **ØMQ**, especialmente en lo relativo al sistema *client-broker-worker* (**cbw**) con socket ROUTER-ROUTER.

Las actividades incluyen 7 hitos, cada uno concretado en un directorio con un nombre.

La **primera sesión** se refiere al manejo básico de Docker, y debería incluir la generación de la imagen inicial (0_PREVIO), dos actividades (1 y 2) sobre el esquema cbw router-router de la práctica 2, y se presenta el sistema de referencia `cbw_ftc` consistente en una reimplementación de cbw con tolerancia a fallos (`ft`) y clases de trabajo (`c`).

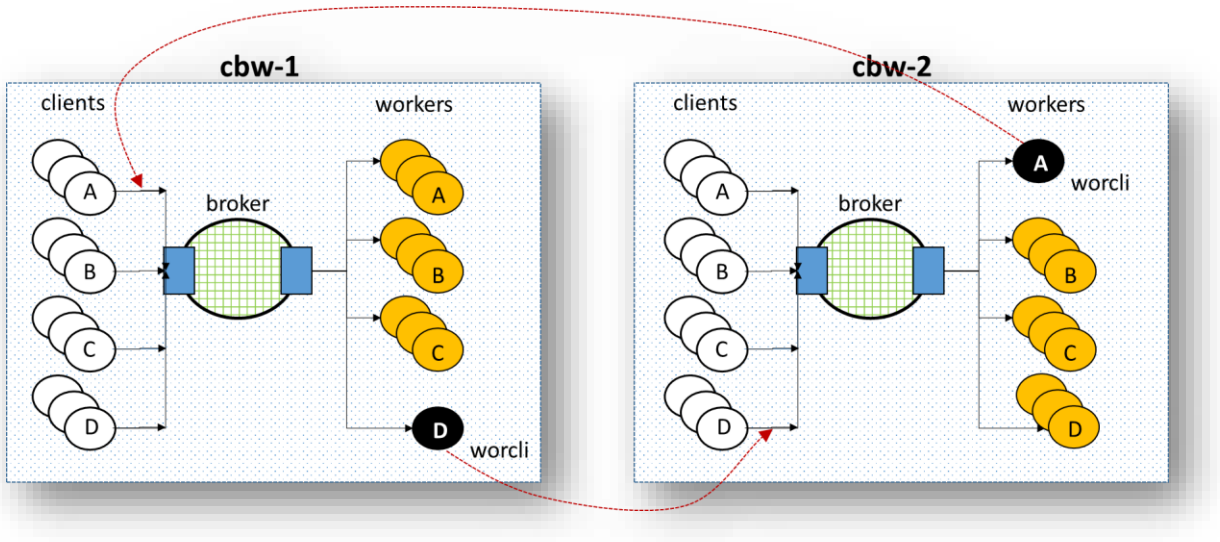
- Desde este momento ya no se necesitará modificar el código de clientes y trabajadores.

En la **segunda sesión** se espera finalizar el despliegue de `cbw_ftc`, realizar pruebas de funcionamiento y se añade un nuevo agente (`logger`) que debe ser integrado en el despliegue (4_CBW_FTCL).

- Ya no se necesitará modificar el código de clientes, trabajadores, broker ni logger.

También se pretende estudiar la incorporación de agentes externos al anfitrión, como un cliente ejecutándose en los equipos de escritorio (5_CBW_FTCL_CLEXT)

La tercera y **última sesión** propone la interconexión de dos despliegues (`6_CBW_FTCL_WORCLIEXT`) incorporando un nuevo agente (`worcli`) que intermedia sin alterar las interfaces de los demás componentes. Es una actividad que se apoya en todas las anteriores.



Para esta práctica se necesita:

1. Los materiales accesibles en PoliformaT (`tsr_lab3_material.zip`) en el directorio correspondiente a la tercera práctica. Puedes comparar su contenido con la imagen del último anexo.
   - Una **máquina virtual de portal**, que contiene ya una instalación de Docker y permite realizar los ejercicios planteados en esta práctica. En caso de que el servicio (demonio) `docker` no se encuentre en marcha, deberás ejecutar:

```
systemctl start docker
```

CONTENIDOS

0 Introducción 4
1 Sesión 1. Primeros pasos con Docker 6
1.1 Construyendo la imagen base con Ubuntu, NodeJS y ØMQ 6
1.2 Despliegue de las imágenes individuales de client/broker/worker 6
1.3 Desplegando cbw de la práctica 2 7
1.4 Nueva base para alta disponibilidad y clases de trabajo 7
2 Sesión 2. Despliegues de nivel intermedio 9
2.1 Anotando los diagnósticos 9
2.2 El componente logger y su efecto en el broker 10
2.3 Nuevas dependencias de los componentes 10
2.4 Acceso a almacenamiento duradero desde logger 10
2.5 Despliegue conjunto del nuevo servicio CBW_FTCL 11
3 Aspectos previos a la última sesión 11
3.1 Redefiniendo el servicio y cliente externo 12
4 Sesión 3. Encadenando varios sistemas cbw: el worcli 13
4.1 Preguntas sobre worcli 14
5 ANEXOS 15
5.1 Previo: construir imagen tsr2021/ubuntu-zmq 15
5.2 Anexo 1_A mano 15
5.3 Anexo 2_CBW (básico) 16
5.4 Anexo 3_CBW_FTC (clases de trabajo y tolerancia a fallos) 17
5.5 Anexo 4_CBW_TFCL (clases de trabajo, tolerancia a fallos y logger) 21
5.6 Anexo 6_CBW_FTCL_WORCLI-EXT 23

0 INTRODUCCIÓN

Los servicios son el resultado de la ejecución de una o varias instancias de cada uno de los componentes software empleados para implementarlos.

1. Uno de los problemas en el momento del despliegue de un servicio es el **empaquetamiento** de cada uno de sus componentes de manera que la instanciación de esos componentes sea repetible, y que la ejecución de las instancias de componentes se aísle de la ejecución del resto de instancias de cualquier componente.
2. Otro problema a abordar consiste en la **configuración** de cada una de las instancias a desplegar.
3. También destacamos la necesidad de especificar la **interrelación** entre los diferentes componentes de una aplicación distribuida, especialmente el enlace entre *endpoints*: cómo se pueden definir y resolver. Una forma de comprobar que dicha relación está bien construida será sometiendo el servicio a operaciones de **escalado**.
4. Por último, una aplicación distribuida real contempla mayor variedad de agentes y situaciones que las que vemos aquí, afectando a componentes y anfitriones. Para iniciarnos en esta complejidad introducimos **variaciones** que inicialmente pueden parecer incompatibles con nuestra infraestructura de laboratorio.

Una gran parte de los conceptos que se ponen en juego en esta práctica tienen su base en el escenario descrito en el primer ejemplo del apartado 6.5.2 de la guía del alumno del tema 4, aunque también intervienen otros condicionantes prácticos que no pueden ser ignorados.

En este laboratorio exploramos formas de construir componentes, configurarlos, conectarlos y ejecutarlos para formar aplicaciones distribuidas escalables de una forma *razonablemente* sencilla. Para ello nos dotamos de tecnologías especializadas en este ámbito.

1. Nos enfrentamos al primer problema con la ayuda del *framework* Docker. Tal y como se ha estudiado en el tema 4, Docker nos provee de herramientas para preparar de forma reproducible toda la pila software necesaria para la instanciación de un componente.
2. Para resolver adecuadamente el segundo problema se necesita especificar la configurabilidad de cada componente. Dada nuestra elección de tecnología (**Docker**) deberemos entender cómo dar a conocer las dependencias para que el *framework* de Docker las resuelva. Concretamente necesitaremos conocer cómo cumplimentar un ***Dockerfile*** y cómo referenciar a informaciones contenidas en él.
    - Es imprescindible que el código a desplegar sea configurable; en caso contrario no se podrá adaptar a los detalles procedentes de cada despliegue concreto.
3. Para abordar la tercera necesidad descrita procederemos incrementalmente a partir del esquema **cbw** con *router-router* de la práctica 2.
    - Inicialmente desplegaremos todos los componentes de un servicio manualmente, usando las informaciones de configuración como parámetros de las órdenes docker.
    - Posteriormente automatizaremos esta actividad mediante **docker-compose**. Esto nos obligará a entender los fundamentos de este nuevo programa, y la especificación necesaria para construir el fichero **docker-compose.yml** con la interrelación entre los componentes de nuestra aplicación distribuida.

- Añadiremos **nuevos componentes** y situaciones, modificaremos el código necesario en los otros componentes y trazaremos nuevos planes de despliegue.

El escalado forma parte de la funcionalidad ofrecida por **docker-compose**. La dificultad principal radica en la adecuación de los componentes de la aplicación distribuida para permitir y aprovechar dicho escalado.

4. Para disponer de un punto de partida realista, añadimos tolerancia a fallos y soporte para clases de trabajo. Es muy complejo comprender el código resultante si no se entienden las funcionalidades básicas introducidas previamente. Tomando este sistema como punto base, realizamos las siguientes actividades:
   - Despliegue del sistema: los cambios en los componentes involucrados introducen diferentes clases de clientes y trabajadores.
   - Pruebas del sistema: comprobar la reacción ante situaciones de fallo de trabajadores.
   - Separar el logger (no forma parte del mismo despliegue), duplicar cbw_ft_class y desplegar cada uno de los 3 sistemas (primero logger, anotar IP e incorporar esa información en el despliegue de los otros 2)

Los ejercicios concretos que se exponen utilizan aplicaciones realizadas por nosotros (código en NodeJS, con módulos) que podemos modificar para alcanzar nuestros objetivos.

1 Sesión 1. Primeros pasos con Docker

1.1 Construyendo la imagen base con Ubuntu, NodeJS y ØMQ

Necesitamos el punto de partida con el que estamos familiarizados, pero aplicado a nuestros contenedores. El ejemplo 1 del apartado 6.5.2 de la guía del alumno del tema 4 resume cómo generar `tsr2021/ubuntu-zmq`. Te incluimos el Dockerfile y la orden indicados.

1.1.1 Dockerfile

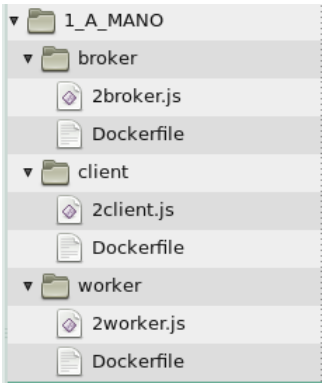
```
FROM ubuntu:focal
RUN apt-get update
RUN apt-get install -y curl ufw gcc g++ make gnupg
RUN curl -sL https://deb.nodesource.com/setup_12.x | bash -
RUN apt-get -y install nodejs
RUN apt-get -y upgrade
RUN npm install zeromq@5
```

1.1.2 Orden

```
docker build -t tsr2021/ubuntu-zmq .
```

Comprueba que la imagen existe tras ejecutar la orden (con *docker images*).

1.2 Despliegue de las imágenes individuales de client/broker/worker



Debes generar las imágenes de los tres componentes, pero necesitas **adaptaciones** respecto al código original de la práctica 2:

- El código del cliente debe emitir 10 solicitudes[1] antes de finalizar.
- Los clientes toman 1 parámetro de la línea de órdenes: el URL del socket correspondiente del broker
- Los workers toman 1 parámetro de la línea de órdenes: el URL del socket correspondiente del broker
- El broker toma 2 parámetros de la línea de órdenes: el puerto del socket para clientes, y el puerto del socket para workers. También se realiza un *cambio menor* en el código.

Sin esta preparación los componentes no podrán interconectarse. A las versiones modificadas les colocamos un prefijo "2" en la ilustración anterior, en la que cada componente cuenta con su propia carpeta con Dockerfile, código fuente y los archivos que necesite.

Los detalles se incluyen en el anexo correspondiente.

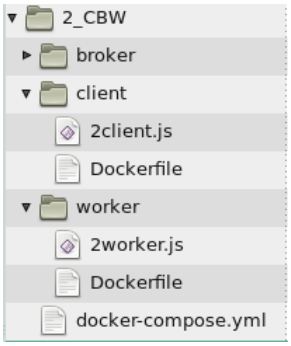
A continuación debes generar (*¿ya sabes cómo?*) 3 imágenes (`imclient`, `imbroker`, `imworker`), abrir 5 ventanas y ejecutar estas instrucciones para comprobar que todo encaja:

- Ventana 1: `docker run imbroker`.
  - Averigua y anota la IP que recibe el contenedor que ejecuta `imbroker`, y modifica adecuadamente los Dockerfiles de los otros.

[1] Debes realizar la modificación

  - La orden necesaria es `docker inspect`, pero debes averiguar el identificador del contenedor
- Ventanas 2 y 3: `docker run imworker`
- Ventanas 4 y 5: `docker run imclient`

1.3 Desplegando cbw de la práctica 2



El apartado **6.7.4** de la guía del alumno del tema 4 menciona cómo construir un despliegue orquestado de varios componentes para crear una aplicación (**cbw**) distribuida.

Desde el material del apartado anterior únicamente necesitamos aplicar cambios en el archivo de despliegue (`docker-compose.yml`) para poder crear variables de entorno ($BROKER_URL) que puedan aprovecharse en el nuevo Dockerfile que deberás crear para clientes y trabajadores.

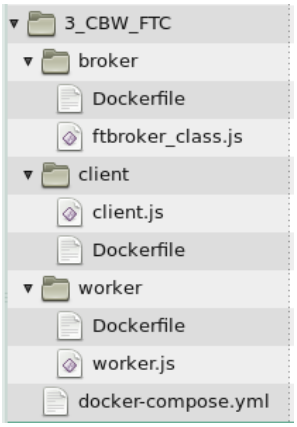
Más información en el anexo 2_CBW. ¿Puedes prever algún problema por el uso que se hace de `identity`? No puede ser una constante, sino un valor aleatorio, ya que no puede haber dos clientes con la misma identidad.

Para ejecutar 4 clientes y 2 trabajadores usaremos:

```
docker-compose up --scale cli=4 --scale wor=2
```

A partir del siguiente apartado cambiamos el código base de los componentes

1.4 Nueva base para alta disponibilidad y clases de trabajo



Como recordarás, a lo largo de la práctica 2, específicamente en el apartado 7 de la "**Documentación de referencia sobre ØMQ**" (`RefZMQ`) se proponen modificaciones sobre el código de varios de los componentes que forman el sistema **cbw** con broker ROUTER-ROUTER. De todos ellos seleccionamos la unión de estas modificaciones:

- Patrón broker tolerante a fallos para que el broker (`ftbroker_class.js`) detecte y recupere fallos de workers (descrito y resuelto en el apartado 7.1 de la documentación citada de la práctica 2).
- Modificación para permitir **tipos de trabajo**, especificados por clientes, admitidos por trabajadores, y soportados por el broker. Esta modificación fue descrita, pero no resuelta, en la práctica 2.

Sin embargo, cambiamos de código de referencia para el broker, con una **implementación diferente** que facilita cambios posteriores. En resumen…

La **tolerancia a fallos** se consigue mediante un temporizador que establece una ventana de tiempo en el que se espera la respuesta. Si no llega en ese intervalo, se interpreta que esa respuesta ya no llegará, y se reenvía este trabajo a otro de la misma clase.

El soporte para **clases de trabajo** puede conseguirse con, al menos, dos alternativas:

- Colocar un broker diferente para cada clase. Es una opción interesante si pueden unificarse los puntos de servicio en uno solo.
- Multiplicar muchas estructuras internas para separar colas por tipos de trabajo. Ésta ha sido la opción seleccionada.

Por otro lado, el nuevo broker toma los mensajes con una construcción genérica (`arguments`) de la que se debe extraer la información deseada, y el broker es el único encargado de mostrar información de seguimiento por pantalla.

Es oportuno reflexionar acerca del archivo de configuración del despliegue (`docker-compose.yml`). Si abordamos el despliegue de 2 tipos de componente (trabajadores y clientes) tomando cada una de las 3 clases de trabajo como un elemento diferencial, necesitaremos 2 x 3 = 6 casos. Nos encontraremos con una *"explosión"* de entradas en el archivo de configuración del despliegue y en la estructura de directorios que requiere. ¿Cómo puede abordarse?

1. En lo relativo al **fichero de configuración del despliegue**, si deseamos un control fino sobre el número de instancias de cada tipo, nos obligamos a poder referenciarlas por separado en la orden `docker-compose run`, como argumento que acompaña a la opción `--scale`. Dicho de otra forma, no podemos controlar esos componentes si no los diferenciamos en la configuración del despliegue, reconociendo la imposibilidad de simplificación en este aspecto.
2. Sin embargo, en lo relativo a la estructura de directorios sí que podemos reducir su complejidad a cambio de modificar el código de los componentes client y worker: la idea es suministrar el tipo de trabajo como un **parámetro de invocación que puede ser establecido en el despliegue** (como el URL del broker).
    - De esta forma contaríamos con un único código y Dockerfile para todos los clientes, y otro par para todos los trabajadores.

Dispones del código y otras informaciones en el anexo correspondiente. Debes desplegarlo y escalar a 3 clientes y 2 trabajadores de cada clase.

Antes de iniciar el despliegue debes observar que las imágenes a utilizar en este apartado tienen los mismos nombres que las utilizadas en el apartado anterior, pero sus programas son diferentes. Debido a esto, convendrá eliminar previamente las generadas en 2_CBW. Revisa qué órdenes deberás utilizar para ello.

¿Qué debes valorar para comprobar que el funcionamiento es correcto?

- Cada cliente recibe respuesta a SU petición y no a las de otros
- No hay trabajadores libres si quedan peticiones pendientes de su clase
- Comprobar tolerancia a fallos: hacer fallar un trabajador mientras atiende una petición y comprobar la reacción

Diseña modificaciones o formas de uso que faciliten las anteriores comprobaciones.

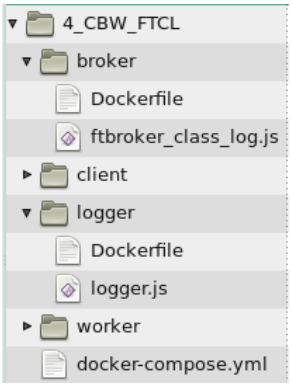
1.4.1 Pregunta

Respecto a las clases de trabajos, tanto en número como en cuanto a sus nombres, el broker puede conocer esos valores **estáticamente**, sin posibilidad de modificación a lo largo de su ejecución, o bien **dinámicamente**, una vez ya se encuentra en funcionamento. En la implementación incluida en esta práctica se ofrece un comportamiento en el que los tipos de petición se van descubriendo a medida que los mensajes pasan por el broker.

- Esta flexibilidad se encuentra muy limitada si en el despliegue necesitamos mencionar las clases de trabajo existentes.
- Reflexiona sobre la posibilidad de idear alguna alternativa en la que no se necesite incluir los tipos en `docker-compose.yml`, pero permitiendo controlar el número de instancias desplegadas para cada tipo.

2 SESIÓN 2. DESPLIEGUES DE NIVEL INTERMEDIO

2.1 Anotando los diagnósticos



Los componentes deben mostrar diagnósticos para notificar cómo progresan y si hay incidencias. Es frecuente que una buena parte de esas notificaciones empleen la salida estándar, pero es una práctica extendida que, salvo urgencia, **esos diagnósticos se acumulen** cronológicamente en algún archivo para su posterior consulta; de hecho incluso existen formatos reconocidos para dichas anotaciones que permiten a aplicaciones externas *digerir* esa información. No es nuestro caso.

Podríamos elegir que cada componente guarde sus anotaciones en un fichero, pero no es cómodo contar con muchas fuentes de información. Si pretendiésemos que todos los componentes anotaran directamente sus diagnósticos en un único fichero centralizado, estaríamos completamente fuera de lugar… ¿un sistema distribuido con un fichero compartido?. **¡Descalificad@!**.

Como posible alternativa podemos desarrollar un componente (***logger***) capaz de recibir órdenes de escritura equivalentes a los `console.log()`

- Por simplicidad, solo usaremos este servicio desde el componente broker, pero es sencillo generalizarlo a todos los demás.

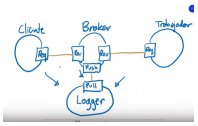
Aspectos destacables:

- Es importante que el archivo usado por el logger no pierda su contenido entre invocaciones. Recuerda que la naturaleza efímera de los contenedores es aquí un problema a resolver. Necesitarás usar un volumen Docker para conectar ese fichero con un espacio del anfitrión.
- Si algún componente debe considerar la existencia de este *logger*, deberá formar parte de su despliegue y será una dependencia a resolver.

- Un asunto interesante es el tipo de socket ØMQ aplicable: PULL para *logger* y PUSH para el resto será suficiente (aunque otras variaciones permitirían otras características).

2.2 El componente logger y su efecto en el broker

Se comunica con el resto de procesos actuando como un recolector, con patrón *push-pull*. Su código, disponible en el anexo sobre CBW_FTCL, es sencillo para quien ya ha experimentado ØMQ. Además de la elección de sockets destaca la escritura en fichero.



Por claridad únicamente el broker hace uso del servicio de anotaciones, por lo que el código y el despliegue del broker deberán tenerlo en cuenta. Necesitamos tres pequeñas cosas…

- Un socket tipo PUSH para conectar con el componente logger

```
sl    = zmq.socket('push')
```

- Tomar el URL del logger como argumento

```
var lURL  = args[2]
```

- En la función `annotate`, cambiar console.log por envío del mensaje

```
function annotate(prefix, id, cid) {
  var str=util.format(prefix+' %s (class "%s")', id, cid)
  //console.log(str)
  sl.send(str)
}
```

El Dockerfile del logger es prácticamente idéntico a otros ya estudiados, destacando el argumento con la ruta del directorio del contenedor (¡¡no lo confundas con el del anfitrión!!)

El código completo de ambos componentes se encuentra en el anexo.

2.3 Nuevas dependencias de los componentes

El broker necesitará conocer cómo conectar con *logger*. Esta situación es similar a la que ya relacionaba cliente y trabajador con el broker, y supone la necesidad de colocar una variable de entorno a sustituir en el despliegue. La última línea del Dockerfile del broker quedará:

```
CMD node mybroker 9998 9999 $LOGGER_URL
```

2.4 Acceso a almacenamiento duradero desde logger

Es necesaria una consideración que no nos había preocupado hasta ahora: ¿cómo se relaciona el directorio con anotaciones (`/tmp/cbwlog`) del contenedor con el sistema de ficheros del anfitrión?. Mediante una sección `volumes`[2] en la descripción del despliegue.

Supone que ya hemos creado el directorio `/tmp/logger.log` en el anfitrión

Si únicamente deseamos desplegar este componente, y no toda la aplicación distribuida, deberemos emplear una invocación de `docker run` con una opción equivalente a la sección `volumes`

```
docker run –v /tmp/logger.log:/tmp/cbwlog parámetros
```

[2] Dispones de información adicional en el material de referencia del tema 4

2.5 Despliegue conjunto del nuevo servicio CBW_FTCL

En este fragmento del archivo de configuración del despliegue de la aplicación CBW_FTCL se muestra únicamente las modificaciones respecto al despliegue sin *logger*.

```
version: '2'
services:

...

  bro:
    image: broker
    build: ./broker/
    links:
      - log
    expose:
      - "9998"
      - "9999"
    environment:
      - LOGGER_URL=tcp://log:9995
  log:
    image: logger
    build: ./logger/
    expose:
      - "9995"
    volumes:
# /tmp/logger.log DIRECTORY must exist on host and be writeable
      - /tmp/logger.log:/tmp/cbwlog
    environment:
      - LOGGER_DIR=/tmp/cbwlog
```

La puesta en marcha con `docker-compose` no tiene ninguna novedad. Es interesante que, desde el anfitrión, puede accederse a las anotaciones en el directorio `/tmp/logger.log`.

Prueba a realizar: el despliegue básico requiere un archivo de anotaciones vacío, y la ejecución de una combinación compuesta por 4 clientes de tipo B, 2 trabajadores de tipo B, 1 broker y 1 logger.

Cuestión: reflexiona, sin necesidad de ejecutar, qué ocurriría si intentáramos desplegarlo en los siguientes escenarios:

- 2 clientes B, 1 trabajador B, 2 brokers, 1 logger
- 2 clientes B, 1 trabajador B, 1 broker, 2 loggers

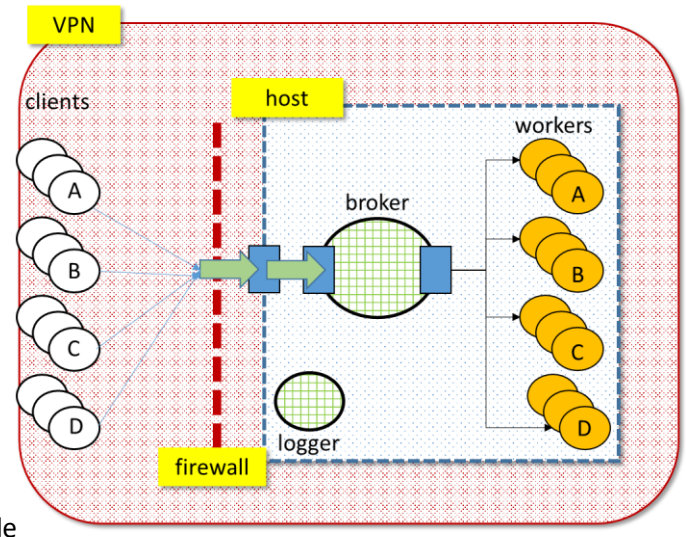
3 ASPECTOS PREVIOS A LA ÚLTIMA SESIÓN

El modelo básico **cbw** con sus enriquecimientos sigue encontrándose distanciado de los problemas técnicos que pueden encontrarse en una aplicación distribuida. En este apartado pretendemos incorporar algunos detalles que nos lleven en esa dirección, de manera que podamos aplicarlos en la última sesión.

El primer paso que tomamos nos facilita la reducción parcial de la complejidad de las clases de trabajo; el segundo se refiere a la reformulación del papel que juegan los clientes en este esquema, y cómo se habilita el acceso desde el exterior del anfitrión; el tercero cubre la adición de un nuevo componente (**worcli**) que aprovecha la preparación anterior.

3.1 Redefiniendo el servicio y cliente externo

Los clientes no son parte de la aplicación distribuida, por lo que deberán interactuar con el servicio mediante algún punto bien conocido para poder encargarle trabajo. Problemas que aparecen:



¿Cuál es el *endpoint* del servicio? El URL del broker.

- Si su IP puede cambiar en cada ejecución, no funcionará.
- Si los clientes se encuentran fuera del anfitrión que hospeda al resto de componentes, tendremos un problema de acceso (las IPs de Docker son locales dentro del anfitrión).

Ambos problemas pueden ser resueltos reservando un puerto del anfitrión que se hará corresponder con la IP y puertos del broker en el despliegue.

- Si el cortafuegos del anfitrión (*host*) se encuentra configurado correctamente, la línea `ports` de `docker-compose.yml` hará el truco.
- Los equipos del portal en TSR ya están configurados para permitir el acceso desde el exterior a los puertos 8000 a 9000. Para ampliar este soporte hasta el puerto 9999 ejecutamos:

```
ufw allow 9001:9999/tcp
```

Ahora podremos disponer de clientes externos que conectarán con el servicio mediante un URL fijo `tcp://hostIP:9998` que conducirá las peticiones al broker.

- Los **requisitos** de los clientes para su ejecución (NodeJS + ØMQ) **se han satisfecho en el arranque LINUX de todos los equipos de escritorio del DSIC**.
    - En las virtuales de escritorio de los laboratorios ya se dispone de NodeJS, aunque sabemos que el uso de puertos puede ser conflictivo si se usan operaciones tipo `bind` o `listen`.
        - Las de tipo `connect`, como en clientes y trabajadores, NO plantean problemas.
- Los clientes externos al anfitrión no pueden interactuar con el `logger`.
- Los equipos fuera de la VPN deben, además, obtener acceso a la misma.

Convendrá realizar una prueba de funcionamiento con la versión de cbw que desees, pero considera estos pasos:

1. Has adaptado la configuración de despliegue para añadir en la sección del broker:

```
ports:
"9998:9998"
```

2. En el equipo de escritorio ejecutarás el/los cliente/s.
3. También en ese equipo averiguarás la IP de tu servidor en el portal (p.ej. con una orden `ping tsr-milogin-2021.dsic.cloud` que en este ejemplo suponemos que nos devuelve `192.168.105.111`)
   - Es importante mencionar que la IP del broker no es visible desde fuera del anfitrión, pero mediante la sección `ports` se ha ordenado al anfitrión que las peticiones entrantes por ese puerto se dirijan al mismo puerto del broker.

El código del cliente (**client_external.js**) es idéntico[3] al del cliente presentado en el sistema CBW_FTC, pero se ejecuta en el equipo de escritorio, lo que requiere que suministremos los argumentos necesarios.

- Puede darse un problema ocasional con la variable `myID`, cuyo cálculo sería sustituido por la instrucción que en el original aparece comentada.

4. Tomando los datos del ejemplo, y para el caso particular de solicitudes del tipo B, deberíamos invocarlo de esta forma:

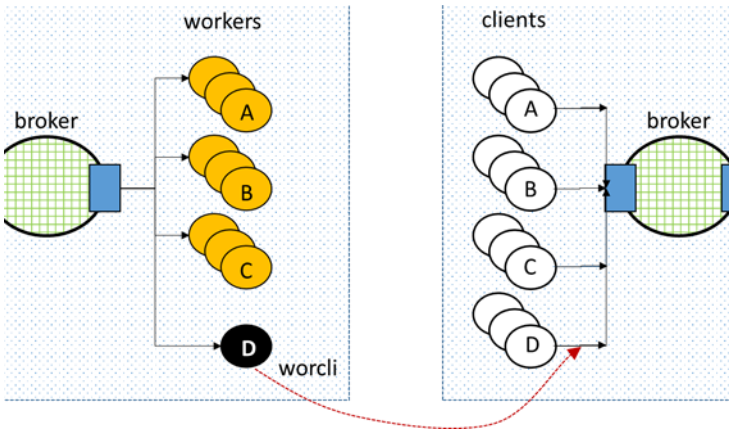
```
node client_external tcp://192.168.105.111:9998 B
```

5. Y en el anfitrión arrancar el servicio mediante `docker-compose up`.

Con estas pruebas y las modificaciones necesarias estamos empezando nuestra preparación para construir componentes que interactúan aunque no se encuentren en el mismo anfitrión.

4 Sesión 3. Encadenando varios sistemas CBW: el worcli

El worcli es un nuevo componente cuyo código procede de la fusión de los roles worker y client para un mismo tipo de trabajo T.



La idea es que este componente redirija peticiones de un broker bk1, que dispone de clientes del tipo T, a otro, bk2, que dispone de trabajadores para ese tipo de trabajo T.

- Para ello se conecta con bk1 como trabajador, y envía la petición como cliente a bk2. El camino inverso requiere recordar en `worcli` el identificador del cliente que se encontraba en el mensaje de petición.
- Para ello presenta un socket req ante cada uno de los 2 brokers

[3] Por esta razón este apartado no posee entrada en los anexos

Para su ejecución requerirá 4 parámetros: el URL de bk1, el URL de bk2, el **retardo de red que añade** (argumento `delay`) y el tipo de trabajo que debe procesar.

Su código aparece en el anexo correspondiente, y solo requiere destacar que requiere dos sistemas cbw_fctl (posiblemente en dos hosts diferentes) para funcionar.

- Como en el cliente externo, puede darse un problema con la variable `myID`, cuyo cálculo sería sustituido por la instrucción que en el original aparece comentada.

Aunque en la ilustración se muestra este componente worcli como parte de uno de los sistemas `cbw_ftcl`, realmente **debe ser ejecutado en las mismas condiciones que el cliente externo**, esto es, en el equipo de escritorio.

Se requiere desplegar ambos sistemas y ejecutar en el equipo de escritorio una instancia de worcli, con las comprobaciones necesarias para confirmar su buen funcionamiento.

4.1 Preguntas sobre worcli

En este apartado la mayor parte de las propuestas son **preguntas abiertas** que invitan a explorar las posibilidades, limitaciones e implicaciones de cada una. Todas toman como punto de partida una situación con 2 anfitriones (*hosts*) en el portal, cada uno con un despliegue que incluye 1 broker (accesible externamente), y múltiples trabajadores.

4.1.1 Pregunta 1

Si el *worcli* se ejecuta en un anfitrión, ¿puede **formar parte de un despliegue** del servicio (mediante un `docker-compose.yml`)?. Como alternativa, ¿podrías ponerlo en funcionamiento "***a mano***"? Argumenta la respuesta a ambas preguntas. Si la segunda es afirmativa, explica cómo lo harías.

4.1.2 Pregunta 2

Supón que alguna de las alternativas nos permite enganchar los sistemas mediante este *worcli*.

- Imaginemos que se especializa en la clase de trabajos D, sin competencia por parte de ningún otro worker de este anfitrión.
- Recuerda, además, que empleamos la implementación con tolerancia a fallos en el broker, con todos los detalles que esto comporta.

Argumenta por qué hay **grandes diferencias** entre las 3 invocaciones siguientes, y qué repercusiones tendrá cuando el broker alojado en el primer anfitrión (`192.168.105.111`) reciba alguna petición de un cliente de tipo D:

```
node worcli tcp://192.168.105.111:9999 tcp://192.168.105.112:9998 200 D
node worcli tcp://192.168.105.111:9999 tcp://192.168.105.112:9998 1200 D
node worcli tcp://192.168.105.111:9999 tcp://192.168.105.112:9998 2200 D
```

(El código de invocación mostrado solo es una ilustración que **no debe tomarse como pista** o respuesta, salvo el cambio en el retardo especificado en cada caso y su relación con el latido)

5 ANEXOS

5.1 Previo: construir imagen tsr2021/ubuntu-zmq

Esta imagen ya se ha generado en los servidores de portal, pero podemos comprobar su existencia y los pasos seguidos para construirla.

5.1.1 Dockerfile

```
FROM ubuntu:focal
RUN apt-get update
RUN apt-get install -y curl ufw gcc g++ make gnupg
RUN curl -sL https://deb.nodesource.com/setup_12.x | bash -
RUN apt-get -y install nodejs
RUN apt-get -y upgrade
RUN npm install zeromq@5
```

5.1.2 Orden

```
docker build -t tsr2021/ubuntu-zmq .
```

5.2 Anexo 1_A mano

5.2.1 Código 2client

```
01: const zmq = require('zeromq')
02: let req = zmq.socket('req');
03:
04: var args = process.argv.slice(2)
05: if (args.length < 1) {
06:   console.log ("node myclient brokerURL")
07:   process.exit(-1)
08: }
09: var bkURL   = args[0]
10: req.connect(bkURL)
11: req.on('message', (msg)=> {
12:     console.log('resp: '+msg)
13:     process.exit(0);
14: })
15: req.send('Hola')
```

5.2.2 Dockerfile incompleto client

```
FROM tsr2021/ubuntu-zmq
COPY ./2client.js /myclient.js

CMD node myclient NEED_BROKER_URL
```

5.2.3 Código 2broker

```
01: const zmq = require('zeromq')
02: let sc = zmq.socket('router') // frontend
03: let sw = zmq.socket('router') // backend
04:
05: var args = process.argv.slice(2)
06: if (args.length < 2) {
07:   console.log ("node mybroker clientsPort workersPort")
08:   process.exit(-1)
09: }
10:
11: var cport = args[0]
12: var wport = args[1]
13: let cli=[], req=[], workers=[]
14: sc.bind('tcp://*:'+cport)
15: sw.bind('tcp://*:'+wport)
16: sc.on('message',(c,sep,m)=> {
```

```
17:     if (workers.length==0) {
18:         cli.push(c); req.push(m)
19:     } else {
20:         sw.send([workers.shift(),'',c,'',m])
21:     }
22: })
23: sw.on('message',(w,sep,c,sep2,r)=> {
24:     if (c!='') sc.send([c,'',r])
25:     if (cli.length>0) {
26:         sw.send([w,'',
27:             cli.shift(),'',req.shift()])
28:     } else {
29:         workers.push(w)
30:     }
31: })
```

5.2.4 Dockerfile broker: construir con docker build

```
FROM tsr2021/ubuntu-zmq
COPY ./2broker.js /mybroker.js
EXPOSE 9998 9999
CMD node mybroker 9998 9999
```

5.2.5 Código 2worker

```
01: const zmq = require('zeromq')
02: let req = zmq.socket('req')
03: req.identity = 'Worker1'
04:
05: var args = process.argv.slice(2)
06: if (args.length < 1) {
07:   console.log ("node myclient brokerURL")
08:   process.exit(-1)
09: }
10: var bkURL   = args[0]
11: req.connect(bkURL)
12: req.on('message', (c,sep,msg)=> {
13:     setTimeout(()=> {
14:         req.send([c,'','resp'])
15:     }, 1000)
16: })
17: req.send(['','',''])
```

5.2.6 Dockerfile incompleto worker

```
FROM tsr2021/ubuntu-zmq
COPY ./2worker.js /myworker.js

CMD node myworker NEED_BROKER_URL
```

5.3 Anexo 2_CBW (básico)

Mantenemos sin cambios el código de los 3 componentes. Los Dockerfile de cliente y trabajador están parametrizados, y se resuelven las dependencias mediante docker-build

5.3.1 docker-compose.yml

```
version: '2'
services:
  cli:
    image: client
    build: ./client/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9998
  wor:
```

```
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9999
  bro:
    image: broker
    build: ./broker/
    expose:
      - "9998"
      - "9999"
```

Cuando los Dockerfile estén preparados, desplegar con `docker-compose up`

5.3.2 Dockerfile client

Únicamente cambiamos la última línea

```
CMD node myclient $BROKER_URL
```

Construir con `docker build`

5.3.3 Dockerfile worker

Únicamente cambiamos la última línea

```
CMD node myclient $BROKER_URL
```

Construir con `docker build`. Ya se puede desplegar y probar

5.4 Anexo 3_CBW_FTC (clases de trabajo y tolerancia a fallos)

5.4.1 Código client

```
01: // client in NodeJS, classID must be provided as a parameter
02: // 10 messages and exit!
03: // CMD node myclient $BROKER_URL $CLASSID
04:
05: const zmq = require('zeromq')
06: let req = zmq.socket('req')
07:
08: var args = process.argv.slice(2)
09: if (args.length < 2) {
10:   console.log ("node myclient brokerURL class")
11:   process.exit(-1)
12: }
13: var bkURL   = args[0]
14: var cid     = args[1]
15: var nMsgs = 10
16: var myMsg   = 'Hello'
17: var myID    = "C_"+require('os').hostname()
18: //myID = "C_"+Date.now()%100000 // running without own IP
19:
20: req.identity = myID
21: req.connect(bkURL)
22:
23: req.on('message', (msg) => {
24:   if (--nMsgs == 0) process.exit(0)
25:   else req.send([myMsg,cid])
26: })
27:
28: req.send([myMsg,cid])
```

5.4.2 Dockerfile client

```
FROM tsr2021/ubuntu-zmq
COPY ./client.js /myclient.js
# We assume that each client is linked to the broker
# container.
CMD node myclient $BROKER_URL $CLASSID
```

5.4.3 Código ft_broker_class

```
01: // ROUTER-ROUTER request-reply broker in NodeJS.
02: // Work classes.
03: // Worker availability-aware variant.
04: //
05: // As code grows, complexity increases. This version returns to the
06: // original structure with auxiliar functions (sendToWorker & sendRequest)
07:
08: var zmq = require('zeromq')
09: , sc    = zmq.socket('router')
10: , sw    = zmq.socket('router')
11: , util  = require('util')
12:
13: var args = process.argv.slice(2)
14: if (args.length < 2) {
15:   console.log ("node mybroker clientsPort workersPort")
16:   process.exit(-1)
17: }
18:
19: var cport = args[0]
20: var wport = args[1]
21:
22: const ansInterval = 2000
23: var workers = [], clients = [] // clients[] = who{}
24: var busyWks = [] // busy workers
25:          // busyWks[] = tout{}
26:
27: var myID = "B_"+require('os').hostname() // unused
28: //myID = "B_"+Date.now()%100000 // running without own IP
29:
30: function testQueues(cid) {
31:   if (workers[cid]==undefined) {
32:     workers[cid]=[]; clients[cid]=[]
33:   }
34: }
35: function annotate(prefix, id, cid) {
36:   var str=util.format(prefix+' %s (class "%s")', id, cid)
37:   console.log(str)
38: }
39:
40: sc.bind('tcp://*:'+cport)
41: sw.bind('tcp://*:'+wport)
42:
43: annotate('CONNECTED', '', '')
44:
45: // Send a message to a worker.
46: function sendToWorker(msg, cid) {
47:   var myWk = msg[0]
48:   annotate('TO Worker', myWk, cid)
49:   sw.send(msg)
50:   busyWks[myWk] = {}
51:   busyWks[myWk].cid = cid
52:   busyWks[myWk].msg = msg.slice(2)
53:   busyWks[myWk].timeout =
54:     setTimeout(newToutHandler(myWk),ansInterval)
55: }
```

```
56:
57: // Function that sends a message to a worker, or
58: // holds the message if no worker is available now.
59: // Parameter 'args' is an array of message segments.
60: function sendRequest(args, cid) { // (c,sep,m,cid)
61:   if (workers[cid].length > 0) {
62:     var myWk = workers[cid].shift()
63:     var m = [myWk,''].concat(args)
64:    annotate('UNQUEUEING Worker', myWk, cid)
65:     sendToWorker(m, cid)
66:   } else {
67:    annotate('QUEUEING Client', args[0], cid)
68:     clients[cid].push({id: args[0],msg: args.slice(2)})
69:   }
70: }
71:
72: function newToutHandler(wkID) {
73:   annotate('TOUT HANDLER', wkID, '')
74:   return () => {
75:     var msg = busyWks[wkID].msg
76:     var cid = busyWks[wkID].cid
77:     delete busyWks[wkID]
78:     annotate('TOUT EXPIRED', wkID, cid)
79:     sendRequest(msg, cid)
80:   }
81: }
82:
83: sc.on('message', function() { // (c,sep,m,cid)
84:   var args = Array.apply(null, arguments)
85:   var cid  = args.pop()
86:   testQueues(cid)
87:   annotate('FROM Client', args[0], cid)
88:   sendRequest(args, cid) // (c,sep,m,cid)
89: });
90:
91: function processPendingClient(wkID, cid) {
92:   if (clients[cid].length>0) {
93:     var nextClient = clients[cid].shift()
94:     var msg = [wkID,'',nextClient.id,''].concat(nextClient.msg)
95:     sendToWorker(msg, cid)
96:     return true
97:   } else return false
98: }
99:
100:   sw.on('message', function() { // (w,sep,c,sep2,r,cid)
101:     var args = Array.apply(null, arguments);
102:     var cid = args.pop()
103:     if (args.length == 3) {// if (c=='') -> (w,sep,'',cid)
104:      testQueues(cid)
105:      annotate('REGISTERING Worker', args[0], cid)
106:       if (!processPendingClient(args[0],cid)) {
107:         annotate('QUEUEING Worker', args[0], cid)
108:         workers[cid].push(args[0])
109:       //return
110:       }
111:     } else {
112:         var wkID = args[0]
113:         clearTimeout(busyWks[wkID].timeout)
114:         args = args.slice(2)
115:         annotate('TO Client', args[0], cid)
116:         sc.send(args)
117:         if (!processPendingClient(wkID,cid)) {
118:           annotate('QUEUEING Worker', wkID, cid)
119:           workers[cid].push(wkID)
120:       }
121:     }
```

```
122:  })
```

5.4.4 Dockerfile broker

```
FROM tsr2021/ubuntu-zmq
COPY ./ftbroker_class.js /mybroker.js
EXPOSE 9998 9999
CMD node mybroker 9998 9999
```

5.4.5 Código worker

```
01: // worker server in NodeJS, classID must be provided as a parameter
02: // 1.0 sec service, infinite loop!
03: // CMD node myworker $BROKER_URL $CLASSID
04:
05: const zmq = require('zeromq')
06: let req   = zmq.socket('req')
07:
08: const Tcpu = 1000
09: var replyText = "Done!"
10: var args = process.argv.slice(2)
11: if (args.length < 2) {
12:   console.log ("node myworker brokerURL class")
13:   process.exit(-1)
14: }
15: var bkURL = args[0]
16: var cid   = args[1]
17: var myID  = "W_"+require('os').hostname()
18: //myID = "W_"+Date.now()%100000
19:
20: req.identity = myID
21: req.connect(bkURL)
22:
23: req.on('message', (c, sep, msg) => {
24:   setTimeout(() => {
25:       req.send([c,'',replyText,cid])
26:   }, Tcpu)
27: })
28:
29: req.send(['',cid])
```

5.4.6 Dockerfile worker

```
FROM tsr2021/ubuntu-zmq
COPY ./worker.js /myworker.js
# We assume that each worker is linked to the broker
# container.
CMD node myworker $BROKER_URL $CLASSID
```

5.4.7 docker-compose.yml

```
version: '2'
services:
  cliA:
    image: client
    build: ./client/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9998
      - CLASSID=A
  cliB:
    image: client
    build: ./client/
    links:
      - bro
    environment:
```

```
      - BROKER_URL=tcp://bro:9998
      - CLASSID=B
  cliC:
    image: client
    build: ./client/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9998
      - CLASSID=C

  worA:
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9999
      - CLASSID=A
  worB:
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9999
      - CLASSID=B
  worC:
    image: worker
    build: ./worker/
    links:
      - bro
    environment:
      - BROKER_URL=tcp://bro:9999
      - CLASSID=C
  bro:
    image: broker
    build: ./broker/
    expose:
      - "9998"
      - "9999"
```

5.5 Anexo 4_CBW_TFCL (clases de trabajo, tolerancia a fallos y logger)

5.5.1 Código logger

```
01: // logger in NodeJS
02: // First argument is port number for incoming messages
03: // Second argument is file path for appending log entries
04:
05: var fs = require('fs');
06: var zmq = require('zeromq')
07: ,   log = zmq.socket('pull')
08: var args = process.argv.slice(2);
09: if (args.length < 2) {
10:   console.log ("node logger loggerPort filename")
11:   process.exit(-1)
12: }
13:
14: var loggerPort = args[0] // '9995'
15: var filename   = args[1] // "/tmp/cbwlog.txt"
16:
17: log.bind('tcp://*:'+loggerPort);
18:
19: log.on('message', function(text) {
20:   fs.appendFileSync(filename, text+'\n');
```

```
21: })
```

5.5.2 Dockerfile logger

Requisitos en anfitrión: directorio preexistente con permisos

```
FROM tsr2021/ubuntu-zmq
COPY ./logger.js /mylogger.js
VOLUME /tmp/cbwlog
EXPOSE 9995
CMD node mylogger 9995 $LOGGER_DIR/logs
```

5.5.3 docker-compose.yml

Todo lo relacionado con clientes y trabajadores permanece igual, pero hemos de dar entrada al nuevo servicio logger, y a las nuevas dependencias para que…

- El broker pueda conectar con el logger (`$LOGGER_URL`)
- El logger pueda conocer el directorio de trabajo cedido por el host (`$LOGGER_DIR`)

A continuación de muestra el **fragmento significativo** de `docker-compose.yml`

```
version: '2'
services:

...

  bro:
    image: broker
    build: ./broker/
    links:
      - log
    expose:
      - "9998"
      - "9999"
    environment:
      - LOGGER_URL=tcp://log:9995
  log:
    image: logger
    build: ./logger/
    expose:
      - "9995"
    volumes:
# /tmp/logger.log DIRECTORY must exist on host and writeable
      - /tmp/logger.log:/tmp/cbwlog
    environment:
      - LOGGER_DIR=/tmp/cbwlog
```

5.6 Anexo 6_CBW_FTCL_WORCLI-EXT

5.6.1 Código worcli-ext

```
01: // worcli
02: // invoked with "node worcli bk1URL bk2URL delay class"
03: // all 5 parameters are mandatory
04: var zmq = require('zeromq')
05: , rw = zmq.socket('req')
06: , rc = zmq.socket('req')
07:
08: var args = process.argv.slice(2)
09: if (args.length < 4) {
10:   console.log ("Usage: node worcli bk1URL bk2URL transfer_delay class")
11:   console.log ("Redirects bk1's class requests to bk2 broker, increasing delay ms")
12:   process.exit(-1)
13: }
14:
15: var w2bk = args[0] // worcli connected to bk1 as a worker
16: var c2bk = args[1] // worcli connected to bk2 as a client
17: var myID  = "WC_"+require('os').hostname()
18: //myID = "W_"+Date.now()%100000
19: var delay = parseInt(args[2]) // transfer delay, in ms
20: var cid   = args[3]
21: var pendingClient
22:
23: rw.identity = myID; rw.connect(w2bk)
24: rc.identity = myID; rc.connect(c2bk)
25:
26: rw.on('message', (c,sep,m) => {
27:   pendingClient = c // only one waiting client, so we don't need a queue
28:   setTimeout(()=>{
29:       rc.send([m, cid])}
30:       , delay/2); // 50% forwarding
31: })
32:
33: rw.send(['', cid])
34:
35: rc.on('message', (m)=> {
36:   setTimeout(()=>{
37:     rw.send([pendingClient,'',m,cid])}
38:     , delay/2); // 50% returning
39: })
```