Sistema de acceso a los clúster de forma remota

Urko Lekuona
ulekuona001@ikasle.ehu.eus

Donostia International Physics Center

Index

Introducción

La práctica consiste en implementar una forma en la que los investigadores puedan conectarse a los clústers del DIPC de forma remota e interactuar con ellos de diversas maneras intentando limitar las posibilidades lo menos posible para poder simular el uso normal que los investigadores puedan darle. En un principio, la aplicación está orientada a usarla desde un smartphone desde cualquier lugar, aunque en algunas ocasiones también se podrá usar desde un ordenador, como ya se verá más adelante.

1 Análisis del entorno

El DIPC tiene a su disposición una serie de clústers (nosotros solo vamos a trabajar con Atlas, Hemera y Ponto) que permiten a los investigadores realizar diferentes cálculos relacionados con su investigación. Hasta el momento, la forma en la que los investigadores se conectan a estas máquinas para trabajar con ellas es mediante el protocolo SSH. Para que se pueda acceder a los clúster desde cualquier lugar pero no tener problemas de seguridad, el acceso a las máquinas solo es posible mediante un servidor 'cortafuegos' llamado AC.

En definitiva, para poder conectarse a un clúster, un investigador tiene que conectarse primero a AC y desde ahí volver a conectarse a un clúster específico. Esto es importante a la hora de implementar nuestro sistema, ya que habrá que elegir un método que soporte conexiones SSH y nos permita alcanzar los clúster a través del 'cortafuegos' mediante una conexión SSH anidada.

Como el sistema está orientado a ser usado desde un smartphone, la alternativa principal que se planteó fue crear una aplicación (en este caso para Android), publicarla y hacer saber a los investigadores que este servicio existía. Sin embargo, también surgió la idea de los 'bots' de Telegram. Estos bots son aplicaciones que se están ejecutando en un servidor que, mediante la API que ofrece Telegram, pueden ofrecer una gran variedad de servicios, siempre que la API lo permita, claro está.

Estas dos opciones son las principales que se han considerado. Las dos tienen sus ventajas y sus desventajas, que las vamos a ver a continuación:

1.1 Android

La aplicación de Android era la elección más obvia. Implementándolo de esta manera, podríamos evitar el esquema Cliente<->Servidor<->Servidor, ya que cada aplicación podría conectarse directamente con el servidor AC. Además, se podría formatear la apariencia de la aplicación a gusto y organizarlo todo respecto a nuestros intereses.

Aun así, implementar una aplicación en Android tiene ciertas desventajas. Por un lado, habría que elegir una versión de Android para la que desarrollar. Dependiendo de la versión para la que se implemente, determinados smartphones no podrían usar la aplicación si se elige una versión muy reciente, o perderíamos muchas funcionalidades y daría una apariencia desfasada a la aplicación, utilizando versiones más antiguas. Por otro lado, tener que publicar la aplicación puede dar ciertos problemas. Lo mas sencillo sería distribuirla vía Play Store, pero para esto habría que pagar a Google una licencia y la aplicación tiene que cumplir ciertos estándares. Para finalizar, desarrollar una aplicación para Android pero no para iOS sería marginar una plataforma que es tan o más común que Android.

1.2 Bot de Telegram

Aunque en principio un bot de Telegram pueda parecer decepcionante o con poco fundamento, si se estudia un poco se puede ver que es una herramienta bastante potente, aunque es importante tener en cuenta sus limitaciones en todo momento. Estos bots funcionan prácticamente con el esquema Cliente<->Servidor, solo que los mensajes se envían mediante una API que se nos proporciona. Una de las grandes ventajas que ofrece, que a la vez es una de las mayores desventajas, es que hay que olvidarse completamente de la implemetanción de la parte del cliente. Esto facilita mucho las cosas, ya que permite centrarse en la parte del servidor y olvidarse del protocolo, porque habrá que usar el protocolo de la API de Telegram.

A su vez, al funcionar mediante una API, Telegram nos permite elegir el lenguaje en el que queramos implementar el servidor, lo cual puede facilitar mucho todo el proceso y también permite que añadamos funcionalidades más complejas que quizás no hubiésemos sido capaces de implementar en otro lenguaje. Otra ventaja de esta opción es el poder usar la aplicación de escritorio de Telegram, haciendo viable el uso de nuestro sistema también desde un ordenador.

Las desventajas de esta elección tienen que ver sobre todo con la parte del cliente. Como ya se ha comentado, no se puede modificar apenas la parte del cliente. Al ser un chat, funciona mediante mensajes y es la única manera que tiene el bot de interactuar con el cliente. Es cierto que estos mensajes pueden ser fotos, audios, vídeos o documentos, pero sigue siendo una limitación importante respecto a una aplicación en Android. Otra desventaja sería la API como tal. Esta API impide el envío de mensajes de más de 4096 caracteres UTF8 (esto es un problema para una funcionalidad que nos interesa que se explica más adelante), bloquea al bot si envían más de 30 mensajes por segundo (puede ser un problema si hay muchos usuarios utilizándolo al mismo tiempo) o tiene retardos en el envío de mensajes dependiendo de problemas externos a nuestra aplicación (por ejemplo, que pueda estar saturado el servidor de Telegram). Otro inconveniente quizás menos obvio es el de la seguridad. Para conectarse a los clúster hacen falta unas credenciales personales. Obviamente, mandar este tipo de datos mediante un mensaje de chat no es lo más seguro. No tanto porque Telegram pueda almacenar estos mensajes (que ya de por sí puede ser razón suficiente porque es información bastante sensible), sino porque cualquiera que cogiese nuestro móvil o estuviese cerca mientras enviamos el mensaje tendría acceso directo a nuestra cuenta.

1.3 Decisión

Habiendo considerado las ventajas y desventajas de cada opción, se ha elegido como alternativa más viable el bot de Telegram. Puede que no haya dado esta sensación a la hora de listar los puntos de cada uno, pero mientas las desventajas de Telegram se consideran "solucionables" (se puede encontrar remedios o puede que no den problema para el tipo de uso que se le va a dar), las desventajas de Android no son así. Aun así, se ha acabado implementando también una aplicación para Android para poder resolver ciertas limitaciones de Telegram, además de dar a los investigadores una alternativa.

2 Desarrollo

2.1 Creación del Bot

Lo primero que habrá que hacer para desarrollar cualquier bot de Telegram será crearlo. Para esto es necesario una cuenta de Telegram. Una vez estemos dentro de Telegram, habrá que buscar en el buscador de chats un contacto llamado 'BotFather'. Este bot es el 'Bot Maestro' de Telegram. Desde aquí se pueden crear o eliminar bots, gestionarlos, editarlos... Si iniciamos la conversación con él, tendremos que introducir el comando '/newbot' para crear un nuevo bot y seguir los pasos que nos indique (nombre y nombre único). Al finalizar, BotFather nos proporcionará un token único para nuestro bot. Muy importante apuntar bien este token y no compartirlo con nadie, ya que para leer los mensajes que le llegan al bot es imprescindible, y si algún otro bot está utilizando el mismo token que nosotros, los dos bots dejarían de funcionar correctamente. Una vez finalizado el proceso, el bot estaría creado y podremos acceder a él buscándolo por su nombre único en el buscador de Telegram. Se recomienda probar a enviar algún mensaje al bot en este momento ya que, aunque no nos vaya a responder porque no hay un servidor que esté procesando esos mensajes, mediante esta URL: `https://api.telegram.org/bot<TOKEN>/getUpdates` (sustituir <TOKEN> por el token de nuestro bot) podremos ver en formato JSON el contenido de los mensajes. Importante mencionar también que en el momento en el que algún bot lea un mensaje de la API, dicho mensaje se borrará de esta URL.

Desde el chat de BotFather también se recomienda utilizar los comandos '/setabouttext', '/setuserpic' y '/setcommands', para poder diferenciar al bot de otros y que los usuarios tengan una experiencia correcta. El comando '/setcommands' facilita mucho el uso del bot auto-completando los comandos que se introducen, aunque es probable que no seamos capaces de configurarlo desde un principio a no ser que sepamos desde el primer momento el listado de comandos que tendrá el bot.

2.1.1 Implementación del servidor

Para crear la parte servidora del bot en Python no es necesaria gran cosa. Con tener una versión de Python compatible con alguna de las bibliotecas que implementan la API de Telegram y una conexión a Internet sería suficiente. Además, está compuesto por solo 3 ficheros que no ocupan mucho (uno de ellos es de configuración y el otro contiene respuestas que se enviarán al cliente) por lo que es muy portable. Aun así, al ser nuestro bot algo más complejo que la mayoría de bots, son necesarias varias bibliotecas adicionales que no vienen preinstaladas con Python.

Asimismo, para solucionar el problema de las contraseñas enviadas por chat, se ha puesto en marcha un servidor web que se encargue de guardar las credenciales. Sobre este servidor se hablará con más detalle en otra sección, pero es importante saber que utiliza Apache2 y bibliotecas de PHP. Por todas estás razones, se ha decidido que la mejor forma de implementar el servidor del bot sería en una máquina virtual, ubicada en el clúster de servicios del DIPC.

Para simplificar el proceso, hacerlo fácil para reconfigurar y evitar problemas en la instalación (sobre todo de red), se ha decidido usar Vagrant. Vagrant nos permitirá usar una de sus "boxes" (máquinas virtuales configuradas de determinada manera para usarlas con Vagrant sin esfuerzos) y simplificar la instalación. En nuestro caso, se ha elegido la box 'ubuntu/trusty64' aunque hay muchas otras que hubieran válido también. Esta box contiene Ubuntu 14.04.

Para instalar Vagrant y configurar una máquina, se recomienda utilizar el manual que ellos ofrecen: `https://www.vagrantup.com/intro/getting-started/index.html`. Aquí solo se subrayarán los puntos más importantes y las diferencias entre nuestra instalación y la que ellos utilizan como ejemplo en su tutorial. El fichero Vagrantfile (ubicado en la carpeta en la que se haya ejecutado el comando 'vagrant init') es el que contiene todas las opciones de configuración de nuestra máquina. Las siguientes son las líneas que ha habido que modificar o añadir al fichero:

- config.vm.box = "ubuntu/trusty64"
- config.vm.network "public_network", ip: "158.227.173.195", netmask: "255.255.254.0"

La primera línea indicará a Vagrant cuál es la box que tiene que utilizar para ejecutar esta máquina. La segunda línea dará una IP pública a la interfaz de red de la máquina virtual, importante para poder acceder al sitio web que creemos.

El fichero Vagrantfile también dispone de una opción llamada provisioning, que permite ejecutar ciertos comandos en la máquina virtual cuando se inicia o se crea. Esto se suele usar para instalar paquetes o cambiar configuración. Nosotros no lo hemos usado y hemos colocado estos comandos en un script aparte llamado 'configure.sh'. Este fichero contendrá los comandos para instalar todos los paquetes necesarios para poder usar tanto el bot de Python como el servidor web.

Si hemos instalado la máquina virtual correctamente, ahora habrá que crear los ficheros del bot. Estos ficheros se recomienda ubicarlos en la carpeta '/var/www/***' (*** se puede reemplazar por un nombre cualquiera, no es importante). La ubicación es importante porque el bot usará información de otros ficheros y si lo colocamos en otro lugar, habrá que modificar el código del bot para que siga haciendo referencia a los mismos ficheros.

El servidor está compuesto por tres ficheros de extensión '.py'. 'allMessage.py' contiene mensajes que se utilizan en diferentes puntos del programa principal, para enviar como respuesta al cliente. 'config.py' contiene diferentes ajustes del servidor, como la dirección host por defecto del SSH o el token del bot. El fichero 'bot.py' es el programa principal y el que habrá que ejecutar utilizando el comando 'python bot.py'. El programa tiene una estructura poco común y como en un futuro puede que se quiera modificar alguna funcionalidad o incluir nuevas, se explicará el código más detenidamente para poder entenderlo.

2.1.2 Funcionamiento del programa

Al principio del programa estarán importadas todas las bibliotecas que se utilizan, de aquí no hay mucho que destacar exceptuando la línea que dice 'import telebot'. Telebot es una de las muchas implementaciones que existen para Python de la API de Telegram. El link al repositorio de la biblioteca

es este: `https://github.com/eternnoir/pyTelegramBotAPI`. Las razones para haber escogido esta en vez de otra son dos: Por un lado, esta es la segunda más popular, nos ofrece todo lo que necesitamos y sigue estando en desarrollo. Por otro lado, es la opción que utilizaba 'vzemtsov' en su bot (`https://github.com/vzemtsov/ssh-Connection-In-Telegram`). Este bot es el que se ha utilizado como base para el proyecto, ya que también utiliza conexiones SSH.

En el programa se utiliza una clase llamada User, que contiene toda la información de un usuario concreto. Esta información se carga cuando el usuario hace login y algunos datos (como el clúster o la dirección actual) se modifican a medida que se usa el bot. Todos estos usuarios se guardan en una lista llamada 'knownUsers' y se utiliza el id del chat del usuario como identificador. La variable 'bot' hace referencia a nuestro bot y es un objeto de la implementación de la API de Telegram que usamos. Para utilizar cualquier función de la API (send_message() o message_handler() por ejemplo) habrá que hacer referencia a él. Por último, para hacer referencia al mensaje recibido se suele llamar 'message' a la variable que contiene dicho mensaje. Estos cuatro elementos son bastante utilizados en toda la aplicación y es importante entenderlos.

Al final de programa se encuentra la función 'main'. Esta función no es muy importante para entender el funcionamiento del programa ya que casi todo el bot consiste en detectar un evento (la llegada de un nuevo mensaje) y procesarlo de la manera que le corresponda. Todo esto se hace con las funciones 'message_handler()' que se explicarán más tarde, pero hay que decir que la función 'main' se utiliza para inicializar el programa nada más. La línea 'bot.polling(none_stop=True)' es la que hace que el bot se ponga en marcha y empiece a esperar mensajes de la API.

Connect Unas funciones que juegan un papel importante en el bot son las de conexión con el servidor AC mediante SSH. Hay tres de ellas:

- **connect**: Esta función se conecta al servidor AC, usando el cliente de la biblioteca 'paramiko'.
- **connect_cluster**: Esta función se conecta a un clúster en concreto (el que haya elegido el usuario). Al tener que conectarnos mediante el AC, lo que se hace es llamar a la función 'connect' y abrir un nuevo canal con el transporte de la conexión a AC. Después nos conectaremos al clúster al igual que en la función 'connect', solo que utilizando como socket el nuevo canal que hemos creado. Esta solución se ha encontrado en **esta página**.

  En un principio se había pensado usar SSHPass para conectarse, pero tiene varios incovenientes y no tiene sentido usarlo sabiendo que se puede hacer de la manera en que ya se ha explicado.
- **connect_cluster_sftp**: Funciona igual que 'connect_cluster', solo que crea un cliente SFTP del transporte del cliente SSH al clúster. Esta función se utilizar cuando haya que obtener un fichero de algún clúster. En nuestro caso para conseguir informes de DCRAB.

Message Handlers Los 'message_handler' son las funciones que forman el grueso del programa. La mayoría son bastante sencillas y es fácil entender su funcionalidad. Sobre los 'message_handler' hay que explicar algunas cosas:

- Se ejecutan en el orden en el que están escritos. Esto es, en cuanto llega un mensaje, empezará a comprobar si cumple las condiciones del primer 'message_handler', después del segundo, del tercero, ..., así hasta que encuentre un 'message_handler' al que satisfaga. En este momento, se ejecutará la primera función definida después del 'message_handler' y, en cuanto termine, se parará de procesar ese mensaje y se volverá al inicio. **No sigue comprobando si cumple las condiciones de los siguientes**.
- Se puede forzar que el siguiente mensaje sea procesado por una función en concreto. Esto se hará con 'register_next_step_handler'. Esto es útil para algunos comandos que puedan necesitar de varios mensajes, como el de DCRAB, donde primero se pide el comando y más tarde el id del trabajo que se quiere obtener.
- Para que un 'message_handler' procese un mensaje, el mensaje tendrá que cumplir ciertas condiciones que nosotros introduzcamos. A menudo, esto se ha realizado con las funciones lambda de

Python. Son muy sencillas pero no es necesario entenderlas del todo. Basta con saber que el mensaje deberá cumplir la condición lógica, esto es, que devuelva 'True'. Además, un 'message_handler' puede tener varios tipos de condiciones diferentes. Estos están listados y explicados en el siguiente enlace: `https://github.com/eternnoir/pyTelegramBotAPI#message-handlers`

Un 'message_handler' a destacar sería el del comando 'connect'. Este comando lo utilizamos para elegir el clúster al que nos queremos conectar. Para evitar tener que escribir el nombre del clúster y para poder ver cuáles están disponibles, lo que se hace es usar algo llamado 'ReplyKeyboardMarkup'. Esto nos permite añadir una lista de elementos a un teclado personalizado y hacer que este se le muestre al usuario. Importante borrar este teclado una vez haya servido su función.

La función 'message_handler' quizá más engorrosa y complicada de entender es la que procesa el comando 'queue'. El propósito de esta función es mostrar al usuario la lista de trabajos suyos de la cola de trabajos del clúster al que está conectado. Esto se consigue ejecutando el comando 'showq' de maui. El problema es que la salida de este comando está formateada para ser vista en una pantalla de escritorio, y desde luego no para enviarla por un mensaje de texto en una aplicación de un smartphone. Por esta razón, se ha tenido que modificar el mensaje recibido para que el mensaje final que vea el usuario sea lo más legible posible perdiendo la menor cantidad de información posible.

Para conseguir esto, ha habido que eliminar la columna del usuario que está realizando el trabajo, ya que siempre va a ser la del usuario que está pidiendo que se le muestre la cola, y modificar la columna de tiempo de inicio y tiempo de cola, borrando el año y el día de la semana y modificando el mes y la hora, y dejar solo las iniciales del estado del trabajo (R, I o B). Todo este proceso y más se hace de una manera muy particular, pensada específicamente para procesar el comando 'showq' de maui.

Otro problema con esta función es que la API de Telegram no permite mandar mensajes de más de 4096 caracteres UTF8. Este límite puede ser fácilmente sobrepasable por un usuario que tenga varios trabajos en cola. Por esto se ha tenido que crear una manera de cortar el mensaje resultante en trozos, sin romper el formato. En mi caso específico, se ha separado el mensaje en tres (Active, Idle, Blocked) y en cada uno de ellos se busca el carácter 3000 y se corta en el siguiente salto de línea y se envía. **Atención**: La API de Telegram no puede garantizar el orden de recibo de los mensajes que se envían. Esto quiere decir que puede que el usuario reciba la lista desordenada. En los casos de prueba que se han realizado, y han sido bastantes, no ha habido una sola vez en la que los mensajes no hayan llegado en orden. Aún así, es importante tenerlo en cuenta y quizás implementar una solución, como numerar los mensajes o esperar a que el usuario reciba un mensaje para enviar el siguiente.

Los últimos dos 'message_handler' que se van a explicar son los que sirven para ejecutar comandos tanto en AC como en los clúster. Como funcionan de la misma manera, solo se comentará uno de ellos. De estas funciones se pueden distinguir tres partes:

- **Comandos 'cd'**: Como ya se ha explicado, cada vez que se quiera ejecutar un comando mediante SSH, habrá que abrir una conexión, ejecutar el comando, recoger la salida y cerrar la conexión. El problema de este sistema es que cada vez que abramos una nueva conexión, el directorio en el que estaremos ubicados se reinicia al directorio por defecto. Esto no es un problema para comandos como 'showq' o 'cu', pero si se quieren realizar varios comandos uno detrás de otro en un directorio que no sea el 'home', habrá que realizarlo de otra manera.

  La solución que se ha aplicado ha sido guardar las rutas en las que se encuentra cada usuario en unos atributos de la clase 'User' (Uno para AC y otro para los clúster). Por esta razón, cada vez que un usuario intenta ejecutar el comando 'cd', no se ejecuta dicho comando en el servidor, sino que se almacena la nueva ruta. De esta manera, cada vez que se quiera ejecutar un comando, no solo se ejecutará dicho comando, sino que también un 'cd' al directorio en el que está trabajando el usuario ('cd <directorio>; <comando_a_ejecutar>').

  Así, cuando se reciba un comando del usuario, habrá que comprobar si es el comando 'cd' o no. Si lo es, habrá que trabajar con él, también de diferente manera dependiendo del contenido. Por ejemplo, si es una ruta absoluta habrá que sobrescribir la antigua ruta, mientras que si es relativa

habrá que añadir la nueva ruta a la antigua. De todas las diferentes posibilidades, la más complicada de implementar ha sido la de los comandos que intentan acceder a un directorio padre (los que empiezan por '..'). Esto es así porque puede que no solo quiera acceder al directorio padre, sino al padre del padre, o más aún. La forma actual de implementarlo ha funcionado correctamente en las pruebas que se ha realizado, pero aun así no se puede asegurar que esté libre de bugs.

- **Comandos no permitidos**: Al no trabajar con una terminal y ser la salida mensajes de texto de Telegram, no es posible ejecutar ciertos comandos y esperar que funcionen correctamente. La mayoría de estos comandos son denominados 'interactivos' (como editores de texto o comandos estilo 'more') aunque hay otros que tampoco funcionan siempre (el comando 'ping' no termina de ejecutarse sin un argumento específico, por ejemplo). Para evitar que estos comandos se intenten ejecutar e interrumpan el funcionamiento normal del programa, se ha creado un archivo llamado 'command_whitelist.txt' que contiene la lista de comandos que **sí** se pueden utilizar. Esta lista es ampliable pero se recomienda tener cuidado y tener en cuenta los distintos argumentos del comando que se quiera añadir, ya que alguno de ellos puede hacer que el comando se comporte de alguna forma no permitida.
- **Comandos permitidos**: En caso de que el comando esté permitido, habrá que juntarlo con la ruta en la que se encuentra el usuario y ejecutar dicho comando. Algunos comandos pueden no tener una respuesta (como el comando 'chmod'), por lo que se enviará al usuario un mensaje avisándole de que el comando ha sido ejecutado.

Login Como ya se ha comentado antes, usar el bot de Telegram supone un problema de seguridad, ya que los usuarios tienen que introducir sus credenciales de alguna manera y hacerlo desde un chat no es lo más recomendable. Como solución, se ha implementado un servidor web con Apache2 en la máquina virtual en la que está el bot. En este servidor alojaremos un fichero PHP llamado login.php que permitirá al usuario introducir la contraseña de forma más segura (utilizando HTTP y formularios HTML, mucho mejor que escribirlo en un chat de mensajes de texto).

La forma en la que todo este sistema funciona es la siguiente:

1. Se escribe el comando '/on' en el bot. El bot generará un token alfanumérico aleatorio y le mandará un mensaje al usuario con el enlace a login.php y el token (http://bot-telegram.sw.ehu.es/login.php ?token=<TOKEN>). A la vez, el bot guardará dicho token y el id del chat de usuario al que está asociado en un fichero XML llamado tokens.xml que estará en el directorio /var/www/html (mismo directorio que login.php).
2. El usuario accederá a dicho enlace, donde se le mostrará un formulario de login básico (usuario y contraseña). El usuario deberá introducir los credenciales de la cuenta que utiliza para conectarse vía SSH. El servidor web almacenará dichos credenciales en un fichero XML llamado credentials.xml en la misma ruta que login.php. Además de estos datos, también se guardará el id del chat asociado al token que se haya recibido mediante la URL. Como el token ha sido utilizado, se borrará de tokens.xml el token y su chat.
3. Al iniciar el bot se pone en funcionamiento una clase 'Observer' que durante toda la ejecución del bot estará a la espera de que credentials.xml sea modificado. En el momento en el que sea modificado este fichero, el 'Observer' leerá el contenido del fichero y creará un usuario con él. Después borrará el contenido del fichero credentials.xml. De esta manera, los usuarios se cargan automáticamente en el programa una vez hayan hecho login, aunque es necesario que esté el bot en funcionamiento.

2.2 Posibles mejoras para el bot

En este apartado se listarán ciertas mejoras que se pueden aplicar al bot para intentar optimizarlo lo máximo posible o hacerlo más sintácticamente correcto. Estas mejoras no añadirán nada nuevo a la funcionalidad del bot como tal, aunque pueden hacer que responda mejor en determinadas circunstancias.

- Una posible mejora para el rendimiento general del bot, y que posiblemente sea obligada si el bot fuese a ser usado por muchos usuarios al mismo tiempo, sería separar la estructura del bot en dos, haciendo que una parte se encargue del programa principal y de recibir los mensajes, mientras que

la otra los procesaría y respondería a los usuarios. Esta estructura es muy común en servidores TCP con conexiones concurrentes, para evitar que el servidor se bloquee con una única petición.

Para realizar este cambio se han estudiado las diferentes alternativas que ofrece Python y se han considerado las más adecuadas dos: 1. Usar la función os.fork() para crear un programa hijo cada vez que se reciba una petición y 2. Utilizar la clase 'Process' de la biblioteca 'Multiprocessing' de Python. Quizá la más adecuada de las dos sería la segunda, ya que internamente utiliza os.fork() también y es de más alto nivel, aunque es recomendable probar las dos y comparar. El problema principal para implementar este nuevo esquema es hacer que los procesos hijos no reciban mensajes de la API, ya que sería efectivamente igual que tener dos o más bots recibiendo peticiones del mismo Token y, como ya se ha dicho, esto no es posible y recibiríamos continuos errores de la API. Esto se puede conseguir parando el sondeo constante que hace el objeto 'bot' con la funcion 'bot.polling(none_stop=True)' o desactivando los 'message_handler'. Además de todo esto, sería necesario 'matar' los procesos hijos una vez acaben su función y recoger su estado de salida para evitar procesos 'zombie'.

- Al haber implementado las funciones que formatean la salida del comando 'showq' personalmente, es posible que la forma en la que se ha hecho no sea la más correcta, ni en cuanto a rendimiento ni en cuanto a sintaxis (sobre todo siendo Python tan diferente a otros lenguajes en cuanto a tratamiento de 'strings').
- Pasa lo mismo con la forma de procesar el comando 'cd ..'. Es posible que haya formas mejores en las que implementarlo, aunque la actual funcione correctamente en los casos de prueba que se han realizado.
- Por falta de tiempo, no se ha podido implementar la funcionalidad que se iba a encargar de recibir los correos de torque/maui y enviárselos a los usuarios. Lo que si se ha hecho es estudiar las diferentes formas en las que se puede hacer. La mejor solución que se ha pensado ha sido establecer un servidor de correo al que mandar todos los mensajes de torque/maui, que sería uno de ya implementó en su momento otro estudiante de prácticas. Desde nuestro bot se recibirán los correos almacenados en dicho servidor de correo y se enviará cada uno a su respectivo usuario, siempre que tenga una sesión abierta en el bot. Además. habrá que hacer una gestión del servidor de correos desde el bot, para poder eliminar los mensajes que ya se han procesado o marcarlos como leídos. Para hacer todo esto desde Python, la mejor alternativa sería utilizar la biblioteca 'imaplib'. Sin embargo, habiendo leído la práctica del alumno que implemento el servidor de correo, se cree que dicho servidor no soporta IMAP, por lo que probablemente haya que usar el protocolo POP3 con la biblioteca 'poplib' de Python.
- Actualmente, el servidor web está alojado en el servicios-15 del clúster de servicios. Este clúster es bastante restrictivo en cuanto a conexiones externas por lo que de momento el servidor web solo es accesible desde dentro de la red de la UPV o mediante VPN. Esto no es muy cómodo para que un usuario pueda conectarse desde casa, por lo que se considera una mejora implementar todo el sistema en algún lugar más accesible.

2.3 Aplicación para Android

Para solventar algunas de las limitaciones del bot de Telegram y dar a los usuarios otra alternativa de uso y que cada uno pueda elegir la forma de acceso a los clusters que prefiera, se ha decidido implementar también una aplicación para Android con las mismas funcionalidades que el bot. En este apartado se explicará como funciona esta aplicación, parte del código (estará explicado con más detalle en el propio código) y como utilizarla. Para poder centrarnos en lo relevante, se va a suponer que se tienen nociones básicas de programación para Android con Java.

LoginActivity.java y activity_login.xml
La aplicación empieza con una actividad de Login. Esta actividad se ha creado utilizando una plantilla de Android Studio que implementa ya un formulario de inicio de sesión y algunas funciones que tratan con los datos del formulario. Esta plantilla ha sido modificada para que satisfaga nuestras necesidades. El layout está compuesto por una imagen con el logotipo del DIPC, una barra de progreso que está oculta y se

mostrará cuando intentemos iniciar sesión, dos campos de texto para el usuario y contraseña, y un botón que utilizaremos para intentar iniciar sesión. En cuanto se pulse el botón, se llamará al método attemptLogin(), que si alguno de los campos no es válido antes de comprobar las credenciales con el servidor. Si los campos son válidos, se ejecutará una tarea asíncrona en otro hilo que se encargará de conectarse con el servidor AC y alguno de los cluster. Estas tareas asíncronas están compuestas de tres partes que siempre se van a ejecutar en orden: onPreExecute(), doInBackground() y onPostExecute(). Algo importante que hay que mencionar sobre las actividades asíncronas es que no es posible modificar elementos de la interfaz gráfica de la aplicación desde doInBackground(). onPreExecute() y onPostExecute() son los que se encargan de realizar estos cambios antes y después de la ejecución de esta tarea. En el apartado onPreExecute() de esta tarea asíncrona solo nos encargaremos de hacer la barra de progreso visible para hacer saber al usuario que se está realizando una conexión. En doInBackground(), intentaremos abrir una conexión SSH utilizando la biblioteca JSch. La forma de hacerlo es un poco más compleja que una conexión SSH estándar utilizando esta librería. El objetivo es conseguir conectarnos a alguno de los cluster y extraer la lista de cluster a los que el usuario que está intentando iniciar sesión tiene acceso.

Para empezar, abriremos una sesión con el servidor AC en el puerto 22. A continuación nos intentaremos conectar con un cluster específico y si no somos capaces lo intentaremos con el siguiente hasta conseguirlo o que seamos incapaces de hacerlo con ninguno de los tres. Cada una de estas conexiones funciona de la siguiente manera: 1. Borraremos la redirección del puerto local 2233 si esta redirigido, 2. Redirigiremos el puerto local 2233 al puerto 22 del cluster al que intentamos conectarnos, 3. Nos conectamos con AC e abrimos una segunda sesión desde el puerto local 2233 que estará redirigido al cluster, 4. Extraemos la lista de servidores a los que tiene acceso el usuario utilizando un método de la clase Utils que explicaremos más tarde, 5. Cerramos las sesiones que hemos abierto.

Dependiendo del resultado de la función doInBackground(), habrá dos posibles conclusiones que trataremos en onPostExecute(): 1. El intento de login ha sido satisfactorio o 2. El intento de login ha fallado. Si ha fallado, se avisará al usuario con el respectivo fallo que haya ocurrido y nos mantendremos en la actividad actual. Si ha sido satisfactorio, habrá que abrir una nueva actividad y enviar a ella cierta información como los credenciales y la lista de servidores.

ResourcesActivity.java, activity_main.xml, nav_header_main.xml y drawer_view.xml
Una vez hayamos sido capaces de iniciar sesión, se nos redirigirá a la actividad de recursos. En está actividad un usuario podrá consultar la información sobre sus créditos (consumidos y cuota máxima), así como la cantidad de almacenamiento que ha utilizado en su directorio home y en su directorio scratch. El layout de esta actividad es muy diferente del de la actividad de login, pero muy parecido al del resto de actividades de la aplicación. En el fichero main_layout.xml las vistas no están ordenadas de la manera en que se van a explicar porque dependiendo del orden, unos elementos se cargarán encima de otros y alterará la apariencia final de la actividad.

En la parte superior de esta actividad se puede ver un elemento 'toolbar' que tiene el nombre del DIPC y un botón que abrirá un menú desplegable (este menú también se puede abrir arrastrando el dedo desde el borde izquierdo de la pantalla hacia la derecha). El menú desplegable está compuesto por dos grupos de elementos: la cabecera y la lista de elementos. El contenido de la cabecera está en el fichero nav_header_main.xml. que está compuesto por una imagen con el logo del DIPC y un campo con el nombre del usuario conectado. La lista de elementos se encuentra en el fichero drawer_view.xml y contiene dos elementos: Recursos y Informe DCRAB. Además de estos dos, añadiremos la lista de clusters disponibles de forma dinámica. Para terminar de listar los elementos de la actividad de recursos habrá que explicar que tiene dos secciones (una superior y otra inferior). En cada una de ellas habrá una barra de progreso que muestra que se está cargando información. La inferior solo esta compuesta por un par de vistas con texto que rellenaremos con el espacio de almacenamiento del usuario. La superior tiene una vista de texto que servirá para indicar la información de los créditos del usuario, pero además se ha añadido un gráfico de la biblioteca AnyChart para Android.

En cuanto al código de la actividad, hay ciertas cosas que destacar. Algo que no se ha explicado aun pero que funciona igual en el resto de actividades exceptuando la de login es la forma en la que se rellena el menú desplegable con la lista de clusters disponibles. De la lista de clusters que hemos recogido al hacer

login, nos quedaremos con los que sean "ATLAS", "HEMERA" o "PONTO" y añadiremos un nuevo elemento a la lista de elementos del menú desplegable por cada uno (ids 1, 2 y 3). Después habrá que modificar el listener que actúa cuando se elige un elemento de la lista de elementos y hacer que por cada elemento se inicie una actividad nueva con los parámetros que se necesiten (en el caso de los cluster se necesita saber el nombre del cluster porque la actividad será la misma para los tres y hay que diferenciarlos de alguna forma), a no ser que el elemento elegido sea el de la actividad actual, en cuyo caso se cerrará el menú. En la tarea asíncrona de esta actividad extraeremos cuatro tipos de información: consumo realizado, máximo consumo, espacio utilizado en scratch y espacio utilizado en home. La forma en la que se añade información al gráfico está basada en el **ejemplo sobre gráficos de columnas del repositorio de AnyChart**.

QueueActivity.java, activity_queue.xml, list_row_child.xml y list_row_group.xml
Si seleccionamos uno de los cluster desde el menú desplegable accederemos a la actividad de visionado de la cola de trabajos. Esta actividad tiene como objetivo mostrar al usuario la cola de trabajos del cluster seleccionado con información sobre los trabajos del usuario únicamente. La lista se separa en tres sublistas por lo que se ha decidido mostrar la información en una lista extensible. Otra alternativa que posiblemente sea una mejora pero que no se ha conseguido implementar y que cambiaría la forma de funcionamiento de la actividad es incluir esta lista extensible en el menú desplegable y que dentro de la actividad solo se encuentre la una lista con el tipo del trabajos escogido del cluster escogido. Para utilizar una lista extensible hay que implementar un adaptador, en nuestro caso ExpandableListViewAdapter.java (extraído de una guía sobre la listas extensibles).

Sobre el layout de esta actividad, hay que decir que es muy parecido al de la actividad de recursos. En vez de tener el contenido dividido en dos partes, se mantiene todo en una sola, donde se encuentra la barra de progreso de carga y la lista extensible. El formato de la lista está definido en los ficheros list_row_group.xml y list_row_child.xml, donde se da el formato de las cabeceras de las sublistas y de cada elemento de las sublistas, respectivamente. Otra diferencia respecto a la actividad de recursos es que se ha incluido un botón para actualizar la información de la lista.

Respecto al código de la actividad, se puede decir que es igual en estructura y funcionamiento que la actividad de recursos. La única diferencia notable sucede en la tarea asíncrona. En el método doInBackground() de esta tarea, se extrae un string con el estado en cola de los trabajos del usuario. En el método onPostExecute(), se trata este string, eliminando lineas en blanco, separándolo en tres partes y añadiendo estas listas a la lista extensible.

DcrabActivity.java, activity_dcrab.xml y dcrab_report_table.xml
La última actividad implementada en la aplicación es DcrabACtivity.java. Esta actividad se encarga de recoger un informe DCRAB de un trabajo en ejecución del usuario y mostrárselo. Para acceder a la actividad habrá que seleccionar el elemento 'DCRAB reports' del menú desplegable. Esta actividad se diferencia del resto en que tiene dos estados (el estado de selección del trabajo y el estado de visualización del informe DCRAB). El hecho de tener dos estados hace que el funcionamiento de la actividad sea algo más complejo, aunque la estructura general siga siendo la misma. Para mostrar el informe se han implementado tres alternativas:

- Descargar el informe con formato HTML y abrirlo en el navegador por defecto del usuario. Para implementar habrá que ser capaz de eliminar los anteriores informes DCRAB (para evitar saturar el espacio de almacenamiento del usuario), guardar el nuevo y leerlo. Para conseguir hacer todo esto habrá que solicitar permisos de lectura y escritura en disco, implementar una clase FileProvider de Android y añadir una etiqueta '<provider>' a nuestro AndroidManifest.xml para poder utilizar el fichero que nos hemos descargado (a partir de la API 24 es obligatorio, mas información **aquí**). Además de los problemas de implementación que todo esto representa (habría que tener en cuenta todos los posibles errores que puedan surgir y solventarlos), el informe DCRAB en formato HTML no está pensado para ser visualizado en un smartphone, por lo que el resultado final sería subóptimo.
- Descargar el informe con formato HTML y abrirlo en un 'WebViewer'. Los problemas de implementación son exactamente los mismos que los de la opción anterior. Esta alternativa hace mas

cómoda la visualización del informe ya que no hay que salir de la aplicación para verlo, pero el navegador que utiliza Android en el 'WebViewer' suele ser menos potente que el resto y el informe tiene una apariencia aun peor.

- Crear un informe DCRAB con formato de texto y visualizarlo en un TableLayout. Para poder utilizar esta opción habrá que crear el informe en formato de texto con un formato específico. Se ha decidido que el formato sea: '<Nombre del campo> = <Valor del campo>;'. En los clúster debería haber un informe llamado '/scratch/urkole/dcrab_report.txt' que se ha utilizado para hacer pruebas y que probablemente ayude mejor a entender el formato del fichero.

La mejor opción probablemente sería utilizar una combinación de estas opciones. Mostrar la tercera opción y dar la opción al usuario de visualizar también el informe en HTML si así lo desean, ya sea con la primera o segunda opción.

El layout de la actividad es igual al del resto en el primer estado, siendo el contenido un 'spinner' de tipo 'dropdown' que contendrá la lista de trabajos activos del usuario. En el segundo estado varía dependiendo de la forma de implementación escogida para visualizar el informe, aunque se va a asumir que se ha escogido la que utiliza el informe en formato de texto. Para poder adaptarnos al número de campos del informe y al tamaño de cada campo, se ha envuelto el contenido en un ScrollView y un HorizontalScrollView. El contenido se mostrará desplegado en un TableLayout e introduciremos los campos de forma dinámica para admitir cualquier informe (siempre que siga el formato especificado).

El código sigue la misma estructura que las anteriores dos actividades. En el primer estado, extraeremos del clúster los IDs de los trabajos activos del usuario y se los mostraremos. En cuanto un id sea seleccionado, se volverá a llamar a la tarea asíncrona, esta vez con el segundo estado. En este estado, leeremos el informe DCRAB en formato de texto e insertaremos los campos en la tabla.

Utils.java
Utils.java es una clase auxiliar que se ha implementado para evitar repetir trozos de código y hacer la aplicación algo más escalable. Exceptuando los dos primeros métodos, que se utilizan para establecer el color de la barra de estado de Android, todos los métodos son de ejecución de comandos en el servidor. Como esto se hace siempre de la misma manera, estos métodos son todos parecidos.

2.4 Posibles mejoras para la aplicación

Al no haber tenido todo el tiempo que se hubiese deseado para desarrollar la aplicación y por mala práctica mía, algunos aspectos de la aplicación no se considera que estén en su estado final. Algunos de estos se pueden ver utilizando la aplicación, mientras que otros solo son visibles estudiando el código. A continuación se listarán las mejoras que se han ideado para solventar estos problemas:

- Mejoras visuales: En general, se ha intentado que la aplicación tenga un aspecto actualizado y al nivel de otras aplicaciones de su estilo. Sin embargo, algunos elementos han resultado más complicados de modificar y, para evitar consumir mucho tiempo resolviendo este problema que se considera menos importante, muchas veces se ha dejado a medias. Algunos ejemplos de este problema son: el spinner del primer estado de DcrabActivity.java, la lista extensible de QueueActivity.java, el no haber introducido la lista extensible en el menú desplegable, el botón de actualizar tanto de QueueActivity.java como de DcrabActivity.java, ...

- Escalabilidad de la aplicación: Al ser una aplicación de pequeña envergadura, muchas partes del código han acabado siendo reescritas y la estructura general de la aplicación ha degenerado en general. Si algún día se quiere expandir la aplicación y añadir nuevas actividades, hay ciertas partes del código que se recomienda modificar para hacer la tarea menos tediosa. Por ejemplo, si se quisiese introducir una nueva actividad al menú desplegable, habría que modificar el código de todas las actividades para añadir el nuevo elemento, lo cual no es recomendable. Otro ejemplo es el hecho de utilizar tareas asíncronas muy similares en todas las actividades, pudiendo usar una común y modificar ciertos aspectos dependiendo de la actividad.

- Problemas de conectividad: Durante el desarrollo de la aplicación, se ha probado su funcionamiento con diferentes tipos de redes (datos móviles, WiFi y conexión VPN tanto con datos como con WiFi) y se han sacado algunas conclusiones importantes. Una de ellas es que no se puede utilizar la aplicación con 'eduroam', probablemente porque el puerto 22 esté cerrado. Otra más importante aun es que tras varias conexiones en periodos pequeños de tiempo (8-9 conexiones por minuto) deja de ser posible seguir conectándose durante un tiempo. Esto solo ocurre desde redes externas a la UPV (datos móviles o WiFi desde casa, por ejemplo) y no pasa nunca si se está conectado mediante VPN. Probablemente ocurra porque hay algún tipo de sistema de 'softban' o regulador de conexiones (throttler) implementado que límite el número de conexiones desde redes externas. Solucionar estos problemas puede hacer que la aplicación sea mucho más útil a ojos de un usuario.
- Pruebas en diferentes dispositivos: Toda la implementación ha sido probada únicamente desde un teléfono personal mío (Xiaomi MiA1, Android 8.1.0, API 27). Para implementar la aplicación se han usado solo dimensiones relativas, 'match_parent', 'wrap_content' y 'dp' (density independent pixels) para vistas y 'sp' (scale independent pixels) para tamaños de letra, que es lo recomendado y debería hacer que la aplicación se viese igual en cualquier dispositivo. También se ha tenido en cuenta la versión de la API, siendo la aplicación compatible desde la API 19 (15 si no usásemos las gráficas de AnyChart), haciéndola compatible con el 96% de los dispositivos (véase `https://developer.android.com/about/dashboards/`). Aun así, esto no se puede garantizar y se recomienda probarla antes en algún otro dispositivo para comprobar que todo funciona correctamente.

3 Bibliografía

Bot de Telegram

- Documentación sobre los bots de Telegram: https://core.telegram.org/bots
- Documentación sobre la API de Telegram: https://core.telegram.org/bots/api
- Repositorio de la implementación en Python de la API de Telegram: https://github.com/eternnoir/pyTelegramBotAPI
- Repositorio del proyecto de vzemtsov: https://github.com/vzemtsov/ssh-Connection-In-Telegram
- Documentación sobre el cliente SSH de Paramiko: http://docs.paramiko.org/en/2.4/api/client.html
- Documentación sobre los canales SSH de Paramiko: http://docs.paramiko.org/en/2.4/api/channel.html
- Conexiones SSH anidadas con Paramiko: https://stackoverflow.com/questions/35304525/nested-ssh-using-
- Conexiones SFTP con Paramiko: http://docs.paramiko.org/en/2.4/api/sftp.html
- Modificación de variables de entorno para conexiones con Paramiko: https://stackoverflow.com/questions/14834251/setting-session-variable-for-paramiko-session
- Guía de instalación de Vagrant: https://www.vagrantup.com/intro/getting-started/index.html
- Configuración de una red pública en Vagrant: https://www.vagrantup.com/docs/networking/public_network.html
- Implemetanción de la clase Observer:
  - https://stackoverflow.com/questions/18599339/python-watchdog-monitoring-file-for-changes
  - https://stackoverflow.com/questions/47125974/python-watchdog-src-path-inconsistent

Aplicación de Android

- Conexiones SSH con JSch: http://www.jcraft.com/jsch/
- Tutorial para implementar un menú desplegable: https://medium.com/quick-code/android-navigation-drawer
- Tutorial para usar una barra de progreso: https://stackoverflow.com/questions/12559461/how-to-show-progress-barcircle-in-an-activity-having-a-listview-before-loading
- Repositorio de AnyChart para Android: https://github.com/AnyChart/AnyChart-Android
- Ejemplo de gráfico de columnas de AnyChart: https://github.com/AnyChart/AnyChart-Android/blob/master/sample/src/main/java/com/anychart/sample/charts/ColumnChartActivity.java
- Conexiones SSH anidadas con JSch: https://stackoverflow.com/questions/28850188/ssh-tunneling-via-jsc
- Documentación sobre los layouts de Android: https://developer.android.com/guide/topics/ui/declaring-layout
- Añadir elementos al menú desplegable de forma dinámica: https://stackoverflow.com/questions/34482404/how-to-add-an-item-to-a-menu-group-in-navigationview/34543766
- Guía sobre el uso de TableLayout: https://developer.android.com/guide/topics/ui/layout/grid
- Conjunto de widgets disponibles:
  - https://developer.android.com/reference/android/widget/package-summary

  - https://developer.android.com/reference/android/support/v4/widget/package-summary
  - https://developer.android.com/reference/android/support/v7/widget/package-summary
- Tutorial sobre las listas expansibles: http://www.androidtutorialshub.com/android-expandable-list-view-tu
- Listas expansibles en el menú desplegable: https://stackoverflow.com/questions/35956789/how-to-add-a-collapsible-menu-item-inside-navigation-drawer-in-android
- Listas expansibles en el menú desplegable: https://stackoverflow.com/questions/32419446/adding-expandablelistview-to-navigationview/32664433#32664433
- Ocultar teclado en Android: https://stackoverflow.com/questions/13593069/androidhide-keyboard-after-13593232
- Evitar la función onItemSelected() al introducir datos a un spinner: https://stackoverflow.com/questions/13397933/android-spinner-avoid-onitemselected-calls-during-initialization
- Abrir documento HTML con el navegador: https://stackoverflow.com/questions/7293786/opening-local-html-file-with-android-browser
- Explicación sobre FileProvider: https://inthecheesefactory.com/blog/how-to-share-access-to-file-with-en
- Implementación de FileProvider: https://stackoverflow.com/questions/38200282/android-os-fileuriexpos
- Cargar documento HTML en WebView: https://stackoverflow.com/questions/5749569/load-html-file-into
- Reutilizar layouts: https://developer.android.com/training/improving-layouts/reusing-layouts
- Añadir scrolls a una vista: https://stackoverflow.com/questions/6513718/how-to-make-a-scrollable-table
- Utilizar scrolls tanto horizontal como verticales: https://stackoverflow.com/questions/1399605/how-can-i-make-my-layout-scroll-both-horizontally-and-vertically
- Añadir elementos de forma dinámica a una tabla: https://stackoverflow.com/questions/14074632/initializing-tablelayout-in-xml-and-programmatically-filling-it-android-java