Memoria Final

Evolución de una plataforma de chat desde una solución basada en despliegues tradicionales a una basada en contenedores.

MUII - Trabajo final de Máster

Junio 2018

Tutora: Córdoba Cabeza, María Luisa
<mcordoba@fi.upm.es>

Alumno: Pérez Ramos, José María
<josemaria.perez.ramos@alumnos.upm.es>
<jmperez@tuenti.com>

Resumen

Las tecnologías de contenedores y su aplicación a la ingeniería del *software* son una irrupción reciente, sin embargo, están revolucionando completamente el proceso de desarrollo, prueba y despliegue gracias a la agilidad que proporcionan en todos los entornos. El hecho de proporcionar un sistema de empaquetado y despliegue agnóstico de la tecnología más allá del sistema operativo aporta una ventaja diferencial en cuanto a tiempos de desarrollo se refiere.

En este documento se explora el uso de estas tecnologías en una situación real en la que un sistema se transforma de una metodología tradicional a otra basada en contenedores. Se ha documentado todo el proceso y los distintos entornos afectados.

Por último se aplica la solución obtenida en la etapa anterior para dar un paso más y desplegar este sistema de chat en un *cluster* de orquestación basado en contenedores.

Abstract

Containerization and its application to software engineering is a young technology, however, this technology is changing completely the methodology traditionally used to develop, test and deploy software solutions. This revolution is due to the game-changing speed containerization provides in all environments, thanks to the ease of use an environment-independent (save for the operating system) packaging and deployment tool provides.

This document explores the use of the containerization technology in a real world situation where a traditional chat system is migrated from the traditional develop, test and deploy system to another one based in containers. This process has been documented as well as all the environments affected.

In the end, this solution is used to deploy the chat system over an orchestation platform, which is the next logical step in this system evolution.

Tabla de contenidos:

1.Introducción y objetivos....1

2.Estado del arte....3

3.Conceptos previos y herramientas....5

3.1.Docker....5

3.2.Docker-compose....6

3.3.Kubernetes....6

3.4.Erlang....7

3.5.Puppet....7

3.6.Marionette Collective....7

4.Situación inicial....9

4.1.Despliegue....9

4.2.Ciclo de desarrollo del chat....11

5.Chat en docker....13

5.1.Despliegue....13

5.2.Ciclo de desarrollo del chat con docker....16

5.3.Alternativas y cambios realizados....17

5.4.Actualización sin pérdida de disponibilidad....18

6.Chat en Kubernetes....21

6.1.Despliegue....22

6.2.Ciclo de desarrollo....25

6.3.Alternativas y cambios realizados....25

7.Líneas futuras....27

8.Conclusiones....29

9.Bibliografía....31

Listado de figuras

5.1.1 Esquema del despliegue inicial.......9
5.1.2 Esquema de máquina de chat.......10
5.1.3 Esquema de instancia de chat.......10
6.1.1 Esquema de conectividad de un dominio de chat.......14
6.1.2 Esquema de instancia de chat con docker.......14
6.2.1 Esquema de proceso de desarrollo con docker.......17
6.4.1 Proceso de despliegue con configuraciones combinadas.......19
7.1.1 Esquema de funcionamiento básico de Kubernetes.......21

1. Introducción y objetivos

Tuenti, empresa en la que desempeño mi actividad profesional, ofrece entre sus servicios un chat para la comunicación en tiempo real entre los usuarios de su aplicación. Este servicio de chat se utiliza también para toda comunicación instantánea entre la aplicación y los servidores de Tuenti, como puede ser la señalización de las llamadas por Voz IP, notificaciones de especial relevancia o todas las notificaciones en caso de la aplicación web.

Este servicio de chat utiliza el protocolo XMPP (RFC 6120) modificado para la comunicación, está programado en erlang (Walerud, J. (1998) *Erlang/OTP Released as Open Source™*) y se despliega como una serie de ficheros con bytecode que ejecuta sobre la máquina virtual de erlang instalada en la máquina real. Cada una de las instancias del servicio conoce mediante unos ficheros de configuración en qué interfaces tiene que escuchar qué puertos, qué máquinas forman el cluster y qué dominios proporcionan el servicio. Esto implica que el equipo de sistemas tiene que atender, además de todas sus responsabilidades relacionadas con la comunicación y despliegue de máquinas, tareas propias del servicio de chat.

En un esfuerzo por agilizar el desarrollo y despliegue de los servicios, así como desacoplarlos de las máquinas en las que éstos ejecutan, Tuenti ha decidido migrar toda su estructura de servicios a Kubernetes (http://kubernetes.io) y Docker (https://www.docker.com/), estableciendo una frontera con responsabilidades definidas en los distintos equipos. Esta frontera permite a los equipos transversales, como el equipo de sistemas, ofrecer una infraestructura unificada y bien definida y a los equipos propietarios de servicios mayor independencia y, por ello, rapidez, en el ciclo de desarrollo y despliegue.

El chat se considera un servicio intermedio, ya que no sigue el patrón petición-respuesta, sino que mantiene una conexión abierta, por lo que su migración a kubernetes no se contempla salvo para entornos de pruebas. Sin embargo, el uso de docker para su desarrollo y despliegue supone una gran facilidad, al estar unificado con el resto de la infraestructura de Tuenti.

Además del chat, hay otros servicios que encajan en la misma zona intermedia, como es la infraestructura de voz, que no se tratará en este documento.

El objetivo de este trabajo consiste en la transformación de la plataforma del servicio de chat ofertado por Tuenti a un modelo basado en contenedores, lo que conlleva la investigación, diseño y desarrollo tanto de las modificaciones necesarias en el código del chat como en todas aquellas herramientas utilizadas durante el mantenimiento y evolución de esta infraestructura.

Por último, una vez completada la transformación se harán los cambios necesarios para poder desplegarlo sobre una plataforma de orquestación de servicios para su uso como entorno de desarrollo por otros equipos dentro de Tuenti.

2. Estado del arte

En ingeniería del software un producto rara vez está finalizado una vez termina el desarrollo inicial. Las necesidades cambian con el tiempo, y con ello los requisitos de los sistemas que haya en funcionamiento en ese momento.

Es por este motivo que los productos necesitan ser actualizados, ya sea para corregir errores o para añadir o modificar funcionalidades en función de los nuevos requisitos. Esto es más acuciado en los modelos de desarrollo de software como desarrollo ágil (Hunt, Andy; Thomas, Dave (1999) The Pragmatic Programmer, Addison-Wesley) o desarrollo en espiral, pero eso no significa que en modelos de desarrollo más tradicionales, como el desarrollo en cascada (Benington, Herbert D. (1983). "Production of Large Computer Programs") no sea necesario.

El proceso de desarrollo requiere que se disponga de un entorno lo más similar al entorno de ejecución real para sus tareas pero sin poder afectarlo de ninguna forma. Esta configuración cada vez se torna más tediosa debido a la multitud de dependencias, ya sea de código o servicios, o tecnologías distintas requeridas para su funcionamiento. Asimismo los entornos de pruebas automáticas también requieren de esta misma configuración y como requisito añadido, debe ser automático revertirla a un estado consistente.

En el caso de sistemas que proveen un servicio constante, el paso de un entorno de desarrollo a un entorno de producción es crucial, ya que es imperativo que el usuario perciba la mínima irregularidad o interrupción del servicio posible.

El proceso de despliegue de un servicio en producción depende de muchos factores, como pueden ser si el código es compilado directamente a código máquina o es interpretado, pasando por todo el rango de compilación a distintas etapas de *bytecode*, si el servicio que se proporciona mantiene una conexión persistente o se basa en el paradigma petición respuesta, etc.

En cualquiera de los casos descritos anteriormente es necesario ser capaz de convertir una máquina en un entorno de desarrollo o pruebas así como ser capaz de distribuir el nuevo sistema a la máquina en cuestión y actualizar el servicio.

Existen actualmente varias formas para obtener un entorno de desarrollo o pruebas, hasta ahora las más utilizadas para las dependencias que no son de código han sido la automatización de la creación de máquinas virtuales (con tecnologías como vagrant), la generación de las dependencias como paquetes instalables por el sistema operativo, o símplemente la creación por los propios desarrolladores de las herramientas necesarias para

cada una de las dependencias. Las tecnologías utilizadas para solucionar las dependencias de código suelen depender del lenguaje, como puede ser maven para java, composer para php o rebar para erlang.

Por su parte, desplegar el servicio en producción requiere la transferencia de los ficheros a la máquina, de nuevo distintos lenguajes o tecnologías utilizan distintos métodos, siempre teniendo en cuenta que las dependencias en el entorno real deben estar satisfechas.

Desde 2013 existe una herramienta, Docker, que permite simplificar todo este proceso en los servidores basados en Linux, independientemente de la tecnología utilizada.

Docker permite la creación de 'imágenes' que contienen todos aquellos ficheros incluidos por el creador de la imagen. Los procesos (a nivel del sistema operativo) que ejecuten dentro de la imagen leen estos ficheros en vez de los que tuviera el *host* en la misma ruta, y son incapaces de escribir ficheros salvo en rutas determinadas explícitamente.

Con una o varias imágenes es posible tener en funcionamiento un entorno de desarrollo inmediatamente, ya que la propia imagen incluye todas las dependencias y configuración necesarias. Con la propiedad de que el entorno es de sólo lectura los sistemas de pruebas automáticas obtienen todas las ventajas.

Docker aísla los procesos dentro de sus imágenes y ofrece una API bien documentada para las distintas opciones de configuración, permitiendo definir una frontera clara entre la provisión de la máquina y la del servicio y haciendo más veloz el desarrollo al reducir las dependencias. Las imágenes están versionadas, y son independientes de la naturaleza del servicio que contienen, por lo que son un objeto idóneo como unidad de despliegue.

Docker ha supuesto una revolución en el proceso de desarrollo y despliegue, lo que ha llevado a su utilización como base para la creación de sistemas más complejos.

Gracias al aislamiento de procesos a nivel de sistema operativo, sin necesidad de máquinas virtuales, en una única máquina pueden ejecutar simultáneamente multitud de servicios distintos compartiendo mejor los recursos. Uno de estos sistemas es Kubernetes.

Desde 2014, Kubernetes proporciona la capacidad de crear un *cluster* de máquinas, todas ellas ejecutando Docker y con resolución de nombres de dominio a nivel de contenedor. Esto ha supuesto el desacoplamiento prácticamente total de los servicios del *hardware* sobre el que ejecutan. Con Kubernetes los contenedores son efímeros, una imagen de un servicio puede estar ejecutando en una máquina y por un pico de carga, pararse y volver a ejecutar en otra distinta.

3. Conceptos previos y herramientas

A continuación se exponen brevemente los conceptos y herramientas más importantes utilizados durante el desarrollo de este trabajo.

3.1. Docker

Docker es una tecnología que permite el aislamiento de procesos a nivel de Sistema Operativo aprovechándose de los *namespaces* y *control groups* existentes en el núcleo de Linux. Se utiliza el término 'contenedor' para referirse a estos procesos en ejecución.

El aislamiento sucede a varios niveles:

- **Red:** Gracias al namespace de red, según la configuración de docker, el contenedor puede tener acceso completo al *stack* de red, no tener acceso a la red o únicamente acceso a una subred virtual propia de docker utilizada para la comunicación con otros contenedores con NAT hacia internet. Docker permite la creación de estas subredes virtuales así como que un contenedor pueda estar conectado a cualquier número de ellas, permitiendo configuraciones complejas.

- **Sistema de ficheros:** Los contenedores tienen una visión particular del sistema de ficheros. Salvo determinadas rutas establecidas a la hora de ejecutar el contenedor, los ficheros pertenecientes al contenedor así como las modificaciones que hace a cualquier fichero son efímeras, sólo existen mientras el contenedor está en ejecución y no se ven reflejadas en los ficheros del *host*. Esto es posible gracias al uso de sistemas de ficheros como AuFS u OverlayFS

- **Árbol de procesos:** Cada contenedor considera que es el único proceso en ejecución.

- **Límites de recursos:** Los contenedores no son conscientes de los límites que puedan tener en ejecución, que son establecidos por docker.

Este aislamiento permite la ejecución de múltiples contenedores en una única máquina, como si estuvieran virtualizados, pero sin el sobrecoste que eso conlleva al estar todo el aislamiento integrado por el sistema operativo

Imagen de docker:

Una imagen de docker es un archivo que tiene todos los ficheros escogidos por el creador de la imagen en una ruta determinada, así como un punto de entrada, o comando que ejecutar. Las imágenes deben incluir todas aquellas dependencias necesarias para la ejecución de su punto de entrada.

Ejecución de un contenedor:

Cuando docker crea un contenedor a partir de su imagen, crea un proceso en el que las rutas de los ficheros de la imagen ocultan los propios de la máquina. Asimismo, cualquier escritura se lleva a cabo únicamente en su visión del sistema de ficheros. Por último, ejecuta el punto de entrada de la imagen.

El proceso que entra en ejecución ejecuta como PID 1, es decir, como si fuera el único proceso en ejecución Tiene las limitaciones que le haya impuesto docker y el sistema operativo desde el *host*, pero no es consciente de ellas.

3.2. Docker-compose

Docker-compose se utilizará en esta memoria como un fichero de configuración capaz de definir un conjunto de servicios con sus imágenes de docker, una subred virtual asociada y unas opciones de ejecución para cada uno de los servicios.

Docker-compose interactúa con docker para, a partir de estos ficheros de configuración, poner en ejecución todos los servicios definidos con sus opciones e interconectarlos.

3.3. Kubernetes

Kubernetes es una tecnología que, desplegada sobre un conjunto de máquinas, permite ejecutar un conjunto de servicios basados en contenedores en ellas.

La unidad básica de Kubernetes es el *pod* (juego de palabras, ya que pod es un conjunto de ballenas -logo de docker- en inglés, así como una vaina), que consiste en varios contenedores ejecutando todos en el mismo *namespace* de red.

Kubernetes mantiene varias instancias de un *pod* en ejecución en el *cluster* y crea un punto de entrada de red que balancea entre las distintas instancias del *pod*. Esto permite crear una red de servicios comunicándose entre ellos dentro del *cluster* con alta disponibilidad.

Los *pods* de Kubernetes son efímeros, en cualquier momento un pod puede ser eliminado por alguna causa interna o externa. Kubernetes se asegura de poner en marcha otro en sustitución.

3.4. Erlang

Erlang/OTP es un lenguaje de programación funcional diseñado por Ericsson para construir sistemas distribuidos, tolerantes a fallos, de alta disponibilidad y masivamente concurrentes. Entre otras características útiles está la capacidad para actualizar el código en caliente, es decir, sin necesidad de parar el proceso en ejecución.

Es el lenguaje de programación elegido para el chat de tuenti entre otros componentes.

El código de un programa en erlang se compila a *bytecode* para su máquina virtual y tradicionalmente se empaqueta en un archivador. Es posible incluir la máquina virtual en este paquete para su distribución, pudiendo generar un programa ejecutable en la mayoría de los sistemas basados en Linux

3.5. Puppet

Puppet es un sistema de gestión de configuración con una arquitectura cliente-servidor.
El agente de puppet ejecuta en la máquina con permisos de administración, manteniendo actualizada la configuración con la definida por el servidor.

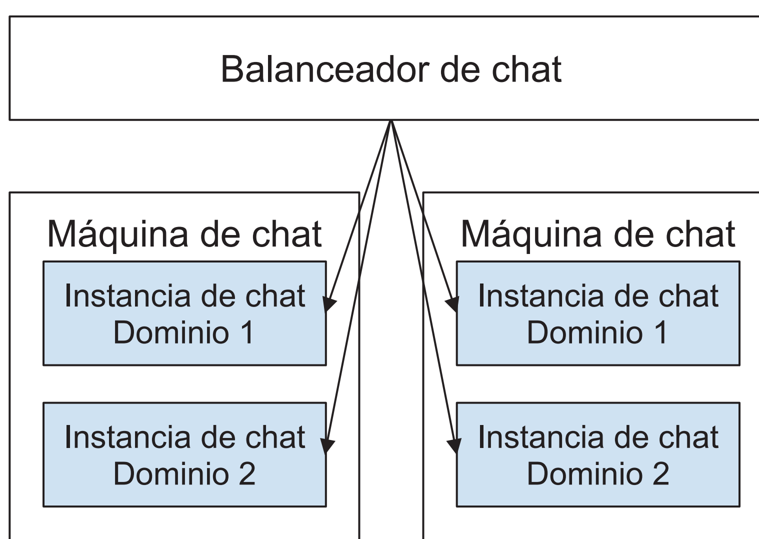
3.6. Marionette Collective

Marionette Collective es un subsistema de puppet utilizado para realizar acciones de configuración en múltiples máquinas simultáneamente.

4. Situación inicial

A continuación se expone el estado inicial del servicio de chat, tanto de despliegue como de generación de versiones. Este estado corresponde con una metodología más tradicional de despliegue, aunque se haya seguido una metodología ágil para su desarrollo.

4.1. Despliegue



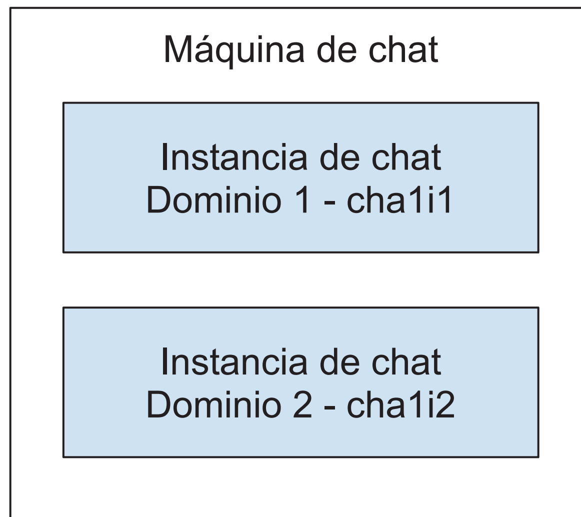
5.1.1 Esquema del despliegue inicial

El servicio de chat está dividido en dominios, a cada usuario se le asigna un dominio y hasta que esta asociación no cambie, conectará siempre con el mismo dominio. Cada dominio tiene una dirección IP pública distinta.

Cada uno de los dominios de chat está formado por 4 instancias de chat activas y 2 inactivas, constituyendo entre ellas un *cluster* a nivel de erlang. Si alguna de las instancias quedara fuera de servicio, el resto se reparten la carga de forma semitransparente. Los dominios se comunican entre sí mediante el protocolo XMPP.

Las instancias están repartidas entre máquinas reales, de forma que cada dominio tiene sus instancias en *hardware* distinto, pero una máquina puede servir varios dominios independientemente.

Estas máquinas de chat son máquinas especialmente dedicadas para su rol, cada una de ellas posee varias interfaces de red y una cantidad de memoria RAM excesiva para sus requisitos actuales, ya que son heredadas de la época en la que Tuenti era una red social y todos sus usuarios estaban permanentemente conectados al chat.

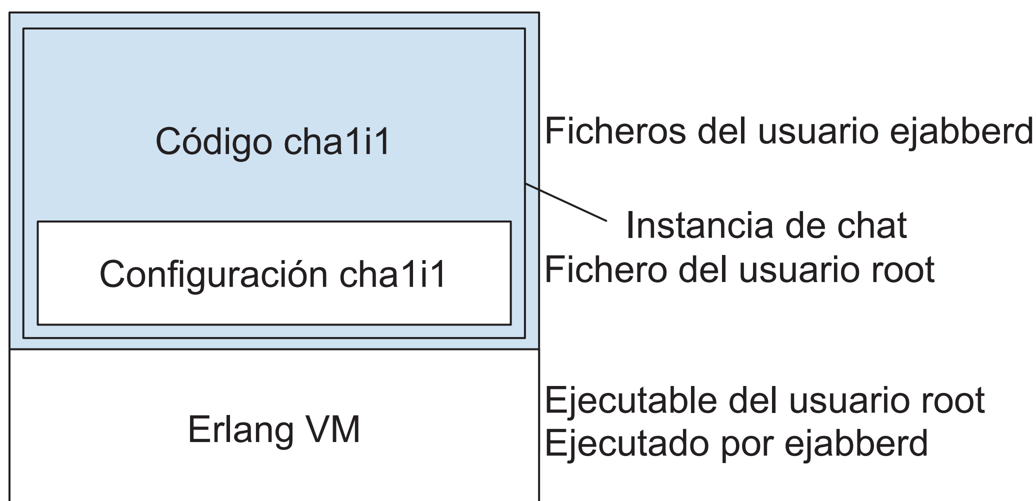


5.1.2 Esquema de máquina de chat

Cada una de las instancias de cada una de las máquinas tiene un nombre de la forma `chaXiY` donde chaX es el nombre de la máquina e Y el número de la instancia.

Cada instancia de chat tiene su propio fichero de configuración estática que lee al arrancar y que está definido y desplegado mediante puppet.

Hay dos usuarios creados en la máquina: ejabberd, que es el usuario que ejecuta el chat, y root, que es el usuario que utiliza el equipo de sistemas. Nadie fuera del equipo de sistemas tiene la posibilidad de utilizar el usuario root y únicamente las herramientas de este equipo ejecutan como usuario root.



5.1.3 Esquema de instancia de chat

Cabe destacar que el chat escucha, entre otros, en el puerto 80. La máquina virtual de erlang tiene habilitada la *capability* para escuchar en ese puerto.

A continuación, se muestra un ejemplo de las líneas más relevantes de uno de los ficheros de configuración más sencillos (en sintaxis erlang):

```
%%%   ==========
%%%   DEPLOYMENT

{deployment_domains, [{"xmpp1.tuenti.com",
[{generic,{192,168,1,X1}},
{generic,{192,168,1,X2}},
{generic,{192,168,1,X3}},
{generic,{192,168,1,X4}}]}
]}.


{deployment_instances, [  {'chmia-prod-1i1',generic,{192,168,1,X2}},
  {'chmia-prod-2i1',generic,{192,168,1,X1}},
  {'chmia-prod-3i1',generic,{192,168,1,X3}},
  {'chmia-prod-4i1',generic,{192,168,1,X4}}
]}.


%%%   ==============
%%%   SERVED DOMAINS

{hosts, ["xmpp1.tuenti.com"]}.
```

Como se puede apreciar, el servidor de chat requiere conocer la dirección IP de sus vecinos.

4.2. Ciclo de desarrollo del chat

El proceso de generación de versiones en el chat es muy manual y necesita una actualización:

Tras introducir un cambio en el repositorio del chat y ejecutar los tests automáticos (es necesario tener instaladas la versión de erlang y las bibliotecas utilizadas en producción), se decide generar una nueva versión.

El paquete correspondiente del despliegue se tiene que compilar en una máquina especial (`chatbuilder-dev`), que posee exactamente la misma configuración (versión del sistema operativo, versión de erlang y bibliotecas) que las máquinas de producción. Esto es necesario ya que producción tiene una versión del sistema operativo distinta y hay un módulo pequeño pero imprescindible utilizado por el chat escrito en C.

La nueva versión se prueba durante un tiempo (determinado por la importancia de los cambios) con uso real en unos servidores internos que tienen, de nuevo, la misma configuración que las máquinas de producción (aunque los internos sirven más dominios de pruebas simultáneamente).

Una vez se ha validado este despliegue, la nueva versión se puede empujar a un repositorio de artefactos y a los servidores de producción, de máquina en máquina.

Durante el proceso de despliegue, se deshabilita el balanceador que redirige a la máquina, se expulsa de forma progresiva a los usuarios que aún tengan una conexión abierta, se actualiza el artefacto y se rehabilita la máquina en el balanceador.

5. Chat en docker

A continuación se expone la solución adoptada para el funcionamiento del servicio de chat sobre contenedores, tanto en su despliegue como en la generación de artefactos.

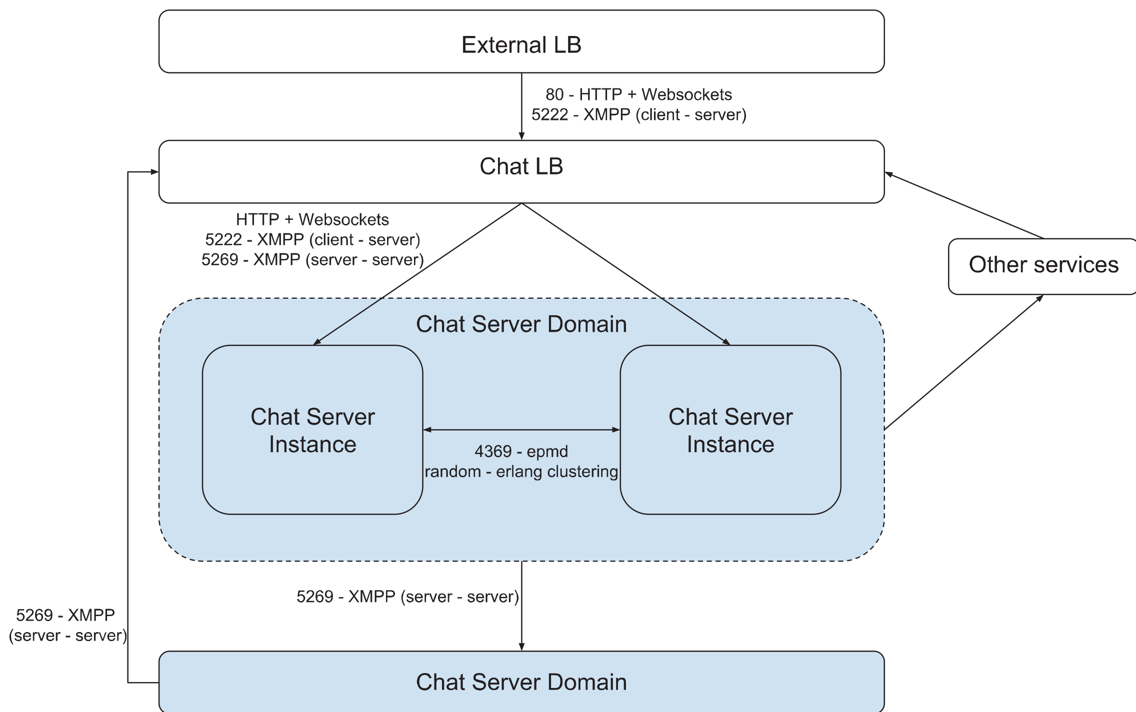
5.1. Despliegue

La infraestructura proporcionada por el equipo de sistemas consiste en que cada máquina dispondrá de docker y docker-compose así como un agente de Puppet que mantiene la configuración actualizada.

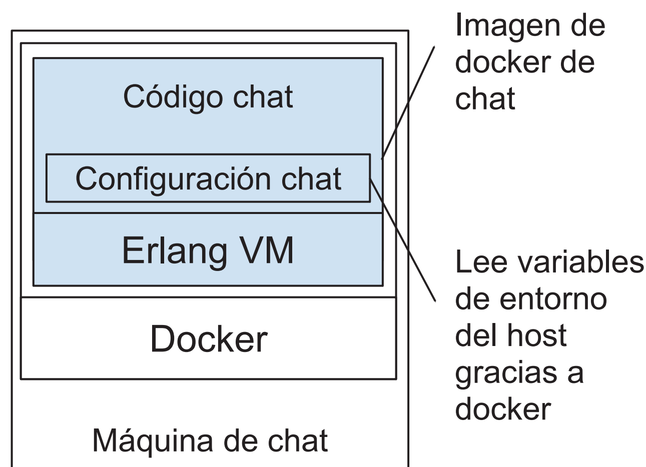
Las máquinas dispondrán también de un agente gestionado por Marionette Collective para la actualización de las versiones.

El servicio de chat será responsable de la notificación al balanceador de que quiere recibir tráfico mediante una respuesta HTTP 200 OK en la ruta /health-check, mientras que cualquier otro valor o su ausencia determinará que el balanceador no envíe tráfico.

La conectividad de la máquina y la resolución de nombres de dominio se gestionará por el equipo de sistemas y sólo existirá una instancia del servicio de chat por máquina. De esta forma se asocia una instancia de chat a una máquina, con una única dirección IP que los responsables del servicio de chat desconocen.



6.1.1 Esquema de conectividad de un dominio de chat



6.1.2 Esquema de instancia de chat con docker

Se ha decidido utilizar docker-compose como formato ya que el servicio de chat necesita

acceso ilimitado a la pila de red del host, así como ciertos directorios compartidos entre el docker y el host para los registros de eventos.

Docker-compose facilita esta configuración de forma consistente y con documentación fácilmente accesible.

A partir de este momento, un artefacto del servicio de chat se corresponde con una imagen de docker y un fichero de configuración de docker-compose, debidamente versionados. Se reproduce la plantilla para generar estos ficheros de configuración a continuación:

```
$ cat chatserver-$DOCKER_TAG.yaml
---
version: '2'
services:
  chat:
    image: $DOCKER_IMAGE_PATH:$DOCKER_TAG
    container_name: chat
    restart: always
    network_mode: host
    volumes:
      - /var/xconfig:/var/xconfig
      - /chat/etc/override:/chat/etc/override
      - /chat/storage:/chat/storage
    environment:
      - XMPP_DOMAINS
      - MEMTABD_HIGH_LIMIT
      - MEMTABD_LOW_LIMIT
      - MEMTABD_NODES
      - CONFIG_ENVIRONMENTS
...
```

Esta plantilla contiene la imagen de docker versionada (tanto el fichero de configuración de docker-compose como la imagen de docker comparten versión) así como las variables de entorno que docker-compose tiene que compartir entre el host y el contenedor.

También se pueden ver los directorios compartidos entre ambos sistemas para la configuración dinámica y los registros de eventos.
Si no se compartieran estos directorios, y por cualquier casual se reiniciara el contenedor, se perderían todos los ficheros escritos por el mismo.

El servicio de chat requiere configuración acerca de qué instancias forman un dominio de chat así como de los dominios de los que esa máquina es responsable. Hasta ahora esta información se le proporcionaba mediante los ficheros de configuración con nombre e IP en formato erlang, a partir de ahora esta información se corresponde únicamente con el nombre

de la máquina y el servicio de chat las lee de variables de entorno definidas en puppet.

Un ejemplo de esta definición que permite al chat calcular automáticamente los nombres de sus vecinos es:

```
xmpp_domains: xmpp1.tuenti.com

memtabd_nodes_low_limit: 1
memtabd_nodes_high_limit: 4
```

Mientras que otro ejemplo que establece explícitamente los nombres de los vecinos es:

```
xmpp_domains: xmpp2.tuenti.com

memtabd_nodes: "chat@ch-prod-5.fqdn.tail:\
                chat@ch-prod-6.fqdn.tail:\
                chat@ch-prod-7.fqdn.tail:\
                chat@ch-prod-8.fqdn.tail"
```

De esta forma, se consigue dividir la configuración estática del servicio en dos partes, la que gestionan los responsables del servicio de chat de configuración general del servicio, dentro del repositorio de chat, y la que corresponde a infraestructura y depende de la máquina, en el repositorio de puppet.

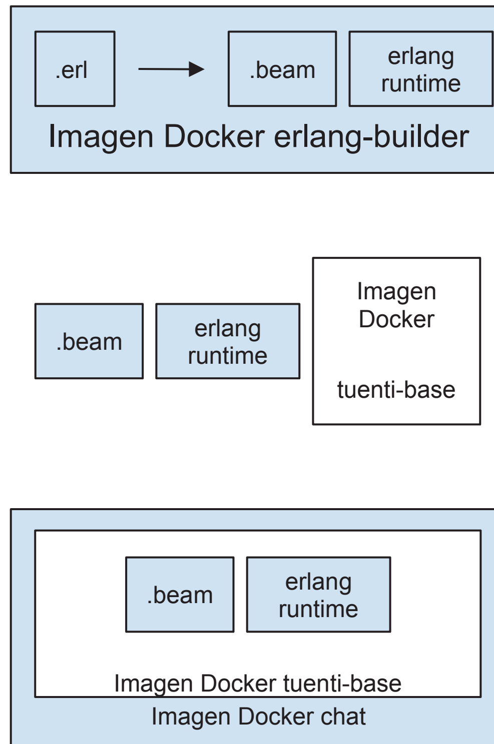
El chat utiliza un servicio interno para configuración dinámica, *xconfig*, que es desplegado por el equipo de sistemas en las máquinas de forma estándar.

5.2. Ciclo de desarrollo del chat con docker:

Gracias a docker, es posible simplificar el proceso de desarrollo.

Para sustituir a la máquina `chatbuilder-dev` se ha creado una imagen de docker que tiene todos los requisitos necesarios para compilar el chat. Esta imagen permite no sólo abandonar el uso de `chatbuilder-dev`, sino que cualquier persona pueda desarrollar sobre el chat simplemente utilizando esta imagen. Esta imagen se utiliza exclusivamente para la compilación y ejecutar los tests automáticos.

Una vez el nuevo bytecode ha sido generado, se incluye junto con la máquina virtual de erlang en una nueva imagen que es la que se despliega posteriormente. Este proceso tiene la ventaja añadida de que la imagen base utilizada para ejecutar el chat es una imagen preparada por otro equipo con un conjunto de herramientas comunes.



6.2.1 Esquema de proceso de desarrollo con docker

Al estar basado en docker, este proceso se puede ejecutar en cualquier sitio en el que esté instalado, como ordenadores de desarrollo, sistemas de compilación remota o pcs personales en caso de necesidad.

Como requisito para el despliegue, además de la publicación de la imagen de docker se publica también un fichero de configuración para docker-compose.

En el momento de empujar la nueva versión a los entornos de preproducción o producción, el sistema es similar hacia el responsable, pero es necesario reescribir los pasos ejecutados por puppet.

5.3. Alternativas y cambios realizados:

En tuenti utilizamos kubernetes para otros sistemas, que requiere que los contenedores sean docker, por lo que en ese aspecto no ha habido alternativa.

Los cambios realizados requeridos, por proyecto:

Chat:

- Gestión de interfaces en las que el chat debe escuchar en el sistema de generación de configuración estática
- Gestión de los dominios de los que un servidor es responsable en la configuración estática
- Gestión de nodos del *cluster*
- Integración con el ecosistema de docker en el proceso de desarrollo:
    - Creación de una imagen de compilación de erlang
    - Adaptar el proceso de publicación de artefactos para tratar con imágenes docker como resultado del desarrollo
    - Adaptar el proceso de pruebas para tratar con imágenes docker

Puppet:

- Definición de la información que puppet debe poseer acerca de las máquinas
- Configuración en un formato más accesible que el formato de erlang
- Reescritura del proceso de despliegue de una instancia de chat

5.4. Actualización sin pérdida de disponibilidad:

Durante la transición de un sistema a otro, es imprescindible que no se deje de prestar servicio, para lo cual hemos decidido seguir un proceso con pasos intermedios.

Para este paso ha sido imprescindible el hecho de que las instancias, tanto las antiguas como las nuevas, hayan sido definidas con la opción -name en erlang. Si no hubiera sido el caso, las máquinas nunca hubieran podido formar un *cluster* por limitación del lenguaje.

Para cada dominio:

1. Se definen las nuevas máquinas en puppet teniendo a las antiguas como parte del *cluster*, pero aún no se crean.
2. Se publica una nueva versión del chat en el que la máquinas antiguas tienen a las nuevas como parte del *cluster* y se despliega de la forma habitual.
3. Se crean una por una las máquinas nuevas y se comprueba que su funcionamiento es el correcto, compartiendo la carga del dominio con las antiguas y comunicándose con ellas.
4. Una vez validado el funcionamiento de las máquinas nuevas, se apagan las antiguas una por una.
5. Se realiza un despliegue en las máquinas nuevas eliminando las referencias a las máquinas antiguas.

6.4.1 Proceso de despliegue con configuraciones combinadas

Un ejemplo de la configuración de una de las máquinas nuevas durante este proceso:

```
xmpp_domains: xmpp1.tuenti.com

memtabd_nodes: "chmia-prod-1i1@192.168.1.X2:\
                chmia-prod-2i1@192.168.1.X1:\
                chmia-prod-3i1@192.168.1.X3:\
                chmia-prod-4i1@192.168.1.X4:\
                chat@ch-prod-1.fqdn.tail:\
                chat@ch-prod-2.fqdn.tail:\
                chat@ch-prod-3.fqdn.tail:\
                chat@ch-prod-4.fqdn.tail"
```

6. Chat en Kubernetes

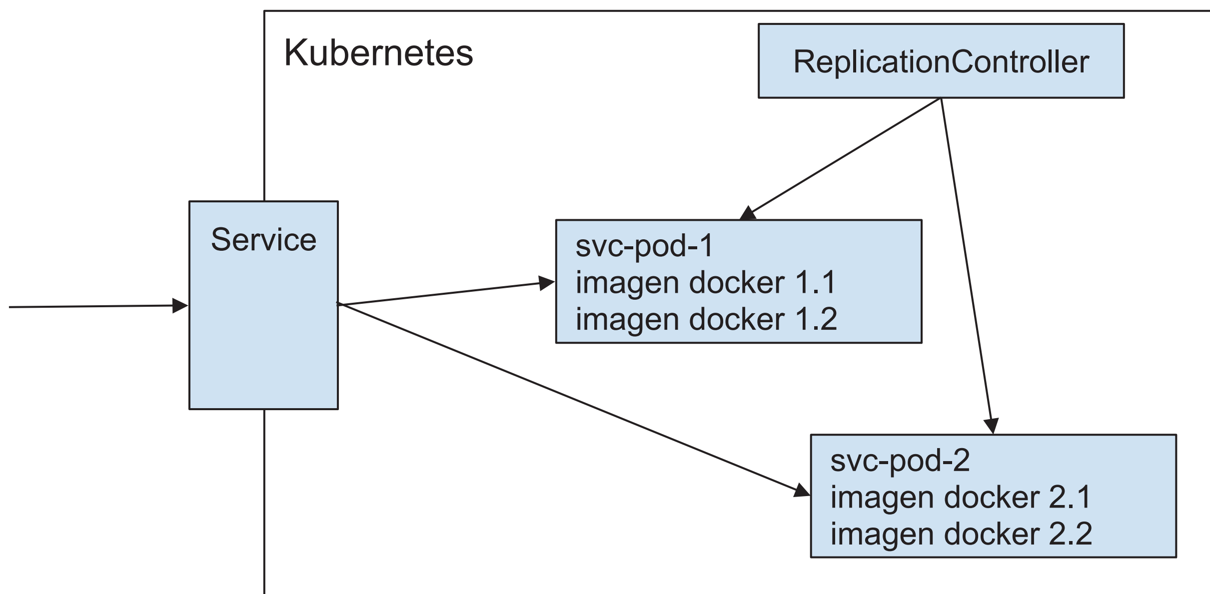
En los apartados anteriores se ha expuesto la transformación en una solución basada en contenedores de un servicio de chat, y es la solución actualmente implementada en tuenti en la mayoría de los entornos.

Sin embargo, durante el proceso de desarrollo es necesario proporcionar el servicio de chat a otros equipos. Teniendo en cuenta la cantidad de entornos de desarrollo distintos (al menos uno por proveedor) a los que es necesario dar soporte, el uso de máquinas dedicadas o máquinas virtuales no es viable.

Kubernetes nos soluciona esta necesidad, pero sólo en el caso de desarrollo, ya que el chat tiene un perfil particular (elevado uso de RAM y conexiones persistentes) y en preproducción y producción es preferible tener máquinas dedicadas.

Kubernetes es un sistema de orquestación que permite mantener contenedores activos de forma automática, comunicándose entre ellos mediante peticiones de red.
La unidad más básica de kubernetes es el pod, que corresponde con un conjunto de contenedores ejecutando en el mismo *namespace* de red.

Los pods tienen dos facetas: Hay una clase de recurso de Kubernetes que crea o destruye el pod y hay otra clase de recurso que le redirige tráfico desde otros pods o desde el exterior.



7.1.1 Esquema de funcionamiento básico de Kubernetes

6.1. Despliegue

En el caso de kubernetes, el chat se despliega como cualquier otro servicio, es decir, consta de ciertos recursos.

ReplicationController:

Un recurso del tipo ReplicationController define la creación y destrucción de pods, pero los *pods* de un RC obtienen un *hostname* pseudoaleatorio. El chat necesita que todas sus instancias conozcan a priori los nombres del resto, algo que no se puede realizar con un ReplicationController.

StatefulSet:

Un StatefulSet, sin embargo, permite que los pods obtengan nombres predecibles, por lo que es el necesario para el despliegue del chat. Todo StatefulSet necesita un Service de kubernetes para mantener la identidad de red de los pods.

En la definición del StatefulSet, además de las imágenes de docker que tiene que ejecutar, es necesario especificar qué puertos escucha el *pod* así como los volúmenes compartido y las peticiones que kubernetes puede hacer para detectar si el pod está funcionando correctamente.

A continuación se muestran algunas líneas de la plantilla utilizada para la definición del StatefulSet:

```
   spec:
      volumes:
        {{ common.volumes_app() }}
      containers:
      - name: chat
        image: {{ docker.registry }}/{{ docker.image }}:
{{ docker.tag }}
        imagePullPolicy: Always
        volumeMounts:
          {{ common.mounts_app() }}
        ports:
        - name: web
          containerPort: 80
        - name: backdoor
          containerPort: 8000
        - name: xmpp
          containerPort: 5222
        - name: xmpp-internal
```

```
          containerPort: 5269
        - name: epmd
          containerPort: 4369
        env:
        - {name: MEMTABD_HIGH_LIMIT, value: "4" }
        - {name: XMPP_DOMAINS, value: "{{ domain + '.tuenti.io' }}"}
        - {name: CONFIG_HELPER_NO_MODE_FLAG, value: "true"}
          {{ common.env_app() }}

        readinessProbe:
          httpGet:
            path: /health-check
            port: 80
        livenessProbe:
          httpGet:
            path: /health-check
            port: 80
        lifecycle:
          preStop:
            exec:
              command: ["bin/ejabberdctl", "chat", "ctl", "I", "local",
"shutdown", "0"]
...
```

Es conveniente destacar `CONFIG_HELPER_NO_MODE_FLAG`, que se mencionará en el siguiente apartado.

Service:

Los Service son los encargados de redirigir tráfico desde el exterior o desde otros pods al servicio. El chat necesita dos Service adicionales al del StatefulSet: Uno para conexiones desde el exterior y otro para las conexiones desde el interior. En ambos casos el tráfico es TCP.

A continuación se muestran algunas líneas de las plantillas que definen estos dos servicios:

```
# External service to expose ports to be accessed by the apps
# - kube2lb does not understand string targetPort, it must be port
number
kind: Service
apiVersion: v1
metadata:
  name: {{ domain }}-external
  annotations:
    kube2lb/port-mode: |
      {"xmpp":"tcp","web":"tcp"}
```

```
  labels:
    app: {{ k8s_name }}
    domain: {{ domain }}
    service: {{ service_name }}-{{ domain }}
spec:
  ports:
  - name: xmpp
    port: {{ external_ports.xmpp }}
    targetPort: 5222
  - name: web
    port: {{ external_ports.web }}
    targetPort: 80
  selector:
    app: {{ k8s_name }}
    domain: {{ domain }}
  type: LoadBalancer
  loadBalancerIP: "{{ external_ip.get(domain) }}"
...
---
# Internal service to be accessed by the services @ {{ domain }}
kind: Service
apiVersion: v1
metadata:
  name: {{ domain }}
  labels:
    app: {{ k8s_name }}
    domain: {{ domain }}
    service: {{ service_name }}-{{ domain }}
spec:
  ports:
  - name: backdoor
    port: 8000
    targetPort: backdoor
  - name: xmpp-internal
    port: 5269
    targetPort: xmpp-internal
  selector:
    app: {{ k8s_name }}
    domain: {{ domain }}
  type: ClusterIP
...
```

En el ámbito de las direcciones IP externas es conveniente destacar que en los entornos de producción y preproducción el puerto que se expone hacia internet es siempre el 443, para evitar firewalls. Esto provoca que todos los dominios tengan que estar en direcciones públicas.

Reservar direcciones públicas para todos los dominios de los entornos de desarrollo es inviable, por lo que cada entorno de desarrollo tiene 4 dominios de chat, y todos los entornos tienen las mismas direcciones IP públicas para los dominios de chat, multiplexando por el puerto de acceso.

6.2. Ciclo de desarrollo

El ciclo de desarrollo no cambia, ya que se despliega una imagen de docker. En este caso no hace falta el fichero de configuración de docker-compose.

Para el despliegue se utiliza una herramienta interna para gestionar los entornos kubernetes con plantillas.

6.3. Alternativas y cambios realizados

Al igual que en el caso de docker, kubernetes ya se estaba usando como infraestructura, por lo que no hubo consideración de una alternativa.

Los cambios realizados requeridos, por proyecto:

Chat:

- Los servicios en kubernetes, debido al despliegue de tuenti, utilizan unas rutas de configuración distintas, por lo que ha sido necesario modificar el chat para poder funcionar con ambos conjuntos de rutas. La variable de entorno `CONFIG_HELPER_NO_MODE_FLAG` es la que se utiliza para determinar qué conjunto de rutas utilizar.

Definiciones de Kubernetes:

- Creación de plantillas
- Creación de particularizaciones de plantillas

7. Líneas futuras

Como posible continuación de este trabajo, sería interesante probar a utilizar docker-compose como plataforma de orquestación (Swarm) o kubernetes en modo de un único nodo. Se está trabajando en docker-compose para que pueda ser una alternativa a kubernetes, pero kubernetes tiene ventaja y un gran equipo detrás. Utilizar kubernetes en modo de un único nodo podría ser factible, pero quizás la complejidad de este despliegue es demasiada para algo que docker-compose soluciona con facilidad.
La principal ventaja de esta línea sería la reducción del número de tecnologías utilizadas globalmente.

Otra posible continuación de este trabajo es la investigación de la viablidad de rkt, rkt es un sistema de contenedores análogo a Docker pero construido desde cero, con la ventaja de que en su etapa de diseño ya conocen los casos de uso y con la desventaja de que es necesario reimplementar muchas de las características de Docker.

Otra posibilidad adicional es la modificación del chat para poder acceder a la API que expone Kubernetes a todos los pods desde el interior, esta modificación nos permitiría poder tener mayor control sobre el número de nodos y sus nombres, pudiendo de esta forma gestionar mejor el estado de cada uno de los dominios internamente.

Como propuesta final, sería necesario considerar la creación de un *cluster* de kubernetes adicional única y exclusivamente para el despliegue del chat. Al ser un conjunto de nodos separado, no habría problemas con los picos de carga del resto de servicios y se podría asumir una disponibilidad similar a la generada por el despliegue con docker-compose. Esta propuesta permitiría el uso de las herramientas internas actualmente diseñadas para kubernetes también con el chat, unificando de esta forma todos los despliegues.

8. Conclusiones

Como conclusiones de las tareas, en primer lugar me gustaría destacar la calidad de la documentación tanto de docker como de kubernetes, que es excelente y me ha facilitado la tarea de recopilación de información. En el caso de puppet, también es buena, pero me ha sido más útil consultar ejemplos existentes.

Sin embargo, en el caso del servicio de chat la documentación es inexistente, por lo que la única posibilidad ha sido el estudio del código. Es mi intención que este documento sirva también como ayuda en Tuenti acerca del despliegue de chat.

El diseño de una solución basada en contenedores ha sido relativamente directa una vez he conocido las características y limitaciones de docker, aunque para su implementación ha hecho falta modificar gran cantidad de *scripts* utilizados en y por el chat además de los cambios requeridos por el propio chat. Como tarea pendiente queda modificar el entorno de chat para incluir un Makefile o herramienta similar.

Mientras que el despliegue hubiera debido ser algo fácil, el hecho de tener que mantener la disponibilidad mientras un cluster mantenía máquinas con la nueva configuración junto con máquinas con la antigua ha complicado esta tarea más de lo necesario. La causa de esta complicación ha sido el sistema de despliegue antiguo, extremadamente enrevesado.

Por último como conclusión de las tareas, el desarrollo de la solución basada en sistemas de orquestación fue realmente frustrante hasta el descubrimiento de los StatefulSets, ya que consideré que incluso tenía que modificar el chat para que los nombres de los nodos del *cluster* pudieran ser dinámicos, que es una tarea ingente.

En cuanto a las herramientas utilizadas, Docker es una tecnología revolucionaria que permite simplificar en gran medida la generación y publicación de servicios, pero no sólo sirve para eso. Al poder compartir volúmenes entre la máquina host y el contenedor, tienes a tu disposición configuraciones efímeras que puedes explotar para agilizar el desarrollo o hacer compilaciones destinadas a máquinas con configuraciones distintas, todo ello sin el sobrecoste que implica una máquina virtual.

Gracias a los contenedores, cualquier persona con acceso al repositorio tiene a su disposición todas las herramientas necesarias para desarrollar, probar y publicar un servicio sin necesidad de instalar otras herramientas además de docker, lo que agiliza el desarrollo y la generación de versiones.

Además, al servir de frontera entre la máquina y el servicio, Docker permite delimitar responsabilidades desde el principio, de nuevo reduciendo las interrupciones y el tiempo invertido en la sincronización necesaria entre los equipos que gestionan ambas responsabilidades.

Otro excelente ejemplo de la potencia de docker es kubernetes, que te permite configurar una nube de microservicios con unos ficheros de configuración sencillos.

No obstante, hay que tener en cuenta que el hardware sobre el que se ejecuta puede ser distinto, lo que puede generar errores si los juegos de instrucciones no son compatibles. No ejecuta como una máquina virtualizada, es un proceso más del sistema operativo.

Por otro lado, la transformación de un servicio de chat a contenedores presenta retos debido a la naturaleza distribuida pero con estado y conexión persistente de su funcionamiento, lo que choca frontalmente con la filosofía de "servicio efímero y sin estado" subyacente en kubernetes. Sin embargo, no deja de ser posible aprovechar las ventajas de cada una de las tecnologías para cada caso concreto.

Tanto docker como kubernetes son tecnologías jóvenes pero ya han demostrado que tienen un hueco merecido entre las herramientas de los ingenieros de software, no sólo entre aquellos dedicados exclusivamente a la administración de sistemas. En el caso de docker, aún más que merecido.

Aunque es la más madura, docker no es la única tecnología de contenedores disponible, otras soluciones como rkt existen, sin embargo, sea cual sea la que termine triunfando, los conceptos aprendidos para cualquiera de ellas muy probablemente sirvan para las demás, con los cambios de sintaxis que sean necesarios.

A título personal, este trabajo me ha resultado muy interesante ya que siempre he tenido interés por el funcionamiento del sistema operativo y no es un área en el que, trabajando con erlang en Tuenti, me pueda introducir a menudo. Particularmente, Kubernetes ha supuesto un verdadero desafío porque la forma que tenemos de gestionar los nodos no se ajusta al modelo de uso más común de esta tecnología.

Por otro lado, este desarrollo me ha resultado extremadamente gratificante pues, aunque el chat de Tuenti no tiene el uso ni mucho menos como hace unos años, toda la metodología expuesta en esta memoria está ahora mismo funcionando en un entorno real.

9. Bibliografía

Armstrong, J (2003). "Making reliable distributed systems in the presence of software errors" Ph.D. Dissertation. The Royal Institute of Technology, Stockholm, Sweden.

Armstrong, J.; Däcker, B.; Lindgren, T.; Millroth, H. (2011) "Open-source Erlang – White Paper".

Benington, Herbert D. (1983). "Production of Large Computer Programs" *IEEE Annals of the History of Computing*. IEEE Educational Activities Department. **5** (4): 350–361. doi:10.1109/MAHC.1983.10102.

Brown, Neil. Recuperado en marzo de 2017 https://www.kernel.org/doc/Documentation/filesystems/overlayfs.txt (OverlayFS)

Docker. Inc. Recuperado entre marzo y junio de 2017, https://docs.docker.com/

Docker. Inc. Recuperado entre marzo y junio de 2017, https://docs.docker.com/compose/ (Docker-Compose)

Example42. Recuperado entre marzo y mayo de 2017, https://github.com/example42/puppi (Puppet)

Erlang.org Recuperado en abril de 2017, http://erlang.org/doc/

Franceschi, A. Recuperado entre marzo y mayo de 2017, https://puppet.com/presentations/automating-applications-deployments-puppi (Puppet)

Google. Recuperado entre mayo y junio de 2017, https://kubernetes.io/

HashiCorp. Recuperado en marzo de 2017, https://www.vagrantup.com/docs/index.html (Vagrant)

Hunt, A.; Thomas, D. (1999) "The Pragmatic Programmer", Addison-Wesley

Internet Engineering Task Force. Recuperado en febrero de 2017, https://tools.ietf.org/html/rfc6120 (XMPP)

Puppet. Recuperado entre marzo y mayo de 2017, https://docs.puppet.com/

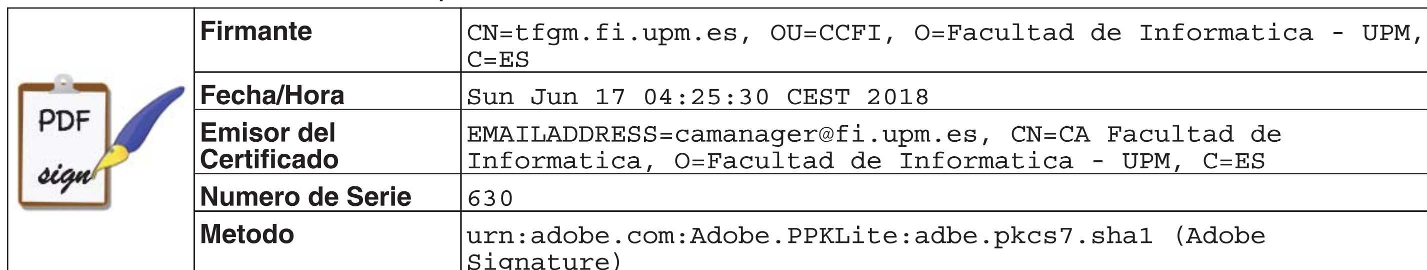
Puppet. Recuperado entre marzo y mayo de 2017,
https://puppet.com/docs/mcollective/current/index.html (Marionette Collective)

Rebar3. Recuperado entre febrero y mayo de 2017, https://www.rebar3.org/docs

Shchepin, A. Recuperado entre marzo y abril de 2017, www.ejabberd.im

Walredud, J. (1998) "Erlang/OTP Released as Open Source™, 1998-12-08".

Este documento esta firmado por



	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=Facultad de Informatica - UPM, C=ES
	Fecha/Hora	Sun Jun 17 04:25:30 CEST 2018
	Emisor del Certificado	EMAILADDRESS=camanager@fi.upm.es, CN=CA Facultad de Informatica, O=Facultad de Informatica - UPM, C=ES
	Numero de Serie	630
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)