Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Proyecto fin de carrera

Sistema de gestión de estaciones autónomas

Autor: Guillermo Vayá Pérez
Tutor: Sergio Paraíso Medina

Lectura: Madrid, 25 Julio 2018

Sistema de gestión de estaciones autónomas

Guillermo Vayá Pérez

Lectura: Madrid, 25 Julio 2018

Tutor:

Sergio Paraíso Medina (sergio.paraiso@upm.es)

La composición de este documento se ha realizado con LaTeX.
Diseño de Oscar Cubo Medina.

© 2018, Guillermo Vayá Pérez

Esta obra está bajo una licencia Atribución-NoComercial-CompartirIgual 4.0 Internacional (CC BY-NC-SA 4.0) Creative Commons. Para ver una copia de esta licencia, visite:
http://creativecommons.org/licenses/by-nc-sa/4.0/deed.es.

Si quieres que sucedan cosas diferentes
deja de hacer siempre lo mismo.

Sonia Fernández-Vidal

Agradecimientos

Quiero agradecer a mis padres la paciencia y apoyo que han tenido durante estos años, acompañandome en cada paso del camino.

A mi esposa, con la que comparto cada parte de mi vida y esta es otra que no habría logrado sin su ayuda.

A mis compañeros de facultad, con los que he vivido muchas cosas estos años y con los que aprendí mucho mas de lo que veiamos en la facultad. En especial a César, quien no perdió la fe de verme comenzar este documento.

A mis compañeros de trabajo, pasados y presentes. Pues son una fuente inacabable de experiencias que me enseñan cosas nuevas cada día.

A Pilar Herrera y Sergio Paraiso, por ayudarme en los ultimos pasos dados en la facultad.

A la Facultad de Informática, por todo lo que he aprendido de ella y por prepararme para lo que venía después.

Resumen

Este Proyecto de Fin de Carrera trata sobre el diseño, implementación y puesta en marcha de un sistema de gestión de estaciones autónomas.

Dicho sistema, es utilizado para controlar una serie de ordenadores situados en la calle que gestionan el servicio al cliente, mediante este proyecto se quiere dar solucion al problema de gestionar dicha infraestructura independientemente de las diversas condiciones que puedan existir.

Es una aplicacion cliente-servidor que permite monitorizar, configurar y enviar comandos a las estaciones, de manera que se pueda saber en todo momento el estado y controlar cualquier casuistica que pueda suceder.

Este sistema ha sido desarrollado por el equipo técnico de Ride On Consulting S.L., una startup dedicada a la movilidad eléctrica urbana con proyectos en España y los Estados Unidos de América. Dada la separación geográfica, surgió la necesidad de este proyecto para facilitar el mantenimiento y gestión de los distintos proyectos.

Palabras Clave: *cliente-servidor, kiosco, gestión, grpc, rest, golang*

Índice

Índice VII

Índice de figuras XI

PARTE I MOTIVACIÓN Y OBJETIVOS

1 Introducción 3

2 Descripción del problema 5

3 Objetivos y Requisitos 7

4 Estructura del documento 11

PARTE II ESTADO DE LA CUESTIÓN

5 El problema en Ride On Consulting 15

6 Herramientas similares 17

7 Lenguajes evaluados 21

7.1 Clojure 22

7.2 Javascript 24

7.3 Python 24

7.4 Go 24

8 Bases de datos 27

PARTE III PLANTEAMIENTO Y SOLUCIÓN

9 Especificación de requisitos 31

9.1 Perspectiva del producto 32

9.2 Suposiciones y dependencias 32

9.3 Convenciones 33

10 Metodología de desarrollo 37

10.1 Componentes de Scrum 39

11 Tecnologías de apoyo 41

11.1 GRPC 42

11.2 Git/Github 42

11.3 ShellScript 43

11.4 Api Restful 44

11.5 Concurrencia mediante corrutinas 44

12 Implementación 47

12.1 Descripción del protocolo de comunicación 48

12.2 Descripción de la aplicación servidor 51
12.2.1 Arquitectura general 51
12.2.2 Autodescubrimiento 52
12.2.3 Interfaz API Restful 53
12.2.4 Descripción del modelo de BB.DD 55
12.3 Descripción de la aplicación cliente 57
12.3.1 Arquitectura General 57
12.4 Fases del desarrollo 58

PARTE IV CONCLUSIONES Y LÍNEAS FUTURAS

13 Conclusiones 63
13.1 Estado actual del proyecto 64
13.2 Problemas encontrados 64

14 Líneas futuras 67

Bibliografía 73

Índice de figuras

3.1 Visión General 9

7.1 Evolución de uso de lenguajes 22

10.1 Adopción de las Metodologías de Scrum 38

10.2 Proceso de Scrum 40

12.1 Flujo de mensajes en las comunicaciones 48

12.2 Esquema general de hilos en el servidor 51

12.3 Comunicaciones si el operador provee el túnel 53

12.4 Comunicaciones si Ride On Consulting provee el túnel 54

12.5 Modelo de BBDD de Redis 56

Parte I

Motivación y objetivos

Capítulo 1

Introducción

Ride On Consulting S.L. es una empresa española de movilidad eléctrica actualmente focalizada en las bicicletas eléctricas con proyectos de ámbito internacional.

La bicicleta eléctrica ha demostrado ser una de las mejores herramientas para facilitar el tránsito de los ciudadanos para trayectos cortos y medios, mejorando sustancialmente la fluidez del tráfico en grandes ciudades y contribuyendo a reducir la polución ambiental debidas a las emisiones de vehículos propulsados por gasolina y gasóleo.

De cara a poder ofrecer opciones de aparcamiento y recarga de las bicicletas eléctricas, Ride On Consulting ha de instalar estaciones a pie de calle. Estas estaciones están gobernadas por un pequeño ordenador con soporte de temperaturas de rango industrial (de -20ºC a 50ºC). Este ordenador se comunica con otros servicios y sistemas mediante distintos métodos: desde fibra óptica a comunicaciones por redes móviles. El tipo de conexión depende mucho de la zona de instalación, ya que cada una tiene una serie de características que obliga a enfrentarse a problemas diferentes y dificulta la estandarización que normalmente se busca.

Debido a que este ordenador no está siendo operado por ninguna persona, nos obliga a tener mecanismos que:

- Monitoricen el correcto funcionamiento.
- Permitan la ejecución de órdenes en remoto.
- Abran un canal de comunicación directo en caso de operativa que requiera interacción humana.
- En caso de fallo en la comunicación, permita encolar ordenes, para ser resueltas en el momento que la máquina vuelva a tener conectividad.

Estos requisitos son problemas habituales en máquinas remotas, resueltos mediante distintas herramientas de orquestación (Ansible, Puppet, Chef, etc.) y comunicación (SSH (**S**ecure **SH**ell)/Telnet).

Capítulo 2

Descripción del problema

En Ride On Consulting tenemos una variedad de tamaños de proyectos. Desde complejos de oficinas con necesidad de unas pocas estaciones, a un condado entero de EE.UU. que contiene varias ciudades de distintos tamaños.

Si el proyecto es pequeño (por ejemplo: un complejo de oficinas) donde hay un numero pequeño de estas estaciones (típicamente inferior a 10), la gestión de las mismas se puede hacer de manera manual y un pequeño equipo técnico con formación de sistemas, pero al ir aumentando de tamaño del proyecto, esto se vuelve inviable y hemos de tratar de automatizar y estandarizar tanto como nos sea posible.

Es la variedad de comunicaciones la que nos ha presentado un problema que de otra forma estaría cubierto por las herramientas del mercado mencionadas anteriormente. En concreto la comunicación por redes móviles (sin contratar servicios específicos de túneles y de direccionamiento IP público) plantea el problema de no tener una conexión directa con el ordenador de la estación. Las operadoras móviles utilizan protocolos similares al usado por los enrutadores (NAT) en sus APN, por lo que se asigna una IP privada dentro del mismo grupo y lo que los vuelve inaccesibles desde Internet. Esta situación obliga a que debe iniciarse siempre la comunicación desde el dispositivo satélite.

Incluso en los casos donde el tipo de conexión sí permite el acceso directo, tampoco es deseable tener un acceso permanentemente abierto por motivos de seguridad. El protocolo SSH ha demostrado ser muy fiable y aguantar el test del tiempo, pero como cualquier puerta, es mejor si la deshabilitamos hasta el momento de necesidad.

Por todo lo anterior se plantea desarrollar un sistema de gestión de estaciones autónomas que nos permita tener un control de las mismas independientemente de cual sea la difícil situación que nos plantee el mundo real.

Capítulo 3

Objetivos y Requisitos

Este proyecto consiste en una aplicación cliente-servidor que permita cubrir varias necesidades de cara a mantener un parque de estaciones autónomas desplegadas a lo largo de un área metropolitana (o incluso comarcal amplia), con conexiones diversas (y en ocasiones intermitentes) y en las que el acceso físico de personal especializado resulta caro económicamente.

Para la toma de requisitos, se evaluó la forma habitual de trabajo del equipo encargado del mantenimiento de dichas estaciones en diversos proyectos. A grandes rasgos podemos definir varios requisitos. Estos se ampliarán posteriormente en la sección dedicada a ello:

1. El sistema debe funcionar de manera aislada, no dependiendo de los mecanismos ya establecidos para el sistema de bicicleta eléctrica ni influenciar en estos salvo por motivos operativos.
2. El uso principal es la gestión de maquinas a las que no se tiene acceso físico.
3. Como uso secundario, debe dar una visión general del estado de las estaciones.
4. Debe contemplar distintas configuraciones de red, con especial énfasis en aquellas que presentan alguno de estos problemas:
5. Intermitencias en la comunicación.
6. Toda comunicación debe empezar desde el cliente, es decir no hay acceso directo al dispositivo.
7. Debido al requisito 4b, debe poder habilitar un canal directo de comunicaciones si fuera necesario.
8. Tratar de minimizar el gasto de comunicaciones, ya que estas pueden ser facturadas por el uso que se haga de las mismas.

Los objetivos de este proyecto, se ven parcialmente reflejados por la siguiente imagen:

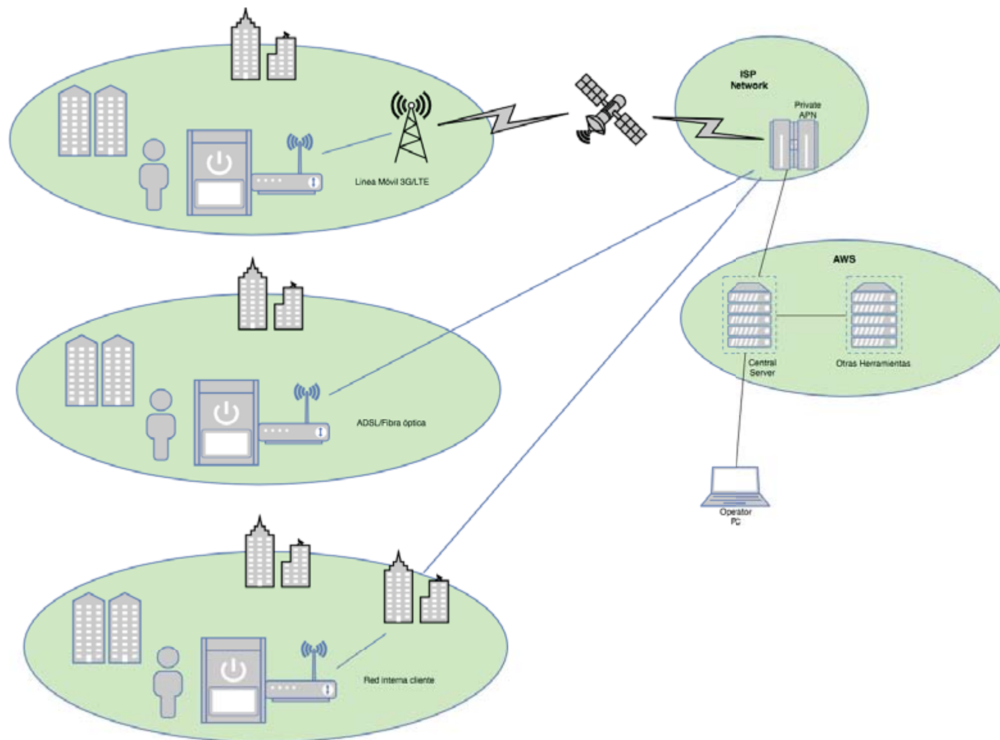

Figura 3.1: Visión General

Capítulo 4

Estructura del documento

Este documento se divide en cuatro partes:

Parte I Introducción. Es la sección en la que estamos donde se intentará dar una visión global del documento, del problema y del ámbito.

Parte II Estado de la cuestión. Donde se analizará el estado previo al desarrollo de la aplicación, así como un análisis de las tecnologías utilizadas para la realización del proyecto.

Parte III Planteamiento y Solución. Análisis de la fase de desarrollo de la aplicación, tanto de la parte cliente como de la parte de servidor, cubriendo desde los detalles de diseño hasta las decisiones de alto nivel tomadas.

Parte IV Conclusiones y líneas futuras. En esta parte se hará un resumen de lo conseguido con el proyecto, tanto a nivel de aplicación como en su puesta en marcha y se verán las posibles líneas de desarrollo futuro que se plantea la empresa con este proyecto.

Aparte de estas secciones, también hay una sección de glosario para tratar de aclarar las distintas siglas utilizadas en el documento y un par de Anexos con información adicional relativa al proyecto.

Parte II

Estado de la cuestión

Capítulo 5

El problema en Ride On Consulting

Antes de la creación del proyecto, la gestión de las estaciones se hacia de manera casi manual. Si la estación lo permite se hace la conexión por SSH, pero en muchos casos (sobretodo en proyectos pequeños) la conexión es mediante líneas móviles de 4G con APN (**A**ccess **P**oint **N**ame) compartidos o en instalaciones donde la red del cliente no permite tener puertos abiertos para SSH. En estos casos, se establece un demonio en el sistema que mantenga un túnel SSH inverso con un bastión predefinido.

Una vez conseguido el acceso a la máquina, se han elaborado scripts en Bash para las operaciones más comunes, aunque, al ser una empresa joven, aún hay mucha operativa que requiere pasos manuales y conocimientos de sistemas. A lo largo de la vida de la empresa, es de esperar que cada vez se tenga un mayor número de operativa identificada y automatizada, lo que repercute ampliamente en reducir la velocidad de actuación de sus encargados.

Capítulo 6

Herramientas similares

En el mercado existen distintas herramientas para la configuración y orquestado de servidores, si bien la mayoría de ellas están pensadas para una topología cercana entre sí, más pensando en la comodidad de gestión de un "datacenter" (sea este físico o en la nube). Es por eso que no ha sido posible encontrar una herramienta que pueda cubrir las distintas configuraciones a las que nos enfrentamos en la instalación y mantenimiento de un proyecto en la calle.

Hay que aclarar que lo anterior no quiere decir que no exista tal herramienta, ya que Ride On Consulting no es la primera compañía que instala kioscos en la vía pública o en recintos privados, por lo que es de suponer que estas compañías han podido solventar el problema de alguna manera, pero mantienen estas herramientas para uso interno. Esto puede ser debido a políticas de empresa o por estar ligados al propio núcleo de la operativa de negocio. Sin embargo, un requisito que Ride On Consulting se propuso al iniciar el proyecto es, justamente, que no se ligue al concepto de "electric bike sharing". Los motivos para marcar el interés en esta separación son principalmente porque los planes de la compañía pasan por otras verticales y la separación permitiría reaprovechar la herramienta sin necesidad de hacer un esfuerzo.

Un ejemplo de compañía que hace algo así sería Fon, para gestionar sus "foneras" (pequeños "routers" para crear wifis públicas) utiliza un sistema supuestamente similar al desarrollado. Lamentablemente, este sistema no es público, ya que es parte principal de su plan de negocio, por lo que no se puede tener acceso a él por motivos de no facilitar el trabajo de la competencia. Hay que hacer notar como anécdota, que de una conversación sobre este tema con extrabajadores de la compañía es de donde surgió la idea para este proyecto.

El principal problema encontrado a la hora de evaluar herramientas open source como Chef, Puppet o Ansible, es que en todos ellos el servidor principal se encarga de enviar la información necesaria a los clientes, quedando estos únicamente responsables de hacer un registro inicial (como mucho) ya que se supone control sobre las redes en las que se hace el despliegue y herramientas de descubrimiento de servicios que facilitan resolver la desorganización presente en redes muy pobladas.

El caso de Ride On Consulting es justamente el contrario, cada nodo está en una red completamente separada y, en algunas configuraciones, solo se puede iniciar la comunicación desde el propio nodo, por lo que resulta inviable que el nodo maestro envíe o solicite información a los nodos secundarios.

Un ejemplo claro es Ansible, quien elimina la necesidad de nodo maestro y permite el lanzamiento de recetas a grupos de servidores de la propia red. Para realizar estas operaciones, Ansible se conecta mediante SSH y ejecuta los "playbooks" en las máquinas objetivo. Para poder optar a hacer algo así, requeriríamos poder establecer la conexión por SSH al servidor objetivo, pero al tener la necesidad de que ésta fuera establecida desde el objetivo, el túnel SSH a establecer sería uno inverso y por lo tanto ha de estar establecido antes de tener la necesidad, lo que en el caso de Ride On Consulting quiere decir que sería un túnel permanente. Esto es problemático ya que la conexión no es siempre estable y requiere tener bastiones con conexiones permanentes a las estaciones para poder utilizarlas cuando haga falta. Esto además de la complejidad de mantenimiento, supone un gasto extra en recursos. Otro problema habitual es cuando uno de estos túneles se desconecta, ya que puede llegar a tardar bastante tiempo en recuperarse la conexión, lo que no es admisible en momentos críticos.

Capítulo 7

Lenguajes evaluados

Si bien el lenguaje es una herramienta para conseguir un objetivo y posiblemente se puedan conseguir productos similares usando lenguajes muy diferentes, cada lenguaje tiene una serie de características que facilitan o dificultan el desarrollo de un determinado proyecto. Algunas de estas características no son intrínsecas al lenguaje en sí, sino del propio ecosistema.

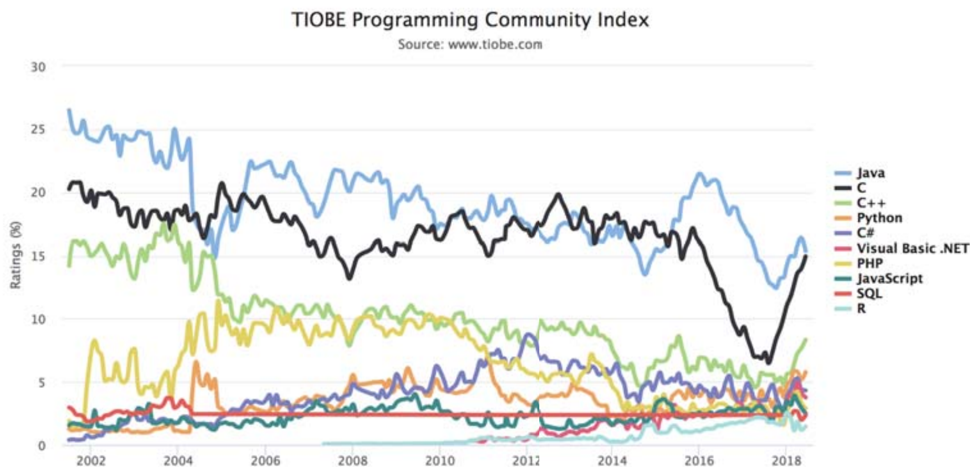

Figura 7.1: Evolución de uso de lenguajes

7.1. Clojure

Uno de los lenguajes más utilizados en Ride On Consulting es Clojure. Un dialecto de Lisp para la JVM (**J**ava **V**irtual **M**achine). Principalmente funcional y con un buen soporte de concurrencia. En Clojure es muy sencillo desarrollar DSL (**D**omain **S**pecific **L**anguage) específicos para la tarea y genera aplicaciones altamente maleables. Como otros Lisps, el aprendizaje es muy rápido (sobretodo con apoyo de un equipo de desarrollo que lo conozca) debido a que tiene muy poca sintaxis y reglas. Casi todo el lenguaje está construido desde unas pocas primitivas y en general es considerado muy conciso.

Clojure tiene muchos puntos fuertes, por los que es usado en Ride On Consulting:

- Concurrencia flexible. Dispone de múltiples mecanismos de concurrencia a usar en cada caso.

- Inmutabilidad. Inicialmente puede ser vista como una carga, pero es esta característica la que hace que la concurrencia sea mucho más fácil de llevar que otros lenguajes, sabiendo que nada puede modificar el valor, se eliminan muchas condiciones de carrera y bugs.
- Ejecuta sobre la JVM, la cual está presente en casi todas las arquitecturas disponibles.
- Permite la programación exploratoria mediante su REPL (**R**ead **E**val **P**rint **L**oop), lo que facilita la iteración y prueba de diferentes soluciones. En casos extremos la REPL permite conectarse a una aplicación en ejecución y alterarla en caliente, sin caída de servicio.

Fue una de las primeras opciones barajadas, pero descartado por varios motivos:

- Ecosistema. En Clojure se sufre que el ecosistema no tenga madurez. Esto se debe a que no tiene una acogida mayoritaria y que sus desarrolladores tienden más a hacer las cosas por ellos mismos (y en muchos casos a dejar una librería a medias) dada la maleabilidad del lenguaje. Para entender este aspecto, se recomienda la lectura del desarrollador de lisp bipolar.
- Usuarios objetivo. Dado que los principales usuarios de esta herramienta vendrán con un transfondo de Sistemas y DevOps (**Dev**eloper **Op**eration**s**) es poco habitual el conocimiento de este lenguaje. Si bien es sencillo de aprender, como hemos mencionado antes, requiere tener conocimientos de desarrollo funcional, algo poco habitual en el entorno de sistemas y arquitectura, más orientados a trabajar con estado y programación imperativa.
- JVM. Es un proceso pesado y lento de arranque, lo que añadido a que ya tenemos otros elementos corriendo, no parece la solución más idónea para el cliente. Si bien podría usarse en el servidor, es preferible tener todo el sistema en el mismo lenguaje salvo que haya ventajas importantes en la separación de los mismos (como pasa en una aplicación web, en la que tradicionalmente se usa alguna variante de Javascript en el frontal y otro lenguaje para el servidor)
- Sin soporte de GRPC oficial. Aunque hay otras opciones, no tiene un buen soporte para una de las tecnologías que teníamos claras desde el principio. Lo que restaría valor a la propuesta de GRPC de hacer contratos entre las partes.

7.2. Javascript

Si Clojure tiene una deficiencia de ecosistema, en Javascript es quizás justo lo contrario, hay multitud de librerías para casi todo. Es un lenguaje que ha cogido mucha fuerza debido a su presencia en la web y que con “Node.js” ha traspasado el muro del navegador entrando tanto a los servidores como a las aplicaciones.

En Ride On Consulting se utiliza Javascript para el desarrollo de interfaces de usuario, desde los “dashboards” que gestionan la plataforma mediante React, hasta las aplicaciones móviles haciendo uso de ReactNative.

Si bien en estos dos campos es un lenguaje que apoyo completamente dada la cantidad de beneficios que aporta, no tiene la misma consideración cuando lo movemos a la parte de desarrollo de aplicaciones de servidor o de sistemas. La cantidad de casos esquina, complejidades ocultas e idiosincrasias propias del lenguaje lo hace complicado a la hora de desarrollar, lo que ha provocado utilizar librerías y pasos de compilación a JS (**J**ava**S**cript) previos (“transpilación”, en realidad) para tratar de minimizar estos problemas. Son muy numerosos los videos y artículos que detallan los problemas de Javascript como lenguaje por lo que se recomienda hacer una búsqueda en la web para conocerlos antes de tomar la decisión de usarlo.

7.3. Python

Un lenguaje mucho más maduro que los anteriores y con buen ecosistema. Sin embargo, en este caso el descarte fue por el propio equipo de Ride On Consulting, al no haber apenas gente en el mismo que desarrolle Python con fluidez.

7.4. Go

Finalmente llegamos al lenguaje seleccionado. Go es un lenguaje que resulta sencillo de aprender, tiene un amplio ecosistema favorecido por Google, buena recepción en la comunidad, buen soporte de concurrencia y es muy usado como leguaje de desarrollo de aplicaciones para servidor, tanto generalistas como de herramientas para mantenimiento de servidores.

Docker y Terraform son ejemplos de herramientas de gestión de aplicaciones y servidores hechas en Go. Dentro de Ride On Consulting se utilizan ambas herramientas.

Además, Go está presente en buena parte de las herramientas de apoyo para la infraestructura. Así como algunos desarrollos pequeños de microservicios "serverless" sobre AWS (**A**mazon **W**eb **S**ervices) Lambda que se encargan de gestionar pequeñas partes de la lógica de negocio.

La compilación de Go genera binarios pequeños y rápidos, lo que los hace ideales para correr en un entorno restringido como es el ordenador de un kiosco, además de facilitar su instalación y actualización por no consumir mucho ancho de banda en transferirlo. Como ejemplo contrario: el archivo comprimido (un ".jar" desarrollado en Clojure y Clojurescript) que utiliza el kiosco para dar servicio ronda los 100 MB (**M**ega**B**ytes), lo que hace que las actualizaciones sean más lentas y caras. Esto además ejemplifica también los problemas de red enfrentados ya que, en redes de servidores, mover un archivo de 100 MB apenas tarda unos segundos, mientras que en un kiosco en la calle con una conexión por redes móviles 3G puede llegar a tardar horas si la red sufre cortes o no tiene velocidad suficiente.

Como desventajas de Go:

- la falta de tipos genéricos obliga a apoyarse mucho en herramientas de generación automática de código, así como tener duplicidad en funciones de uso común.
- El sistema de tipos es demasiado simple en comparación con otros lenguajes, lo que por un lado facilita su adopción, pero a la larga complica el desarrollo y facilita la aparición de malos hábitos que eliminan algunas de las ventajas previas. Por ejemplo: para algunas cosas se usan tipos de interfaz genérica, lo que hace que el compilador no pueda optimizar el código ni comprobar que las cosas sean correctas, lo que hace un código más lento y puede dar problemas en tiempo de ejecución.

Pese a esto último, Go parece, bajo nuestro criterio, la herramienta apropiada para el desarrollo de esta aplicación.

Capítulo 8

Bases de datos

Como casi cualquier aplicación que vaya a estar corriendo de manera continuada, necesita de algún tipo de persistencia del estado, de manera que podamos ir modificando el mismo a lo largo del tiempo.

Para el caso actual, se considera que no hace falta una BBDD (**B**ases de **D**atos) relacional. Si bien se barajó y podría usarse, no se encontró ningún beneficio, ya que los datos a mantener son escasos y poco estructurados. De haber sido una BBDD relacional, se habría optado por PostgreSQL, la cual ya se está utilizando en la empresa, tiene una eficacia probada, es open source y está soportado por el servicio de Amazon RDS.

Sin embargo, para este proyecto se opta por otra opción: Redis. Es una BBDD de clave-valor, sencilla de utilizar y rápida. De nuevo opensource y de probado valor. También se usa dentro de Ride On Consulting como caché intermedia de valores, de manera que la BBDD principal de la plataforma esté más descargada y para sincronizar pequeñas tareas. A lo largo de mi carrera, la he utilizado múltiples veces para todo tipo de proyectos. Como extra, soporta varios tipos de estructuras de datos, lo que hace que no sea simplemente clave-valor, sino que forma objetos mucho más útiles sin necesidad de hacer transformaciones sobre scripts, lo que suele llevar a generar defectos imprevistos en el software.

Por norma general, Redis funciona como base de datos en memoria, aunque puede persistirse. Evaluando las necesidades de persistencia del proyecto, nos damos cuenta que la pérdida de datos de manera puntual no supone realmente mucho problema, ya que serán otros proyectos y personas las que exploten esta información y debe por tanto considerarse volátil. Esta decisión simplifica el mantenimiento de la herramienta y mejora la velocidad de respuesta.

Una de las características más interesantes de Redis, es que su servidor es mono-hilo, lo que es una restricción importante, pero a la vez ayuda a asegurar que ningún otro elemento esté consultando o modificando la estructura de datos sobre la que se está trabajando en una misma transacción.

Parte III

Planteamiento y solución

Capítulo 9

Especificación de requisitos

Este capítulo quiere dar una visión más formal y de alto nivel del problema, con las que se espera mostrar las necesidades del proyecto.

9.1. Perspectiva del producto

El proyecto se divide en dos subsistemas, como es habitual en su naturaleza de cliente-servidor. Las comunicaciones estarán siempre iniciadas desde el cliente, siendo el servidor un elemento pasivo que recibe información y almacena las órdenes a la espera de ser ejecutadas por los clientes. Ambos deberán correr sobre sistemas Unix.

El cliente tendrá principalmente dos funciones:

- Recopilación de información de estado. El cliente monitorizará algunos recursos del sistema e informará periódicamente de los mismos al servidor para su posterior análisis y monitorización de la salud de la plataforma.

- Ejecución de órdenes. El cliente debe preguntar al sistema de manera periódica si hay alguna orden pendiente de ejecutar en el sistema objetivo, en caso de haberla, se recibirá la misma y se devolverá el resultado una vez finalizada.

El servidor sin embargo es un mero receptor. Por un lado recibirá el estado de cada cliente, lo que permite luego enviarlo a los sistemas de monitorización. Por otro, recibirá la lista de ordenes a aplicar en los dispositivos y despachará las mismas una vez estos pregunten.

9.2. Suposiciones y dependencias

Para este proyecto se han tenido las siguientes suposiciones:

El lenguaje de desarrollo será comun en cliente y servidor, debido a no haberse visto una ganancia en tenerlos separados. Futuros desarrollo (GUI, por ejemplo) pueden provocar cambios.

Para la comunicacion entre ambos se usara la libreria GRPC para asegurar el contrato entre ambas partes.

Como en todos los proyectos de Ride On Consulting, no se incluiran librerías que puedan poner en peligro el proyecto, esto es que no tengan una prevision de mantenimiento aceptable hasta donde se pueda conocer en el momento de incluirla. En caso de considerarse que va a cambiar esta situación en una librería ya añadida, se buscará un sustituto tan pronto como sea posible y se hospedará una version estable de la misma en un servidor de la empresa por si hicera falta hacer algún tipo de ”*rollback*”.

9.3. Convenciones

Para la identificación de requisitos utilizaremos el siguiente formato: TR-XX.

TR clasificará el tipo de requisito: funcional (RF) o no funcional (RN)

XX será el identificador numérico secuencial.

Es decir:

- RF-003 será el tercer requisito generado y es un Requisito Funcional
- RN-004 será el cuarto requisito generado y es un Requisito No funcional

Para cada requisito generaremos una tabla como la siguiente:

TR-XX	Resumen del requisito
Explicación más detallada del mismo requisito de manera que pueda comprenderse el alcance y detalles, minimizando así posibles equívocos y falta de detalle.	

A continuación, detallaremos los requisitos planteados en la Parte I.

TF-01	Funcionamiento aislado del resto de sistemas
Este proyecto debe ser una herramienta que facilite las tareas del equipo, pero no debe interferir o ser necesario para el funcionamiento de la plataforma principal. De esta manera se plantea una planificación separada que se adapte a futuros proyectos.	

TF-02	Funcionamiento del cliente sobre sistemas Unix-like.
Como poco, el cliente debe funcionar sobre sistemas Linux con arquitectura x86 al ser estas las características de los ordenadores de las estaciones. Aunque es deseable que tenga soporte de otras arquitecturas, de cara a futuros proyectos.	

TF-03	Soporte de redes inestables
El sistema debe soportar que un cliente no esté 100 % online. Por lo que debe ser capaz de esperar que este vuelva a aparecer más adelante cuando las condiciones de la conexión sean más favorables.	

TF-04	Soporte de redes privadas
El sistema no debe depender de que el cliente tenga una ip accesible.	

TF-05	Capaz de ejecutar ordenes en el cliente
El servidor mandará comandos o “recetas” que deben ejecutarse en el cliente y ser capaz de recibir el resultado de los mismos. Hay que hacer especial énfasis en el comando de establecer túnel con un bastión, de manera que se pueda interactuar con el dispositivo cliente de manera manual si fuera necesario.	

TF-06	Conocer el último estado del sistema
El cliente debe reportar periódicamente con el servidor de manera que se pueda conocer distintos parámetros de sus sensores/software asi como saber si está experimentando problemas en la estabilidad de la conexión.	

TF-07	Uso de protocolo extensible
Dado que el futuro de esta herramienta es evolucionar con la compañía, es deseable que el protocolo a utilizar entre cliente y servidor sea extensible con facilidad.	

TF-08	Envío de órdenes a múltiples sistemas
Al enviar comandos, debe poder especificarse múltiples objetivos, de manera que se pueda actuar sobre grupos.	

TF-09	Interfaz REST (**RE**presentational **S**tate **T**ransfer) para desarrollo/integración con otros sistemas
Otras herramientas pueden depender de esta, o ampliar su funcionalidad. Por ello debe ofrecer una interfaz fácil de implementar en otras herramientas/sistemas.	

TN-10	Persistencia de información
El servidor, mantendrá un registro de las últimas interacciones con los clientes, para posterior consulta.	

TN-11	Sin lógica de negocio
Dado que es una herramienta separada y que puede llegar a dar soporte a distintos tipos de plataformas, no debe contener lógica de negocio.	

TN-12	Sin información identificable de usuarios
Al ser sistemas que están en la calle, son sistemas mucho más vulnerables a robo. Por ello el cliente no debe tener información de usuarios.	

TN-13	Fácilmente escalable
Debido a que algunos proyectos pueden crecer, la herramienta debe tener una estrategia planteada para dar soporte al mayor número de elementos posibles.	

Capítulo 10

Metodología de desarrollo

Para el desarrollo de este proyecto se ha seguido una metodología de desarrollo ágil. Los motivos para usar esta metodología tan frecuente últimamente son:

Requisitos sin cerrar. El proyecto no tuvo los requisitos completamente cerrados, al ser una herramienta de apoyo. Por lo que fue tomando forma según la empresa necesitaba e incluso algunos de los requisitos originales fueron descartados al descubrirse posteriormente formas de solucionarlos alternativas o que no tenían la suficiente solidez como para mantenerlos en el proyecto.

Uso en el día a día. El proyecto pudo ser usado casi desde el principio, lo que permitió observar qué necesidades estaba cubriendo y cuales no tenían la forma adecuada. Gracias a las metodologías agiles, no tuvimos que esperar todo un ciclo completo de desarrollo para comenzar a mover una arquitectura que igual no lo hubiera soportado.

Motivación del desarrollo. Además, el poder usarla en el día a día motiva a continuar desarrollándola, al convertirse en una parte indispensable de nuestra gestión de infraestructura.

De las múltiples variantes de metodologías ágiles, en Ride On Consulting y en especial para este proyecto se sigue la metodología Scrum. La que posiblemente es la más común de todas las metodologías aplicadas a software actualmente.

Figura 10.1: Adopción de las Metodologías de Scrum

Creado en 1986 en Japón, es una metodología específicamente diseñada para trabajar sobre proyectos que sufren cambios constantes. Algo muy habitual en el ecosistema “startup”, donde Ride On Consulting está ahora mismo.

10.1. Componentes de Scrum

Scrum consta de dos conjuntos de trabajo principales coexistiendo simultáneamente:

- Product Backlog. Es el repositorio de trabajo completo. Donde se almacenan las nuevas características, así como los defectos encontrados durante las pruebas. Está en constante revisión y puede sufrir cambios drásticos a lo largo del tiempo, al ser el lugar donde se tiene una planificación a largo plazo.
- Sprint Backlog. Es un subconjunto del anterior en el que se trabaja en un ciclo corto de tiempo para producir un resultado visible en el proyecto. Es más raro que sufra cambios dado el corto espacio de tiempo en el que está activo. Su duración es la duración del sprint, del que hablaremos a continuación.

La unidad de tiempo en Scrum es el Sprint, que suele tener una duración corta (de 2 a 4 semanas) dentro del cual está todo tan planificado como sea posible. Esto permite dar un marco de estabilidad al equipo de desarrollo sin que afecten los cambios del Product Backlog. En Ride On Consulting, hacemos los Sprints de dos semanas, por considerar que nos permite pivotar con las necesidades y gestionar los constantes cambios que una “startup” sufre. En caso de usar casi un mes de sprint, sería mucho más complicado mantener los cambios al margen, ya que Ride On Consulting se encuentra en un punto donde hay mucho competidor y cada cliente tiene unas necesidades especificas y cambiantes. Durante este lapso de tiempo, se espera que se hagan todas las fases de desarrollo (requisitos, diseño, implementación y pruebas), teniendo una nueva iteración en el Sprint siguiente. De esta manera el equipo ve como el producto crece sin verse atrapado por las decisiones.

Aparte de los integrantes de un equipo de desarrollo normal (llamados Scrum Team), en Scrum se destacan dos actores principales que deben compaginar otras actividades con las propias de estos roles:

Product Owner. Es la persona o grupo, que deciden y priorizan los distintos requisitos de la aplicación. Deben conocer el espacio donde el proyecto debe existir de manera que puedan tener una visión clara de los pasos a tomar para un mayor impacto de la herramienta. En

este caso, las personas a cargo de la infraestructura fueron los encargados de tomar este papel.

Scrum Master. Es el coordinador del equipo de desarrollo. Se encarga de la correcta ejecución de los sprints, asi como de liderar cualquier reunión que sea necesaria, como los "daily standups".

Una práctica muy habitual en los equipos de Scrum (aunque altamente recomendable para cualquier metodología) son los "daily standups". Estas son reuniones diarias para conocer el progreso (o su carencia) de cada miembro del equipo. Son reuniones cortas (un par de minutos por integrante) en las que se habla de lo que se hizo el ultimo día, si hubo alguna dificultad y del plan que se tiene para el día siguiente. Típicamente se hacen de pie (de ahí su nombre) y tienen un carácter informal para que sea algo rápido y no entorpezca las labores del equipo.

La siguiente imagen trata de resumir los conceptos anteriores de una manera clara y sencilla:

Figura 10.2: Proceso de Scrum

Capítulo 11

Tecnologías de apoyo

Aparte de la elección del lenguaje y BBDD mencionadas en la Parte II, este proyecto sería inviable sin apoyarse en muchos otros que se desarrollaron previamente y que facilitan enormemente la tarea. En esta sección se pretenden mencionar los más significativos.

11.1. GRPC

Desde el comienzo del proyecto, se sabía que se iba a usar esta tecnología (o en caso de no tener soporte por parte del lenguaje: una que ofreciera unas características similares). GRPC ofrece un lenguaje y framework común para desarrollar aplicaciones que se comuniquen entre sí (cliente-servidor habitualmente, pero puede usarse para otros escenarios).

El motivo de tener clara la elección se debe a que el protocolo de comunicación es uno de los puntos clave de la herramienta, por lo que tener un sistema que nos permita establecer contratos entre los nodos hace que nos tengamos que preocupar de una cosa menos, ya que el código que se encarga de transmitir la información de un punto a otro es autogenerado, por lo que en caso de surgir la necesidad de cambio de protocolo, obliga a tener ambas partes correctamente sincronizadas.

Además, genera código para servidores de *"streaming"*, lo que podemos aprovecharlo de cara al envío de comandos, lo que simplifica enormemente la forma de tratar las tareas al considerarlo unitario en vez de empaquetado. Es decir, en vez de mandar un único paquete juntando todas las ordenes disponibles, se puede convertir a paquetes indivisibles y mandar varios seguidos, de manera que se puedan procesar individualmente y sin pasos intermedios.

GRPC utiliza un lenguaje de descripción de servicios que posteriormente se compila al lenguaje objetivo (en el presente caso: Go), reduciendo así buena parte de la complejidad inherente a las comunicaciones y estableciendo canales de comunicación claros y adecuados para el objetivo.

11.2. Git/Github

Probablemente el par de tecnologías más utilizado de los últimos tiempos. La mayor parte del código fuente actual utiliza algún tipo de Control de Versiones, especialmente los descen-

tralizados y dentro de estos: Git. Este DCVS (**D**ecentralized **C**ontrol **V**ersion **S**ystem) creado por Linus Torvalds asegura el versionado del código de todo tipo de proyectos, incluyendo los personales. Y es que no hace falta trabajar en equipo para darse cuenta de las enormes ventajas de este sistema.

Junto con Git, está Github, la empresa que hace de servidor central para buena parte del código de *"startups"*, organizaciones Open Source y grandes empresas en general. Ofrece un sistema seguro y cómodo donde desarrolladores y empresas pueden almacenar y compartir su código.

La unión de ambas ha demostrado no solo ser efectiva y sino además una apuesta segura en los pocos años de vida tanto de la tecnología como de la plataforma.

En el caso que nos ocupa, el uso de ambas ofrece una doble ventaja: por un lado, colaborar con el desarrollo de manera habitual y, por otro, ser un repositorio de scripts con los que poder gestionar los nodos de la plataforma.

En Ride On Consulting están ambos muy integrados en el flujo de desarrollo de los miembros del equipo.

11.3. ShellScript

Si bien Shellscript es un lenguaje de programación orientado a la configuración de sistemas, no es un candidato apropiado para el desarrollo de las partes principales de este proyecto. Sin embargo, es una parte integral para terminar de dar forma a la herramienta, al ser estos scripts los que realicen los comandos en el cliente.

Además, es ampliamente utilizado por Administradores de Sistemas y DevOps, quienes deberán extender esta parte de la herramienta para asegurar el correcto funcionamiento de todo el proyecto con el menor esfuerzo posible.

A futuro, muchos de estos scripts serán convertidos a otro lenguaje, pero ofrecen una herramienta rápida para juntar las distintas piezas requeridas para operaciones de mantenimiento en un servidor Unix.

11.4. Api Restful

Si bien la comunicación cliente-servidor irá utilizando GRPC para asegurar el contrato. A futuro, se desea también el poder interactuar con la herramienta desde otros proyectos o incluso generar una pequeña interfaz de usuario gráfica que permita tener un mejor control sobre los nodos. Para ello se plantea una pequeña API (**A**pplication **P**rogramming **I**nterface) Restful que facilite la integración con cualquier otra plataforma.

REST es un estilo de arquitectura muy utilizado en HTTP (**H**yper**T**ext **T**ransfer **Protocol**) para la gestión de servicios. Las URI (**U**niversal **R**esource **I**dentifier) representan objetos sobre los que se realizan operaciones (llamados verbos en este estilo).

Una de las características más interesantes de este tipo de interfaces es que están planteadas para ser ajenas al estado, por lo que no puede llevarse este de una comunicación a otra. Esto puede ser una desventaja para algunos tipos de aplicación, pero en el caso de interfaces con otras herramientas es una restricción que simplifica el diseño tanto de la misma interfaz como de la propia interacción.

11.5. Concurrencia mediante corrutinas

El modelo de concurrencia a utilizar es el utilizado por el propio Go, es decir un modelo de concurrencia basado en corrutinas.

En el caso de Go, se hace una mezcla entre un modelo corrutinas puro (como veríamos en Node.js de JS o en la librería "Async.io" de Python) pero implementado sobre un grupo de hilos que permiten ir más allá de la solución mono-hilo que presentan los otros ejemplos.

Para complementar esta solución, Go además hace uso de varios mecanismos habituales en la concurrencia (*"mutexes"*), pero sobretodo de uno en especial: los canales.

Mediante los mismos se pueden sincronizar corrutinas así como compartir información que de otro modo daría muchos quebraderos de cabeza y posiblemente defectos difíciles de reproducir.

Los canales permiten una variedad amplia de comportamientos entre productores y consumidores, lo que les da una flexibilidad interesante. Cuando creamos un canal, podemos hacerlo de dos maneras:

Sencillo. Solo puede haber un elemento en el canal, lo que bloquea al consumidor hasta que un productor escriba en el mismo y bloquea también al productor hasta que un consumidor lo lea. Esto hace que dos procesos separados tengan un punto común de sincronización.

Buffered. Se aceptan N valores en el canal, no llegando a bloquear hasta que no se llena/vacía. Se puede considerar el sencillo, una versión simple de este. Además de lo anterior podemos leer de manera bloqueante o no. Ya que en ocasiones no es deseable que el sistema se bloquee si el buffer estuviera lleno. Un ejemplo claro es querer utilizar un mecanismo bien de error o de *"deadletter"* si se ha llegado a llenar el buffer.

Al igual que los ejemplos presentados, Go tiene una interfaz I/O asíncrona que evita que un hilo de ejecución se quede bloqueado debido a una necesidad de comunicarse con otros elementos.

Por todo ello, es un modelo apropiado para este proyecto, facilitado la escalabilidad y la concurrencia sin pagar un precio alto por ello.

Capítulo 12

Implementación

12.1. Descripción del protocolo de comunicación

El aspecto principal de este sistema reside en la comunicación entre las partes, ya que es el elemento que debe resolver los problemas encontrados por la empresa. Es por ello que fue lo primero que se definió de este proyecto y lo primero que se presenta en esta memoria. A grandes rasgos, el protocolo de comunicación está definido en el siguiente diagrama:

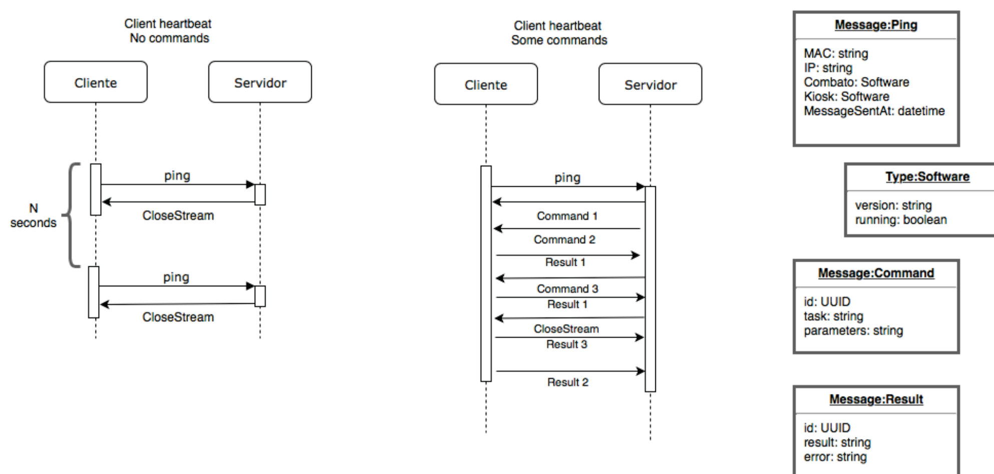

Figura 12.1: Flujo de mensajes en las comunicaciones

En el diagrama podemos apreciar tres tipos de mensajes diferentes. Se planteó uno extra de saludo, pero dada la naturaleza de las redes intermitentes se consideró que generaría más problemas ya que este primer intercambio puede verse afectado por los mismos problemas que se tratan de solucionar.

Ping. Es un mensaje recurrente, se da cada N segundos (por defecto se suele poner cada 5 min.) La idea detrás de este mensaje es una comunicación periódica que nos permita saber que la estación está funcionando correctamente.Está formado por los siguientes campos:

El campo Mac, fue lo que se decidió que se utilizaría como identificador de las estaciones. Esto es debido a que es un identificador único (salvo que se clone, pero eso es algo que no nos planteamos hacer) y fácil de obtener del propio sistema operativo.

El envío de la dirección Ip se debe a que es interesante saber cuál es esta dirección en caso de que la red permita acceso directo. De esta manera podemos conectar directamente si

Campo	Tipo	Explicacion
Mac	String	Identificador único del kiosco
Ip	String	IP pública actual
Combato	Software	Versión y estado del subsistema
Kiosk	Software	Versión y estado del subsistema
MessageSentAt	DateTime	Fecha y hora de envio de este mensaje

hay ocasión. En los casos donde este dato no nos permita hacer acceso directo, queda como anecdótico únicamente.

Kiosk y Combato son dos procesos que queremos monitorizar dentro de la aplicación. Se ha creado un tipo específico que nos permita saber qué versión están ejecutando y si elservicio está efectivamente corriendo. Mediante este tipo, es fácil extender a otros servicios cuando estos crezcan.

MessageSentAt. Este campo contiene la fecha y hora a la que se envió el mensaje. Junto con la hora de la recepción del mismo nos da una idea de cómo de congestionada está la red.

Command. Este mensaje sirve para distribuir las ordenes a los clientes. Forma parte de un stream de mensajes que suceden del servidor hacia el cliente, como respuesta a a un mensaje de Ping. De esta manera el cliente siempre inicia la conversación y al tener un mensaje periódico, nos aseguramos que en algún momento le llegará la orden. Este mensaje debe contener toda la información necesaria para poder ejecutarla. Está formado por los siguientes campos:

Campo	Tipo	Explicacion
Id	UUID	Identificador único de tarea
Task	String	Nombre de la tarea a realizar
Parameters	String	Parámetros de la tarea a realizar

El campo ID está formado por un UUID (Universally Unique Identifier) que nos permita identificar de manera única la tarea. Este identificador está compartido entre las distintas estaciones a las que se le haya pedido el mismo comando.

Tanto el campo Task como el campo Parameters son strings. Dado que no sabemos qué clase de parámetros de entrada será necesarios, se deja como tarea de los administradores o de

las herramientas que se integren el definirlos.

Result. Finalmente, el cliente tiene otro tipo de mensaje, para informar al sistema de que ha terminado de ejecutar la orden del mensaje de *Command* y devuelve el resultado. Está formado por los siguientes campos:

Campo	**Tipo**	**Explicacion**
Id	UUID	Identificador único de tarea
Result	String	Resultado final de la tarea
Errors	String	Cualquier error que se haya querido notificar

El campo ID es la correlación para saber qué tarea terminó de ejecutar.

Tanto el campo Result como el campo Errors, son strings, dado que no sabemos qué clase de resultado se nos puede devolver y es tarea de los administradores o de las herramientas que se integren el tratarlos. Como ya se ha mencionado varias veces, se trata de no acoplar el proyecto a otros elementos y por lo tanto se envía un string, dejando los detalles a cuadrar por los programas que interactúan.

De todos estos mensajes, hay uno muy claro que cambiará junto con el proyecto en el futuro: el mensaje de Ping. Con el tiempo irá incluyendo cada vez más detalles de la estación para su observación futura y toma de decisiones. Pero los datos presentados aquí son los considerados "mínimo necesario" para que el proyecto tuviera sentido.

Para la implementación de este protocolo, se ha utilizado una librería llamada GRPC que permite establecer un contrato entre clientes y servidor. Para ello se crea un fichero en un DSL (como el proporcionado en el Anexo I) mediante el cual se especifican los detalles que debe implementar tanto el cliente como el servidor para que puedan entenderse. Esto además permite aprovecharnos de herramientas de generación automática de código que nos quite de en medio algunos de los errores típicos de la comunicación entre nodos de una red.

12.2. Descripción de la aplicación servidor

12.2.1. Arquitectura general

El servidor consta principalmente de 3 corrutinas que se encargan de gestionar el 90 % de la funcionalidad. Lo que estamos descartando en esta parte es la sección de API RESTful, que cumple un papel más de integración con otros elementos y se hablará de ella más adelante.

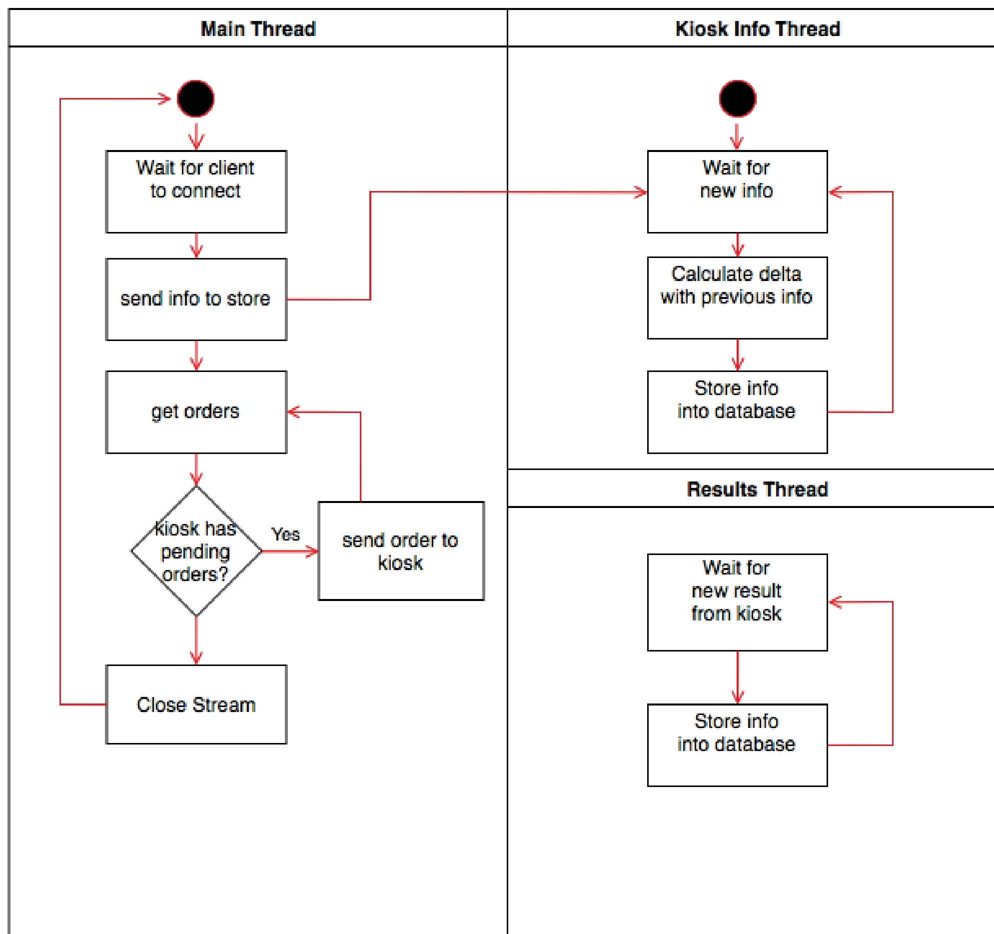

Figura 12.2: Esquema general de hilos en el servidor

El hilo principal (arriba a la izquierda) es el que se encarga de la recepción de información de los clientes. Está permanentemente escuchando hasta que se dé la orden de apagado del servidor. Si bien inicialmente cumplia esta función y la del hilo de kiosk info (arriba a la derecha en la

imagen), se consideró que era mejor separar la funcionalidad, para evitar que se bloqueara la escucha de nuevas peticiones, dejando un hilo principal liviano que únicamente se dedicará a la parte de comunicación con el cliente. Por otra parte, se generan otros dos hilos, uno para escuchar resultados del kiosko y otro para almacen de info de kiosko. El primero se debe más a la estructura que te da la propia librería grpc, que facilita esa separación. El segundo a lo explicado anteriormente: separar funcionalidad.

12.2.2. Autodescubrimiento

En las primeras fases del diseño de la aplicación y concretamente en el protocolo de comunicación, se planteó el problema de cómo tratar las estaciones de cara a darse de alta. Si tenía que estar dada de alta previamente, o si había que hacer algún mensaje específico por parte del kiosco que le permitiera identificarse. La razón principal para plantearse esta característica fue por motivos de seguridad: evitando así que alguien pudiera añadir a una instalación un elemento no deseado.

Esto se descartó por varios motivos:

- El primero es que se vio innecesario: no se está gestionando información sensible, sino únicamente de gestión, orquestración y monitorización. Por ello, si alguien copiara el protocolo y tratara de introducir un kiosco ficticio, no haría más que desvirtuar ligeramente los datos recabados. Esto podría corregirse desde la herramienta con la que se visualice la información una vez se detectara.

- Otro motivo es la facilidad de crecimiento, a lo largo de un despliegue en un nuevo proyecto, es muy habitual que las estaciones se desplieguen de forma continuada. Quitar pasos intermedios, elimina barreras, obstáculos y posibles problemas en cascada. Por último, una reducción de mensajes a enviar implica una reducción de costes en comunicaciones. Dependiendo de cómo se plantease este tipo de mensajes, podría llegar a ser muy costoso.

Los contras que tiene el autodescubrimiento es la falta de seguridad y la duplicidad de información.

La falta de seguridad puede ser un problema en ciertas redes si éstas son abiertas y por lo tanto alguien puede escuchar el intercambio y repetirlo posteriormente. Este problema deja de ser tan grave cuando ponemos en consideración el tipo de datos que se van a tratar y las posibles consecuencias de una brecha de seguridad. No habría problemas con datos de clientes, y los problemas serían más de tipo económico: alguien roba un equipo y deja algo simulando aun estar ahí, lo que resulta caro para el propio delincuente. Además, en muchos casos esto se resuelve mediante herramientas ajenas al propio proyecto, asegurando el canal de comunicación en lugar del protocolo. Estos mecanismos se pueden conseguir de dos formas: bien por contratos con operadoras de telefonía móvil o construyéndolo por nuestra cuenta. La forma de conseguirlo en cualquiera de los dos casos es planteando un túnel punto a punto con un bastión en AWS mediante el cual se asegure el canal de comunicación, el otro extremo del túnel depende del camino elegido. En el caso de la operadora, el túnel estará a nivel de APN. El propio APN deberá ser privado y rechazar la conexión de cualquier dispositivo que no esté permitido.

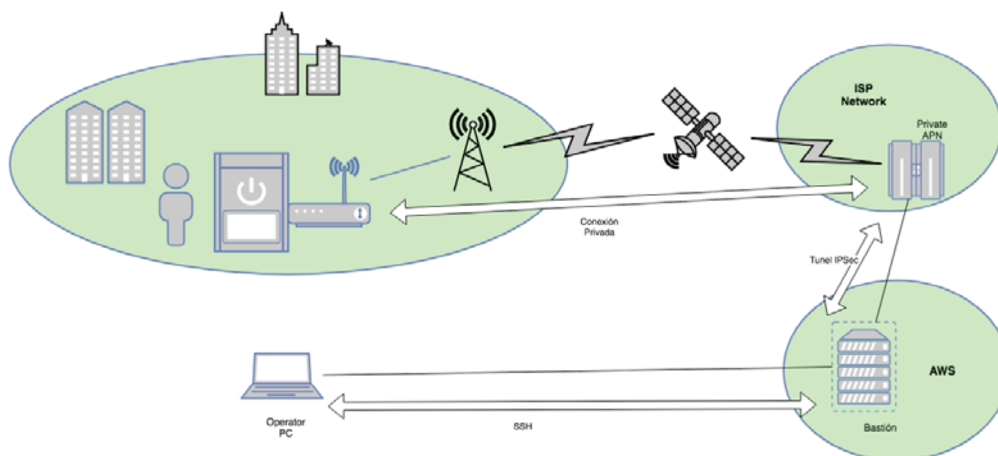

Figura 12.3: Comunicaciones si el operador provee el túnel

En el caso de hacerlo de manera interna por Ride On, el túnel se construye desde el router de la estación.

12.2.3. Interfaz API Restful

Como ya se ha comentado en otras partes del documento, se plantea una interfaz API RESTful. La motivación para esta interfaz es para la integración con otros sistemas, automa-

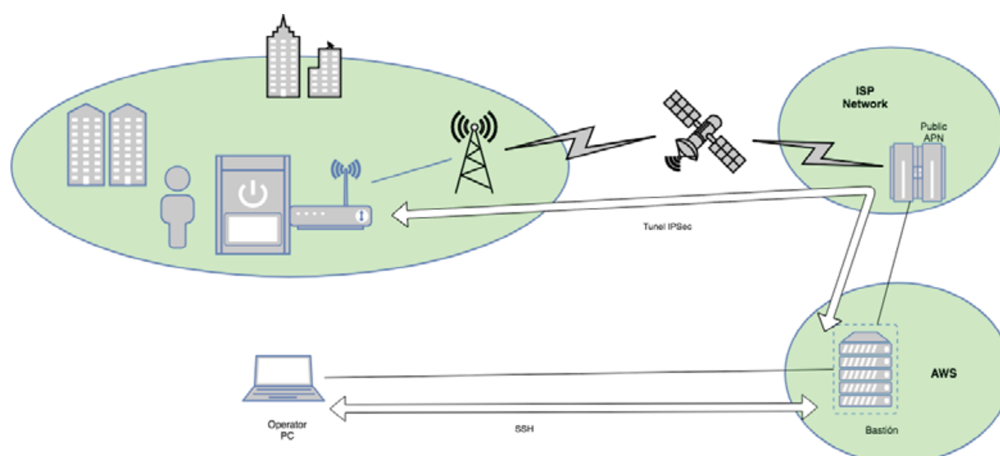

Figura 12.4: Comunicaciones si Ride On Consulting provee el túnel

tización y desarrollos futuros. Las interfaces mediante HTTP con arquitectura REST son muy comunes y resulta una manera sencilla de intercomunicar sistemas. Hay además herramientas como Postman, que acercan estas interfaces a usuarios de perfil menos técnico. A futuro, se plantea además dotar al proyecto de una interfaz web que aproveche esta fuente de datos y acciones.

Para definir la API y mantener un contrato continuado con los futuros clientes del proyecto, se decide utilizar OpenApi v3.0. Esto es un DSL específico de APIs RESTfuls basado en la idea original de swagger, pero tratando de ser un estándar libre no dominado por una compañía y sus intereses (si bien hay herramientas para compatibilizar entre uno y otro). De esta manera, una vez acordado el esquema de la interfaz (rutas, respuestas, parámetros, etc.) se puede generar el código fuente base tanto del servidor como del cliente. Mediante esta autogeneración de código se asegura que no haya problemas en la comunicación de los datos y eliminamos así una fuente de problemas.

Esto además permite versionar la propia interfaz, por lo que se pueden tener distintos servidores, según estén en una versión u otra, lo que facilita la migración o incluso mantener ambas si los requerimientos del proyecto lo necesitan.

Se puede encontrar en el anexo, la especificación de la API.

Respecto a como está definido respecto a la arquitectura general del resto del servidor, es un nuevo hilo que únicamente se dedica a consultar a Redis con los datos de la petición API, pero llegado el caso, podría separarse como un proceso independiente. Dada la escasa previsión

de uso que tiene esta interfaz no se cree necesario separarlo por el momento, como acabamos de comentar no está planteado que se abra al público, por lo que solo un grupo reducido de gente tendría acceso, aparte de otras aplicaciones de uso interno. Si esta situación cambiara, la recomendación sería separarlo en dos procesos: gestión de clientes y API de control.

12.2.4. Descripción del modelo de BB.DD

Aun cuando Redis no sea una BBDD relacional, esto no quiere decir que no tengamos que tener un modelo claro planteado.

El uso de Redis implica que no estamos demasiado preocupados por la persistencia de los datos. Dado que es una herramienta pensada para dar apoyo en la gestión de la plataforma y en muchos casos, el historico de hace un mes no es ya relevante. Y para la información de monitorización, si queremos guardar el histórico, tenemos otras herramientas a las que se puede exportar y analizar con mayor facilidad.

Por lo anterior y por las propia arquitectura de Redis de clave-valor, el modelo planteado resulta bastante sencillo. Si bien esto no es completamente cierto ya que valor cobra un sentido algo distinto en redis de lo que estamos acostumbrados. Redis permite almacenar estructuras de datos (listas, hashmaps, geodata, conjuntos, conjuntos ordenados, colas, etc.) por lo que va un paso más allá que otras BBDD de clave-valor.

Para simplificar, dividiremos las estructuras en dos grupos: los objetos que contienen información de un determinado elemento y las relaciones, que serán listas de los elementos anteriores. De esta manera con las herramientas proporcionadas por redis tendremos un conjunto de elementos que cumplen perfectamente nuestras necesidades.

A diferencia de una BBDD relacional, la propia clave es parte importante para relacionar unos elementos con otros, ya que es la única indexada y la búsqueda por otras partes, aunque posible, es mucho mas lenta por lo que para usos habituales está desaconsejada.

En la imagen ha tratado de utilizarse un diagrama de entidad-relación para representarlo, aunque puede que haya estándares mejores para representarlo, pero buscar un lenguaje común es una parte importante del desarrollo de un proyecto aun cuando esto suponga un cierto esfuerzo extra.

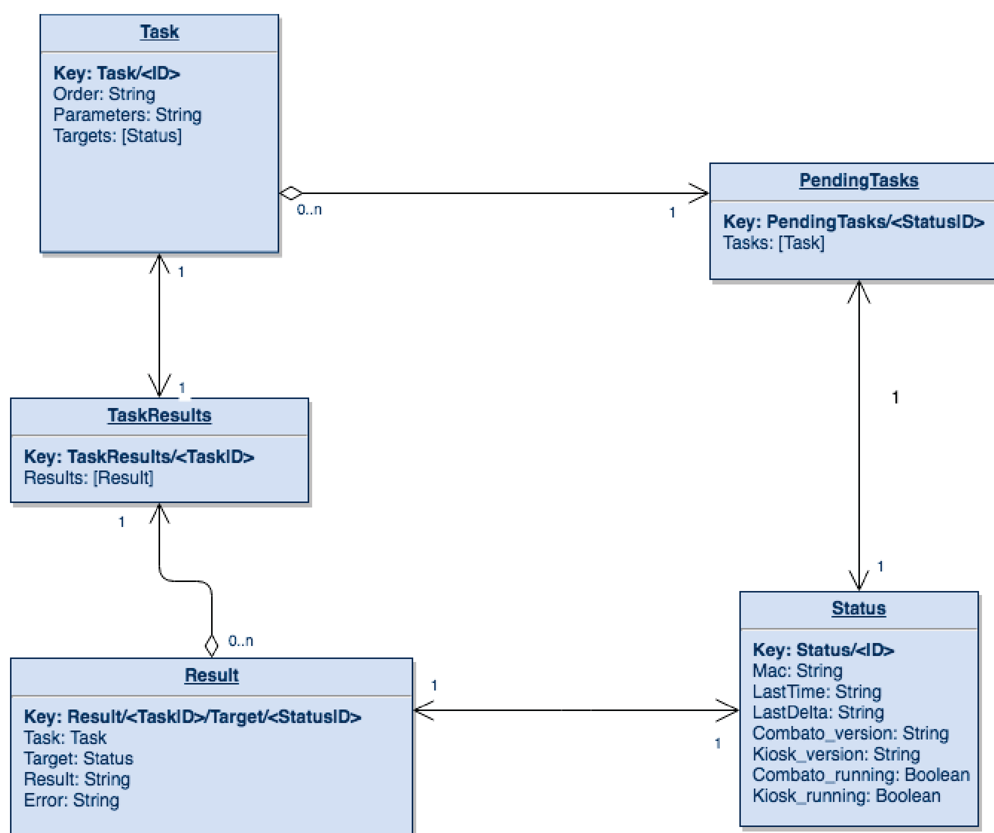

Figura 12.5: Modelo de BBDD de Redis

Hay dos elementos importantes:

status que contiene la información relativa a un kiosko.

task que contiene la orden original dada al sistema de cara a su distribución.

A partir de estos dos estructuras, se generan el resto de elementos que han de dar soporte al funcionamiento del servidor (el cliente es muy sencillo y no tiene necesidad alguna de mantener el estado mediante una BBDD).

Cada kiosco tiene un pendingtask asociado que mantiene la lista de tareas a ser enviadas al cliente en un momento dado. Esta estructura funciona como una cola y se irán eliminando elementos por orden de llegada.

Una vez el kiosko responda con el resultado, creará un Result y lo asociará a la tarea mediante taskresults.

Por tanto, TaskResults y Pending tasks funcionarían como el equivalente a una campo que generara la relación 1:n de una BBDD SQL tradicional.

Como ya anticipamos, el modelo planteado es tremendamente simple y utilizar una base de datos relacional iba a suponer más un coste que una ventaja aun cuando fuera perfectamente aplicable al modelo.

12.3. Descripción de la aplicación cliente

12.3.1. Arquitectura General

El cliente es a grandes líneas mucho más sencillo que el servidor. Sus labores se limitan a 3 grandes objetivos:

- Recabar información del sistema anfitrión.
- Ejecutar acciones en el sistema anfitrión.
- Comunicarse con el sistema central de manera periódica.

Es el último objetivo el que va a marcar el ritmo de los otros dos, ya que el cliente por si solo no tiene sentido y necesita del servidor para que su funcionalidad tenga razón de ser. Si la información recabada se queda en el cliente no tendríamos visibilidad del mismo, la ejecución de acciones en el sistema es algo que ya te daría cualquier Shell de Linux. Es al combinarlo con un sistema centralizado, que nos da la comodidad de la gestión remota.

El sistema se divide en un hilo de ejecución continuado, que se encarga de la comunicación periódica con el servidor central y es el que va a marcar el ritmo de otros dos hilos temporales:

- El primero se encarga de recabar la información. Lee de distintas fuentes del sistema varios parámetros. No se prevé que haya ningún elemento que provoque retrasos, por lo que, a menos que establezcamos ciclos de comunicación muy rápidos (inferiores al segundo),

no tiene sentido multiplexar este recabadado de información en distintas corutinas. Por lo que se hará de manera secuencial para simplificar el desarrollo. Si en algún momento se quisiera complicar, se recomendaría hacer un pequeño servidor de estado de hardware cuya tarea fuera únicamente mantener actualizado un vector de información del sistema, de manera que el cliente únicamente leyera la misma. Pero como ya se ha comentado, esto se puede hacer secuencialmente por el momento.

- El otro hilo, será el encargado de ejecutar las tareas en el sistema y devolver el resultado de la misma. Para ello, se utilizarán dos canales unidireccionales, uno para enviar las tareas a realizar y otro para devolver el resultado.

12.4. Fases del desarrollo

Como ya se ha mencionado, la metodología a utilizar tiene un carácter iterativo muy apropiado para el desarrollo de software. Es por ello que tener un plan de ruta claro es tremendamente útil para no caer en ciclos de desarrollo meramente correctivos tratando de rectificar decisiones que a corto plazo eran buenas, pero que viéndolo con cierta perspectiva se podrían haber evitado. Tampoco es bueno caer en un intento de planificación absoluto. Si esa fuera la idea, se iría a un modelo de desarrollo más cercano al tradicional "waterfall", el cual es también muy típico en el desarrollo de software, pero que se ha ido quedando relegado a entornos más conservadores. Además, plantear versiones intermedias que ya puedan tener un uso nos permite ir avanzando en otros frentes que también hay que tener en consideración, como es el afrontar el despliegue y la configuración en las estaciones.

Fase 0 En esta fase, únicamente se espera que el sistema reporte un estado parcial de la estación al servidor y que este lo almacene. Lo consideramos 0 por la escasa utilidad que proporciona.

Fase 1 Con esta fase, ya tenemos algo de uso, ya que se implementa el envío de comandos. Poder lanzar un comando de "*shellscript*" es el objetivo de esta fase. No se espera recibir el resultado.

Fase 2 Recepción de resultados de un comando. Recuperamos la salida del comando y la mandamos de vuelta al servidor central.

Fase 3 Notificaciones. En caso de ciertos eventos, levantar un aviso de problemas. Ejemplos de esto puede ser: última comunicación de la estación hace más de 15 min., temperatura muy alta, alguno de los “daemons” está parado, etc. En el caso de Ride On Consulting, esto significa una interfaz directa con Pager Duty, que es el sistema contratado para la gestión de incidencias.

Fase 4 Tipos de Comandos. En esta fase del desarrollo se implementan los tres tipos posibles de comandos: internos (implementados en el propio cliente), recetas (comandos habituales) y Shell scripts (este tipo en realidad ya está hecho, pero pueden plantearse mejoras). Además, debe crearse la forma de actualizar las recetas, para asegurar la consistencia de resultados entre los distintos clientes.

Fase 5 Interfaz API. Hasta esta fase, la interacción con el sistema se hace mediante el acceso a Redis y consultas sobre la misma. Desde este punto, ya se pueden hacer peticiones a la API REST de una manera completa. Si bien puede que algunas partes de la interfaz RESTful se implementen en alguna de las fases anteriores, no se prevee hacerla completa hasta este punto ya que no es indispensable para interactuar con el sistema, sino un facilitador.

Parte IV

Conclusiones y líneas futuras

Capítulo 13

Conclusiones

13.1. Estado actual del proyecto

El proyecto se encuentra actualmente en uso y en continuo desarrollo. Como ya se ha comentado anteriormente, se planteó un diseño por fases, que permitiera sacar rendimiento del desarrollo y pararlo/continuarlo según fuera permitiendo el estado de la compañía y sus proyectos. Las primeras fases están cubiertas y actualmente se está integrando con el sistema de notificaciones (ver fase 3 más arriba).

Aunque aun no se ha probado con un proyecto a gran escala (tenemos planes a corto plazo de soportar más de 120 estaciones en un solo proyecto) nada de lo probado hasta el momento hace prever ningún problema al aumentar el número de clientes en una cantidad considerable.

13.2. Problemas encontrados

Identificación unívoca de los kioskos. En Ride On Consulting tenemos muchas herramientas y cada una estaba identificando la estación de una manera diferente. A partir de este proyecto, se tomó la decisión de estandarizar el sistema y utilizar la Mac del ordenador como referente interno. Además de cara a otros ámbitos se utilizan nombres más fácilmente pronunciables para humanos, pero siempre sabiendo que estos son únicamente alias para facilitar la identificación, manteniendo la dirección de la capa MAC (**M**edia **A**ccess **C**ontrol) como referente unívoco.

Tiempos de desarrollo. Aunque la herramienta era claramente una necesidad de cara al mantenimiento del sistema, no es una que apoye la cara más marketiniana de la empresa, por lo que es fácil que no se le de tantos recursos como a otros elementos que si facilitan el ganar nuevos proyectos. Es por ello, que incluso las fases iniciales, tuvieran tiempos de desarrollo más largos de lo esperado.

Cobertura de tests. Ride On Consulting, es una compañía que, a nivel general, se toma tanto el testeo como las labores de QA (**Q**uality **A**ssurancce) muy en serio. Sin embargo, debido a la falta de tiempo y a que algunas librerías de Go no facilitan las labores de testeo, la herramienta no tiene la cobertura de test que gustaría para dar más fiabilidad. Se espera

que a futuro esta situación cambie, pero es un inconveniente con el que tenemos que seguir desarrollando.

Capítulo 14

Líneas futuras

Desde el comienzo del proyecto, se han planteado muchas ideas de lo que podría llegar a hacer, si bien tres han sido las más claras:

Detección de "tampering". Poder saber si alguien ha tocado el sistema y levantar la alarma de ello. Para ello la información recuperada por el cliente será mucho más abundante y además deberá compararse bien con datos previos o con los resultados de otras estaciones, de manera que puedan apreciarse las anomalías en el funcionamiento.

Autoconfiguración del sistema. Una de las líneas futuras es la autoconfiguración de una estación, de manera que se le instala la imagen, se registra en el servidor como nuevo elemento y cualquier valor que pueda necesitar. Al arrancar el sistema, se comunica con el servidor central y lanza las ordenes necesarias para dar la parte de unicidad (creación de claves de SSH, establecer variables de entorno, recuperar claves de acceso, etc.) Si bien esta idea se sabe que requerirá un estudio concienzudo por ver los cambios que implicará.

Interfaz gráfica. Al ser una herramienta interna, no se ha prestado mucha atención a los detalles de interfaz, pero hay claramente mejoras en este aspecto, comenzando por dar una interfaz sobre el API Restful de manera que un operador sin conocimientos pueda interactuar con la plataforma. En el momento que este tipo de usuarios entra en juego, no solo hay que tener consideraciones de usabilidad (transmitir correctamente la información de manera clara) sino también de seguridad y para prevenir accidentes.

Siglas

API **A**pplication **P**rogramming **I**nterface. 44, 51, 53–55, 59, 68, 69

APN **A**ccess **P**oint **N**ame. 16, 69

AWS **A**mazon **W**eb **S**ervices. 25, 69

BBDD **B**ases de **D**atos. 28, 42, 55–57, 69

DCVS **D**ecentralized **C**ontrol **V**ersion **S**ystem. 43, 69

DevOps **Dev**eloper **Op**eration**s**. 23, 43, 69

DSL **D**omain **S**pecific **L**anguage. 22, 69

HTTP **H**yper**T**ext **T**ransfer **Protocol**. 44, 54, 69

IP **I**nternet **P**rotocol. 69

JS **J**ava**S**cript. 24, 44, 69

JVM **J**ava **V**irtual **M**achine. 22, 23, 69

MAC **M**edia **A**ccess **C**ontrol. 64, 69

MB **M**ega**B**ytes. 25, 69

QA **Q**uality **A**ssurancce. 64, 69

REPL **R**ead **E**val **P**rint **L**oop. 23, 69

REST **RE**presentational **S**tate **T**ransfer. 35, 44, 51, 53, 54, 59, 69

SSH **S**ecure **SH**ell. 4, 6, 16, 19, 68, 69

URI **U**niversal **R**esource **I**dentifier. 44, 69

Bibliografía

[But16] Matt Butcher. *Go in Practice*. Manning, 2016.

[Cox09] Russ Cox. Go data structures. 2009.

[Cox11] Russ Cox. Profiling go programs. 2011.

[Ken16] William Kennedy. *Go in Action*. Manning, 2016.

[Nyg18] Michael Nygard. *Release It!* Pragmatic Programmers, 2018.