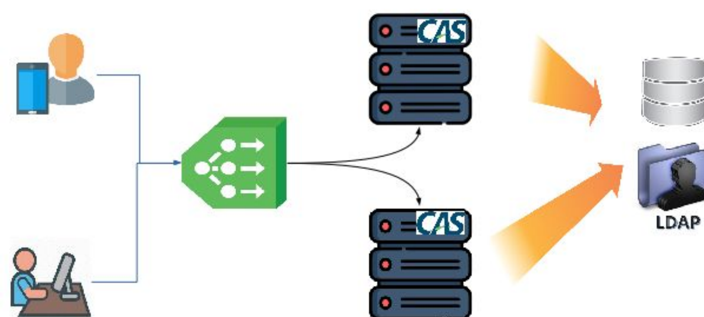
Universitat
Oberta
de Catalunya

Implantación y pruebas de rendimiento de sistema de autenticación centralizada en alta disponibilidad con Apereo CAS



Moisés Ramón Rodríguez Barrera

Titulación: Grado de Ingeniería Informática
Tutor: Joaquín Lopez Sanchez-Montañes
Trabajo Fin de Grado - Convocatoria: Junio 2019

Contexto

En las últimas décadas, el concepto de informática ha pasado a referirse desde “el ordenador personal” a “la nube y sus servicios”. Las organizaciones han visto cómo la digitalización ha sido un requisito y han tenido que adaptar muchos de sus procedimientos manuales de manera que sean accesibles a través de un programa informático o internet. Debido a que en algunas ocasiones los servicios publicados en una red, como podría ser internet, deben ser solamente accesibles para un interesado o conjunto de interesados, surge la necesidad de identificar a las personas antes de que puedan acceder.

Los sistemas de identificación han evolucionado adaptándose a las nuevas tecnologías y convirtiéndose en un servicio imprescindible para las organizaciones, llegando a ser un servicio fundamental que cumple con la identificación telemática de los usuarios que acceden a sus servicios.

Es por todo lo anterior y debido a la creciente demanda que está habiendo en el sector de los servicios de internet, realizaré este trabajo de fin de grado con el fin de analizar el amplio abanico de soluciones tecnológicas, seleccionar la más conveniente e implantar la solución.

Object

Over the last decades, the computing concept has evolved from “personal computer" to "the cloud". Organizations have noticed how digitization become standard, and they have been forced to adapt many of their manual process, enabling them to be accessible from computer programs or the internet. Because in some occasions services published in a network such as the Internet could only be accessible to an interested party or group of interested parties, here appears the requirement to identify people before they can access to it.

Identity systems have evolved adapting to new technologies and becoming an essential service for organizations, turning about as a fundamental service that complies with the telematic identification of users who access their services.

This is all to say that due to the growing demand in the sector of Internet services, I will perform this end-of-degree work in order to analyze the wide range of technological solutions, selecting the most convenient and implement the product for my academical idea.

Índice

Contexto 1

Object 1

Índice 2

1. Introducción 4

1.1 Contexto y justificación del trabajo 4

1.2 Objetivos del trabajo 4

1.3 Enfoque y método seguido 5

1.4 Planificación del trabajo 5

1.5 Productos obtenidos 6

2. Documentación de Apereo CAS 7

2.1 Prefacio y características generales 7

2.2 Requisitos Hardware y Software 8

2.3 Alta disponibilidad de Apereo CAS 9

3. Análisis y aspectos económicos 10

3.1 Balanceador 10
Nginx 11
HAProxy 11

3.2 Directorio de usuarios 11

3.3 Estudio económico 12
Hardware 12
Software 13
Personal técnico 13

4. Instalación e implantación de la solución 15

4.1 Instalación de las máquinas 15

4.2 Parametrización del directorio de usuarios. 17
4.2.1 Instalación de OpenLdap. 17
4.2.2 Aprovisionamiento de usuarios 20

4.3 Parametrización de nodos de CAS. Configuración standalone. 21
4.3.1 Instalación y configuración de componentes 21
4.3.2 Configuración de componentes extra 23
4.3.3 Configuración de aplicación DemoCAS 24

4.3.4 Empaquetar y ejecutar el servicio de CAS 25
4.3.5 Instalación de Nginx como frontal de cada nodo de CAS. 28
4.4 Parametrización de balanceador 31
4.4.1 Configuraciones generales 31
4.4.2 Añadir un servicio de frontend 32
4.4.3 Añadir los servicios de backend 32
4.4.4 Habilitar las estadísticas de HaProxy 33
4.4.5 El resultado de la configuración 34
4.5 Parametrización de nodos de CAS con alta disponibilidad 37
5. Ejecución de pruebas de rendimiento 40
5.1 Introducción y contexto de las pruebas 40
5.1.1 Software de pruebas de rendimiento 40
5.1.2 Comparativa de software de pruebas de rendimiento 41
5.1.3 Instalación de Jmeter 43
5.1.4 Objetivo 43
5.2 Directorio de usuarios Openldap 44
5.2.1 Aspectos generales 44
5.2.2 Resultados de la prueba y análisis 46
5.3 Nodo CAS en standalone 46
5.3.1 Resultados de la prueba y análisis 46
5.4 Servicio de CAS en alta disponibilidad. 49
5.4.1 Comparativa de rendimiento de diferentes tecnologías de replicación de tickets. 51
5.4.2 Prueba de rendimiento con dos nodos activos replicando tickets a través de Hazelcast 52
6. Conclusiones y posibles mejoras 55
7. Glosario 56
8. Índice de Figuras 57
9. Bibliografía 58

1. Introducción

La memoria generada en este estudio aportará al lector un resumen ordenado de todos los conocimientos adquiridos durante este trabajo y de las decisiones tomadas durante el mismo. Además se detallarán los pasos más relevantes para poder llegar a disponer de un servicio de autenticación con Apereo CAS en alta disponibilidad.

1.1 Contexto y justificación del trabajo

En toda empresa o administración pública que ofrezca servicios a sus usuarios a través de internet necesita tener un sistema de autenticación e incluso de autorización adecuado a sus necesidades. Aunque muchas organizaciones ya disponen de sistemas de identificación que les permite autenticar a los usuarios, se ha detectado que no siempre los sistemas de identificación son centralizados y que el usuario acaba introduciendo repetidas veces su usuario y contraseña o incluso diferentes contraseñas en función del programa al que quieren entrar.

Si bien en determinados ámbitos el uso de diferentes contraseñas podría considerarse deseable aludiendo motivos de seguridad, la realidad es que el rápido crecimiento de los sistemas de información de las organizaciones, junto con una mala planificación de los responsables de tecnologías de la información pueden haber facilitado la existencia de estos no deseables sistemas de validación inconexos.

1.2 Objetivos del trabajo

Se plantea cumplir con los siguientes puntos al finalizar el trabajo:

- Evaluar los componentes necesarios para montar adecuadamente un servicio de autenticación.
- Disponer de un servicio de autenticación en alta disponibilidad basado en el producto Apereo CAS.
- Realizar pruebas de rendimiento de la infraestructura y optimizar la parametrización del servicio en función a los resultados que se vayan obteniendo.
- Se dará prioridad al uso de software libre frente a una solución de pago siempre que el software libre cumpla con los requisitos del servicio.

1.3 Enfoque y método seguido

Debido a que se debe respetar la temática del proyecto, la cual consiste en implantar una solución con Apereo CAS, la línea de trabajo consistirá en realizar un estudio de los componentes necesarios para obtener un producto final que permita disponer de un sistema en alta disponibilidad de validación centralizada de usuarios en una organización. Además y como medida certificadora del correcto funcionamiento del trabajo, se realizarán pruebas de rendimiento que buscarán datos para realizar la posible optimización del montaje inicial del sistema, teniendo en cuenta la limitación de recursos que se disponen para este proyecto académico.

1.4 Planificación del trabajo

Según la estimación de tareas para este proyecto se ha preparado un diagrama de Gantt. Este diagrama nos permitirá modelar la planificación y su objetivo es mostrar el tiempo de dedicación previsto para cada una de las tareas a lo largo del tiempo total asignado para el proyecto.

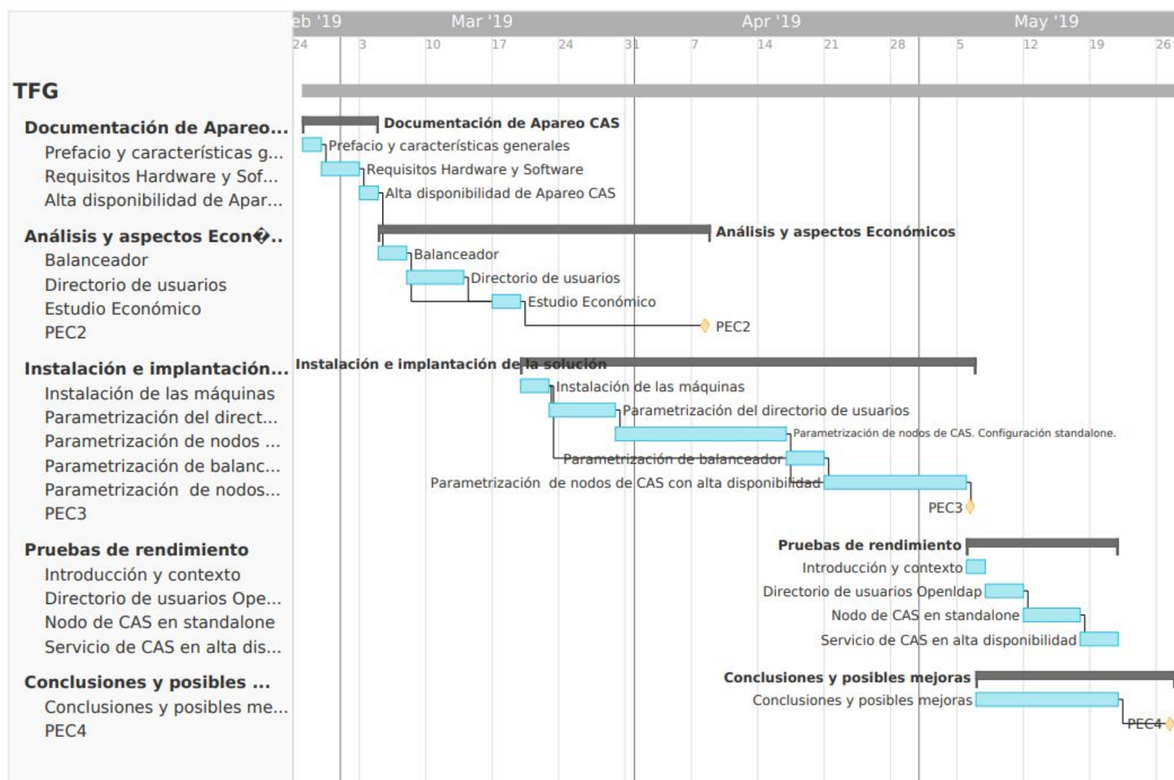


Figura 1: Diagrama de Gantt. Planificación temporal.

1.5 Productos obtenidos

Al finalizar este trabajo de fin de grado, se espera disponer de un servicio de autenticación centralizada en alta disponibilidad que permita autenticar a los usuarios de una institución. Esta infraestructura sería el punto de encuentro inicial y común para todos los usuarios que necesitan validarse en aplicaciones de la institución. Una vez validado el usuario, la aplicación será notificada con los datos del usuario.

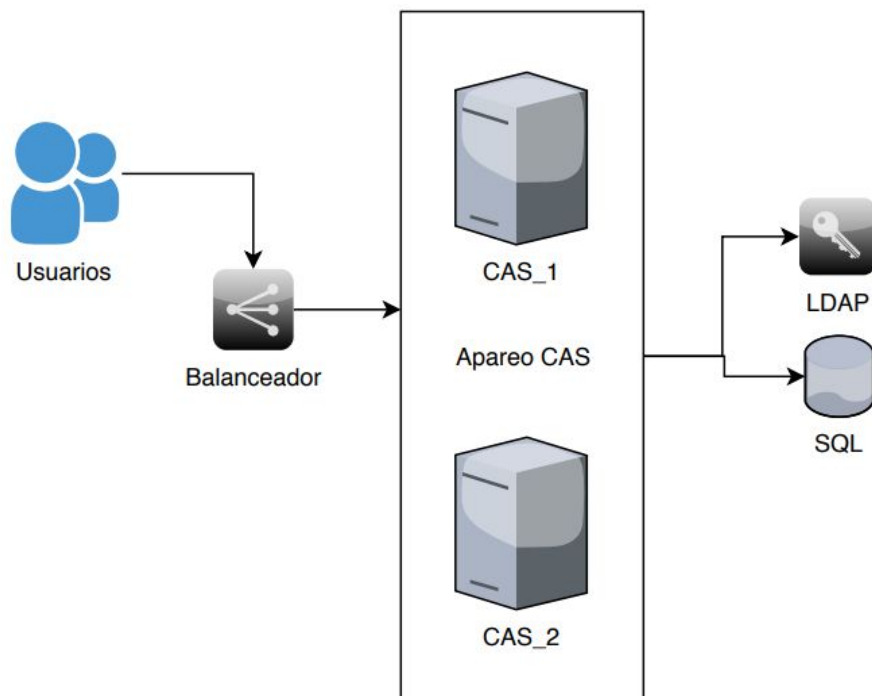


Figura 2: Representación de la Infraestructura de autenticación

2. Documentación de Apereo CAS

2.1 Prefacio y características generales

Como ya se ha ido adelantando en el capítulo anterior, el servicio de autenticación centralizada del inglés (Central Authentication Service o CAS) es un aplicación que implementa un servicio de Single Sign-on o SSO y que permite a los usuarios validarse en aplicaciones de una organización o incluso en aplicaciones de terceros. El principal beneficio de este sistema es que no es necesario que la aplicación disponga de los credenciales, comúnmente el par usuario-contraseña, del usuario para autenticarlo, sino que delega la autenticación a CAS y a través de un cliente de CAS, se completará el protocolo de autenticación para recibir como resultado los atributos de un usuario autenticado o la denegación de credenciales en caso de que el usuario no sea validado.

Desde el punto de vista estructural, el protocolo de CAS involucra por un lado a un servidor de CAS que es dónde se encuentra el núcleo del sistema, luego una aplicación la cual conoce al servidor de CAS y los mecanismos para comunicarse y finalmente al navegador del usuario interesado en acceder a la aplicación. Además de todo esto, se requiere de un sistema que contenga un directorio con todos los usuarios y contraseñas tal como podría ser por ejemplo una base de datos o un servicio de LDAP.

El sistema funciona de la siguiente manera:

- El usuario accede a una aplicación a través del navegador web.
- Si la aplicación requiere conocer el usuario que accede, se redireccionará la navegación al servidor de CAS, para que el usuario escriba sus credenciales.
- El usuario introduce su par usuario-contraseña. Comúnmente, si los credenciales son erróneos, se volverán a solicitar o se redireccionará a una página de error; por otro lado, si los credenciales son correctos, el servidor de CAS generará un ticket de servicio (Service Ticket o ST) y reenviará al navegador del usuario de nuevo a la aplicación con dicho ST.
- Al recibir la aplicación una petición con el ST, realiza una llamada directa contra el servidor de CAS para validar el ST del usuario
- En un paso posterior, el servidor también devolverá atributos del propio usuario como podrían ser, nombre de usuario, email, nombre y apellidos, entre otros.
- Es en este momento en el que el usuario está correctamente autenticado tanto en la aplicación como en el servidor de CAS.

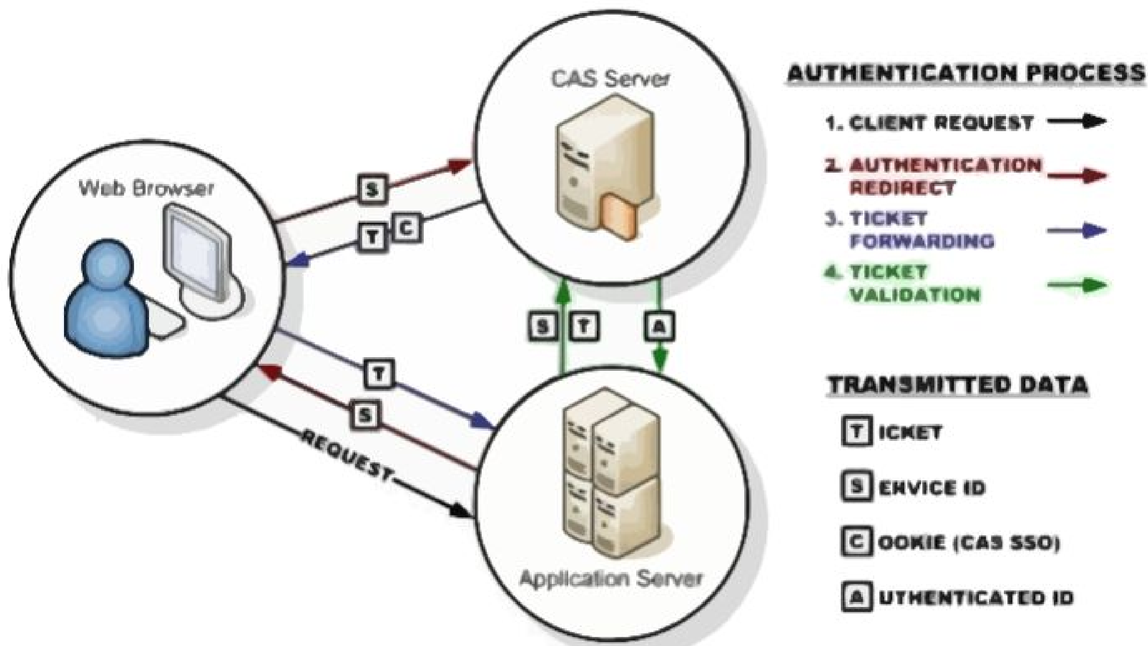


Figura 3: Ilustración flujo de autenticación de Apereo CAS. Modern authentication techniques in Python web applications. PyGrunn talk by Artur Barseghyan. Year 2014.

- El gran beneficio de la tecnología de validación centralizada o SSO es que cuando un usuario autenticado requiere acceder a una segunda aplicación, la aplicación le permitirá el acceso y obtendrá sus datos automáticamente tras validar que el Service Ticket (ST) sigue siendo válido para el servidor de CAS.

2.2 Requisitos Hardware y Software

La documentación de requisitos de instalación proyecto de Apereo CAS[1], indica que el servicio requiere de muy pocos recursos, pero especifica también que en función del número de peticiones que deba atender, el servidor deberá tener mejores componentes. Para iniciar nuestro estudio, nos guiaremos por la recomendación de la página de Apereo CAS. Especifica que un servidor con procesador de doble núcleo a 3Ghz y 8GB de memoria RAM deberían ser suficientes para ofrecer un servicio de CAS. En nuestro caso como el servicio se montará en alta disponibilidad, necesitamos un mínimo de dos servidores con estas características. Además de lo anterior necesitaremos una infraestructura de red y servicios auxiliares como podrían ser un LDAP o servidor SQL.

Debido a los escasos recursos para los que se cuenta en este proyecto académico debemos simular la infraestructura completa con máquinas virtualizadas. Utilizaremos el entorno *VirtualBox* para instalar el conjunto de máquinas necesarias de manera que obtendremos un mayor aprovechamiento de los recursos hardware del ordenador disponible a un mínimo coste.

La máquina host o servidor de la que se dispone será un Intel Core i5-8265U @ 3.90GHz con 4 procesadores y 8 hilos de ejecución, 20 GB de memoria RAM y disco duro 256GB SSD M.2 SATA3. El sistema operativo host será un Ubuntu 18.04 x64.

Se instalarán 4 máquinas guest que permitirán implementar la infraestructura.

- TFG_BAL: Implementará el balanceador por software.
  - 2 procesadores, 2gb RAM, 25gb disco, ubuntu 18.04 x64 server.
- TFG_CAS1 y TFG_CAS2: Alojarán el servicio de CAS.
  - 2 procesadores, 4gb RAM, 25gb disco, ubuntu 18.04 x64 server.
- TFG_DATA: Implementará todos los servicios auxiliares que requiera la infraestructura como será el LDAP o la base de datos.
  - 2 procesadores, 2gb RAM, 25gb disco, ubuntu 18.04 x64 server.

Además de lo anterior, los servidores de CAS requieren una máquina virtual de Java y un contenedor de servlets. Se utilizará según lo indicado en su documentación [1] la JVM openJDK y el contenedor de servlets Apache Tomcat.

2.3 Alta disponibilidad de Apereo CAS

Un sistema en alta disponibilidad(HA) es un sistema orientado a dar servicio en todo momento, aunque individualmente algún componente de este mismo presente fallo. Podría ofrecerse alta disponibilidad al servicio a partir de virtualización de una única máquina, pero el método más común para aportar tolerancia a fallos es la utilización de varios nodos que de manera conjunta y sincronizada presten servicio.

Desde el punto de vista del paradigma del alta disponibilidad las arquitecturas pueden clasificarse en:

- Activo-Pasivo: Se refiere a servicios dónde existen múltiples nodos funcionando, pero es un nodo el que recibe y procesa todas las peticiones y replica el estado al resto de nodos (nodo maestro). En caso de fallo de dicho nodo, el sistema reacciona promocionando un nuevo nodo a maestro y sacando al nodo que falla del servicio.
- Activo-Activo: En este modo de funcionamiento todos los nodos atienden peticiones y replican entre sí los datos que surjan durante su funcionamiento, para que en caso de que un nodo falle, el resto de nodos dispondrá de información suficiente para continuar trabajando sin caída del servicio.

En nuestro caso, nos decantamos por un funcionamiento **Activo-Activo** ya que el servicio a desplegar está diseñado para ello y nos permite aprovechar el 100% de la capacidad de cómputo de ambos servidores. El sistema en alta disponibilidad también debe ofrecer servicio durante momentos de actualización programada y alta concurrencia, por lo que disponer de más de un nodo activo simplificará estas labores.

3. Análisis y aspectos económicos

3.1 Balanceador

Desde el punto de vista que hemos planteado el proyecto, necesitamos un elemento por el que entren las peticiones de los clientes y mediante algún algoritmo de decisión, las reparta adecuadamente en los servidores de CAS. Este elemento, configurado de la manera correcta, aumentará la tolerancia a fallos del servicio y permitirá aumentar el número de usuarios que podrán acceder al servicio.

Principalmente se busca que sea capaz de repartir tráfico basado en puertos TCP de manera que las peticiones que lleguen a la máquina balanceadora TFG_BAL en un determinado puerto o contexto http, sea repartido entre las máquinas de validación TFG_CAS1 y TFG_CAS2 de la manera adecuada y que en caso de que alguna máquina se sature u ocurra que deje de estar disponible, el servicio de balanceo sea capaz de detectarlo; anulando el envío de peticiones a la máquina que está dando problemas y continúe ofreciendo el servicio al resto de clientes sobre la o las máquinas que funcionan de manera adecuada.

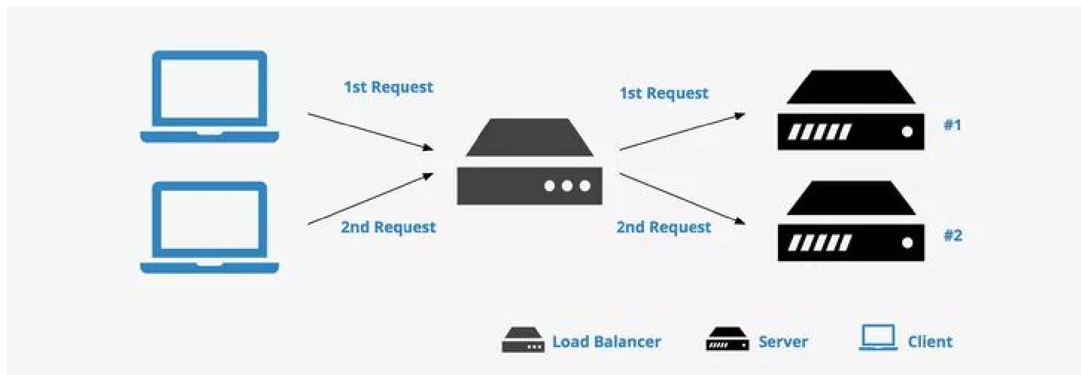


Figura 4: Gráfico del funcionamiento de un balanceador de carga [3.1]

Tras una detenida lectura de diversos documentos y estudios a través de internet, se ha podido sacar como conclusión que los productos Nginx y HAproxy son los balanceadores software que más se utilizan en grandes corporaciones debido a los buenos resultados de estabilidad y rendimiento que ofrecen. Si bien es cierto que años atrás las grandes corporaciones solían disponer de balanceadores hardware comprados a fabricantes de comunicaciones, el aumento de rendimiento de los servidores de propósito general, junto con el desarrollo de soluciones Open Source como las dos anteriores que hemos nombrado, han hecho que estos años las empresas opten instalar directamente uno de estos dos. Si el lector desea obtener más información, puede hacerlo en la bibliografía [3], [3.1] y [3.2].

Se ofrece una breve comparativa de los diferentes puntos que considero interesantes para la elección final del balanceador a desplegar.

Nginx

- \+ Muy útil si se quiere usar como servidor web o proxy inverso además de como balanceador de carga.
- \- Sus métricas son muy básicas y poco legibles. Existe una versión de pago del producto que mejora esta y otras funciones.

HAProxy

- \+ Balanceador de microcódigo optimizado, muy poco consumo de recursos.
- \+ Provee solamente funciones de balanceador de carga.
- \+ Sus métricas son muy variadas y visualmente fáciles interpretar.
- \+ Simple

Se elegirá **HAProxy como balanceador** a desplegar para este TFG debido a que según los estudios, siendo el rendimiento muy similar en ambos casos, HAProxy destaca en la simplicidad de configuración, un consumo bajo de recursos y cumple específicamente con su objetivo, que es balancear el servicio de CAS.

3.2 Directorio de usuarios

Para poder gestionar adecuadamente las validaciones en nuestro sistema CAS, se requiere que detrás exista un sistema que almacene los credenciales de los usuarios. Comúnmente estos sistemas además del nombre de usuario y la contraseña cifrada pueden contener información relevante del usuario, como podría ser el nombre, apellidos, dirección de correo y todos los atributos que se consideren necesarios o útiles y que acompañan a la identidad del usuario.

Entre todas las alternativas existentes, podríamos decir que existen dos aproximaciones básicas principales, cada una con unos beneficios y desventajas que pretendo detallar a continuación [4]:

SQL	LDAP
Pros: ● Fácil de configurar ● Recuperación de datos de usuario a través de consultas SQL. ● Facilidad de implementar para usuarios que manejen SQL previamente. Cons: ● SQL no está orientado a gestionar identidades. ● No se implementan políticas contraseñas, seguridad y recuperación.	Pros: ● Servicio ligero y rápido. ● Implementación de políticas de seguridad relacionadas con gestión de identidad. ● Dispone de comandos para validación y replicación. ● Es un protocolo abierto y estándar para la gestión de usuarios. Cons:

• No dispone de mecanismos de sincronización ni validación.	• Requiere de una configuración inicial más detallada. Políticas, ramas, grupos.

Tras la lectura de [3.2.1],[3.2.2] y analizando la tabla anterior, **nos decantamos por utilizar el protocolo LDAP para autenticar a los usuarios** de nuestro sistema debido a su madurez, la seguridad que aporta un sistema de propósito específico y la ligereza del protocolo.

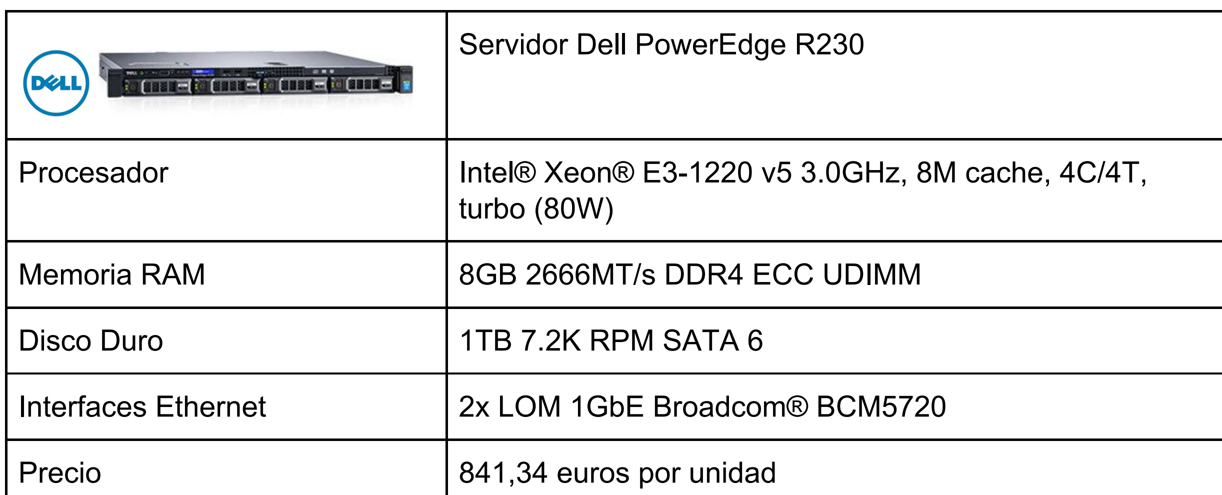
Utilizaremos **OpenLDAP como directorio de usuarios**, un software de código abierto que implementa el protocolo LDAP. Este software permitirá almacenar los credenciales de los usuarios de la organización, por medio de políticas de seguridad y su interfaz de comandos.

3.3 Estudio económico

Para poder implementar este trabajo en un entorno real y partiendo de los datos del punto 2.2 de este mismo proyecto, se ha realizado una evaluación del coste económico de este proyecto. Se asumirá que solamente se comprará hardware para los servidores de CAS y que el resto de servicios auxiliares como balanceadores, bases de datos o ldap, se servirán desde servidores virtuales o máquinas ya existentes en la organización.

Hardware

Se ha elegido el fabricante Dell como proveedor del hardware necesario y se presenta a continuación un presupuesto y características destacadas del modelo de servidor a utilizar. Destacamos que el modelo viene en formato rack, lo cual permite almacenarlo fácilmente en armarios preparados para ello, además de contar con procesador Xeon y memoria suficiente para el servicio, sus dos interfaces de red a 1gb permitirán inicialmente tener suficiente ancho de banda para las peticiones de toda la organización. Otro aspecto importante es que el precio indicado en el presupuesto tiene un año de garantía por parte del fabricante, por lo que sería recomendable la valoración de compra de una extensión de garantía para alargar la vida de la inversión.



	Servidor Dell PowerEdge R230
Procesador	Intel® Xeon® E3-1220 v5 3.0GHz, 8M cache, 4C/4T, turbo (80W)
Memoria RAM	8GB 2666MT/s DDR4 ECC UDIMM
Disco Duro	1TB 7.2K RPM SATA 6
Interfaces Ethernet	2x LOM 1GbE Broadcom® BCM5720
Precio	841,34 euros por unidad

Software

Además del coste de las máquinas, debemos calcular el coste del software de los productos que utilicemos. Se debe tener en cuenta que los desarrolladores de software tienen diferentes tipos de licenciamiento en función del uso que se le vaya a dar al mismo; por ello, se debe leer y comprender las condiciones de la licencia de los productos que utilicemos. Afortunadamente, gracias a nuestra predisposición a la utilización de **software libre** y a que en este campo existen multitud de buenos productos con este licenciamiento, el coste del software utilizado en la infraestructura del sercivio será cero.

Se detalla a continuación el software destacado para llevar a cabo este proyecto:

- Ubuntu Server
- Apereo Cas
- Apache Tomcat
- HAProxy
- OpenLDAP
- Redis Server

Personal técnico

Solo queda detallar el coste de personal técnico implicado en el proyecto para la completa instalación y configuración de un servicio de autenticación en alta disponibilidad. Se estima el precio medio por hora de un Graduado en Informática subcontratado específicamente para esta tarea en 35 euros/hora.

Tarea	Horas de trabajo	Precio Hora	Total
Instalación de las cuatro máquinas	16	35	560
Parametrización de balanceador	10	35	350
Parametrización de nodos de CAS. Configuración standalone.	20	35	700
Parametrización del directorio de usuarios.	10	35	350
Parametrización de nodos de CAS. Configuración en alta disponibilidad	10	35	350
Ejecución de pruebas de rendimiento standalone	10	35	350
Ejecución de pruebas de rendimiento en alta disponibilidad	5	35	175
Ajuste de configuraciones finales.	10	35	350
Totales	91 horas		3185 euros

Coste Total

Podemos estimar que el proyecto podría llevarse a cabo por un total de 4867,68 euros.

Concepto	Unidades	Precio Unitario en euros
Servidores de CAS	2	841,34
Licencias software	-	0
Personal técnico en horas	91	3185
Total		**4867,68 euros**

4. Instalación e implantación de la solución

4.1 Instalación de las máquinas

Para la instalación de las máquinas se realizará la instalación de una maqueta en virtualbox y luego a partir de dicha maqueta se realizarán los clones necesarios de la misma para obtener las cuatro máquinas a usar en este proyecto.

Recurriremos a la página del proyecto Ubuntu para obtener la imagen ISO del sistema Ubuntu Server 18.04 x64. Una vez disponemos de la máquina maqueta y la imagen del sistema operativo a instalar, arrancaremos la instalación live que por medio de unos menús de texto se nos irá preguntando datos básicos de la instalación como nombre de usuario, nombre de la máquina, configuración de la interfaz de red y habilitación del servicio SSH.

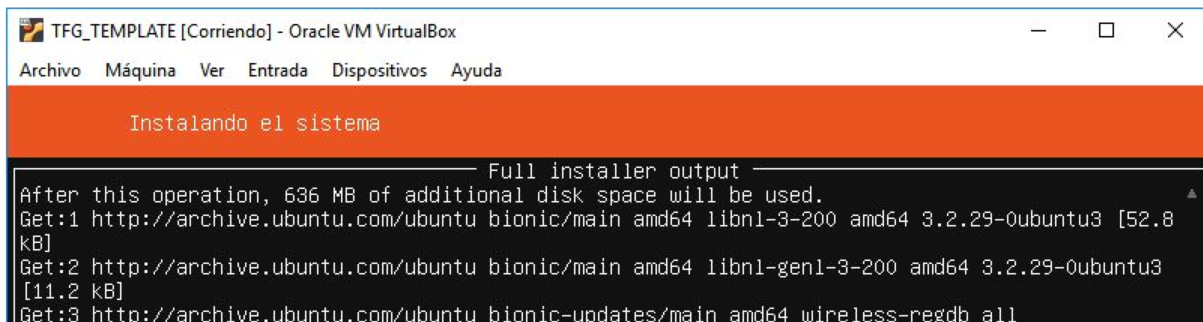


Figura 5: Instalación del Ubuntu Server para las máquinas en VirtualBox

Una vez finalizado el instalador, haremos un reinicio de la máquina. Antes de nada, definiremos dos interfaces de red a nivel de VirtualBox (una NAT y otra “solo Anfitrión” con modo promiscuo en “Permitir todo”). La primera(10.x.x.x) nos permitirá conectarnos a internet para poder realizar las instalaciones de software, la segunda(192.168.56.x) permitirá que podamos acceder desde el anfitrión a la máquina invitado y desde la máquina invitado a otras máquinas invitado.

Tras las añadir las interfaces de red, quedará por añadir a la configuración de red de Linux las IPs adecuadas y rutas necesarias para que la máquina encamine adecuadamente los paquetes por la red interna y externa. Una vez tenemos esto, tenemos una máquina plantilla “TFG_TEMPLATE” que nos permitirá tener una máquina básica Linux que luego podrá ser clonada a través de Virtualbox y generar así el conjunto de máquinas de la infraestructura.

Además, para facilitar las configuraciones tras el clonado de las máquinas, se define una plantilla de interfaces de red en /etc/netplan/1-interfaces.yaml; en función de la máquina final que se está personalizando se deberá descomentar la línea addresses correspondiente.

```
network:
   version: 2
   renderer: networkd
   ethernets:
       enp0s3:
           dhcp4: true
           routes:
             - to: 0.0.0.0/0
               via: 10.0.2.2
               metric: 0
       enp0s8:
           dhcp4: no
           #addresses: [192.168.56.101/24] #TFG_BAL
           #addresses: [192.168.56.111/24] #TFG_CAS1
           #addresses: [192.168.56.111/24] #TFG_CAS2
           #addresses: [192.168.56.121/24] #TFG_DATA
```

Para realizar adecuadamente el cambio de nombre de la máquina se debe ejecutar el siguiente conjunto de comandos:

```
#moises: Requiere de un parámetro "nombre_maquina"
hostnamectl set-hostname $nombre_maquina
sed -i 's/false/true/g' /etc/cloud/cloud.cfg

#Prepare /etc/hosts with nameservers
echo "192.168.56.101 bal.uoc.tfg  tfg_bal" >> /etc/hosts
echo "192.168.56.111 cas1.uoc.tfg tfg_cas1" >> /etc/hosts
echo "192.168.56.112 cas2.uoc.tfg tfg_cas2" >> /etc/hosts
echo "192.168.56.121 data.uoc.tfg tfg_data" >> /etc/hosts
```

```
Ubuntu 18.04.2 LTS TFG_TEMPLATE tty1

TFG_TEMPLATE login: moises
Password:
Last login: Mon Apr  8 16:57:39 UTC 2019 from 192.168.56.1 on pts/0
Welcome to Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-47-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Mon Apr  8 17:24:38 UTC 2019

  System load:  0.0               Processes:           94
  Usage of /:   19.7% of 19.56GB  Users logged in:     0
  Memory usage: 7%                IP address for enp0s3: 10.0.2.15
  Swap usage:   0%                IP address for enp0s8: 192.168.56.150
```

Figura 6: Maqueta de máquina virtual Ubuntu Server recien arrancada

TFG_TEMPLATE (INICIAL) [Corriendo] - Oracle VM VirtualBox

Archivo Máquina Ver Entrada Dispositivos Ayuda

```
moises@TFG_TEMPLATE:~$ route -n
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
0.0.0.0         192.168.56.1    0.0.0.0         UG    0      0        0 enp0s8
0.0.0.0         10.0.2.2        0.0.0.0         UG    100    0        0 enp0s3
10.0.2.0        0.0.0.0         255.255.255.0   U     0      0        0 enp0s3
10.0.2.2        0.0.0.0         255.255.255.255 UH    100    0        0 enp0s3
192.168.56.0    0.0.0.0         255.255.255.0   U     0      0        0 enp0s8
moises@TFG_TEMPLATE:~$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.0.2.15  netmask 255.255.255.0  broadcast 10.0.2.255
        inet6 fe80::a00:27ff:feb8:997b  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:b8:99:7b  txqueuelen 1000  (Ethernet)
        RX packets 2  bytes 1180 (1.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 16  bytes 1760 (1.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.56.150  netmask 255.255.255.0  broadcast 192.168.56.255
        inet6 fe80::a00:27ff:fe5f:f807  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:5f:f8:07  txqueuelen 1000  (Ethernet)
        RX packets 50  bytes 3462 (3.4 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1525  bytes 113293 (113.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        inet6 ::1  prefixlen 128  scopeid 0x10<host>
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 1707  bytes 135161 (135.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 1707  bytes 135161 (135.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

moises@TFG_TEMPLATE:~$
```

CTRL DERECHA

Figura 7: Configuración de red de la maqueta de máquina virtual Ubuntu Server

4.2 Parametrización del directorio de usuarios.

4.2.1 Instalación de OpenLdap.

Necesitamos instalar un LDAP que permita almacenar y validar los credenciales de nuestros usuarios. Para ello haremos lo siguiente:

```
$ sudo apt update
$ sudo apt -y install slapd ldap-utils
```

Tras el comando install comenzará la instalación del servicio de OpenLdap en el que se nos preguntará la contraseña de administrador. El propio instalador a partir de los datos recopilados de la configuración del servidor (dominio, nombre de máquina, etc) generará las estructuras imprescindibles necesarias para el funcionamiento.

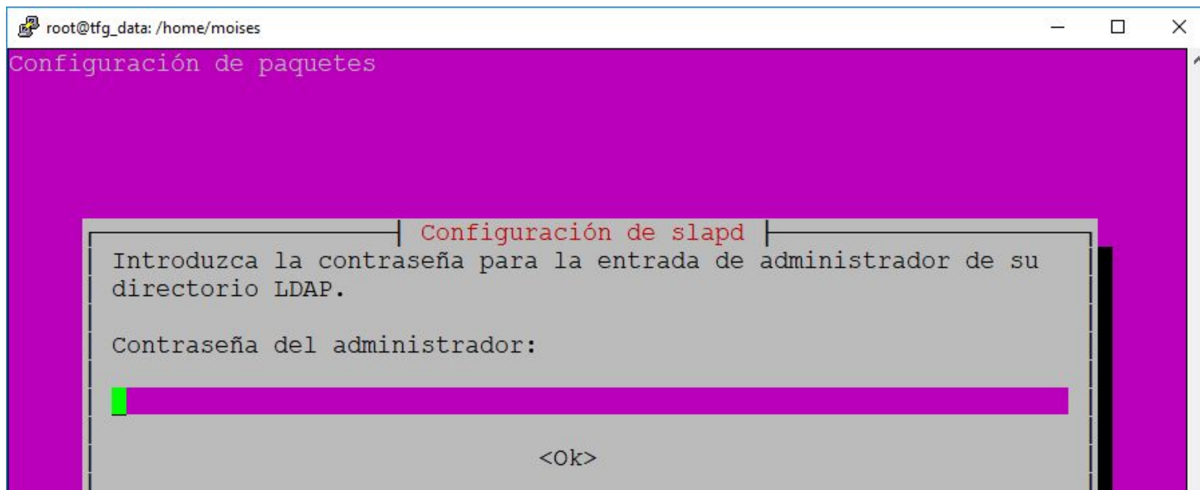


Figura 8: Proceso de instalación de Openldap. Petición de contraseña de administración.

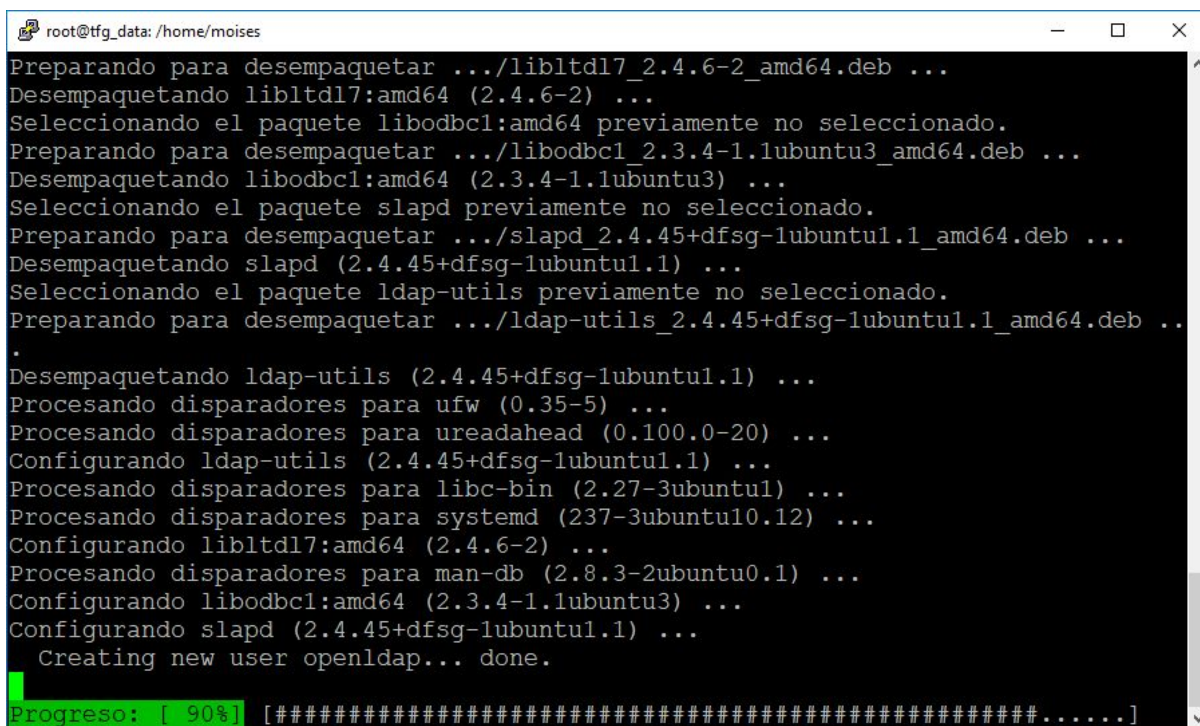


Figura 9: Proceso de instalación de Openldap.

Tras la finalización del proceso de instalación podemos aprovechar la utilidad de exportación de directorio *slapcat* para **examinar el dominio que ha creado la instalación**.

```
root@tfg_data: /home/moises
slapacl     slapauth    slapd       slapindex   slapschema  slattach
root@tfg_data:/home/moises# slapcat
dn: dc=uoc,dc=tfg
objectClass: top
objectClass: dcObject
objectClass: organization
o: uoc.tfg
dc: uoc
structuralObjectClass: organization
entryUUID: 70d7545a-f3af-1038-8867-1b73538793db
creatorsName: cn=admin,dc=uoc,dc=tfg
createTimestamp: 20190415094900Z
entryCSN: 20190415094900.877916Z#000000#000#000000
modifiersName: cn=admin,dc=uoc,dc=tfg
modifyTimestamp: 20190415094900Z

dn: cn=admin,dc=uoc,dc=tfg
objectClass: simpleSecurityObject
objectClass: organizationalRole
cn: admin
description: LDAP administrator
userPassword:: e1NTSEF9S3piZ2Q0TTQvMUpqK291VFp5UFdjZzVYYm13alFleWE=
structuralObjectClass: organizationalRole
entryUUID: 70d820b0-f3af-1038-8868-1b73538793db
creatorsName: cn=admin,dc=uoc,dc=tfg
createTimestamp: 20190415094900Z
entryCSN: 20190415094900.883526Z#000000#000#000000
modifiersName: cn=admin,dc=uoc,dc=tfg
modifyTimestamp: 20190415094900Z

root@tfg_data:/home/moises#
```

Figura 10: Dominio Openldap con la rama dc=uoc,dc=tfg y su administrador cn=admin

Lo siguiente será crear las ramas organizativas de usuarios y grupos a partir de un fichero LDIF. El fichero LDIF define cada una de las entradas o modificaciones que se van a realizar sobre el directorio de LDAP. Como en este caso se desean agregar elementos con el comando ldapadd, bastará con definir el *dn:* o domain name que definirá una ramificación lógica del directorio base "dc=uoc,dc=tfg". Esta ramificación se hace puramente por motivos organizativos, de manera que todos los usuarios y grupos estén separados del resto de elementos. Luego, insertamos los *objectclass* asociados a esa rama; que definen los atributos mínimos que deben tener los elementos que se inserten en ella, de manera que no puedan generarse entradas dentro de esa rama que no contengan dicha información. Finalmente se define el atributo *ou:* u organizational unit el cual viene a dar un nombre resumido del nombre del grupo.

```
root@tfg_data: /home/moises/ldap
dn: ou=Users,dc=uoc,dc=tfg
objectClass: organizationalUnit
ou: Users

dn: ou=Groups,dc=uoc,dc=tfg
objectClass: organizationalUnit
ou: Groups
```

Figura 11: Fichero en formato LDIF con definición de las ramas de grupos y usuarios.

Para aplicar las modificaciones, en este caso nuevas entradas, de un fichero ldif sobre nuestro directorio se ejecutará el comando ldapadd identificandonos como administrador del ldap.

```
$ sudo ldapadd -x -D  cn=admin,dc=uoc,dc=tfg -W -f ramas.ldif
Enter LDAP Password:
adding new entry "ou=Users,dc=uoc,dc=tfg"

adding new entry "ou=Groups,dc=uoc,dc=tfg"
```

4.2.2 Aprovisionamiento de usuarios

Una vez contamos con la instalación y las estructuras lógicas mínimas de un dominio, pasamos a generar los usuarios que serán autenticados durante nuestra prueba.

Para ello se ha creado un shell script que genere una salida en formato LDIF con la inserción de 1000 usuarios de prueba. El script creará usuarios con el nombre “usuarioX” y la contraseña “passX” siendo X un número entre 1 y 1000. Además de lo anteriormente nombrado, destacaremos que el nombre del usuario se especifica en el atributo *cn:* o common name y su apellido viene dado por el atributo *sn:* o surname.

```
#!/bin/bash
for i in $(seq 1 1000); do
        echo "dn: uid=usuario$i,ou=Users,dc=uoc,dc=tfg"
        echo "objectClass: inetOrgPerson"
        echo "objectClass: posixAccount"
        echo "objectClass: shadowAccount"
        echo "cn: usuario$i"
        echo "sn: tfg"
        echo "userPassword: pass$i"
        echo "loginShell: /bin/bash"
        echo "homeDirectory: /home/usuario$i"
        echo ""
done;
                                                  1,11          Todo
```

Figura 12: Shell script de generación de usuarios en formato LDIF

Tras ejecutar el shell script, obtendremos una salida que almacenaremos en usuarios_generados.ldif. A continuación, ejecutamos *ldapadd* tal y como realizamos en el punto previo para generar las ramas de Users y Groups.

```
$ sudo ldapadd -x -D  cn=admin,dc=uoc,dc=tfg -W -f usuarios_generados.ldif
Enter LDAP Password:
adding new entry "uid=usuario1,ou=Users,dc=uoc,dc=tfg"

adding new entry "uid=usuario2,ou=Users,dc=uoc,dc=tfg"

adding new entry "uid=usuario3,ou=Users,dc=uoc,dc=tfg"
```

```
adding new entry "uid=usuario4,ou=Users,dc=uoc,dc=tfg"
..
..
adding new entry "uid=usuario1000,ou=Users,dc=uoc,dc=tfg"
```

Al finalizar, dispondremos de un **servicio ldap completamente funcional** en la máquina data.uoc.tfg:389

4.3 Parametrización de nodos de CAS. Configuración standalone.

4.3.1 Instalación y configuración de componentes

Para el despliegue de un nodo de CAS se requiere tener de manera obligatoria el siguiente software[4.3.1.1]:

- Java JDK 8
- Git
- Repositorio Git de CAS de manera local en la máquina.

Una vez descargado el repositorio del Apereo CAS, debemos colocarnos en la rama git de la versión de CAS que se utilizará en este trabajo de fin de grado como se ve a continuación.

```
$ sudo apt install openjdk-8-jre-headless git
$ git clone https://github.com/apereo/cas-overlay-template
$ cd cas-overlay-template
$ git branch -a
* master
  remotes/origin/4.1
  remotes/origin/4.2
  remotes/origin/5.0.x
  remotes/origin/5.1
  remotes/origin/5.2
  remotes/origin/5.3
  remotes/origin/6.0
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
$ git checkout 5.3
$ ls
build.cmd  etc         maven  mvnw.bat  README.md
build.sh   LICENSE.txt  mvnw   pom.xml
```

Cuando tengamos descargado y estemos en la rama correcta del proyecto de CAS, disponemos de todo lo necesario para construir y ejecutar CAS.

El proyecto que vamos a desplegar de Apareo Cas no se entrega como un binario atómico que provee muchas funcionalidades, sino que el implantador decidirá qué funcionalidades se incorporarán al servicio a través de un conjunto de "capas" denominadas overlays. El fichero en el que se indican estas especificaciones es el **pom.xml** que se encuentra la definición xml de la construcción del proyecto. Por otro lado debemos mostrar tambien atencion a los ficheros del directorio **etc/cas/config/ .** Dentro de este directorio encontraremos **cas.properties** y **log4j2.xml** siendo el primero el responsable de definir las propiedades asociadas a las funcionalidades de CAS y el segundo un fichero específico de la configuración de logs.

Respecto a cas.properties encontraremos las definiciones básicas que personalizamos con las URL de nuestro servicio.

```
cas.server.name: https://cas1.uoc.tfg:8443
cas.server.prefix: https://cas1.uoc.tfg:8443/cas
cas.adminPagesSecurity.ip=127\.0\.0\.1
logging.config: file:/etc/cas/config/log4j2.xml
```

Volviendo al pom.xml, primero observamos que a partir de la etiqueta *cas.version* se identifica la versión exacta de CAS con la que se trabajará y con *app.server* el servidor de aplicaciones embebido con el que se desplegará el producto.

```
<properties>
    <cas.version>5.3.9</cas.version>
    <springboot.version>1.5.18.RELEASE</springboot.version>
    <!-- app.server could be -jetty, -undertow, -tomcat, or blank if you plan to provide appserver -->
    <app.server>-tomcat</app.server>

    <mainClassName>org.springframework.boot.loader.WarLauncher</mainClassName>
    <isExecutable>false</isExecutable>
    <manifestFileToUse>${project.build.directory}/war/work/org.apereo.cas/cas-server-webapp...

    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Figura 13: Definición de versión de CAS y servidor de aplicaciones en pom.xml

El sistema de capas, tal y como se indicaba, permite añadir funcionalidades de CAS a través del fichero pom.xml. Vamos a especificar las dependencias de nuestro proyecto para que el constructor basado en maven añada las librerías necesarias para el correcto funcionamiento de nuestro producto final.

Para la consecución de nuestro objetivo, se requiere la validación de usuarios sobre un directorio de usuarios el cual elegimos que fuese Ldap, por ello, se debe añadir dentro del apartado

de dependencias una nueva *dependency* y anexar al fichero cas.properties existente las configuraciones específicas de nuestro servicio ldap (cas.authn.ldap[0]).

```
<dependencies>
    <dependency>
        <groupId>org.apereo.cas</groupId>
        <artifactId>cas-server-webapp${app.server}</artifactId>
        <version>${cas.version}</version>
        <type>war</type>
        <scope>runtime</scope>
    </dependency>
    <!--
    ...Additional dependencies may be placed here...
    -->
</dependencies>
```

Figura 14: Definición de dependencias de CAS en pom.xml

```
<dependency>
        <groupId>org.apereo.cas</groupId>
        <artifactId>cas-server-support-ldap</artifactId>
        <version>${cas.version}</version>
</dependency>
```

```
cas.authn.ldap[0].type=AUTHENTICATED
cas.authn.ldap[0].ldapUrl=ldap://data.uoc.tfg
cas.authn.ldap[0].useSsl=false
cas.authn.ldap[0].useStartTls=false
cas.authn.ldap[0].baseDn=ou=Users,dc=uoc,dc=tfg
cas.authn.ldap[0].searchFilter=uid={user}
cas.authn.ldap[0].bindDn=cn=admin,dc=uoc,dc=tfg
cas.authn.ldap[0].bindCredential=1234
cas.authn.ldap[0].principalAttributeList=cn,sn,homeDirectory
#Empty parameter removes development users
cas.authn.accept.users=
```

4.3.2 Configuración de componentes extra

Justo como se informó; el funcionamiento de CAS permite que las aplicaciones ,de manera centralizada, autentiquen usuarios sin tener los credenciales del mismo. Debido a que no se desea que cualquier aplicación de terceros pueda autenticar a los usuarios de una organización de una manera malintencionada recuperando así los atributos que el servicio de CAS provea, se necesita añadir un componente extra dónde se autoricen las aplicaciones que pueden consumir el servicio de CAS. Aunque existen multitud maneras de gestionar las aplicaciones dadas de alta en CAS, y en un

entorno profesional lo más útil sería tener una interfaz web [4.3.2.1], debido a la necesidad de simplificar nuestro servicio para la ejecución de este trabajo de fin de grado, se optará por la especificación de las aplicaciones autorizadas a través de ficheros estáticos JSON[4.3.2.2].

Para ello, añadiremos la dependencia correspondiente en el pom.xml:

```
<dependency>
      <groupId>org.apereo.cas</groupId>

<artifactId>cas-server-support-json-service-registry</artifactId>
      <version>${cas.version}</version>
</dependency>
```

Generamos un fichero en la ruta /etc/cas/services/DemoCAS-100.json con lo siguiente:

```
{
 "@class" : "org.apereo.cas.services.RegexRegisteredService",
 "serviceId" : "^(https|http)://.*",
 "name" : "DemoCAS",
 "id" : 100,
  "description" : "Autoriza todas las URI http o https. Trabajo fin de grado
Moisés Rguez.",
 "evaluationOrder" : 10000
 "attributeReleasePolicy" : {
   "@class"   "org.apereo.cas.services.ReturnAllAttributeReleasePolicy"
 }
}
```

Por último, esta dependencia requiere de una propiedad que debe ser anexada a cas.properties indicando el lugar dónde se localizarán los json.

```
cas.serviceRegistry.json.location=file:/etc/cas/services
```

4.3.3 Configuración de aplicación DemoCAS

Para poder demostrar el funcionamiento del servicio necesitamos de una aplicación auxiliar que nos permita simular el consumo de los servicios de CAS cuando el usuario solicita acceder a una información privada.

Para ello hemos buscado un ejemplo simple en java desarrollado por terceros [4.3.3.1] y que a su vez basandose en el cliente de java de Apereo permite cerrar el flujo de autenticación de usuarios con un servicio CAS.

Al igual que otras aplicaciones, necesitamos clonar el repositorio, esta vez en la máquina tfg_data, luego se sustituirán las URI relativas a dónde se encuentra el servidor de CAS y el propio servidor y contexto de la aplicación, por último se copiará el almacén de certificados *thekeystore* de prueba generado posteriormente en el siguiente punto. Finalmente llamar con maven a la ejecución de la aplicación la cual usará el servidor jetty. En nuestro caso, el servidor de CAS está en https://cas1.uoc.tfg:443/cas y la aplicación la ejecutamos en la máquina auxiliar del proyecto bajo la URI http://data.uoc.tfg:9080/DemoCAS/

```
$ git clone https://github.com/cas-projects/cas-sample-java-webapp.git
$ cd cas-sample-java-webapp
$find ./ -type f |
 xargs sed -i 's/https:\/\/mmoayyed.unicon.net:9443/http:\/\/data.uoc.tfg:9080/g'
$find ./ -type f |
 xargs sed -i 's/https:\/\/mmoayyed.unicon.net:8443/https:\/\/cas1.uoc.tfg:443/g'
$find ./ -type f | xargs sed -i 's/sample/DemoCAS/g'
$ scp cas1.uoc.tfg:/etc/cas/thekeystore /etc/cas/jetty/thekeystore #Esto se realiza tras cap 4.3.4
$ mvn clean package jetty:run-forked
```

4.3.4 Empaquetar y ejecutar el servicio de CAS

Disponemos ya de un servicio de CAS preconfigurado y revisado, un ldap con el directorio de usuarios y una aplicación que consume dichos usuarios, estamos en disposición de empaquetar ejecutar el servicio de CAS.

El procedimiento de capas de Apereo CAS, a través del script de build.sh facilita la construcción y ejecución del paquete empaquetado gracias a un servidor tomcat embebido con el propio paquete. Lo primero que haremos es revisar que las dependencias y propiedades añadidas al proyecto generan un .war que permita posteriormente ser ejecutado.

```
$ cd cas-overlay-template
$ ./build.sh package
```

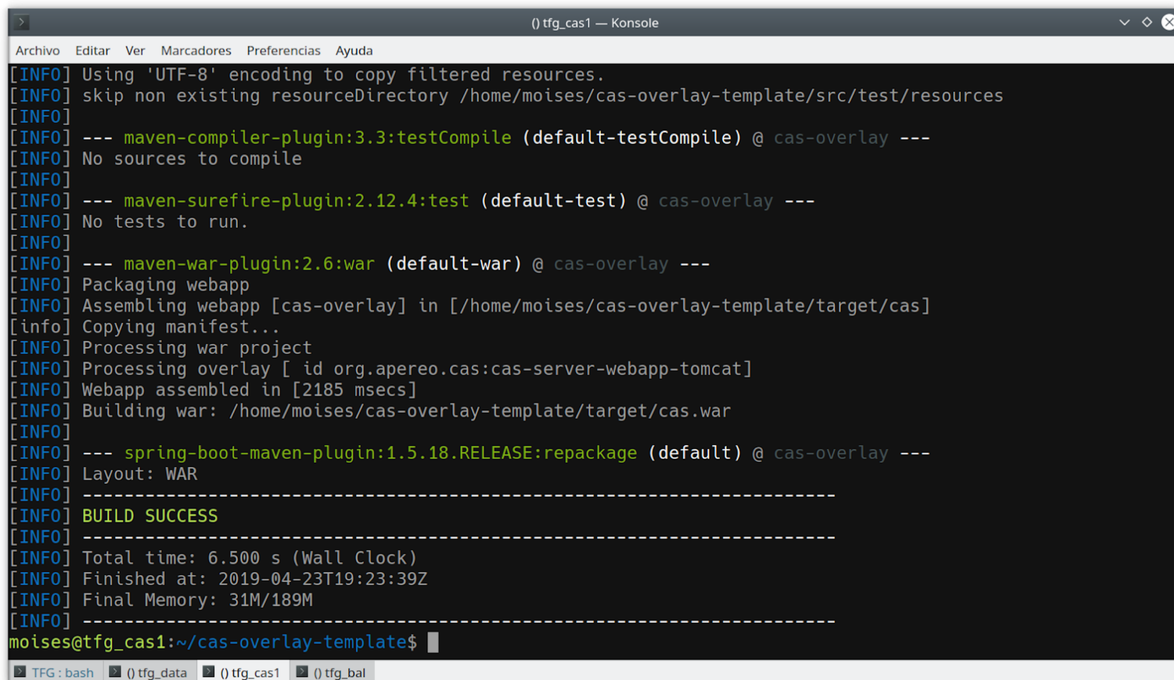


Figura 15: Proceso de empaquetado del fichero cas.war

Tras comprobar que todo se ha generado correctamente vamos probar el fichero cas.war sobre el propio servidor de servlets que lleva embebido en el despliegue.

Antes de nada, debemos tener correctamente mapeados los dos directorios necesarios para que arranque la aplicación y debe existir el almacén de claves que se utilizará para la capa SSL . Por un lado se necesita encontrar los ficheros de configuración anteriormente definidos (cas.properties y log4j2.xml) en el directorio **/etc/cas/config**. Para este trabajo enlazaremos los ficheros definidos en el proyecto git local de CAS. Se hace notar que para un despliegue corporativo, lo recomendable es contar con una herramienta de automatización de despliegues como CHEF que actualice los ficheros desde el propio repositorio remoto de git.

Además de los ficheros de configuración, el usuario que ejecuta el proceso de CAS, el cual realmente es un proceso de tomcat embebido con el cas.war desplegado, necesita acceso a una carpeta donde almacenar los logs de la aplicación y de registro de acceso. Por último se generará un certificado autofirmado con el nombre *thekeystore* el cual es el que viene por defecto en la configuración del paquete CAS[4.3.4.1].

```
$ sudo mkdir /etc/cas/
$ sudo ln -s ~/cas-overlay-templates/etc/cas/config/ /etc/cas/
$ sudo mkdir /etc/cas/logs
$ sudo chown moises:moises /etc/cas/logs
$sudo keytool -genkey -alias cas -keyalg RSA -validity 999 -keystore /etc/cas/thekeystore -ext san=dns:cas1.uoc.tfg
$ ./build.sh run
```

El comando “build.sh run” reempaquetará la aplicación de CAS y la ejecutará. Si todo finaliza bien, podremos acceder en nuestro navegador directamente a la URL del nodo **https://cas1.uoc.tfg:8443/cas** y hacer una prueba de validación. Una vez validados correctamente, somos capaces de comprobar que CAS ha devuelto los parámetros indicados en la sección ldap[0].principalAttributeList de cas.properties.

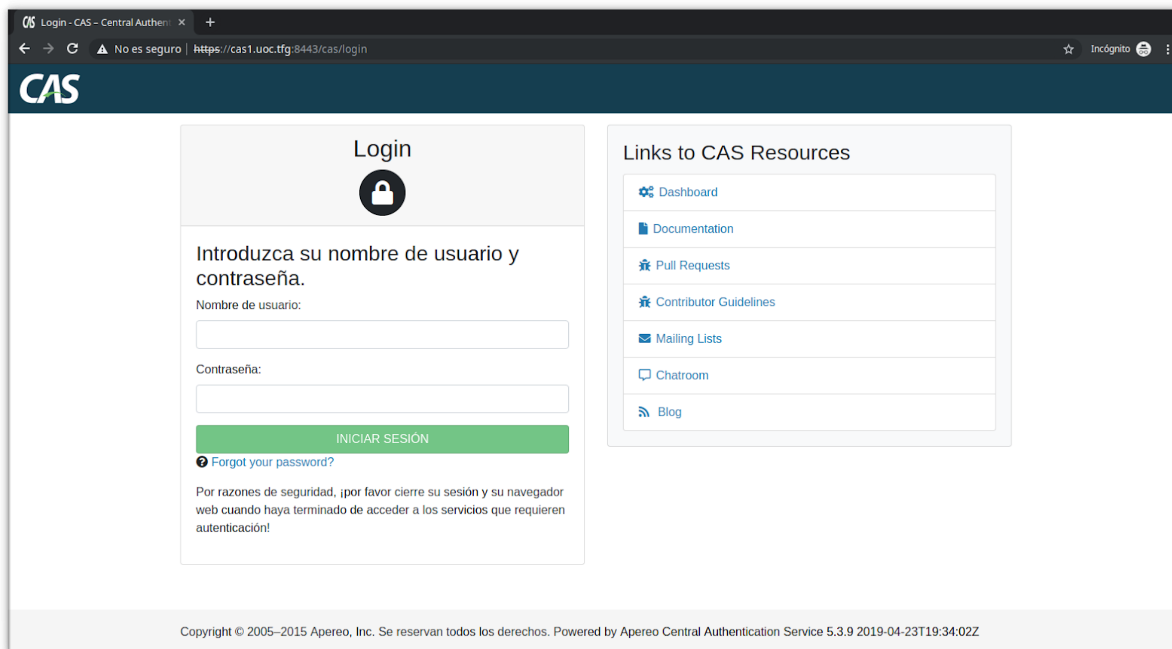


Figura 16: Acceso a validación de CAS en nodo único

Figura 17: Comprobación de parámetros devueltos por CAS tras validación

Se ha comprobado que se dispone de un servicio funcional de CAS funcionando en modo standalone, es decir, sin compartición de sesiones activas entre otros posibles nodos.

4.3.5 Instalación de Nginx como frontal de cada nodo de CAS.

Debemos tener en cuenta que en este punto se responderán todas las peticiones realizadas al puerto 8443. Para mejorar la seguridad del sistema, es una buena práctica instalar un software que haga las funciones de frontal de servicio, para que este sea llamado directamente por el servicio de balanceo. Si bien se descartó **Nginx** para realizar el balanceo de carga debido a que sus amplias funcionalidades y diseño original no fueron pensadas para realizar dicha labor; este software dispone de características que lo convierten en un excelente frontal de servicio con gestión de seguridad y proxy inverso.

Realizaremos una instalación de nginx con ssl-cert que nos proveerá de los certificados de prueba necesarios para poder realizar conexiones a nuestro servidor por el puerto ssl (https) y por último importamos el nuevo certificado de pruebas instalado en el almacén por defecto de java que utilizará CAS para hacer las consultas[4.3.5.1].

```
$ sudo apt install nginx ssl-cert
$ sudo keytool -import -keystore $JAVA_HOME/jre/lib/security/cacerts
        -file /etc/ssl/certs/ssl-cert-snakeoil.pem -alias nginx_tfg
# Copiar el fichero cacerts al resto de máquinas con las que ya se esté trabajando
```

Tras la instalación, editaremos el fichero de configuración del sitio por defecto de nginx localizado en */etc/nginx/sites-enabled/default*. En esta configuración especificaremos el puerto de escucha del servidor, la configuración de los certificados ssl (snakeoil.conf) y la parte más destacable que es "*location /cas*". Lo que aquí especificamos es que todas las peticiones que lleguen a esta máquina por el puerto 443 y el contexto /cas serán redireccionadas a la máquina local en el puerto 8443 y mismo contexto. Por otro lado y debido a que estamos trabajando con un servidor proxy que enmascara nativamente la IP de quien realiza la petición, unido a que en todo servicio de internet se requiere tener identificada a la IP real del cliente que realiza la petición, se añaden las directivas **proxy_set_header** en el servidor, haciendo que para cada petición de proxy inverso se envíe la IP real del cliente al servidor de CAS.

```
##
# Default server configuration
#
server {
	# SSL configuration
	listen 443 ssl default_server;
	listen [::]:443 ssl default_server;
	# Self signed certs generated by the ssl-cert package
	include snippets/snakeoil.conf;
	root /var/www/html;
```

```
        index index.html index.htm index.nginx-debian.html;
        server_name _;

        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;

        location /cas {
                proxy_pass https://localhost:8443/cas/;
        }
        location / {
                try_files $uri $uri/ =404;
        }
}
```

Realizando estas configuraciones estamos en disposición de reiniciar el servicio de Nginx para realizar de nuevo una prueba del servicio de CAS, pero esta vez sobre el puerto 443 de la máquina (https://cas1.uoc.tfg/cas).

```
$ sudo service nginx restart
```

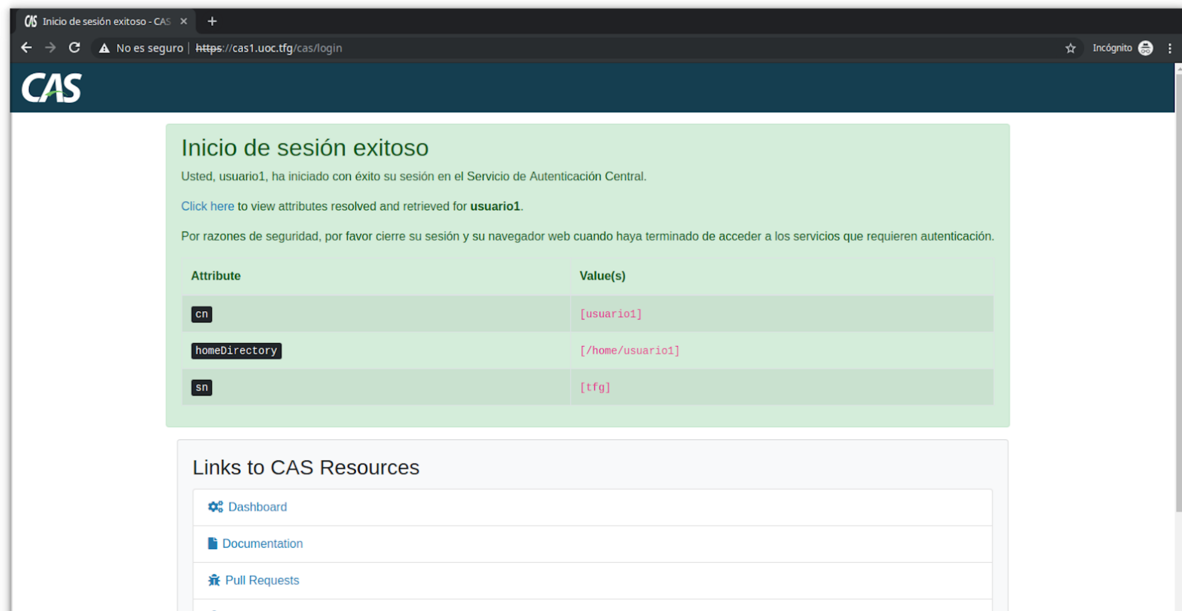


Figura 18: Parámetros devueltos por CAS tras validación con NGINX como proxy inverso.

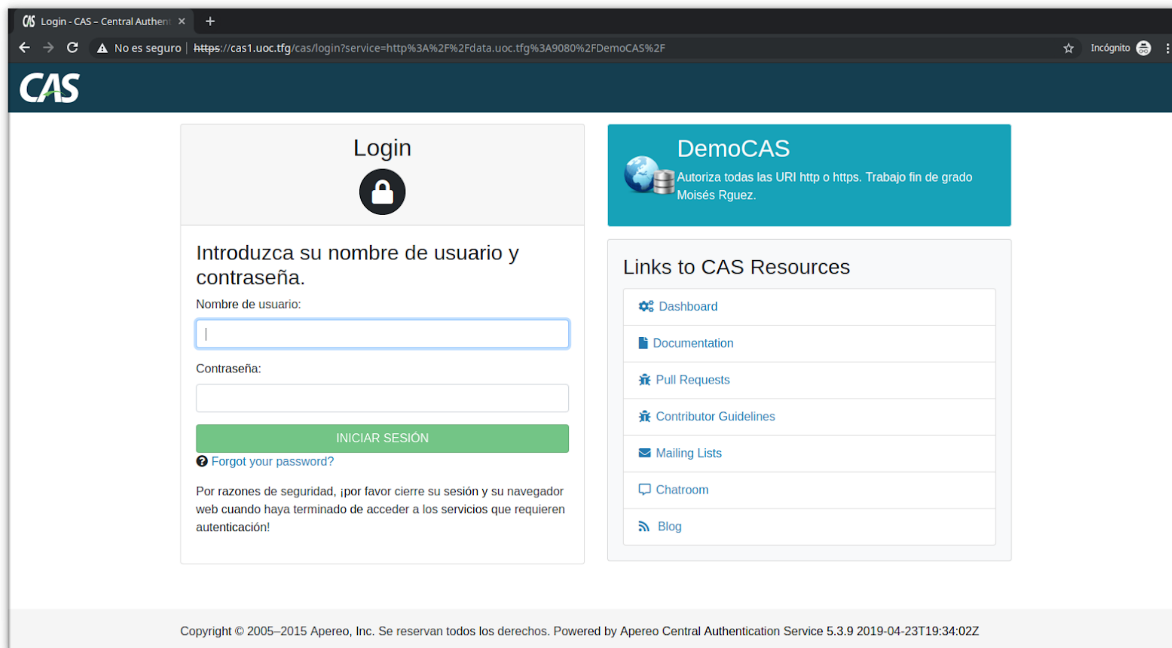


Figura 19: Acceso a validación de CAS en nodo único desde aplicación DemoCAS

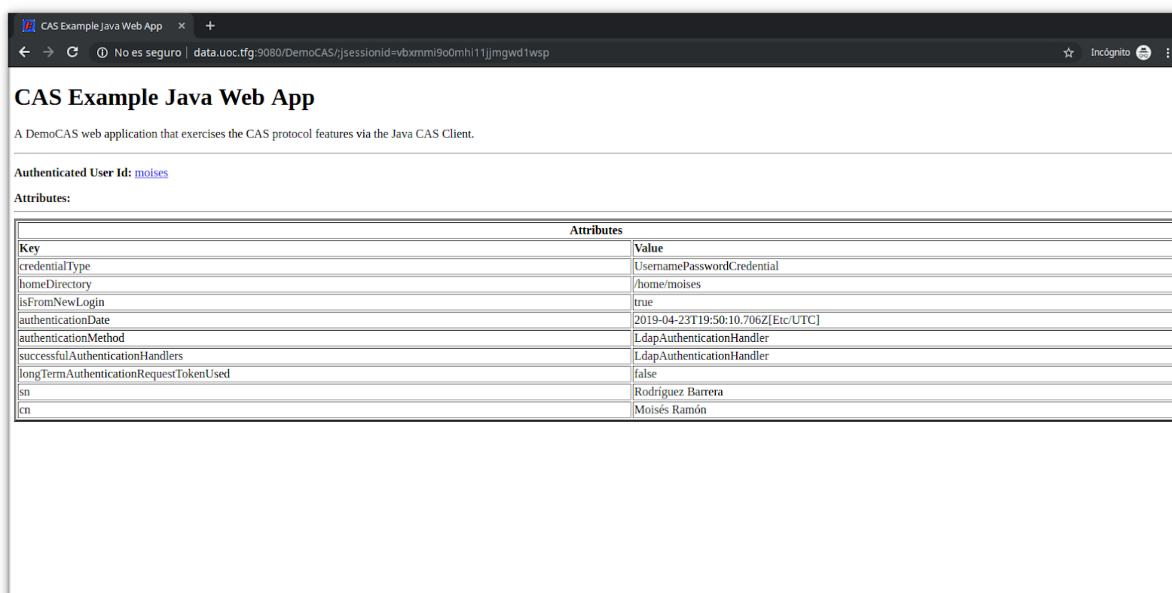


Figura 20: Acceso a DemoCAS trás autenticación del usuario

4.4 Parametrización de balanceador

Nos disponemos a instalar el nodo balanceador de carga. Para ello instalaremos HaProxy a partir de los repositorios oficiales de Ubuntu de la siguiente manera:

```
$ sudo apt update
$ sudo apt install haproxy
$ haproxy -v
HA-Proxy version 1.8.8-1ubuntu0.4 2019/01/24
Copyright 2000-20
```

El archivo de configuración de Haproxy se encuentra definido en */etc/haproxy/haproxy.cfg*.

```
$ sudo vi /etc/haproxy/haproxy.cfg
```

4.4.1 Configuraciones generales

En la parte inicial del fichero de configuración de HAProxy encontraremos siempre dos secciones bien diferenciadas. La primera es “global”; en este apartado se pueden definir configuraciones generales como podría ser la *facility* y el *level* para la especificación de los logs (para aumentar el rendimiento del servicio, con haproxy no es posible escribir los logs sobre fichero), el número máximo de conexiones permitido, el número de procesos que arrancará el demonio del servicio o aspectos de la negociación SSL. En la segunda sección del fichero encontramos el apartado “defaults” en él se definen propiedades que afectarán al resto de apartados definidos a continuación.

```
global
       log /dev/log    local0
       log /dev/log    local1 notice
       chroot /var/lib/haproxy
       stats socket /run/haproxy/admin.sock mode 660 level admin
         \ expose-fd listeners
       stats timeout 30s
       user haproxy
       group haproxy
       daemon

       #SSL
       ca-base /etc/ssl/certs
       crt-base /etc/ssl/private
       ssl-default-bind-ciphers ECDH+AESGCM:DH+AESGCM:ECDH+AES256:
         \ DH+AES256:ECDH+AES128:DH+AES:RSA+AESGCM:RSA+AES:!aNULL:
         \ !MD5:!DSS
```

```
	ssl-default-bind-options no-sslv3


defaults
	log	global
	mode	http
	option	httplog
	option	dontlognull
	timeout connect 5000
	timeout client  50000
	timeout server  50000
	errorfile 400 /etc/haproxy/errors/400.http
	errorfile 403 /etc/haproxy/errors/403.http
	errorfile 408 /etc/haproxy/errors/408.http
	errorfile 500 /etc/haproxy/errors/500.http
	errorfile 502 /etc/haproxy/errors/502.http
	errorfile 503 /etc/haproxy/errors/503.http
	errorfile 504 /etc/haproxy/errors/504.http
```

4.4.2 Añadir un servicio de frontend

En el mismo fichero procedemos a definir de manera muy simple la interfaz que atenderá las peticiones de entrada al servicio balanceado llamada "frontend". Con esta configuración, se arranca en el puerto 443 de la máquina un servicio escuchando en todas sus interfaces de red. Todas las peticiones direccionadas a este puerto serán redirigidas a los backend de CAS que hemos definidos en otra sección posterior como cas_servers.

```
frontend haproxy_frontend
  bind *:443 ssl crt /etc/ssl/private/tfg_uoc.pem
  mode http
  default_backend cas_servers
```

4.4.3 Añadir los servicios de backend

Tras la definición de la capa de frontend se procede a detallar el backend de la arquitectura. En este apartado se indica con *server* el servidor o servidores destino a las peticiones; se definirá un *server* por cada nodo de CAS que tengamos. Además se configura una directiva muy importante **option forwardfor** que debido a que haproxy al balancear actúa como proxy inverso, se especifica que debe reenviar la IP real del cliente al servidor de backend. Otra directiva importante en la sección de backend es la opción que comprueba el estado del servicio en el backend comúnmente llamada *health check*. Añadiendo ***option httpchk HEAD /cas/*** se establece la comprobación de los nodos de backend realizando un HEAD de la página /cas. Si el check recibe un código HTTP 2xx (correcto) o 3xx (redirección) el servicio entiende que todo funciona como debería; si por el contrario se recibe un código 4xx(error de cliente) o 5xx(error de servidor) el nodo de backend se marca

como erróneo y no se envían peticiones al mismo hasta que vuelva a cambiar su estado a correcto. Por último y debido a que un servicio de CAS en alta disponibilidad requiere de persistencia de conección a la hora de que un cliente se autentique, se añade **cookie SERVERID_tfg** para que se genere por cada cliente una cookie de sesión que asigne las peticiones de ese cliente a ese servidor.

```
backend cas_servers
  mode http
  stats enable
  balance roundrobin
  #Send client IP to backends
  option forwardfor except 127.0.0.1
  #Inserted cookie for persistent connection
  cookie SERVERID_tfg insert indirect nocache
  #Check backend status waiting 2xx 3xx response code
  option httpchk HEAD /cas/
  server tfg_cas1 192.168.56.111:443 check cookie tfg_cas1 ssl verify none
  server tfg_cas2 192.168.56.112:443 check cookie tfg_cas2 ssl verify none
```

4.4.4 Habilitar las estadísticas de HaProxy

Aunque no es requisito indispensable, para cualquier administrador es interesante disponer de una interfaz web con las estadísticas de los servicios que están definidos en el balanceador. Añadiendo una interfaz de estadísticas controlaremos de manera visual e intuitiva nuestro servicio accediendo a http://bal.uoc.tfg/stats

```
listen stats
  bind *:80
  mode http
  stats enable
  stats uri /stats
  stats realm HAProxy\ Statistics
  stats auth moises:1234
```

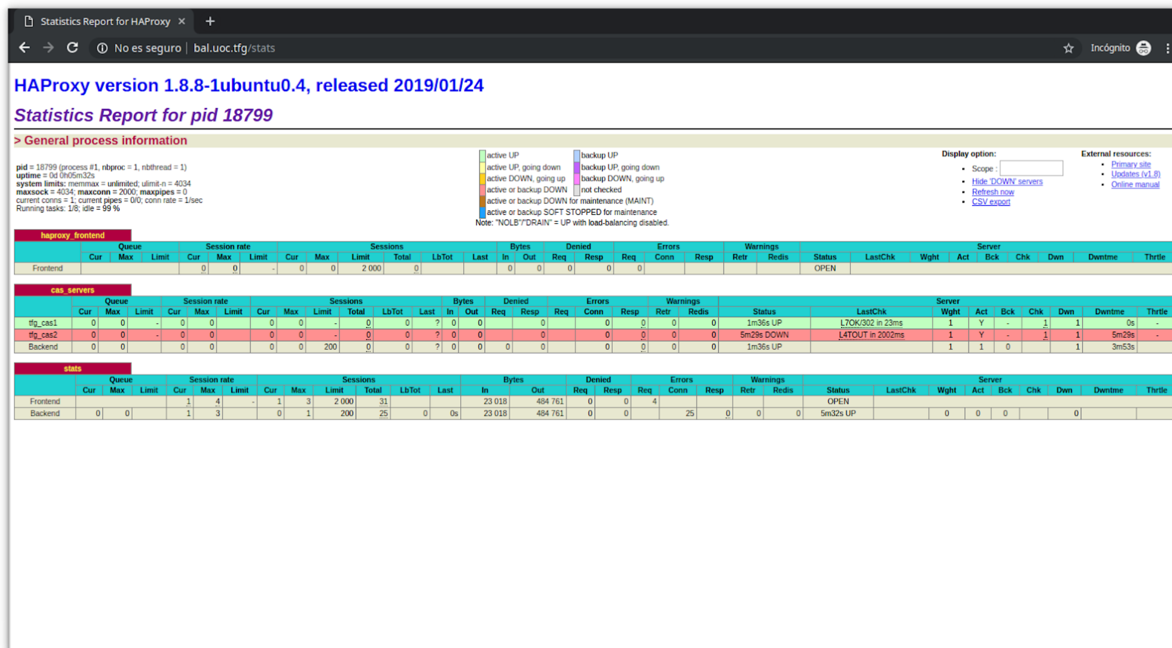


Figura 21: Monitor de estadísticas de HAProxy con nodo tfg_cas1 activo y tfg_cas2 con error

4.4.5 El resultado de la configuración

Tras añadir todas las configuraciones anteriores a nuestro balanceador el archivo /etc/haproxy/haproxy.cfg debería quedar de la siguiente manera.

```
global
        log /dev/log    local0
        log /dev/log    local1 notice
        chroot /var/lib/haproxy
        stats socket /run/haproxy/admin.sock mode 660 level admin expose-fd listeners
        stats timeout 30s
        user haproxy
        group haproxy
        daemon

        #SSL
        ca-base /etc/ssl/certs
        crt-base /etc/ssl/private
        ssl-default-bind-ciphers ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256:
                                 ECDH+AES128:DH+AES:RSA+AESGCM:RSA+AES:!aNULL:
                                 !MD5:!DSS
        ssl-default-bind-options no-sslv3

defaults
        log     global
        mode    http
        option  httplog
        option  dontlognull
        timeout connect 5000
        timeout client  50000
        timeout server  50000
        errorfile 400 /etc/haproxy/errors/400.http
        errorfile 403 /etc/haproxy/errors/403.http
        errorfile 408 /etc/haproxy/errors/408.http
        errorfile 500 /etc/haproxy/errors/500.http
        errorfile 502 /etc/haproxy/errors/502.http
```

```
      errorfile 503 /etc/haproxy/errors/503.http
      errorfile 504 /etc/haproxy/errors/504.http


frontend haproxy_frontend
  bind *:443 ssl crt /etc/ssl/private/tfg_uoc.pem
  mode http
  default_backend cas_servers

backend cas_servers
  mode http
  stats enable
  balance roundrobin
  #Send client IP to backends
  option forwardfor except 127.0.0.1
  #Inserted cookie for persistent connection
  cookie SERVERID_tfg insert indirect nocache
  #Check backend status waiting 2xx 3xx response code
  option httpchk HEAD /cas/
  server tfg_cas1 192.168.56.111:443 check cookie tfg_cas1 ssl verify none
  server tfg_cas2 192.168.56.112:443 check cookie tfg_cas2 ssl verify none

listen stats
  bind *:80
  mode http
  stats enable
  stats uri /stats
  stats realm HAProxy\ Statistics
  stats auth moises:1234
```

Teniendo esta configuración establecida, lo primero que haremos será comprobar que la sintaxis del ficheros es la adecuada, para luego, si todo está correcto reiniciar el servicio de HAProxy y verificar que que todo funciona como debería.

```
$ sudo haproxy -c -f /etc/haproxy/haproxy.cfg
$ sudo service haproxy restart
```

Procedemos ahora a verificar que podemos acceder al CAS a través del balanceador accediendo desde nuestro navegador a la siguiente dirección https://bal.uoc.tfg/cas

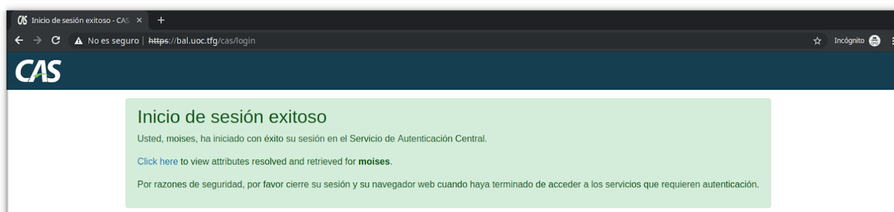


Figura 22: Comprobación desde un navegador de acceso al servicio a través del balanceador

Otra prueba que validará el correcto funcionamiento del servicio balanceado será comprobar que los **health check** se comportan de la manera que hemos indicado. Teniendo el Nginx y el servicio de CAS activos, el nodo tfg_cas1 de backend debería estar activo (color verde), pero desde que uno de los anteriores esté no disponible, el nodo backend debería de causar un error y dejar de dar servicio. Se adjuntan capturas de pantalla con el nodo tfg_cas1 fuera de balanceo debido a que el Nginx no está disponible y una segunda pantalla debido a que el proceso java de CAS no está disponible.

cas_servers	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings			
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
tfg_cas1	0	0	-	0	32		0	12	-	53	53	4m42s	25 467	3 094 217		0		0	0	0	0	5s DOWN	L7STS/502 in 4ms

Figura 23: HAProxy stats con tfg_cas1 caído. No existe proceso java ejecutando CAS

cas_servers	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings			
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
tfg_cas1	0	0	-	0	32		0	12	-	53	53	7m23s	25 467	3 094 217		0		0	0	0	0	2m46s DOWN	L4CON in 0ms

Figura 24: HAProxy stats con tfg_cas1 caído.No existe nginx corriendo en la máquina

Podemos apreciar como para los dos casos más obvios de error el HAProxy se comporta como esperábamos según el health check configurado. Como dato interesante cabría resaltar que el campo ***LastChk* muestra diferentes valores para cada tipo de error**; esto es totalmente lógico debido a que en un caso el error detectado fue en *capa 7* del modelo de referencia OSI o capa de aplicación debido a que no existía el proceso java ejecutando CAS y en el segundo caso, el problema estaba en la *capa 4* o de transporte no pudiéndose establecer conexión TCP con el Nginx. **En todas estas casuísticas y en un entorno en alta disponibilidad, existiendo otro nodo de backend activo, el servicio de CAS continuaría funcionando correctamente.**

4.5 Parametrización de nodos de CAS con alta disponibilidad

Para poder montar el servicio en alta disponibilidad, se deberá configurar un segundo nodo de CAS con idénticas configuraciones al nodo standalone, para ello podemos aprovechar la función de clonado de Virtualbox o montar y personalizar una segunda máquina de CAS realizando los mismos pasos que en los apartados adecuados anteriores con la excepción de especificarle un nombre de máquina e IP diferentes.

Una vez disponemos de una segunda máquina completamente funcional, podemos ejecutar el servicio de CAS y comprobar que la máquina funciona correctamente llamando al servicio en https://cas2.uoc.tfg:443. Posteriormente, podemos probar a entrar al servicios de CAS a través de https://bal.uoc.tfg:443, recordar que ya introdujimos la IP del servidor tfg_cas2 como backend del servicio en el paso anterior. Al hacer la prueba y desde el desconocimiento un administrador inexperto, se podría pensar que el servicio ya se encuentra en alta disponibilidad debido a que con los dos nodos levantados, el servicio funciona a la perfección. Es al bajar el servicio en el nodo en el cual ha caído nuestra petición de autenticación, cuando el balanceador enviará el resto de peticiones al otro nodo, y dicho nodo volverá a solicitar que nos autentiquemos, siendo un comportamiento no deseado para un servicio de alta disponibilidad y que indica que las sesiones de los usuarios no se están compartiendo de manera adecuada.

Tras atenta lectura de la documentación referente al servicio de tickers (service ticket) [4.5.1] de Apereo CAS, detallaremos dos de los múltiples mecanismos de replicación de sesiones:

- Hazelcast: Es un sistema de tickets implementado en memoria caché. El proyecto de CAS ha integrado la librería Hazelcast como método de compartición de tickets de manera dinámica para que cada nodo almacene sus tickets en memoria y se sincronice con el resto de nodos para actualizar los tickets válidos de manera global en el sistema.
- Redis: Es un sistema de tickets que se basa en almacenar los tickets en una base de datos Redis (NoSQL). La base de datos se ejecuta en un servicio externo al proceso de CAS por lo que en determinados entornos podría ser beneficioso. Además la base de datos actúa como lugar de almacenamiento común entre todos los nodos simplificando las operaciones de consulta, replicación y compartición de tickets en la infraestructura.

Pasamos a indicar las estructuras y propiedades requeridas para implementar la compartición de tickets por los dos métodos nombrados con anterioridad; se destaca que los dos métodos son de funcionamiento exclusivo y no pueden ser activados a la vez debido a que producirían errores y degradación de los sistemas. Lo primero será especificar en el pom.xml que era el fichero que le indicaba al sistema de capas de CAS las librerías que se deben anexar y activar durante la construcción del paquete.

En caso de implementación de CAS con alta disponibilidad a través de Hazelcast se debe añadir la siguiente dependencia:

```
<dependency>
        <groupId>org.apereo.cas</groupId>
        <artifactId>cas-server-support-hazelcast-ticket-registry</artifactId>
        <version>${cas.version}</version>
</dependency>
```

En caso de implementar CAS con alta disponibilidad a través de Redis, se debe añadir la siguiente dependencia:

```
<dependency>
        <groupId>org.apereo.cas</groupId>
        <artifactId>cas-server-support-redis-ticket-registry</artifactId>
        <version>${cas.version}</version>
</dependency>
```

El siguiente paso consistirá en editar el fichero /etc/cas/config/cas.properties para añadir las propiedades que se detallan a continuación, destacando previamente que las etiquetadas con **cas.tgc.crypto.* y cas.webglow.crypto.* son propiedades a definir con el mismo valor en todos los nodos, cosa que no está claramente especificado en la web del proyecto oficial.** En caso de no especificar las mismas o ponerles diferentes valores entre nodo, y aún especificando los parámetros de Redis o Hazelcast, estariamos ejecutando un entorno en el que los tickets se compartirán entre los nodos, pero serían ilegibles, debido a que cada nodo estaría usando diferentes claves criptográficas para encriptar, y luego el resto de nodos no tendría la clave para desencriptar, resultado un entorno inconexo.

```
#Not explicitly documented requirement for HA
cas.tgc.crypto.encryption.key=233LQlzL3Sz92d17J2WN0NhKHXyGRxpGGDVq755Dh30
cas.tgc.crypto.signing.key=tVy88heddYqpL8e9eeyoeO4cLlrOtZAD4Ej8JJiArczfwWcHPTHTtN
s0YU4mgddEcroHLKwtMU99FyuRzZFXaw
cas.webflow.crypto.signing.key=DAVsTVE8FvDSD4Z-FSWmec-yaHOLoIntooqzt_RB6ChT9EMPJJ
8Zf016sC947MeyCf-UHTX2SJfW5ljSZjyJTQ
cas.webflow.crypto.encryption.key=orcEP9ABHQBn5vNQBmS0iA
# Hazelcast properties
cas.ticket.registry.hazelcast.cluster.members=192.168.56.111,192.168.56.112
# REDIS properties
cas.ticket.registry.redis.host=data.uoc.tfg
cas.ticket.registry.redis.port=6379
cas.ticket.registry.redis.database=0
cas.ticket.registry.redis.password=
cas.ticket.registry.redis.timeout=2000
cas.ticket.registry.redis.useSsl=false
```

Las configuraciones específicas de Hazelcast requieren únicamente identificar los nodos existentes de CAS existentes en el cluster. Las configuraciones específicas de Redis indican los datos de dónde existe la base de datos remota de almacenamiento de tickets.

Para que funcione correctamente el servicio con redis, hemos instalado en el servidor tfg_data un servicio Redis con configuraciones por defecto.

```
$ sudo apt install redis-server
```

5. Ejecución de pruebas de rendimiento

5.1 Introducción y contexto de las pruebas

En todo nuevo proyecto o implantación es crucial realizar pruebas que permitan garantizar la calidad del producto que se está desarrollando. Las pruebas sobre el software permiten obtener información sobre cómo un sistema se comporta ante determinados aspectos y nos aporta una visión de los problemas no detectados durante la implantación.

Concretamente las pruebas de rendimiento permiten saber cómo se comportará un sistema una vez pase a producción. La importancia de las mismas es perfectamente justificable si nos fijamos en el número de aplicaciones que presentan problemas ante picos de carga o tiempos de respuesta inaceptables para el usuario final.

Para realizar pruebas de rendimiento sobre un sistema, someteremos dicho sistema a la simulación de la actividad real de los futuros usuarios en el sistema. Estas pruebas nos permitirán predecir el comportamiento del sistema para valorar si es capaz de realizar sus funciones sin pérdida de servicio y con un tiempo de respuesta de usuario estable y óptimo. A partir de los datos que obtengamos en las pruebas y tras un análisis adecuado se pueden realizar modificaciones en nuestro sistema para eliminar o en el peor caso mitigar los posibles problemas que aparecieran con un determinado número de usuarios interactuando con nuestro sistema.

Es importante que tras realizar las modificaciones oportunas en el sistema, se realice exactamente la misma prueba de rendimiento para poder comparar los resultados obtenidos y verificar que los cambios realizados actúan de manera positiva en la consecución de objetivos marcados en nuestras pruebas.

5.1.1 Software de pruebas de rendimiento

Para realizar las pruebas de rendimiento existen diferentes productos en el mercado. La misión de un software de pruebas de rendimiento es simular la entrada de múltiples usuarios en el sistema a partir de unas instrucciones o script de órdenes facilitadas por el encargado de las pruebas. Además el software de pruebas de rendimiento debe facilitar algún tipo de informe de estadísticas de la simulación para que el técnico que realiza las pruebas pueda analizar y decidir el grado de adecuación del sistema a la estimación de demanda real a la que se va a someter.

Tras realizar una amplia búsqueda del software que permita realizar pruebas de rendimiento, aparecen dos productos con resaltables referencias; uno de ellos es **Apache Jmeter** y el otro HP LoadRunner que recientemente ha pasado a manos de otra empresa llamandose ahora **MicroFocus LoadRunner**[5.1.1.1]

5.1.2 Comparativa de software de pruebas de rendimiento

Para poder comparar las principales funciones de los productos objetivos a analizar realizaremos una tabla donde se reflejen los diferentes aspectos a valorar[5.1.2.1]

Criterio	LoadRunner (HPE)	JMeter (Apache)
Licencia comercial	Sí	No, gratuita (Apache 2.0)
Precio base	Hasta 50 usuarios virtuales gratis, luego facturación en función de volúmen.	Gratis sea cual sea el número de usuarios.
Cobertura de protocolo	Especialmente interesante en entornos empresariales complejos (SAP y ERP). Amplia variedad de entornos y protocolos de aplicaciones, como web/móvil, servicios web, MQ, HTML5, WebSockets, AJAX, Flex, RDP, base de datos, emuladores de terminal remoto, Citrix, Java, .NET, Oracle	Amplia variedad de protocolos; HTTP (S), servicios web (Soap, Rest), FTP, LDAP y protocolos relacionados con Java (JMS, JDBC) .Soporta más protocolos con complementos de código abierto.
Capacidad de grabación y reproducción	Sí	Sí
Formación / Aprendizaje	Sí, de pago mayoritariamente.	Como JMeter es un programa de código abierto proporcionado por Apache, no hay cursos de aprendizaje de Apache. Existen múltiples recursos por internet para aprender a usar el software por uno mismo además de empresas especialistas que ofrecen cursos de pago.
Monitorización del estado de la prueba en tiempo real	Sí	Sí
Personalización de scripts	Difícil y limitado (lenguaje C, Javascript agregado para la Web recientemente)	Es fácil, ya que JMeter se basa en una arquitectura de complementos y está hecho para ser personalizado (los elementos JSR223 le permiten usar muchos lenguajes de script (Groovy, Javascript, Java ...)). Esto lo convierte en la herramienta más personalizable en comparación con otros.

Informe de resultados	Sí	Sí
Generación de carga	Depende del tipo de licencia	Generación de carga *ilimitada*.
Soporte de esquemas de autenticación y certificados de cliente.	Sí	Sí (Básico, Digest, Kerberos, ...)
Manejo flexible de errores y criterios ajustables de aprobación / fallo	Sí	Sí
Emulación de dispositivos	Sí	Sí, modificando el encabezado User-Agent en Header Manager
Sistema Operativo	Windows. Solamente el componente Load Generator está disponible para Linux	Sí, ya que está basado en Java
Cluster (escalado)	Sí	Sí (horizontal a través del modo cliente / servidor o PAAS)
Número de usuarios virtuales disponiendo de 8 CPU virtuales y 4Gb de RAM	1000	De 1000 a 1500
Prueba de carga como código	Sí	Sí

Tras la resumida comparativa de características y funciones que ofrecen ambos productos, la tabla muestra las **bondades de Jmeter respecto a LoadRunner** para este proyecto. Jmeter, al ser un proyecto coordinado por la fundación Apache y presentar una Licencia de uso Apache, da la libertad al público en general de utilizar el software sin limitaciones cosa que su competidor limita de manera gratuita al uso de 50 usuarios virtuales. A su vez y debido a la gratuidad del componente han aparecido por internet multitud de tutoriales que enseñan al usuario a utilizar la herramienta de manera correcta y analítica.

Aunque no se descarta que un producto como LoadRunner tendrá unas excelentes prestaciones en entornos empresariales complejos como podrían ser pruebas sobre SAP o aplicaciones específicas de Oracle; para un proyecto que se aleje de estas características tan específicas, no se justifica el coste de las licencias con el beneficio que podría aportar respecto a Jmeter.

Es por todo lo anterior que **se utilizará Apache Jmeter para someter a pruebas de rendimiento los sistemas anteriormente diseñados**.

5.1.3 Instalación de Jmeter

Para la instalación de Jmeter bastará con acceder a la página de descargas de Jmeter (https://jmeter.apache.org/download_jmeter.cgi) y obtener un fichero comprimido con los binarios. La versión disponible con la que se realizarán las pruebas es Jmeter 5.1.1.

Debido a que Jmeter se ejecuta con Java, la elección del sistema operativo del equipo o equipos que ejecutan la prueba es totalmente libre con la única condición de que tendrá que tener disponible alguna versión de Java 8 instalada.

Una vez descomprimimos el archivo descargado accederemos a la carpeta bin/ y encontraremos los ejecutables jmeter.sh para Linux o jmeter.bat en caso de querer ejecutarlo en un Windows. Junto con la instalación vienen múltiples complementos y ejemplos de scripts de ejecución que serán muy útiles para aprender y entender el funcionamiento de las pruebas, además de ser fácilmente modificables para nuestro proyecto.

5.1.4 Objetivo

Una manera de tener resultados útiles y que ofrezcan la posibilidad de un análisis respecto a la carga que soporta el servicio de CAS que se ha montado es ir probando de la manera más atómica posible los elementos que lo componen.

Con este objetivo, se realizarán las siguientes pruebas de rendimiento a los diversos componentes que componen la arquitectura intentando buscar los límites de la misma y los posibles cuellos de botella que pudieran surgir para que en caso de ser posible, en una etapa posterior buscar una solución adecuada que elimine o mitigue el error.

- Prueba de carga al directorio de usuarios OpenLdap. Con esta prueba se busca saber el rendimiento del servicio LDAP montado en data.uoc.tfg:389. Principalmente la prueba consistirá en hacer operaciones de **single bind** de manera masiva a un conjunto amplio de usuarios.
- Prueba de carga al nodo CAS en standalone. La prueba busca obtener estadísticas de cuántos usuarios pueden conectarse a un nodo de CAS sin balanceador. Se accede al servicio de CAS montado en el nodo cas1.uoc.tfg:443 que consume los servicios de LDAP de la prueba anterior. Se debe tener un script de prueba, en el que el conjunto de usuarios simula acceder a la aplicación DemoCAS, esta deberá redireccionarlos al CAS para que se validen y posteriormente cuando el usuario simulado introduce sus credenciales retornará a la aplicación DemoCAS obteniendo los atributos adecuados.
- Prueba de carga al servicio de CAS en alta disponibilidad. En esta prueba se comprobará cómo funciona la solución completa. Para ella se configurará el servicio de CAS a través de bal.uoc.tfg:443/cas. El script usado para esta prueba es el mismo que en el anterior caso, variando las URI, ya que en vez de utilizar el nodo cas1.uoc.tfg se utilizarán el balanceador haproxy y los nodos cas1 y cas2.

5.2 Directorio de usuarios Openldap

5.2.1 Aspectos generales

Para realizar la prueba del servicio de Openldap nos basaremos en la documentación de Apache Jmeter [5.2.1] y [5.2.2] y sus ejemplos. Se genera una prueba para el grupo *LDAP Users* en el que configuraremos aspectos tales como URI del servicio y fichero usuarios_pruebas.txt en formato CSV de donde se obtienen los credenciales de los usuarios. Además recurriremos a una “Petición Extendida de LDAP” para realizar el ldapsearch. La prueba consta de 3 pasos principales por “usuario virtual” o UV.

1. Thread bind: El UV se valida contra el directorio.
2. ldapsearch: El UV realiza una petición LDAP recuperando todos sus atributos
3. Thead unbind: El UV se desconecta contra el directorio.

En esta prueba se simulan las peticiones que el servicio de CAS realiza sobre el sistema LDAP por cada usuario que intenta validar. Se pretende comprobar que el sistema de LDAP permite realizar suficientes consultas por segundo como para no generar un cuello de botella en nuestro sistema CAS.

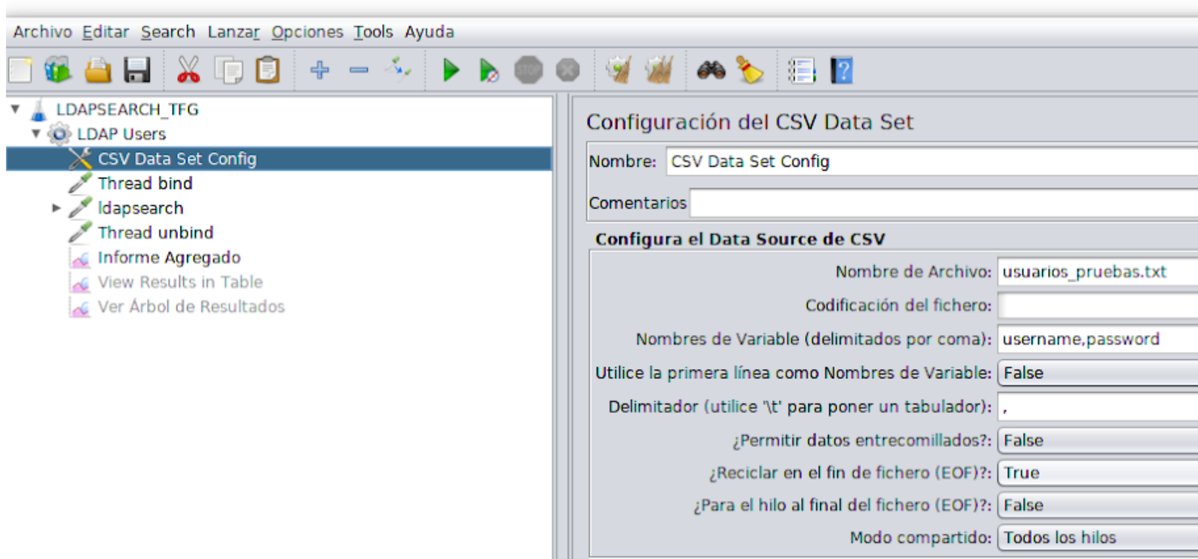


Figura 25: Definición de CSV de usuarios en en prueba de rendimiento LDAP

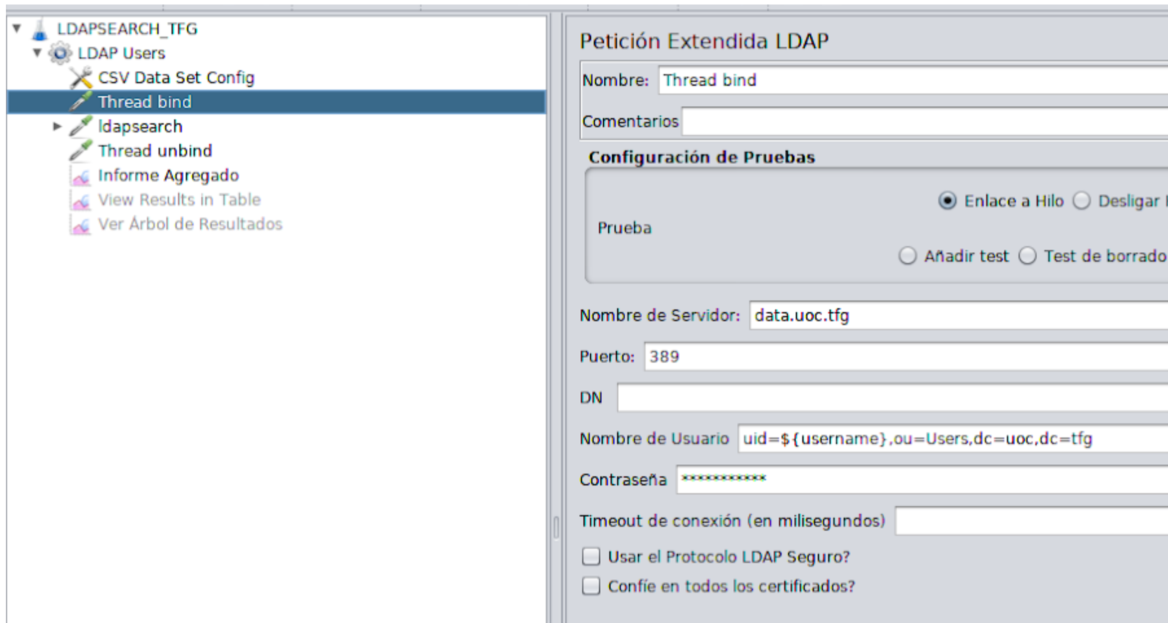


Figura 26: Bind o enlace con credenciales de cada usuario en prueba de rendimiento LDAP

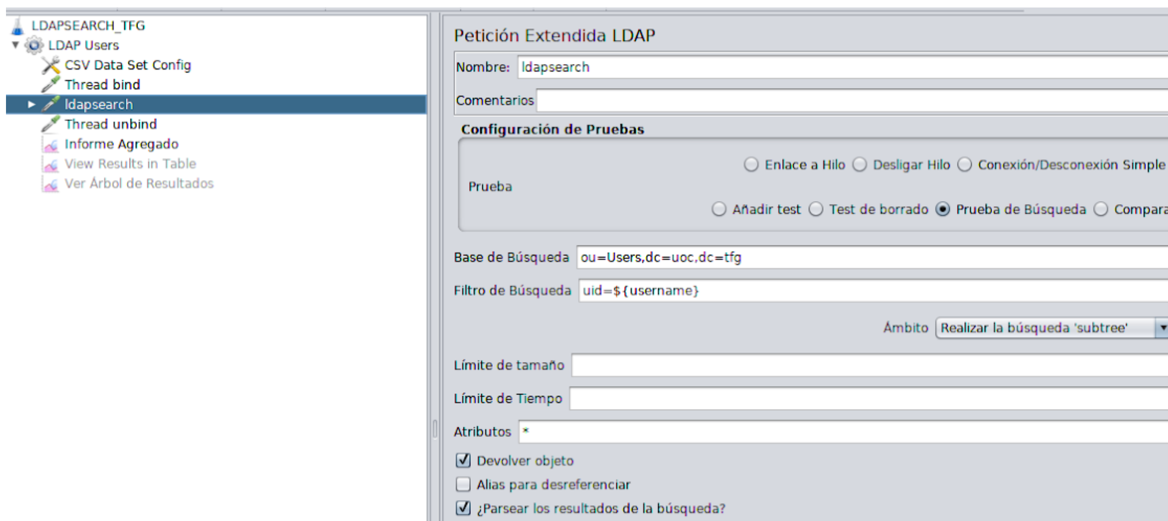


Figura 27: Ldapsearch y recuperación de atributos de usuario en prueba de rendimiento LDAP

Antes de realizar la prueba, se debe generar el fichero de usuarios_pruebas.txt. Para ello se ha utilizado un shell script cuyo fichero de salida debe alojarse en el mismo directorio que el fichero jmx de la prueba de Jmeter.

```
#!/bin/bash
for i in $(seq 1 1 1000)
do
echo "usuario$i,pass$i" >> tmp.txt
done
cat tmp.txt | shuf > usuarios_pruebas.txt
rm -Rf tmp.txt
```

5.2.2 Resultados de la prueba y análisis

Al ejecutar la prueba se ha comprobado que el directorio LDAP alojado solamente en un nodo se comporta de manera adecuada permitiendo más de 1200 peticiones por segundo. Se aprecia que las **operaciones más costosas de tiempo son las de bind**, pero los tiempos de respuesta de la misma para el 95% de los casos son inferiores a los 132 milisegundos valor que se puede considerar más que aceptable. Consideramos por tanto este valor de 1200 peticiones por segundo suficiente para nuestro sistema y se asegura que el servicio de directorio no será un posible cuello de botella siempre que no se sobrepasen ese umbral de peticiones.

Etiqueta	# Muestras	Media	Mediana	90% Line	95% Line	99% Line	Mín	Máx	% Error	Rendimiento
Thread bind	2500	60	37	88	132	1048	0	1079	0,00%	1210,1/sec
ldapsearch	2500	4	1	12	16	43	0	68	0,00%	1210,1/sec
Thread unbind	2500	0	0	0	0	0	0	0	0,00%	1210,7/sec
Total	7500	21	1	50	69	138	0	1079	0,00%	3628,4/sec

Figura 28: Rendimiento del nodo LDAP para 500 peticiones en un segundo realizadas 5 veces.

5.3 Nodo CAS en standalone

Para realizar la siguiente prueba y apoyándonos en la documentación de diseño pruebas de rendimiento recomendadas de Apereo [5.3.1], realizaremos una serie de pruebas con un nodo de cas funcionando sin configuraciones de alta disponibilidad. La prueba se realizará directamente, sin pasar por el balanceador, al puerto 443 de la máquina tfg_cas1. Se simulará el acceso de un usuario a la aplicación de pruebas DemoCAS que estaba alojada en el nodo tfg_data. Al igual que en la prueba anterior, un fichero CSV proveerá de las identidades de usuario. La prueba consta de los siguientes pasos:

1. GET-CAS Login Page: En esta petición se hace una petición a la página de DemoCAS la cual hará una redirección al servidor de CAS. En este paso se capturan variables de sesión que son necesarias para el paso siguiente.
2. POST- Login Credentials: Sería lo equivalente a introducir usuario y contraseña en el navegador. Con estos datos, junto con la variable de sesión anteriormente capturada se valida al usuario en el sistema. Se producen dos redirecciones automáticas dentro del paso, en el que finalmente se acaban obteniendo los atributos que muestra la aplicación DemoCAS. Se utiliza una **aserción de respuesta para validar el correcto funcionamiento cada petición** de manera que entre los atributos devueltos se devuelva un atributo con la cadena “/home/” que corresponde con el homeDiretory de cada usuario. En caso de no recibirlo, se interpretaría como un error y quedaría registrado en el campo *Error %*.
3. GET - User Logout: Se realiza la operación de invocación del contexto /cas/logout simulando que el usuario cierra sesión de CAS.

5.3.1 Resultados de la prueba y análisis

Debido a que en este paso desconocemos el número exactos de peticiones que es capaz de soportar el nodo de CAS, se inician las pruebas para un número bajo de peticiones de UV y se va

ampliando el número de peticiones hasta el punto en el que los tiempos de respuesta empiecen a cambiar a valores no aceptables o en el momento en el que obtengamos errores en las respuestas.

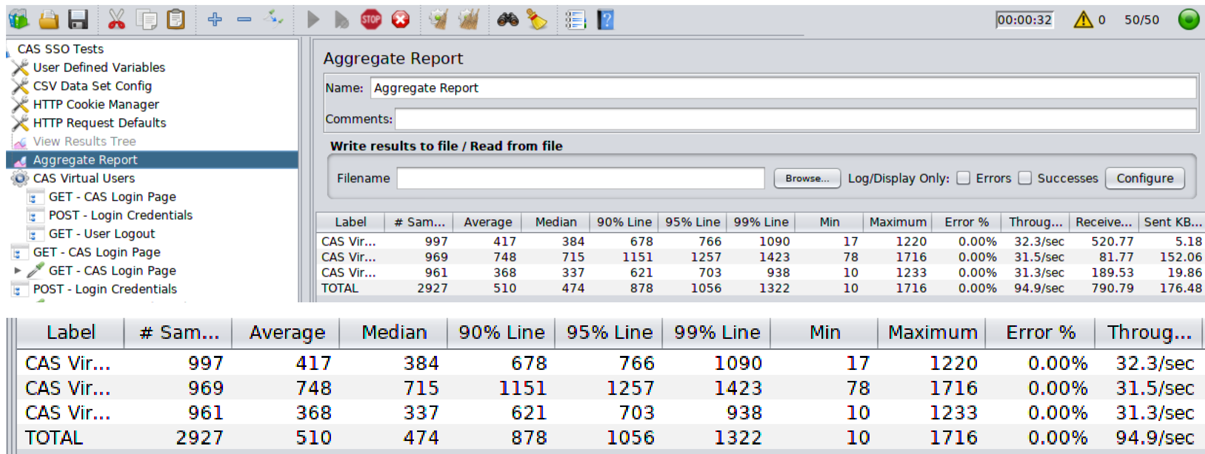


Figura 29: Rendimiento de CAS standalone a los 30 segundos del inicio de una prueba continua con 50 usuarios entrando de manera continuada.

La primera prueba consiste en introducir 50 UV en el sistema de manera continua y ver cómo se comportan los tiempos y la carga del sistema, tras 30 segundos de prueba el 95% de las respuestas se hacen en menos de 1,2 segundos. Debe apreciarse que el rendimiento aproximado del sistema en ese momento fue de unas 31.5 peticiones por segundo.

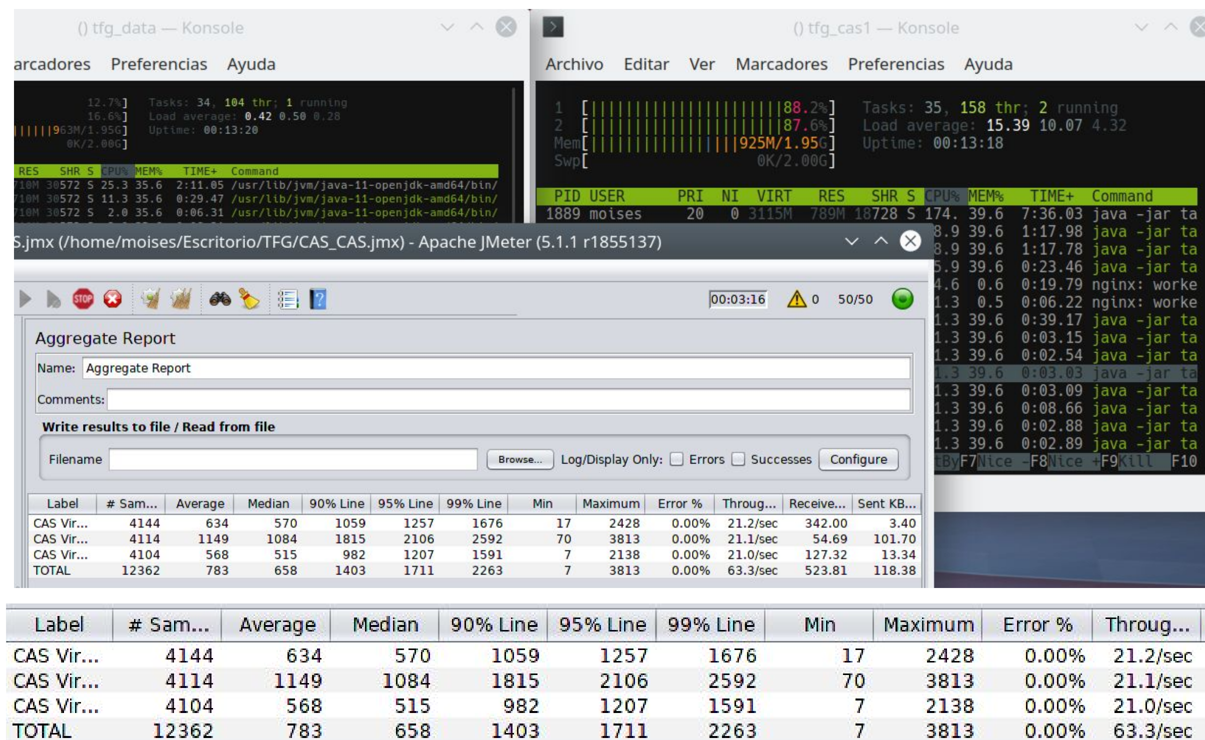


Figura 30: Rendimiento de CAS standalone tras 3 minutos con 50 usuarios entrando de manera continuada y carga de la máquina tfg_cas1

Tras 3 minutos de ejecución con un grupo sostenido de 50UV vemos como el rendimiento del sistema baja y se aproxima más a lo que podría ser el valor de rendimiento real del sistema. Observamos que aún bajando a 21 peticiones por segundo por cada paso, para un conjunto de 50UV el sistema continúa funcionando sin errores y tiempos de respuesta inferiores a los 2.5 segundos en el peor de los casos. Aún así vemos como la carga de la máquina tfg_cas1 ha aumentado a 15.4 lo cual se hace imposible de controlar solamente con dos procesadores que durante la prueba están rozan 100% de carga. Se detecta que en **este momento, el número de procesadores asignados y la potencia de los mismos podría ser el factor limitante para que el servicio de CAS no esté ofreciendo un mayor rendimiento**.

Aunque estamos trabajando con máquinas virtuales y podemos asignar un mayor número de procesadores, vamos a optar por otra opción que consiste en variar las condiciones de la prueba de rendimiento. Conociendo que el sistema acepta la entrada continua de 50UV durante al menos 3 minutos sin producir errores, realizaremos una prueba más realista de nuestro sistema.

Lo siguiente será pasar a ejecutar la entrada de 800 UV durante un periodo de entrada de 30 segundos, intentando simular la entrada masiva de usuarios en un momento determinado a algún servicio de nuestra organización. Esta prueba dará un flujo de generación de usuarios de 26,667 UV por segundo se manera que al acabar los 30 segundos hayan entrado a la simulación todos los usuarios. Con este tipo de prueba también se consigue que los usuarios se repartan de manera homogénea por toda la aplicación durante todo el intervalo de tiempo.

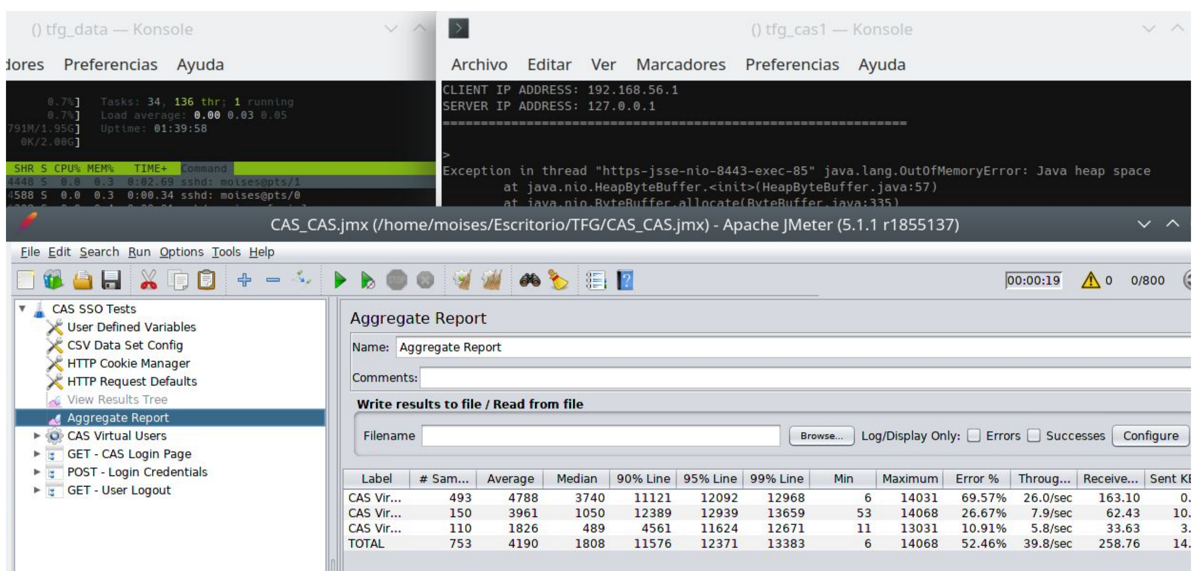


Figura 31: Errores debido a falta de memoria Java en el nodo de CAS standalone para prueba de 800 usuarios en rampa de entrada de 30 segundos.

Tal y como se aprecia en la imagen anterior, a partir del segundo 19 el sistema se sobrecargó. Si observamos la consola de la máquina tfg_cas1 vemos como el proceso que ejecuta CAS da un error **java.lang.OutOfMemoryError: Java heap space**. Se comprueba a que debido a que ejecutamos el servicio de CAS con el comando por defecto "./build.sh run", el proceso que se arranca **no dispone de la suficiente memoria** que requeriría un proceso de servidor. Tras

consultar la documentación de Oracle respecto a la configuracion de memoria de Java [5.3.1.1], se vuelve a ejecutar el proceso de CAS asignando 1Gb de memoria Heap que debería mejorar los resultados anteriores.

```
$ java -Xms1G -Xmx1G -jar target/cas.war
```

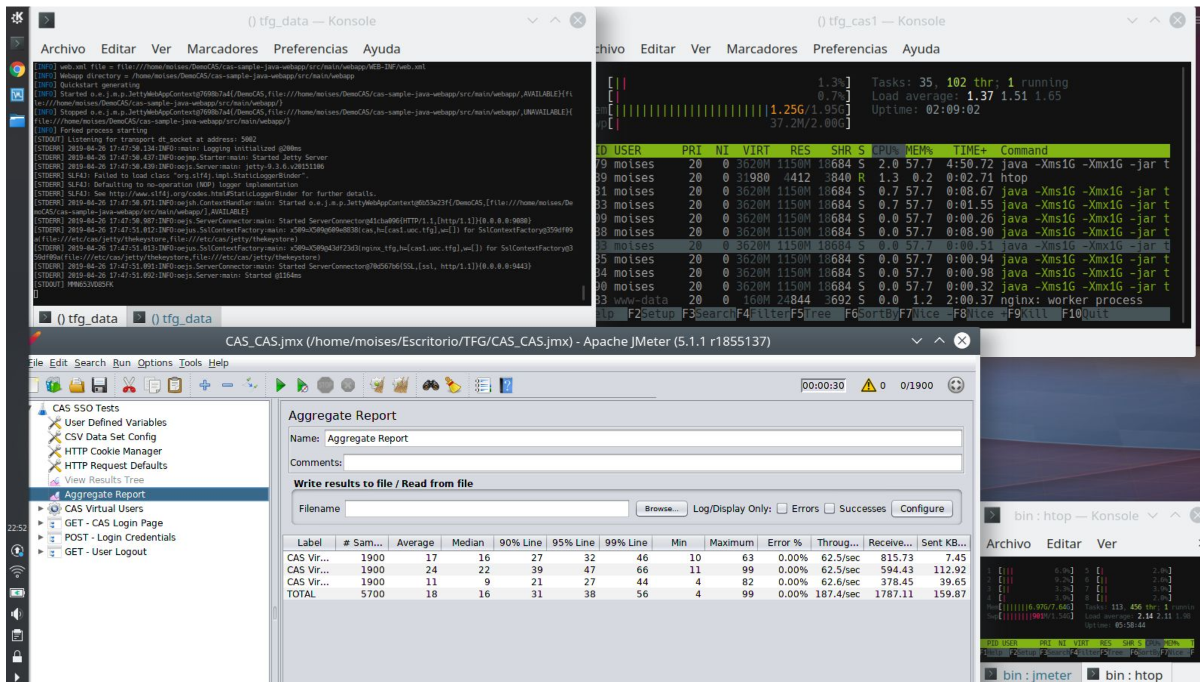


Figura 32: Valores de nodo de CAS standalone con 1Gb para prueba de 1900 usuarios entrando en 30 segundos.

Tras la ejecución de múltiples pruebas se detectó que **aumentando la memoria a 1Gb el sistema es capaz de escalar correctamente** para permitir la entrada sin errores de 1900 usuarios en 30 segundos (63,3 UV/segundo), pasado este valor, se vuelven a obtener errores de memoria Heap. Debido a que no se cuenta con memoria infinita se procede a continuar con las pruebas para el servicio de CAS en alta disponibilidad teniendo este valor y esta configuración como referencia para comprobar si varían ya sea a mejor o peor el número de usuarios al añadir el segundo nodo y el soporte de cluster alta disponibilidad al servicio de CAS.

5.4 Servicio de CAS en alta disponibilidad.

Tras haber interactuado con el sistema funcionando con un solo nodo y tener un breve comportamiento de los valores que puede soportar de manera sostenida en el tiempo, procedemos a definir un orden de pruebas a seguir para garantizar que el sistema se comporte correctamente durante la ejecución del test.

Debido a que en algunas pruebas obtendremos errores graves y hará falta reiniciar el servicio que presente fallos, es imprescindible respetar el orden este bloque de pruebas a modo de calentamiento del sistema ya que intentar lanzar una prueba exigente sobre un sistema recién reiniciado, en muchas ocasiones y sobre todo en sistemas de servidores de aplicaciones, hará que no existan los suficientes pooles de conexión o recursos asignados para tales picos en el instante cero. Tras unas pruebas de rendimiento adecuadas y realizando un análisis de requisitos, se puede establecer un conjunto de propiedades en el arranque del sistema para que se comporte de manera satisfactoria ante estos picos.

Definimos a continuación al conjunto de pruebas que se le realizan al servicio de CAS en alta disponibilidad, a medida que se obtenga algún fallo, se analizará el último mejor resultado y en caso de que sea posible, se adoptarán cambios que permitan mejorar la estabilidad o rendimiento del servicio:

- 30 UV accediendo en 1 segundo
- 60 UV accediendo en 1 segundo
- 300 UV accediendo en 10 segundos
- 1200 UV accediendo en 30 segundos
- 1600 UV accediendo en 30 segundos
- 1800 UV accediendo en 30 segundos
- 1900 UV accediendo en 30 segundos
- 2200 UV accediendo en 30 segundos
- 2500 UV accediendo en 30 segundos
- 3000 UV accediendo en 30 segundos

Se debe tener en cuenta que las pruebas se realizarán varias veces de manera que se despreciaran resultados que hayan ocurrido debido al azar, es decir, no consideraremos que nuestro sistema llega a escalar a 3000 usuarios en 30 segundos a no ser que tras un resultado satisfactorio hayamos comprobado que se puede repetir la prueba múltiples veces con resultados satisfactorios.

Realizaremos una prueba aún con un solo nodo, pero utilizando el balanceador como punto de entrada, intentando comprobar que añadir el balanceador a la infraestructura de autenticación no compromete los resultados anteriores.

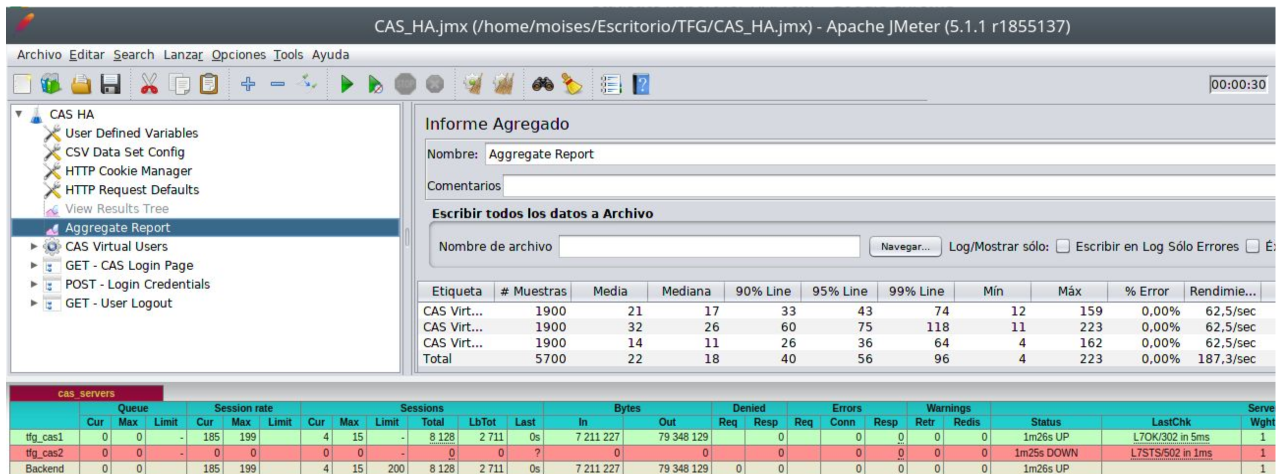


Figura 33: Valores de nodo de CAS standalone a través del balanceador con 1Gb para prueba de 1900 usuarios entrando en 30 segundos.

Comprobamos tras la ejecución de la prueba que los tiempos de respuesta se mantienen y que el balanceador soporta perfectamente cargas de 187,3 peticiones por segundo.

5.4.1 Comparativa de rendimiento de diferentes tecnologías de replicación de tickets.

Tal y como se explicó en capítulos anteriores nos hemos decantado por evaluar dos de las principales tecnologías disponibles en el servicio de CAS para compartir los “service tickets” (que identifican la sesión autenticada) de los usuarios.

Tras haber detectado que en algunos casos 1Gb de memoria no es suficiente para algunas tecnologías que componen el CAS, se realizará una prueba de rendimiento en las mismas condiciones para comprobar qué sistema CAS standalone soporta más usuarios antes de fallar debido a problemas de memoria Heap de Java, tras la prueba, orientaremos las pruebas a la tecnología que sea más eficiente en la gestión de memoria.

Etiqueta	# Muestras	Media	Mediana	90% Line	95% Line	99% Line	Mín	Máx	% Error	Rendimie...
CAS Virtu...	986	1720	817	4183	4662	5519	8	6188	27,59%	57,7/sec
CAS Virtu...	713	1456	773	3833	4287	5069	36	5551	31,28%	41,8/sec
CAS Virtu...	490	925	470	2489	3633	5486	9	6143	4,69%	28,8/sec
Total	2189	1456	684	3923	4455	5395	8	6188	23,66%	128,0/sec

Figura 34: Nodo de CAS standalone configurado con replicación de tickets con Redis en prueba de 1800 usuarios entrando en 30 segundos.

Etiqueta	# Muestras	Media	Mediana	90% Line	95% Line	99% Line	Mín	Máx	% Error	Rendimie...
CAS Virtu...	1800	39	27	80	103	162	14	268	0,00%	58,8/sec
CAS Virtu...	1800	79	67	148	176	234	14	349	0,00%	58,8/sec
CAS Virtu...	1800	38	27	82	102	156	6	234	0,00%	58,8/sec
Total	5400	52	37	113	140	205	6	349	0,00%	176,3/sec

Figura 35: Nodo de CAS standalone configurado con replicación de tickets con Hazelcast en prueba de 1800 usuarios entrando en 30 segundos.

Según muestran las pruebas y con memoria para el proceso de Java de 1Gb, el sistema con compartición de tickets por medio de Hazelcast se comporta sin errores para una prueba de 1800 usuarios entrando en 30 segundos. Por otro lado el sistema Redis ha provocado errores para ese valor comportándose correctamente para la prueba anterior de 1600 usuarios entrando en 30 segundos.

Como se aprecia, el servicio CAS con Hazelcast es más eficiente usando la memoria del proceso, se continuarán las pruebas con dos nodos de CAS en alta disponibilidad con **compartición de tickets a través de Hazelcast**.

5.4.2 Prueba de rendimiento con dos nodos activos replicando tickets a través de Hazelcast

Teniendo ya los dos nodos activos y habiendo comprobado que replican correctamente los tickets de autenticación entre ellos, se comprueba en el balanceador que tras una prueba básica con jmeter, las peticiones son repartidas equitativamente entre los dos nodos de backend.

> General process information

pid = 2370 (process #1, nbproc = 1, nbthread = 1)
uptime = 0d 0h00m17s
system limits: memmax = unlimited; ulimit-n = 4034
maxsock = 4034; **maxconn** = 2000; **maxpipes** = 0
current conns = 25; current pipes = 0/0; conn rate = 61/sec
Running tasks: 4/31; idle = 66 %

active UP | backup UP
active UP, going down | backup UP, going down
active DOWN, going up | backup DOWN, going up
active or backup DOWN | not checked
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
Note: "NOLB"/"DRAIN" = UP with load-balancing disabled.

haproxy_frontend

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings			
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
Frontend				58	60	-	23	34	2 000	461			1 184 704	13 170 257	0	0	0					OPEN	

cas_servers

	Queue			Session rate			Sessions						Bytes		Denied		Errors			Warnings			
	Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk
tfg_cas1	0	0	-	91	90		12	19	-	681	231	0s	589 547	6 572 774		0		0	0	0	0	17s UP	L7OK/302 in 5ms
tfg_cas2	0	0	-	91	91		9	16	-	682	230	0s	595 157	6 597 483		0		0	0	0	0	17s UP	L7OK/302 in 8ms
Backend	0	0		183	179		21	33	200	1 363	461	0s	1 184 704	13 170 257	0	0		0	0	0	0	17s UP	

Figura 36: Estado del balanceador con dos nodos de CAS activos

Se vuelve a ejecutar el conjunto de pruebas hasta llegar a 1800 usuarios en 30 segundos.

Etiqueta	# Muestras	Media	Mediana	90% Line	95% Line	99% Line	Mín	Máx	% Error	Rendimie...
CAS Virtu...	1800	117	93	242	298	400	17	621	0,00%	58,5/sec
CAS Virtu...	1800	188	153	383	448	577	26	863	0,00%	58,3/sec
CAS Virtu...	1800	111	89	238	275	388	10	540	0,00%	58,2/sec
Total	5400	139	109	292	359	492	10	863	0,00%	174,3/sec

Figura 37: Servicio de CAS en HA con replicación de tickets con Hazelcast en prueba de 1800 usuarios entrando en 30 segundos.

Si comparamos los tiempos de respuesta de las muestras con los de la figura *Figura 35*, aunque por ser dos nodos solamente se deberían obtener mejoras; veremos un empeoramiento de los tiempos de respuesta. Una de las razones por las que puede ocurrir esto es que debido a que ambos deben intercambiar información de los tickets activos y eliminados, se emplea tiempo de proceso en intercambiar dichas estructuras por Hazelcast y se empeora para el 95% de las muestras el tiempo de respuesta en el orden de 150 milisegundos por petición.

Aún obteniendo estos datos, se puede pensar que el sistema puede escalar con buenos tiempos a valores cercanos a 2000 o 2500 usuarios, pero tras varias pruebas se comprueba que la memoria del proceso no es suficiente para gestionar tal número de tickets.

Se procede a aumentar los recursos del sistema dotando 3Gb el proceso de java que ejecuta el servicio de CAS.

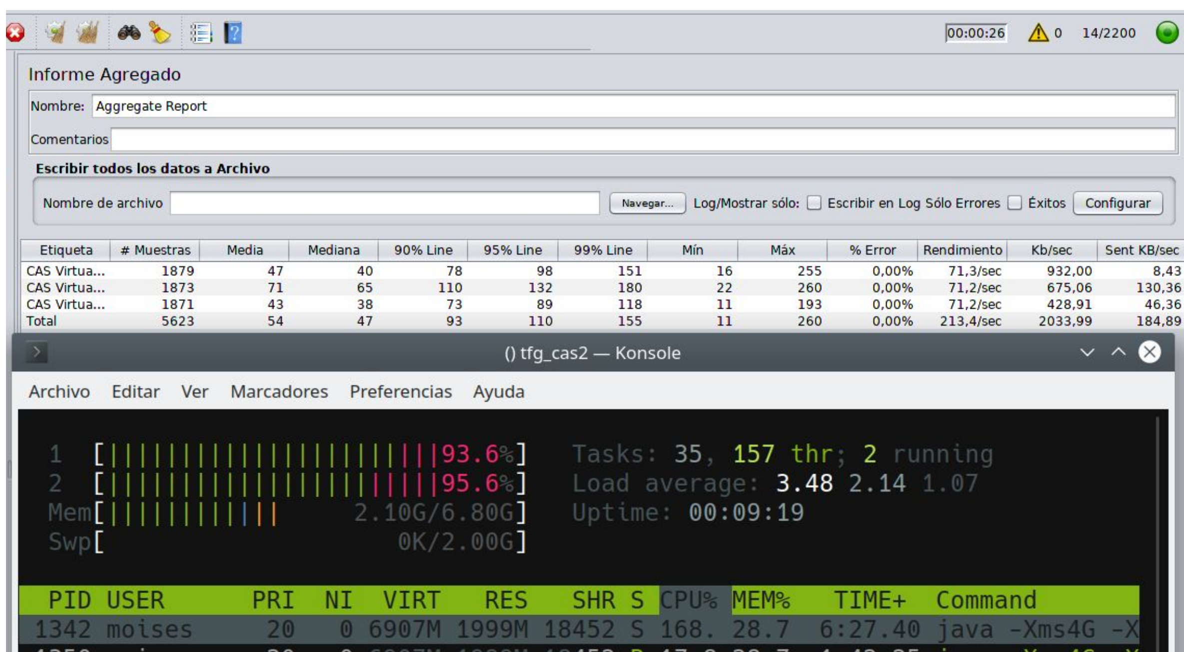
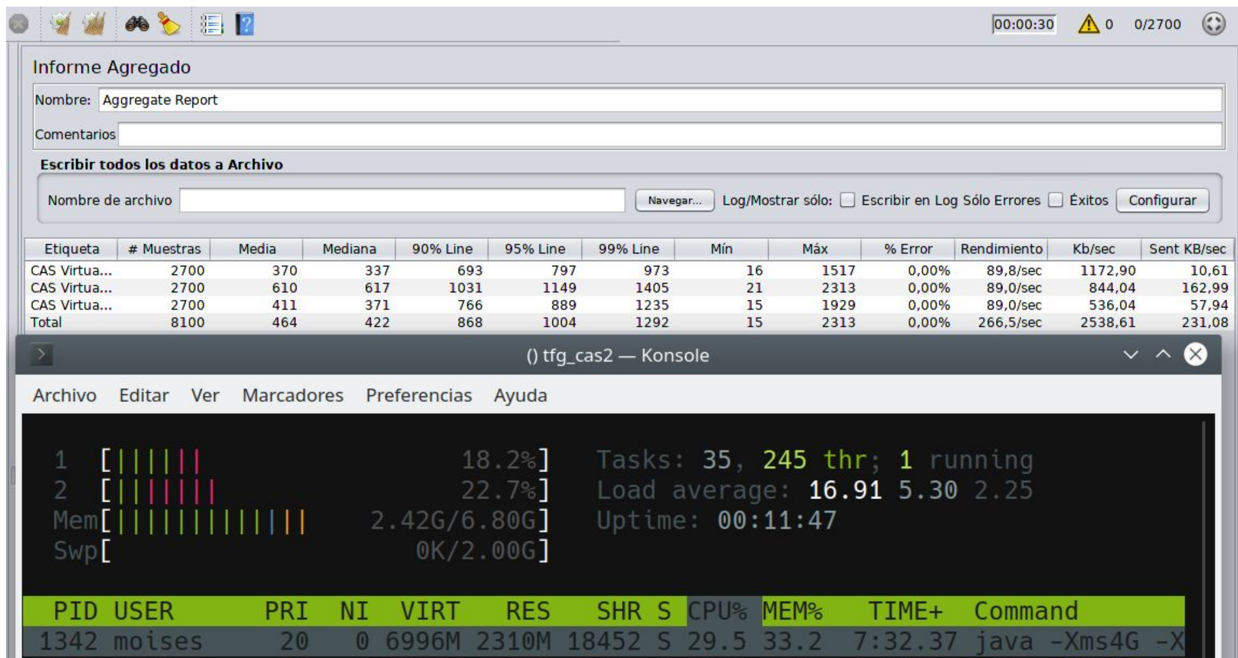


Figura 38: Servicio de CAS en HA con replicación de tickets con Hazelcast y 3Gb de memoria por nodo en prueba de 2200 usuarios entrando en 30 segundos.

Tras algunas pruebas, se detecta que **al aumentar los recursos de memoria**, las máquinas tfg_cas1 y tfg_cas2 **bajan considerablemente la carga de CPU durante las pruebas**. Analizando los factores que podrían afectar a esto, se detecta también que el recolector de basura de la máquina virtual de java podría estar ejecutándose demasiadas veces debido a la poca memoria inicialmente asignada, lo que se traducía en alto consumo de CPU.

Vemos como el sistema ahora llega a soportar 2200 usuarios en 30 segundos de manera adecuada y baja carga en el sistema. Esto nos lleva a pensar que sería capaz de pasar una prueba de carga de larga duración siempre que el flujo de peticiones por segundo se mantenga.



Etiqueta	# Muestras	Media	Mediana	90% Line	95% Line	99% Line	Mín	Máx	% Error	Rendimiento	Kb/sec	Sent KB/sec
CAS Virtua...	2700	370	337	693	797	973	16	1517	0,00%	89,8/sec	1172,90	10,61
CAS Virtua...	2700	610	617	1031	1149	1405	21	2313	0,00%	89,0/sec	844,04	162,99
CAS Virtua...	2700	411	371	766	889	1235	15	1929	0,00%	89,0/sec	536,04	57,94
Total	8100	464	422	868	1004	1292	15	2313	0,00%	266,5/sec	2538,61	231,08

Figura 39: Servicio de CAS en HA con replicación de tickets con Hazelcast y 3Gb de memoria por nodo en prueba de 2700 usuarios entrando en 30 segundos.

Se ha detectado que en determinadas condiciones el sistema podría soportar algún pico de 2700 usuarios con tiempos de respuesta iguales o inferiores al segundo en el 90% de los casos. Pero se destaca que la carga que muestra la CPU de una de las máquinas del servicio de CAS nos indica que esta situación no sería sostenible por un tiempo prolongado.

En este punto se considera más que suficiente el estudio realizado para este trabajo, ya que **hemos conseguido un entorno altamente disponible** y nos hemos encontrado con que la máquina host no es capaz de aportar más rendimiento durante las pruebas en un entorno con cuatro máquinas linux virtualizadas y un Jmeter inyectando usuarios virtuales.

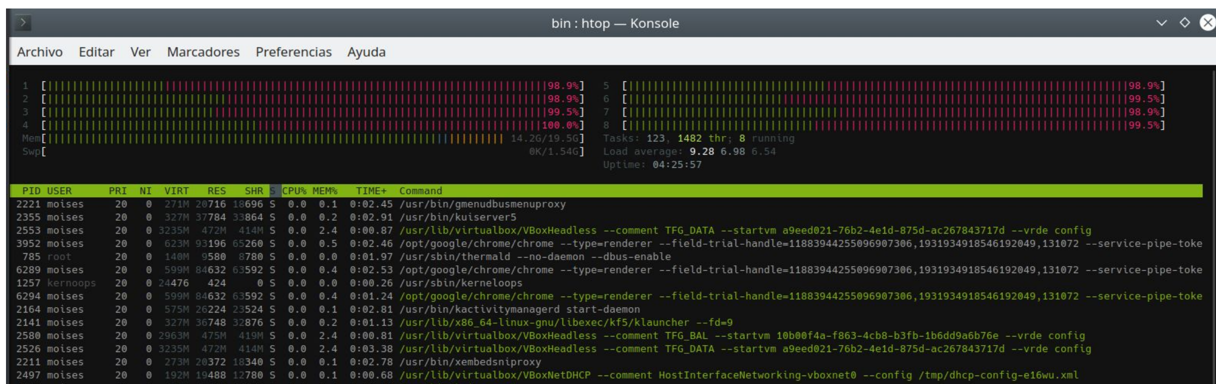


Figura 40: Estado de la máquina host al intentar hacer pruebas de rendimiento más exigentes.

6. Conclusiones y posibles mejoras

Durante la elaboración de este proyecto hemos realizado un estudio de las posibles decisiones que debe tomar un implantador de servicios; hemos hecho un breve análisis de las principales alternativas a elegir, seleccionando alguna que a nuestro juicio tuviera mayores beneficios que el resto. Finalmente se ha instalado un servicio de CAS que es capaz de soportar la caída de alguno de sus nodos sin que el conjunto de usuarios tenga que volver a autenticarse.

Por otro lado y durante las pruebas de rendimiento, hemos demostrado que esta parte de la implantación es una fase crucial y determinante de la calidad que se ofrecerá en el servicio. Se ha simulado el comportamiento de los usuarios en nuestro sistema, nos hemos anticipado a posibles errores por sobrecarga y hemos detectado necesidades de parametrización de nuestro sistema para hacerlo más estable. Podemos afirmar que nuestro sistema de dos nodos de CAS es capaz de soportar alrededor de 2700 usuarios en un pico de 30 segundos o hasta 4400 usuarios por minuto entrando de manera continuada (2200 cada 30 segundos). Extrapolando estos datos, consideramos que el sistema, está suficientemente dimensionado para organizaciones que requieran de hasta 264000 autenticaciones de usuario por hora, no siendo este valor su principal aspecto a destacar, ya que considero de **mayor importancia que el sistema es tolerable a fallos** y permite que, en caso del fallo repentino de un nodo, el resto de nodos continúen atendiendo las nuevas peticiones, asegurando que las anteriores continúen correctamente autenticadas en el sistema.

Se propone como posible mejora del servicio implantado, la generación de un método automático de clonación de nodos de manera que en caso de aumento de peticiones se puedan tener más máquinas tras el balanceador autenticando usuarios. Otra posible mejora a estudiar para nuestro trabajo sería analizar los pros y contras de instalar el servicio de CAS en un Tomcat no embebido, ya que mientras sería práctico en algunos entornos, complicaría la complejidad del despliegue y adaptación a futuras versiones de CAS.

7. Glosario

- **Autenticación**: Proceso utilizado en los mecanismos de control de acceso con el objetivo de verificar la identidad de un usuario, dispositivo o sistema mediante la comprobación de credenciales de acceso.
- **Autorización**: Definición granular de permisos de acceso concedidos a un determinado usuario, dispositivo o sistema, habitualmente implementado mediante listas de control de acceso (ACL).
- **Single Sign-On(SSO)**: Single sign-on (SSO) es un procedimiento de autenticación que habilita al usuario para acceder a varios sistemas con una sola instancia de identificación.
- **LDAP**: Lightweight Directory Access Protocol es un protocolo a nivel de aplicación que permite el acceso a un servicio de directorio ordenado y distribuido para buscar diversa información en un entorno de red. LDAP también es considerado una base de datos (aunque su sistema de almacenamiento puede ser diferente) al que pueden realizarse consultas
- **JVM**: Java Virtual Machine es, un conjunto de programas de software que permiten la ejecución de instrucciones y que normalmente están escritos en código byte de Java. Las máquinas virtuales de Java están disponibles para las plataformas de hardware y software de uso más frecuente.
- **HA**: High Availability o Alta disponibilidad es una aproximación o Diseño que minimiza u oculta a los Usuarios de un Servicio de TI los efectos del Fallo de un Elemento de Configuración. Las soluciones de Alta disponibilidad se diseñan para alcanzar los niveles acordados de disponibilidad y para hacer uso de técnicas como la Tolerancia a Fallos, Resistencia y recuperación rápida para reducir el número de Incidentes y el Impacto de los mismos.FrontEnd: También referenciado como frontales, es la parte de una infraestructura que interactúa con los usuarios
- **Backend**: Es la parte que procesa la entrada desde el front-end
- **LDIF**: LDAP Data Interchange Format. Formato de intercambio de datos de LDAP.
- **CHEF**: Es una herramienta de administración de configuración escrita en Ruby y Erlang. Se utiliza para agilizar la tarea de configurar y mantener los servidores de una empresa.
- **JSON**: JavaScript Object Notation, es un formato de texto sencillo para el intercambio de datos. Se trata de un subconjunto de la notación literal de objetos de JavaScript, aunque, debido a su amplia adopción como alternativa a XML, se considera un formato de lenguaje independiente.
- **Single Bind (Ldap)**: Acción de autenticarse y especificar una versión del protocolo LDAP.

8. Índice de Figuras

Figura 1: Diagrama de Gantt. Planificación temporal. 5
Figura 2: Representación de la Infraestructura de autenticación 6
Figura 3: Ilustración flujo de autenticación de Apereo CAS. Modern authentication techniques in Python web applications. PyGrunn talk by Artur Barseghyan. Year 2014. 8
Figura 4: Gráfico del funcionamiento de un balanceador de carga [3.1] 10
Figura 5: Instalación del Ubuntu Server para las máquinas en VirtualBox 15
Figura 6: Maqueta de máquina virtual Ubuntu Server recien arrancada 16
Figura 7: Configuración de red de la maqueta de máquina virtual Ubuntu Server 17
Figura 8: Proceso de instalación de Openldap. Petición de contraseña de administración. 18
Figura 9: Proceso de instalación de Openldap. 18
Figura 10: Dominio Openldap con la rama dc=uoc,dc=tfg y su administrador cn=admin 19
Figura 11: Fichero en formato LDIF con definición de las ramas de grupos y usuarios. 19
Figura 12: Shell script de generación de usuarios en formato LDIF 20
Figura 13: Definición de versión de CAS y servidor de aplicaciones en pom.xml 22
Figura 14: Definición de dependencias de CAS en pom.xml 23
Figura 15: Proceso de empaquetado del fichero cas.war 26
Figura 16: Acceso a validación de CAS en nodo único 27
Figura 17: Comprobación de parámetros devueltos por CAS tras validación 28
Figura 18: Parámetros devueltos por CAS tras validación con NGINX como proxy inverso. 30
Figura 19: Acceso a validación de CAS en nodo único desde aplicación DemoCAS 30
Figura 20: Acceso a DemoCAS trás autenticación del usuario 31
Figura 21: Monitor de estadísticas de HAProxy con nodo tfg_cas1 activo y tfg_cas2 con error 34
Figura 22: Comprobación desde un navegador de acceso al servicio a través del balanceador 35
Figura 23: HAProxy stats con tfg_cas1 caído. No existe proceso java ejecutando CAS 36
Figura 24: HAProxy stats con tfg_cas1 caído.No existe nginx corriendo en la máquina 36
Figura 25: Definición de CSV de usuarios en en prueba de rendimiento LDAP 44
Figura 26: Bind o enlace con credenciales de cada usuario en prueba de rendimiento LDAP 45
Figura 27: Ldapsearch y recuperación de atributos de usuario en prueba de rendimiento LDAP 45
Figura 28: Rendimiento del nodo LDAP para 500 peticiones en un segundo realizadas 5 veces. 46
Figura 29: Rendimiento de CAS standalone a los 30 segundos del inicio de una prueba continua con 50 usuarios entrando de manera continuada. 47
Figura 30: Rendimiento de CAS standalone tras 3 minutos con 50 usuarios entrando de manera continuada y carga de la máquina tfg_cas1 47
Figura 31: Errores debido a falta de memoria Java en el nodo de CAS standalone para prueba de 800 usuarios en rampa de entrada de 30 segundos. 48
Figura 32: Valores de nodo de CAS standalone con 1Gb para prueba de 1900 usuarios entrando en 30 segundos. 49
Figura 33: Valores de nodo de CAS standalone a través del balanceador con 1Gb para prueba de 1900 usuarios entrando en 30 segundos. 51
Figura 34: Nodo de CAS standalone configurado con replicación de tickets con Redis en prueba de 1800 usuarios entrando en 30 segundos. 51
Figura 35: Nodo de CAS standalone configurado con replicación de tickets con Hazelcast en prueba de 1800 usuarios entrando en 30 segundos. 52
Figura 36: Estado del balanceador con dos nodos de CAS activos 52
Figura 37: Servicio de CAS en HA con replicación de tickets con Hazelcast en prueba de 1800 usuarios entrando en 30 segundos. 52
Figura 38: Servicio de CAS en HA con replicación de tickets con Hazelcast y 3Gb de memoria por nodo en prueba de 2200 usuarios entrando en 30 segundos. 53
Figura 39: Servicio de CAS en HA con replicación de tickets con Hazelcast y 3Gb de memoria por nodo en prueba de 2700 usuarios entrando en 30 segundos. 54
Figura 40: Estado de la máquina host al intentar hacer pruebas de rendimiento más exigentes. 54

9. Bibliografía

CAS en Wikipedia https://en.wikipedia.org/wiki/Central_Authentication_Service
Apereo CAS Official Project https://apereo.github.io/cas/5.3.x/planning/Getting-Started.html

[1] Requisitos CAS
https://apereo.github.io/cas/5.3.x/planning/Architecture.html
[2] CAS en alta disponibilidad
https://apereo.github.io/cas/6.0.x/high_availability/High-Availability-Guide.html
[3] Comparativa de diversos balanceadores de carga
https://www.loggly.com/blog/benchmarking-5-popular-load-balancers-nginx-haproxy-envoy-traefik-and-alb/
[3.1] Comparativa HAproxy vs Nginx
https://www.keycdn.com/support/haproxy-vs-nginx
[3.2] Otra comparativa HAproxy vs Nginx
https://www.freelancinggig.com/blog/2017/04/26/haproxy-vs-nginx-software-load-balancer-better/
[3.2.1] Identity, Authentication, and Access Management in OpenStack by Brad Topol; Steve Martinelli; Henry Nash Published by O'Reilly Media, Inc., 2015
[3.2.2] ¿Por qué elegir LDAP?
https://ldap.com/why-choose-ldap/
[3.4.1] Modelo de servidor elegido
https://www.dell.com/es-es/work/shop/servidores-almacenamiento-y-redes/smart-value-flexi-poweredge-r230-2x35-e3-1220v5-1x8gb-2x1tb-72k-sata-h330-1yr-nbd/spd/poweredge-r230/per2305a
[4.3][4.5] Parámetros de configuración de Apereo CAS
https://apereo.github.io/cas/5.3.x/installation/Configuration-Properties.html
[4.3.1.1] Requisitos de instalación de CAS
https://apereo.github.io/cas/5.3.x/planning/Installation-Requirements.html
[4.3.2.1] Tipos de interfaz de administración de aplicaciones activas en CAS
https://apereo.github.io/cas/5.3.x/installation/Service-Management.html#storage
[4.3.2.2] Interfaz de administración de aplicaciones activas en CAS con JSON
https://apereo.github.io/cas/5.3.x/installation/JSON-Service-Management.html
[4.3.3.1] Repositorio github de la aplicación DemoCAS
https://github.com/cas-projects/cas-sample-java-webapp
[4.3.4.1] Crear certificado SSL SAN (multidominio)
https://support.f5.com/csp/article/K11438
[4.3.5.1] Configuración de un servidor Nginx con proxy inverso
https://www.linode.com/docs/web-servers/nginx/use-nginx-reverse-proxy/
[4.4] Instalación de HaProxy
https://clouding.io/kb/balancear-servicio-web-con-haproxy/
[4.5.1] Tipos de servicio de replicación de tickets para servicios en HA
https://apereo.github.io/cas/5.3.x/installation/Configuring-Ticketing-Components.html
[5.1.1.1] Principales productos destinados a realizar pruebas de rendimiento
http://testorigen.com/loadrunner-vs-jmeter-performance-testing-tools/
[5.1.2.1] Comparativa de funciones de software de pruebas de rendimiento
https://www.edureka.co/blog/jmeter-vs-loadrunner/

[5.2.1] Diseño de una prueba de rendimiento para LDAP con Jmeter
https://jmeter.apache.org/usermanual/build-ldap-test-plan.html
[5.2.2] Ampliación de diseño de una prueba de rendimiento para LDAP con Jmeter
https://jmeter.apache.org/usermanual/build-ldapext-test-plan.html
[5.3.1] Diseño de una prueba de rendimiento para CAS con Jmeter
https://apereo.github.io/cas/5.3.x/planning/Performance-Testing-JMeter.html
[5.3.1.1] Documentación parámetros de memoria de JDK8
https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/sizing.html