



Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

**Desarrollo de Programas en el Sistema
Operativo Fuchsia**

Autor: Josef Manuel Steiner Ramos

Tutor(a): Fernando Pérez Costoya

Madrid, junio 2022

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Título: Desarrollo de Programas en el Sistema Operativo Fuchsia

Junio 2022

Autor: Josef Manuel Steiner Ramos

Tutor:

Fernando Pérez Costoya

Arquitectura y tecnología de sistemas informáticos

ETSI Informáticos

Universidad Politécnica de Madrid

Resumen

Fuchsia es un sistema operativo bastante prometedor hoy en día en el que ya se ha invertido una gran cantidad de tiempo y esfuerzo por parte de Google y de su comunidad para poder verlo en un futuro como el principal sistema operativo del mercado, ya sea para dispositivos móviles como teléfonos celulares como para ordenadores de sobremesa.

El objetivo de este trabajo es ayudar a ese desarrollo aportando información, código y documentación respectiva tanto a Fuchsia como a su núcleo Zircon con el fin de proveer tanto a los nuevos usuarios/desarrollares como a los que ya llevan tiempo involucrados de referencias y bases para conocer y mejorar cada vez más este sistema operativo.

El trabajo se dividirá en una parte de documentación en la que se expondrán las bases del funcionamiento de Fuchsia y en una parte práctica en la que se pondrán a prueba dichos conocimientos desarrollando y probando código para ser ejecutado en este entorno.

Abstract

Fuchsia is a very promising operating system nowadays in which a lot of time and effort has already been invested by Google and its community to be able to see it in the future as the main operating system on the market, whether for devices mobiles such as cell phones or for desktop computers.

The main purpose of this work is to help this development by providing information, code and documentation about both, Fuchsia and its Zircon core, in order to provide both new users/developers and those who have been involved for a long time with references and bases to understand and improve this operating system even more.

The work will be divided into a documentation part in which the basics of Fuchsia's operation will be explained and a practical part in which that knowledge will be tested by developing and running code to be executed in this environment.

Índice de Contenidos

1	Introducción	1
2	Estado del Arte	3
3	Estudio del Microkernel Zircon.....	5
4	Estudio del sistema operativo Fuchsia.....	12
4.1	Fuchsia Layer Cake	12
4.2	Diseño de Fuchsia actual	14
4.2.1	Drivers Layer	15
4.2.2	Fuchsia Runtime Layer.....	17
4.2.3	Component Layer	19
4.2.4	File System Layer.....	27
4.2.5	Application Layer	31
5	Estudio de los aspectos específicos de la programación en Fuchsia .	32
6	Selección de programas a desarrollar y diseño de los mismos.....	37
7	Desarrollo de los programas	38
7.1	Drivers.....	38
7.1.1	Implementación	38
7.1.2	Resultados	45
7.2	Componentes.....	46
7.2.1	Implementación	47
7.2.2	Resultados	57
8	Conclusiones	62
9	Análisis de Impacto	64
10	Bibliografía.....	65

Índice de Figuras

Figura 1. Diferencias entre kernel monolítico y microkernel [8].....	5
Figura 2. Diseño estimado de Fuchsia [12]	7
Figura 3. Esquema de un microkernel [12]	7
Figura 4. Código de un fichero .fidl [13]	8
Figura 5. Esquema de comunicación IPC de Fuchsia [13]	9
Figura 6. Esquema de comunicación IPC de Fuchsia [14]	9
Figura 7. Fuchsia Layer Cake [19]	12
Figura 8. Diseño de la arquitectura de Fuchsia [24].....	14
Figura 9. Comparativa de API vs ABI [26]	16
Figura 10. Device ops [29].....	17
Figura 11. Capacidad de runner a un componente [32].....	18
Figura 12. Registro de un nuevo resolver [35]	20
Figura 13. Ejemplo de las relaciones entre “offer” y “expose”	21
Figura 14. Esquema de los directorios de un hub [44]	24
Figura 15. Esquema de un realm con una colección	25
Figura 16. Monikers relativos	26
Figura 17. Esquema de un namespace [38].....	28
Figura 18. Inicio (modo headless) del emulador de Fuchsia.....	34
Figura 19. Inicio (con GUI) del emulador de Fuchsia	35
Figura 20. Interfaz de ajustes del emulador de Fuchsia.....	35
Figura 21. Shell en el emulador de Fuchsia.....	36
Figura 22. Chromium en el emulador de Fuchsia	36
Figura 23. Test #1 /dev/full	45
Figura 24. Test #2 /dev/full	46
Figura 25. Esquema del realm.....	46
Figura 25. Estado inicial del sistema	57
Figura 26. Creación e inicialización de la instancia de reinfuch.....	58
Figura 27. inicialización de la instancia de fuchsiacomp2.....	59
Figura 28. Comprobación de salidas #1	60
Figura 29. Comprobación de salidas #2	60
Figura 30. Estado de los directorios después de la ejecución.	61

Índice de Tablas

Tabla 1. Relación del tipo vulnerabilidades – cantidad ubicada en los drivers para 2020.....	11
---	-----------

1 Introducción

Como es bien sabido por parte de todos, la empresa norteamericana Google siempre ha estado a la vanguardia en cuanto a innovación y desarrollo tecnológico se refiere, esto le ha permitido, a lo largo de los años, mantenerse puntera en todos los mercados de los que ha formado parte, liderando todo tipo de proyectos en gran cantidad de sectores.

Uno de esos sectores, el de software, es el que detallaremos en las próximas páginas, más concretamente, Fuchsia, sistema operativo descubierto en GitHub en agosto de 2016 y desarrollado totalmente desde cero por parte de Google. Este novedoso software de código abierto plantea la premisa de ser sistema operativo modular de tiempo real (RTOS, siglas en inglés de real-time operating system) independiente del lenguaje de programación e independiente de la plataforma, conceptos en los que ahondaremos más adelante, permitiendo así no solo el desarrollo de software en lenguajes como Go, Rust, Dart, C, etc. Si no también pudiendo ser ejecutado tanto desde smartphones, tablets u ordenadores.

Una de las principales características de este sistema operativo y que también explicaremos con más detalle a lo largo de todo este trabajo, es el hecho de que a diferencia de sus “hermanos” Android y Chrome OS, que están basados en un núcleo Linux, es decir, presentan una arquitectura monolítica donde todos los privilegios o llamadas al sistema (por ejemplo, gestión de memoria, operaciones E/S, gestión de interrupciones hardware o de la pila del CPU) recaen o son responsabilidad del kernel. Fuchsia hace uso de Zircon, microkernel desarrollado también por Google que presenta solo los controladores principales y que está basado en Little Kernel, pequeño sistema operativo para dispositivos embebidos que busca precisamente responder a esa necesidad o deseo de mantener al mínimo las responsabilidades del kernel mode (espacio del núcleo) delegando el resto al user mode (espacio del usuario) para así lograr un núcleo menos complejo y de menor tamaño, permitiendo, por ejemplo, aumentar la portabilidad entre plataformas hardware.

Zircon no solo es un núcleo más ligero y descentralizado, sino que también implementa lo que se conoce como FIDL (siglas en inglés de Fuchsia Interface Definition Language), protocolo para la comunicación entre procesos (IPC, siglas en inglés de Inter-Process Communication) que provee además de un simplificado lenguaje “C-like”, una serie de herramientas que permite al desarrollador del service no preocuparse por tener que realizar una implementación del protocolo de comunicación para cada posible lenguaje que use la parte cliente como serían los ya mencionados anteriormente si no que este solo tendría que usar este sistema para poder lograr la comunicación entre el servicio y los clientes independientemente del lenguaje. [1]

Cuando se habla de Fuchsia hay dos conceptos que hay que tener muy en cuenta a la hora de buscar entender el funcionamiento de este sistema operativo, como lo son Arquitectura modular del sistema y Diseño modular de las aplicaciones.

- Arquitectura modular del sistema, Fuchsia es un sistema operativo que además de contar con su base (Zircon) también cuenta con otras capas o módulos por encima, cada una de las cuales cuenta con sus propias responsabilidades dentro del funcionamiento de este sistema operativo.
- Diseño modular de las aplicaciones, esto quiere decir que todo el software como por ejemplo archivos de sistema o ejecutables están contenidos en unidades modulares denominadas packages los cuales a su vez están formados por pequeñas piezas de software diseñadas para hacer un trabajo específico llamadas components permitiendo así, entre otras cosas, un mayor control de fallos.

Teniendo esto en cuenta, también hay que mencionar una herramienta que tiene un papel clave dentro del ecosistema de Fuchsia, y esa es Flutter. Flutter es un kit de desarrollo de software que por un lado está programado en Dart mientras que por otro y como justamente dice en su página web, permite desarrollar aplicaciones multiplataforma (“Build apps for any screen”) como Android, IOS o Web de una forma bastante sencilla, estas dos razones hacen que sea perfecto para ir de la mano con Fuchsia, cuya base es la de crear un entorno que permita a los usuarios, ya sean desarrolladores o no, tener el mismo sistema operativo y las mismas aplicaciones en todos sus dispositivos sin importar si están escritos en C, Go o Python.

Más adelante, se desarrollarán una serie de códigos de ejemplo que permitan documentar y representar este entorno de programación.

2 Estado del Arte

Si bien es verdad que Fuchsia incorpora una metodología y conceptos muy prometedores y novedosos, los cuales iremos viendo de forma más detallada a lo largo de este trabajo, es recomendable primero echar un vistazo a los avances que han sido realizados hasta la fecha y lo que faltaría por realizar, tanto en el ámbito general de los sistemas operativos y microkernels, como a nivel de Fuchsia en específico, es decir, lo que se ha hecho y lo que falta por hacer.

Previo al desarrollo de Zircon, y en general, de Fuchsia, podemos encontrar dos tecnologías con las que comparten bastantes similitudes, las cuales serían por un lado Little Kernel y por otro Mach.

Zircon, como ya fue mencionado, es un microkernel cuyo código está basado en Little kernel, el cual es un pequeño sistema operativo diseñado para servir de base a dispositivos integrados o embebidos como sería el caso de dispositivos inteligentes, wearables, bootloaders, etc., y que permite la utilización de primitivas del sistema operativo como threads o mutex [2] a cambio de un bajo coste y tamaño, un ejemplo de esto, es su uso en el proceso de ‘booteo’ de dispositivos Android, concretamente en el modo ‘Aboot’ (implementación del protocolo “fastboot” [3] que permite realizar modificaciones en la memoria flash vía USB).

Esto ya es una gran ventaja para Fuchsia y para Zircon, ya que le permite partir desde cero tomando ideas de otras tecnologías del sector como lo es en este caso Little kernel para así desarrollar un sistema operativo que pueda ser usado por cualquier dispositivo y que además que no requiera de tantas prestaciones ni sea tan grande como las otras alternativas del mercado como por ejemplo Linux, que cuenta con más de 20 millones de líneas de código.

Además de esto, también podemos remontarnos a algunas décadas atrás cuando se empezó a desarrollar Mach, microkernel de primera generación desarrollado en la Carnegie Mellon University y que proponía un sistema operativo donde la mayor parte de código, a excepción de lo que respecta a la comunicaciones entre procesos, los threads, direcciones de memoria virtual, etc., estaba ubicado fuera del núcleo en pequeños programas denominados servidores y los cuales se comunicaban mediante puertos similares a los pipes de Linux. [4].

Lo interesante de esta comparación es que Mach también propuso en su momento una tecnología similar a FIDL en Fuchsia llamado MIG [5] (Mach Interface Generator) que, sin ser tan completo como este, sirve para generar automáticamente código en forma de interfaces de llamadas de procedimientos remotos que permitían facilitar la interacción entre procesos.

El código generado hace de mediador entre el cliente y los programas, de forma que, son estas funciones las que toman los argumentos, los codifican y los transforman en mensajes IPC (cada uno con su respectivo ID) que serán enviados a través de los puertos; Una vez enviados esperan la respuesta para luego realizar el proceso inverso y entregar el resultado al cliente (sin que este tenga que involucrarse o saber cómo se está llevando por debajo la comunicación entre procesos, aligerando mucho más la carga sobre los desarrolladores).

Con el paso del tiempo, si bien el proyecto de Mach fue quedando cada vez más abandonado debido a sus constantes problemas de rendimiento, en los que no solo se veía involucrado la sobrecarga del mecanismo de mapeo de memoria que era usado para dar soporte al gran número de mensajes IPC que se transmitían, sino que también, y en mayor medida, estaban involucrados la gran cantidad de trabajo que tenía el kernel a la hora de validar y comprobar los derechos de dichos mensajes o por ejemplo la mala optimización en la gestión de memoria para sistemas multiprocesadores [6], esté sirvió de base para el futuro desarrollo de otros sistemas como macOS o los sistemas UNIX, como por ejemplo el Tru64 [7], de las plataformas Alpha de HP.

La situación que se experimentó a lo largo del desarrollo de Mach evidencia que no ha sido una tarea simple lograr un microkernel que sirva de base para un sistema operativo de forma eficiente, y es aquí donde entra la importancia de Fuchsia y Zircon, los cuales no solo plantean un sistema eficiente y bien trabajado, sino que también plantean un sistema operativo universal apto para cualquier tipo de dispositivo, a diferencia de Linux.

3 Estudio del Microkernel Zircon

Para poder entender cómo funciona Fuchsia, así como su entorno de desarrollo y las ventajas que ofrece al mundo de la tecnología, debemos primero entender cómo funciona la base sobre la cual esta implementada este innovador sistema operativo, y el porqué de su estructura, para esto primero analizaremos las diferencias entre los kernels monolíticos “tradicionales” como serían, por ejemplo, los que sirven de base para Android, y los microkernels.

Un kernel monolítico es un tipo de núcleo de los sistemas operativos, el cual posee la responsabilidad de gestionar todos los servicios del sistema en los que se conoce como el kernel mode, es decir, posee los privilegios sobre los sistemas de archivos, IPC, E/S y gestión de los dispositivos conectados, así como de los procesos y por último y lógicamente del hardware.

Por otro lado, un microkernel, es un tipo de núcleo que solo se reserva la responsabilidad de gestionar los servicios básicos del sistema como sería la IPC y la gestión tanto de subprocessos como del espacio de direcciones de bajo nivel; Estas serían las únicas funciones ejecutadas en el kernel mode o modo administrador, dejando el resto de las funciones como por ejemplo gestión de dispositivos, la pila o sistema de archivos ejecutando como procesos servidores el user mode en forma de módulos.

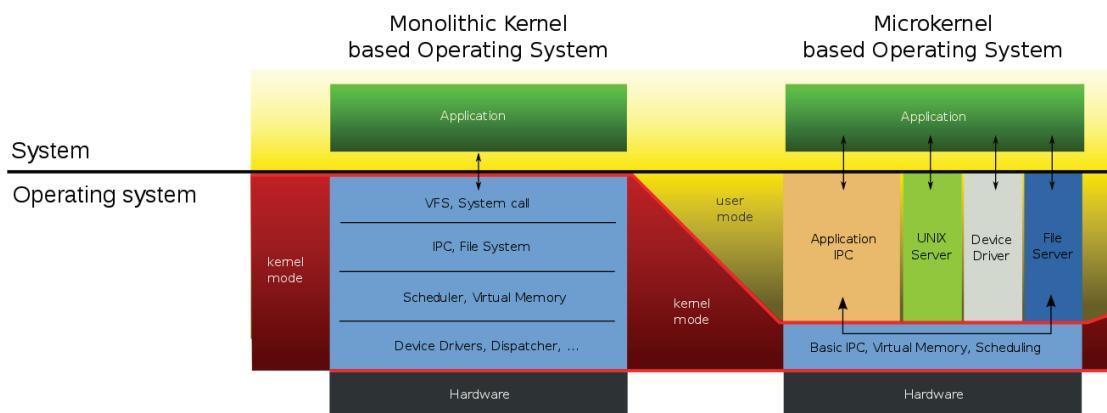


Figura 1. Diferencias entre kernel monolítico y microkernel [8]

A raíz de esta diferencia sobre cómo funciona cada uno de estos dos tipos de kernels podemos encontrar una serie de ventajas y desventajas a la hora de usar uno u otro de ellos como base para nuestro sistema operativo.

- 1- Tamaño, el microkernel presenta un tamaño bastante menor al de un núcleo monolítico debido a que como ya fue mencionado, este solo debe ejecutar el código mínimo para asegurarse de cumplir las funciones básicas, en cambio, el monolítico presenta un código mucho más complejo y extenso.
- 2- Rendimiento, si bien un micro kernel ahorra tamaño a la hora de delegar funciones al user mode y adquirir una estructura modular, lo hace a costa de sacrificar algo de rendimiento ya que en este caso los módulos deben estar en constante intercambio de mensajes para que funcione el sistema operativo, mientras que el monolítico concentra todas las primitivas y funcionalidades dentro de un mismo espacio de memoria (anillo 0) [9] gestionándolas mediante señales y sockets.
- 3- Tolerancia a fallos, ya que, en caso de error, ya sea fatal o no, si cada funcionalidad tiene su ‘espacio’ solo se produce el fallo ahí, por ejemplo, un desbordamiento de buffer, permitiendo que el resto del sistema pueda continuar funcionando de forma estable, en contraste con lo que sucede en un núcleo monolítico donde un error, al estar todo integrado en un mismo espacio de memoria, puede propagarse a todo el núcleo.
- 4- Escalabilidad y Mantenibilidad, un núcleo monolítico, al estar integrado todo el código en un mismo espacio, hace que sea más difícil cambiar, actualizar, eliminar o depurar una funcionalidad o módulo en concreto ya que habría que recompilar todo el código y reiniciar el sistema, al contrario de lo que sucede en un microkernel donde los servicios están divididos en módulos ahorrando tiempo y memoria.

Una vez presentadas las diferencias más relevantes en cuanto al funcionamiento y estructura de estos dos tipos de núcleos podemos ir entendiendo cada vez más por qué Google tomó la decisión de basar su sistema operativo en Zircon.

Zircon (también desarrollado por Google) es una plataforma que está compuesta por el kernel propiamente dicho, así como por un pequeño conjunto de servicios en el user mode, drivers y librerías necesarios para que el resto del sistema operativo que está por encima pueda funcionar [10].

Este núcleo representa el nivel más bajo y con más privilegios de Fuchsia, es decir, el “anillo 0” que se puede apreciar en la figura 2, el cual “provee al sistema de llamadas (“syscalls”) para llevar a cabo la gestión de procesos, threads, memoria virtual (VM , siglas en inglés de Virtual Memory), IPC, además de mecanismo de espera o bloqueo” [11]

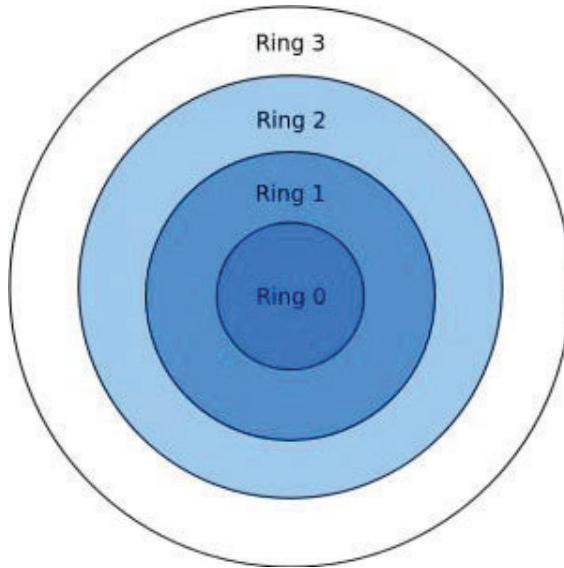


Figura 2. Diseño estimado de Fuchsia [12]

Zircon agrupa diferentes grupos de componentes que se ejecutan en un proceso cada uno, pero manteniendo la separación entre ellos, con esto lo que se busca es mantener una capa de seguridad entre cada módulo para así lograr que en caso de que haya algún tipo de falla en alguno de ellos, no comprometa la seguridad de los otros módulos como se ve en la Figura 3. [12]

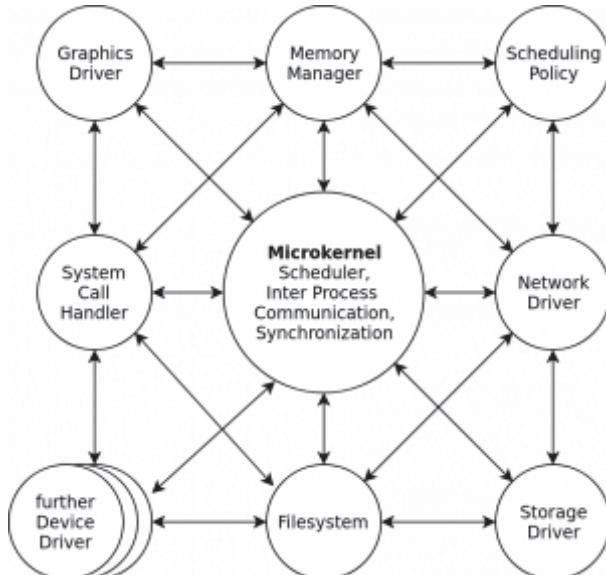


Figura 3. Esquema de un microkernel [12]

Ya explicada la estructura base, toca hablar de cómo se comunican cada uno de esos módulos o capas, y aquí es cuando entra en escena FIDL. FIDL es un lenguaje usado y desarrollado con dos principales usos, el primero, y el más importante es que sirve como interfaz para abstraer la comunicación entre cada uno de los módulos o procesos permitiendo como se mencionó con anterioridad, la comunicación e intercambio de información entre cliente – servicio sin importar el lenguaje de programación usado.

Para lograr esto, el desarrollador crea lo que se conoce como “FIDL Definition file” usando la extensión .fidl, fichero donde se definen los métodos, tipos y parámetros que se usarán en el servicio.

```
1 library fidl.examples.echo;
2
3 @discoverable
4 protocol Echo {
5     EchoString(struct {
6         value string:optional;
7     }) -> (struct {
8         response string:optional;
9     });
10
11     SendString(struct { value string:optional; });
12
13     ->ReceiveString(struct { response string:optional; });
14 }
```

Figura 4. Código de un fichero .fidl [13]

De esta forma, y siguiendo una estructura y sintaxis similar a C, se definen una serie de componentes como serían el nombre de librería que se usará para distinguir la información que está dentro del fichero, así como los protocolos y los métodos usados para realizar las diferentes funcionalidades.

La estructura de los protocolos, como se puede ver en la figura 6, puede ser de tres formas:

- “Two-way methods”: métodos síncronos que usan el operador -> para indicar el return, método EchoString().
- “One-way methods”: métodos asíncronos sin return declarado, método SendString().
- “Events”: usan el operador -> para indicar mensajes no solicitados por el cliente desde el servidor, método ReceiveString().

Una vez creado este fichero, una herramienta denominada “FIDL compiler” (fidlc) valida y compila dicha librería a través de la llamada `fidl()` y la transforma en un “JSON Intermediate Representation” (IR), que luego será la base para generar los “FIDL bindings” [13]; es decir, se encarga de generar automáticamente tanto una implementación del código para la parte cliente como una implementación para la parte de servicio (cada una de ellas siguiendo una estructura tipo) que les permitan usar los protocolos definidos dentro del fichero `.fidl`.

Estos bindings se encargan de codificar y decodificar los “FIDL messages” y enviarlos a través del canal IPC de Zircon para luego realizar la traducción de estos hacia funciones de más alto nivel en sus respectivos lenguajes o para implementarlos.

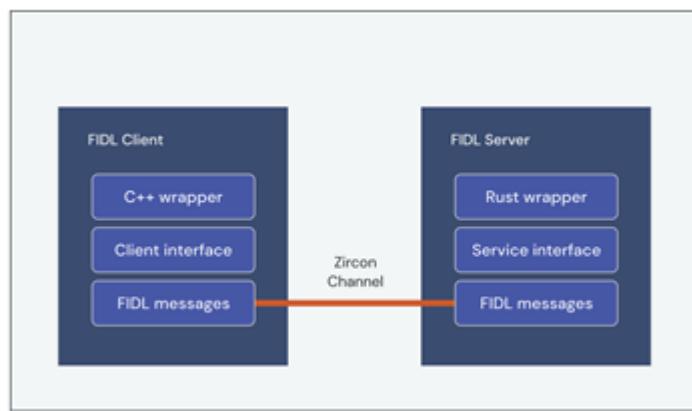


Figura 5. Esquema de comunicación IPC de Fuchsia [13]

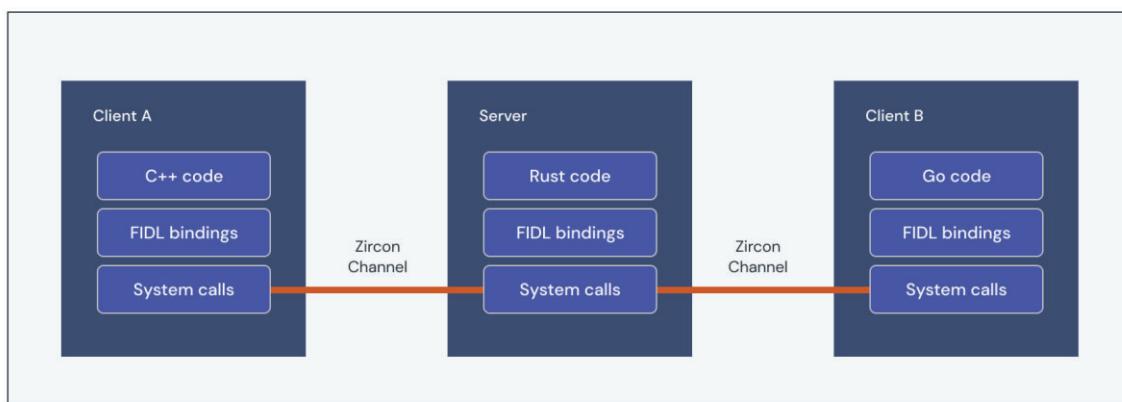


Figura 6. Esquema de comunicación IPC de Fuchsia [14]

Con esto se logra que a partir de un único fichero sea posible generar implementaciones cliente-servicio para una gran variedad de lenguajes sin que el desarrollador tenga que hacerlo manualmente, los lenguajes disponibles de momento serían “C, Low-Level C++, High-Level C++, Dart, Go, Rust”, además de otros lenguajes, los cuales se tiene intención de incluir como Java, JavaScript, etc.

El segundo de los usos de FIDL es que este mismo lenguaje también es usado por Zircon para llevar a cabo la comunicación entre las aplicaciones y los drivers de sus respectivos devices (comúnmente externos); por ejemplo, enviando mensajes de control para dar información que luego será usada al hacer un `read()` del device. [15]

Normalmente, en un entorno POSIX ('interfaz para desarrollar aplicaciones portables basado en UNIX') estas comunicaciones se harían a través de la syscall `ioctl()` y su respectivo controlador en el lado del driver, en Zircon/Fuchsia se hace uso de FIDL gracias a su facilidad de clasificar información y generar automáticamente implementaciones tanto para el driver como para el cliente. De esta forma solo habría que crear el correspondiente fichero `.fidl`, compilarlo para generar los códigos correspondientes y añadir handlers en el driver para que este sea capaz de recibir los mensajes que le envíe el cliente y este a su vez sea capaz de enviarle alguna respuesta en caso de ser necesario.

Al utilizar este protocolo no solo logramos esa libertad de lenguaje que hemos mencionado a lo largo de este trabajo, ya que al “abstraer la definición de mensajes de su implementación”, el “FIDL Compiler” puede generar código en gran variedad de lenguajes diferentes sin necesidad de que los desarrolladores tengan que hacer trabajo adicional, dando más libertad a la hora de diseñar los drivers de algún dispositivo y sobre todo garantizando un mayor número de potenciales terminales compatibles; si no que también se logra un software mucho más seguro y con menor cantidad de bugs.

Para comprender esta mejora en la seguridad hay que partir del hecho de que por compatibilidad con Linux/Unix, la mayoría de los drivers por convención están escritos en C (aunque algunos hicieron la transición a C++). Y si bien es un lenguaje que le permite al desarrollador tener un alto grado de libertad a la hora de optimizar y gestionar la memoria, la pila y el kernel esto también puede ser un arma de doble filo, dando lugar a fallos relacionados con Overflow's, accesos “out of bounds”, accesos “use after free”, entre otros; fallos en los que hay que invertir esfuerzo extra para depurarlos o solucionarlos y que casos más graves pueden derivar en ataques mediante códigos maliciosos, o ataques DoS.

En 2017, año en el que más vulnerabilidades se reportaron según los datos publicados por “CVE Details”, se analizaron 65 de las vulnerabilidades y exposiciones más comunes reportadas (Ejecución de código) para el kernel de Linux y resultó que, de 40 de los bugs prevenibles estudiados (todos relacionados con “memory safety”), 39 estaban ubicados en los drivers del dispositivo. [16]

Si a estos datos les agregamos que 26 de los 51 bugs por Overflow de ese año estuvieron relacionados con los drivers, y que de estos 26, 22 dieron lugar a ataques DoS nos damos cuenta de que solo en 2017 el 56% de los bugs estudiados se originaron debido a carencias en la seguridad de los drivers y la memoria.

Mientras que, por otro lado, en 2020 casi el 39% [17] de este tipo de vulnerabilidades de seguridad tuvieron estas características, cantidad menor pero igualmente a tener en cuenta cuando se quiere evitar bajo cualquier concepto, fallos o ataques por códigos maliciosos.

Año	Vulnerabilidad	Relacionadas con los drivers	
2017	Ejecución de código	39/65 *	56%
2017	Overflow	26/51	
2020	Ejecución de código	1/4	39%
2020	Overflow	6/14	

* 39/40 estaban relacionados con memory safety

Tabla 1. Relación del tipo vulnerabilidades – cantidad ubicada en los drivers para 2020

Claramente no se puede culpar a C de ser la causa de todos los fallos, pero habiendo tantas mejoras en la seguridad, lenguajes que permiten más robustez en sus códigos y tantos avances innovadores como el que precisamente plantea Google con Fuchsia y su manejo de los drivers y el IPC, no sería una mala opción en ir pensando en adoptar su lenguaje para evitar futuras complicaciones.

4 Estudio del sistema operativo Fuchsia

En este punto nos enfocaremos más en cómo funciona el sistema operativo en su totalidad. Cuando se habla de Fuchsia es imposible no mencionar el concepto de ‘modularidad’ y es que, sin ir más lejos, es una de las bases mediante las cuales fue desarrollada su arquitectura.

Para entender esto podemos poner como ejemplo el cómo estaba estructurado Fuchsia en un principio, con lo que se denominaba “The Fuchsia layer cake” [18], esta arquitectura de capas estaba compuesta además de por Zircon, que hacia de base y soporte, por otras tres capaz principales denominadas “Garnet”, “Peridot” y “Topaz”.

4.1 Fuchsia Layer Cake

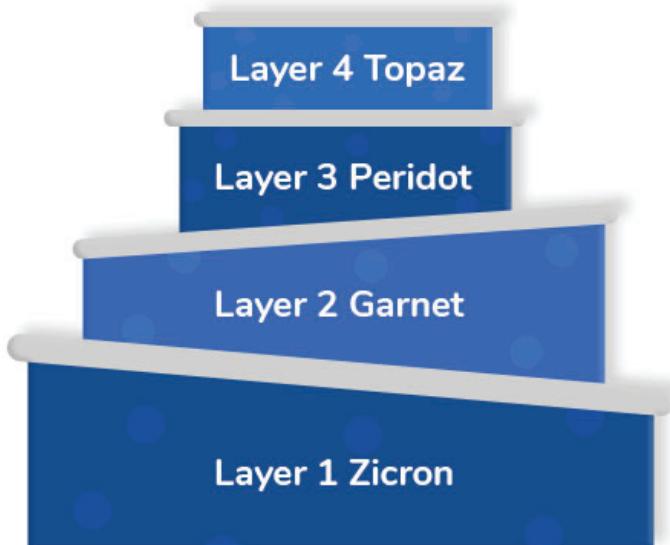


Figura 7. Fuchsia Layer Cake [19]

En esta arquitectura cada una de las capas se encargaba de cumplir sus respectivas tareas. “Garnet”, la capa más inferior luego de Zircon, era la encargada de gestionar los drivers de los dispositivos, los servicios de red, el renderizado de gráficos (mediante su procesador de gráficos “Escher”) así como de instalación de software (mediante su sistema de actualizaciones “Amber”).

Luego de “Garnet” estaría “Peridot”, capa en la cual se gestionaban las aplicaciones, también modulares, del sistema; Dentro de esta capa se encontraban sus dos principales componentes:

- “Ledger”: sistema de almacenamiento distribuido (nube) que permite al usuario almacenar los datos de cualquier aplicación y acceder a ellos desde cualquier dispositivo
- “Maxwell”: componente basado en inteligencia artificial que determina por ejemplo qué procesos deben tenerse en segundo plano en cada momento según el comportamiento del usuario.

Por último, y en el nivel más alto de la pirámide, estaría “Topaz”, capa que se encarga de implementar las interfaces de usuario definidas en los niveles inferiores a ella usando como soporte el SDK de código fuente abierto denominado Flutter.

Si bien este diseño sirve para tener una idea de cómo funciona Fuchsia, no es la forma en la que actualmente están distribuidos todos los componentes del sistema, ya que como es normal, con el paso del tiempo se van encontrando formas mejores y más eficientes de realizar determinadas tareas o procesos. Esto dio lugar a que varios de los componentes que formaban parte de la “Fuchsia Layer Cake” fueran eliminados o quedaran en desuso, y que finalmente, en 2019, se tomara la decisión de sustituir el diseño que se tenía por una versión mejorada de cinco capas, dejando al layer cake obsoleto (“deprecated”) [20].

Componentes como “Ledger” o “Maxwell”, así como la capa de “Peridot” fueron removidos del “Fuchsia tree” ya sea por planes de usarlos de forma externa como librerías, caso de “Ledger” [21], o porque de momento se reconsideró la idea de su uso, caso de “Maxwell”. En cuanto a la capa Garnet, que pasó a estar deprecated [22], algunos de los componentes que estaban dentro de ella fueron trasladados a otras rutas, como es el caso del procesador de gráficos Escher que fue trasladado a `main:src/ui/lib/escher/`, mientras que el resto de sus ficheros o componentes fueron removidos o, según la propia documentación, están pendientes de serlo [23].

4.2 Diseño de Fuchsia actual

Como ya sabemos, la estructura general de un sistema que posea como base un microkernel, es una que está constituida por la parte de más bajo nivel, el kernel propiamente dicho, donde se llevan a cabo las tareas más básicas como serían el IPC o la gestión de procesos, y por un conjunto de módulos o capas que se encargan de realizar el resto de las tareas o responsabilidades del sistema operativo.

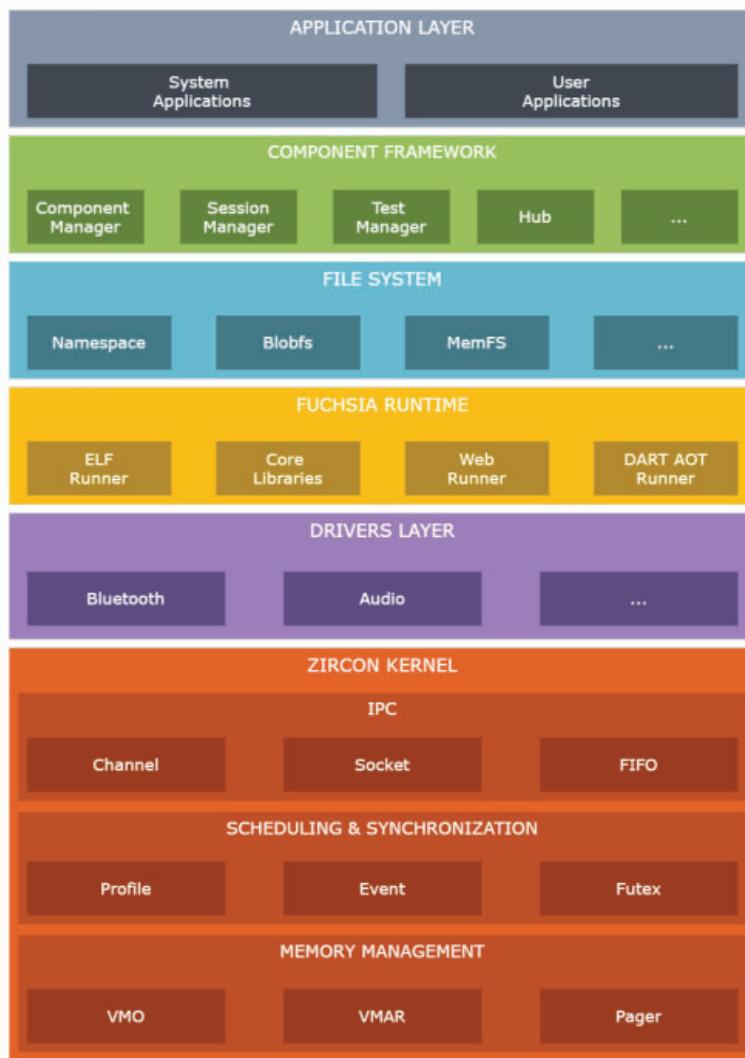


Figura 8. Diseño de la arquitectura de Fuchsia [24]

Como se puede ver en la Figura 8, la arquitectura de Fuchsia está constituida, además de por su base Zircon, de otras 5 capas las cuales desempeñan cada una sus propias tareas y funciones.

4.2.1 Drivers Layer

Dejando atrás el kernel mode, la capa de drivers es la primera y de más bajo nivel de las 5 que componen el user mode de Fuchsia. La principal característica de esta capa es el uso del FDF (siglas de “Fuchsia Driver Framework”), framework que permite y facilita a los desarrolladores todo lo que tiene que ver con la creación, gestión y elaboración de pruebas de los drivers de cualquier dispositivo.

Dentro de este framework podemos encontrar herramientas como el driver manager, el driver host o la librería core (`libdriver`) mientras se hace uso de las ya explicadas interfaces proporcionadas por FIDL y de las interfaces proporcionadas por Banjo.

El driver manager es una herramienta que se encarga de gestionar, cargar y ejecutar, mediante el driver host (binario ejecutado para alojar uno o más drivers), los drivers de los dispositivos en todas las plataformas. Entre sus funciones se encuentran, por un lado, intentar emparejar un driver para cada dispositivo mediante un programa de enlace y gestionar los ciclos de vida de cada uno de los drivers [25], mientras que, por otro lado, también se encarga de montar y hacer de host de un sistema de ficheros virtual ubicado debajo del directorio `/dev` mediante el cual se provee de una ruta de acceso unificada para todos los programas o dispositivos a los drivers a través de sus respectivas interfaces.

Esta conexión entre los dispositivos y los drivers se realiza mediante las interfaces de FIDL y las de Banjo, es decir, mediante una serie de API's y ABI's que permiten estas comunicaciones.

Por definición, las API's tienen como objetivo definir interfaces entre componentes de programa a nivel de código fuente, por otro lado, Las ABI's lo hacen en este caso a nivel de código máquina, estas últimas se encargan de definir interfaces para cubrir aspectos como los tamaños o alineamientos de datos, como se pasan o se recuperan valores de las funciones, el formato binario de los objetos, etc. [26].

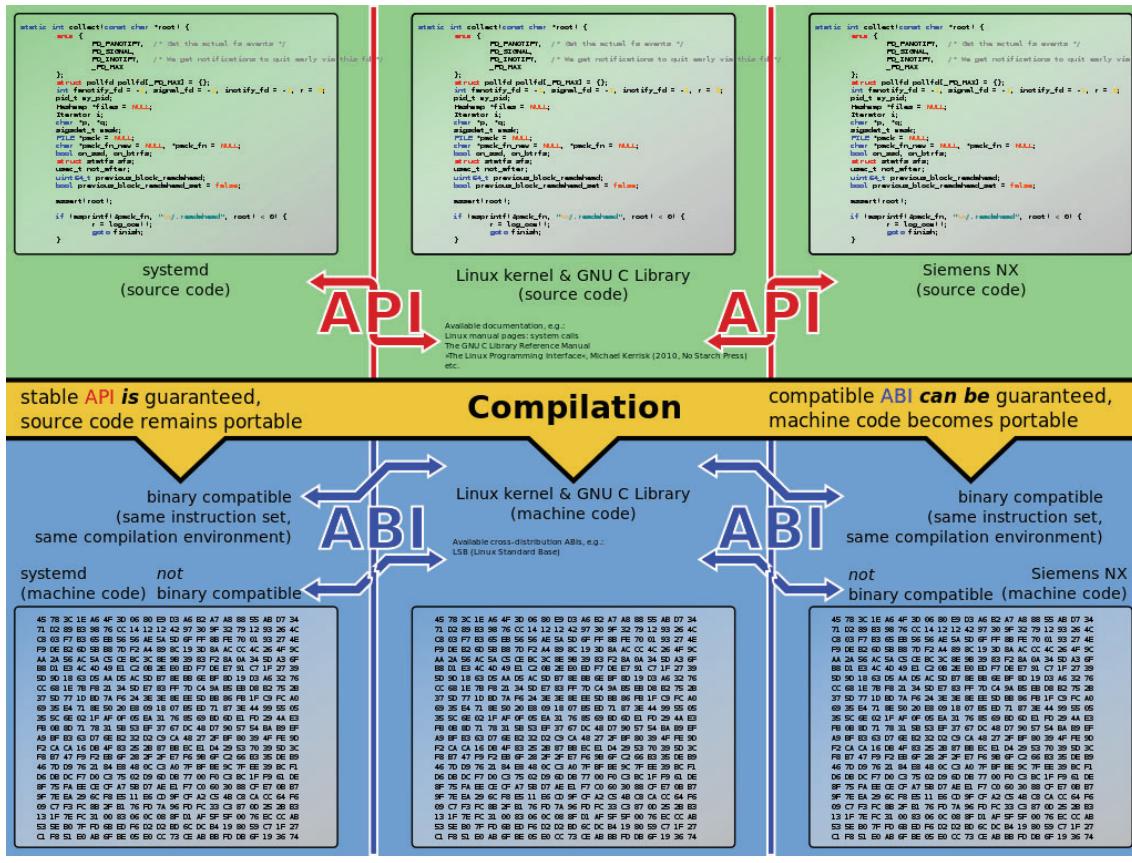


Figura 9. Comparativa de API vs ABI [26]

En el caso de Fuchsia, FIDL juega un papel muy importante a la hora de desarrollar drivers como ya se explicó previamente, ya que es el sistema de comunicación entre procesos. Los devices implementan interfaces mediante FIDL que se encargan de ser protocolos de llamada a procedimientos remotos (RPC) las cuales van a ser usadas luego por las aplicaciones o por los servicios (clientes).

Banjo, por su parte, pese a ser también una especie de compilador como lo sería el “fidlc”, se encarga de generar los protocolos (ABI’s) que serán implementadas por los devices y que serán usadas para que se dé la conexión entre un device hijo y su padre; Dichos protocolos, son generalmente interacciones entre devices de un mismo proceso en un mismo “driver host” [27].

Esto quiere decir que, para que se puedan dar las interacciones entre drivers y devices, un driver tiene que ser capaz de proveer al device una serie de “device ops” y “protocol ops” [28].

Las “device ops” implementan tanto la interfaz externa como los métodos que controlan el ciclo de vida del device (device_open, device_unbind, device_release), los cuales serán llamados por otras aplicaciones o servicios del user space haciendo uso de las interfaces de FIDL y sus mensajes [29].

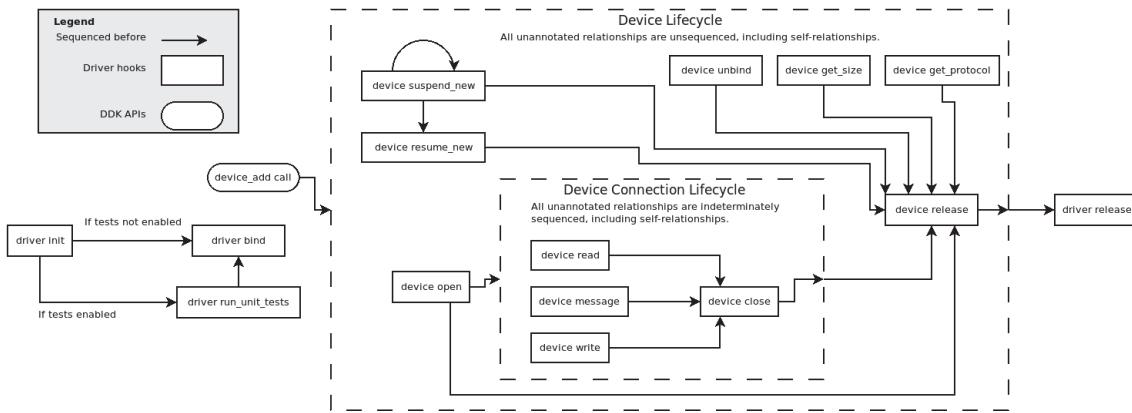


Figura 10. Device ops [29]

Las “protocol ops” implementan los protocolos para ser usados dentro del mismo proceso para el device, es decir, se usa por ejemplo para funciones que van a ser llamadas por otros drivers dentro del mismo proceso para que así, si un driver hijo quiere usar esa función solo debe llamar a esa “protocol op”.

4.2.2 Fuchsia Runtime Layer

Un runtime system o sistema de tiempo de ejecución es el medio por el cual se implementan los modelos de ejecución para el comportamiento de los elementos de un lenguaje, es decir, un grupo de configuraciones que proveen del ambiente necesario para que programas o componentes se ejecuten.

En Fuchsia, los runners, son los responsables de integrar los diferentes runtimes y frameworks para hacer que cada componente ubicado en el user space tenga la información o configuraciones necesarias para ejecutarse [30], brindándole apoyo y liberando de trabajo al component manager al proveer él de los distintos runtimes; En otras palabras, son los encargados de gestionar la ejecución de los componentes dentro de su respectivo runtime interpretando las órdenes provenientes del component manager (servicio que explicaremos más adelante y que se encarga de gestionar a los componentes).

Entre las funciones o responsabilidades de los runners se encuentran iniciar un nuevo proceso para los componentes, gestionar su ejecución dentro de su runtime correspondiente o aislar un componente dentro de una máquina virtual.

```

{
  capabilities: [
    {
      runner: "web",
      path: "/svc/fuchsia.component.runner.ComponentRunner",
    },
  ],
}

```

Figura 11. Capacidad de runner a un componente [32]

En la figura 11 se muestra un ejemplo de cómo se le da la capacidad de runner a un componente dentro de su manifiesto (conceptos que se explicarán más adelante en la capa de componentes); Se indica el nombre identificativo de dicho runtime y el path (ruta) del protocolo a través del cual el component manager enviará, primeramente, la información necesaria para iniciar el componente, y luego, los protocolos que usará para enviarle las órdenes posteriores, tales como reiniciar o parar el componente.

Por otra parte, en caso de que un componente quiera seleccionar un runner de los que ya están registrados, en lugar de declararlo, lo solicitará en la sección “program” de su manifiesto (en lugar de la sección “capabilities”), indicando ahí el nombre [32].

Algunos ejemplos de los runners que vienen por defecto en el ecosistema de Fuchsia podrían ser [30]:

- ELF Runner: es un runner disponible para todos los componentes que sirve para ejecutar ficheros con el correspondiente formato .elf (Executable and Linkable Format), formato muy usado en sistemas tipo UNIX como GNU/Linux, macOS o Solaris al compilar programas C, C++ o Rust.
- Dart AOT Runner: es el responsable de proveer al ambiente necesario para ejecutar los ficheros escritos en este lenguaje como por ejemplo una máquina virtual (Dart VM) con un compilador AOT (Ahead of time) para la producción del código máquina que será usado por el sistema operativo [31].

Para trabajar con esta máquina virtual se utiliza un componente dentro de Fuchsia llamado “Tonic”, herramienta escrita en Dart y compatible con Flutter con utilidades que permiten trabajar con la DartVM API.

- Chromium web Runner: encargado de brindar el runtime necesario para la ejecución de componentes implementados como páginas web.

Además de esto también se podría mencionar la idea de Google para integrar una versión del Android Runtime (ART) diseñada específicamente para Fuchsia, lo que permitiría la ejecución de las aplicaciones Android (ecosistema más popular entre los desarrolladores [33]) dentro de su sistema operativo, facilitando así, la adopción de Fuchsia por parte de los usuarios y desarrolladores.

4.2.3 Component Layer

En la capa de componentes o “Component Framework” es donde se engloba toda la información y las herramientas necesarias para ejecutar los componentes y coordinar la comunicación y acceso a recursos por parte de ellos [34], estos elementos serían: los componentes propiamente dichos y sus manifiestos, así como el component manager, el hub y conceptos como los realms (grupos de componentes y sus hijos) y los environments (conjunto de configuraciones aplicables a los componentes de un realm).

Los componentes son la unidad básica de software ejecutable dentro de Fuchsia, los cuales deben cumplir con la premisa de ser componibles (composable en inglés) y “sandboxed” [34].

Primeramente, que un componente sea “composable” hace referencia a que, por un lado, y como ya se ha mencionado en varias ocasiones a lo largo de este trabajo, debe ser modular, es decir, que pueda ser ejecutado sin necesidad de otros componentes (auto contenido), mientras que por otro lado debe ser capaz de ejecutar cada petición como una transacción independiente (“stateless”) sin importarle el resultado de tareas o transacciones pasadas.

En segundo lugar, el hecho de que sea “sandboxed” quiere decir que, durante el tiempo de ejecución, el componente debe trabajar en un entorno aislado para garantizar la seguridad y la estabilidad del resto del sistema.

En este sistema operativo, cada uno de los componentes funcionan gracias a dos principales servicios como serían el component manager y el hub, el primero es el encargado de gestionar tanto las interacciones entre componentes, como sus ciclos de vida y cada uno de los protocolos o servicios que necesitan durante su tiempo de ejecución, mientras que el segundo, el hub, es el encargado de facilitarle dichas tareas al component manager, ya que en él se guarda toda la información necesaria de cada uno de los componentes durante su tiempo de ejecución (de sus instancias) [44].

Cada componente es dotado de dos principales características para diferenciarlo del resto de componentes, la URL del componente, que sería como su identificación, y su manifiesto (ficheros con la extensión .cml), que es donde se describen, entre otras cosas, como ejecutar el componente y sus capabilities (capacidades), por ejemplo, si tiene la capacidad de ser resolver o el runner del cual va a hacer uso en caso de incluir programas ejecutables [34].

Estos resolvers (componentes a los cuales se les dio dicha capacidad de manera análoga que a un runner) son los encargados de interactuar con el component manager de parte de sus respectivos componentes para obtener a otros componentes (por ejemplo, hijos [30]) a través de sus URL.

Cuando se necesita un componente, el component manager delega la tarea de obtenerlo y localizarlo a los resolvers. Los resolvers se encargan de recuperar componentes desde su origen, es decir, toman la URL como input y producen un manifiesto para un nuevo componente.

```
environments: [
  {
    name: "my-environ",
    extends: "realm",
    resolvers: [
      {
        resolver: "my_resolver",
        scheme: "my-scheme",
        from: "parent",
      }
    ],
  },
]
```

Figura 12. Registro de un nuevo resolver [35]

La figura 12 muestra la forma mediante la cual se registra un nuevo resolver dentro de un environment para que todos los componentes pertenecientes al realm de dicho environment lo tengan disponible (proceso igualmente similar al registro de un runner). En esa declaración hay que especificar la información respectiva a su nombre identificativo, el esquema, que se refiere al formato que usará el component manager como guía para escoger un resolver u otro en función de que componente necesite obtener [35], y el campo correspondiente a “from” que se refiere a la ruta origen en la cual está declarado del resolver, similar a un “import”.

Tanto en el caso de los resolver como anteriormente en el caso de los runners, se mencionó en varias oportunidades la palabra capability, ya que como se explicó, ambos son componentes a los cuales se les dio esa capacidad. En general, las capabilities son un conjunto de derechos combinados a la referencia de un objeto, cuando un programa posee una capability es porque se le ha concedido el privilegio de ejecutarla o de usarla [36].

Dentro del manifiesto de un componente, todas las capabilities desarrolladas por él y todas las capabilities (propias de otros componentes) de las cuales se hará uso durante el tiempo de ejecución se declaran en forma de objetos dentro de las arrays correspondientes a las secciones “capabilities” y “use” respectivamente [37], estas últimas son las que tendrá que leer el component manager para buscarlas e instalárselas en su namespace,

El namespace de un componente es un file system con ficheros, directorios, sockets o servicios (todos ellos representados como objetos con un nombre de referencia mediante el cual hacer uso de ellos) el cual “vive” en la File System Layer y es proporcionado por el environment a cada uno de los componentes que están dentro de él [38]. Un componente puede, o bien usar los objetos de su propio namespace, ya sea protocolos externos “importados”, servicios u objetos propios, o bien puede ofrecer objetos (capabilities) para que otros componentes las usen declarándolos en el array del manifiesto correspondiente a la sección “expose”.

Las capabilities propias pueden también ser agregadas, además de en la sección environment en el caso de runners o resolver, en las secciones “offer” y “expose”, las cuales hacen referencia a las capabilities que son ‘puestas’ visibles para que las use un hijo (acompañadas del parámetro “to”) o un padre (acompañadas del parámetro “from”) respectivamente [39], estas relaciones pueden verse representadas en la figura 13.

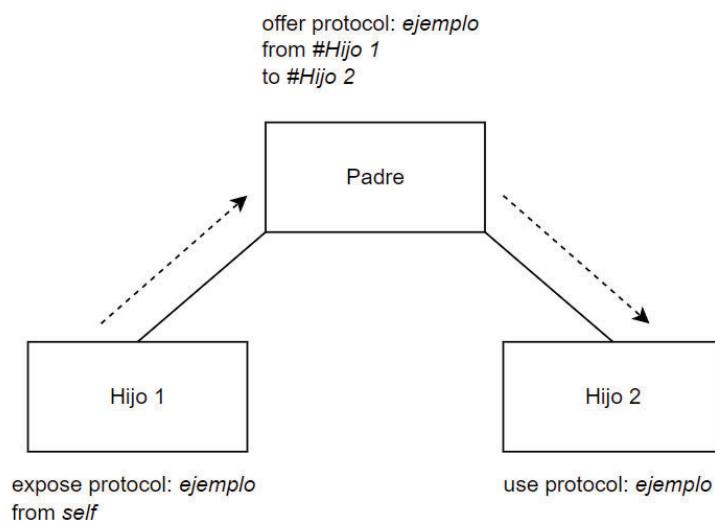


Figura 13. Ejemplo de las relaciones entre “offer” y “expose”

Cuando un componente es inicializado dentro de su respectivo environment, este recibe una lista con la ruta de el o los namespaces a los cuales tendrá acceso mediante sus respectivos handles ya preconectados a estos. Un handle es un objeto mediante el cual un proceso del user space puede referenciar y usar objetos que “vivan” en el kernel space [40] como serian en este caso channels por los cuales comunicarse (mediante protocolos FIDL) y usar los objetos que estén dentro de dicho namespace.

Dentro del namespace se establecen un grupo de directorios en forma de estos handles hacia los canales IPC usados para realizar las interacciones entre componentes [41] y llevar a cabo la comunicación entre procesos.

En función de los objetos que hayan sido declarados dentro de la sección “use” (en este caso, los protocolos que necesitara), el component manager añadirá los handles en el namespace de dicho componente hacia los directorios padre de esos objetos, por ejemplo, si un componente tiene en la sección “use” un objeto {protocol: “p.ejemplo”}, se añadirá en su namespace un handle hacia el directorio `/svc` (rama por defecto del namespace provista por el component manager en la que se ubican las rutas a través de las cuales acceder a las implementaciones de los protocolos) [38].

Cuando un componente quiere usar un protocolo que no es propio de él (lo tiene declarado en la sección use de su manifiesto), por ejemplo, el protocolo “p.ejemplo” mencionado anteriormente; Una vez que el component manager haya creado el respectivo handle `/svc`, creará un nuevo channel y entregará uno de los extremos mediante la instrucción `open()` (instrucción proporcionada por el protocolo FIDL fuchsia.io) ejecutada desde el “directorio” `/svc` e indicando como path: `/svc/p.ejemplo` [39].

El component manager, al ver que el componente realizó la llamada `open()` en el directorio `/svc` tomará ese extremo del channel y buscará al componente del cual proviene el protocolo “p.ejemplo”, mediante lo que se conoce como “capability routing” [39], proceso mediante el cual el component manager revisa las secciones offer y expose de los manifiestos de cada uno de los componentes en el árbol buscando el componente con dicha capability declarada con el parámetro `from: “self”`, es decir, el componente de quien es dicha capability.

Una vez encontrado el componente origen de dicha capability, el component manager le entrega el extremo del channel que recibió mediante la llamada `open()` a él, de forma que, cuando este componente origin entre en tiempo de ejecución, publicará la implementación de ese protocolo a través de una nueva entrada (correspondiente al handle `/svc/p.ejemplo`) en su Outgoing directory [39], directorio donde un componente tiene declarados handles para que otros accedan a sus capabilities (declaradas con el parámetro `from: “self”`) expuestas para que las usen [36].

Ya terminado todo el proceso, el component manager se retira mientras que ambos componentes (cliente y proveedor) hablan entre ellos a través de ese channel.

Otros tipos de capabilities además de las ya vistas (runner, resolver y protocol) podrían ser:

- Service: es una capability que permite el uso de una o más instancias de FIDL services, un componente cliente puede hospedar múltiples instancias que, de forma análoga a las protocol capabilities, estarían en el directorio por defecto /svc del namespace del cliente. Un ejemplo de esto sería:

```
/svc/s.ejemplo/57dfe118a2a8/echo  
/svc/s.ejemplo/57dfe118a2a8/reversed_echo
```

Donde “s.ejemplo” sería el nombre del servicio, “57dfe118a2a8” serían los identificadores de las instancias (generados aleatoriamente por el component manager) creadas de ese servicio mientras que “echo” y “reversed_echo” serían funciones de ese servicio [42].

- Directory: en este caso un componente declara una capability en forma de directorio (objeto) especificando el nombre, los derechos (lectura, escritura) y el path del Outgoing directory por el que pueden acceder a esa información. Por su parte, el cliente, que también debe especificar el nombre y los derechos, puede añadir el path de su namespace donde quiere que se ubiquen los contenidos de ese directorio compartido [43].

El hub, como ya vimos anteriormente es una directory capability, por lo cual, puede ser accedido por su respectivo componente solicitándolo en la sección use de su manifiesto indicando los parámetros [44]:

```
{ directory: "hub", from: "framework", rights: [ "r*" ] }
```

Con esto, se creará una entrada en su namespace con la ruta /hub a través de la cual acceder a su información.

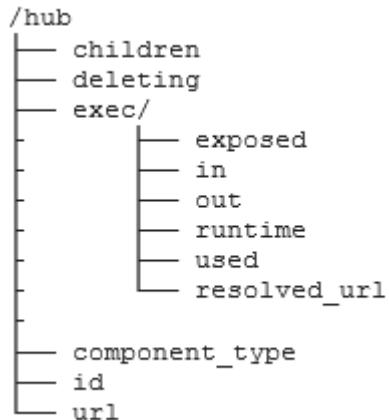


Figura 14. Esquema de los directorios de un hub [44]

Gran parte del contenido del directorio hub es de carácter inmutable a excepción de `/hub/exec/out/`, que es donde se ubica el contenido referente al Outgoing directory [44] de la instancia de ese respectivo componente, aquí como ya vimos se encuentran los handles a las capabilities que se tienen en expose, los cuales pueden abrirse, cerrarse o consumirse; es debido esta razón que es el único de estos directorios que puede estar sujeto cambios.

Otras rutas del hub serían, por ejemplo, `/hub/exec/exposed/` donde se listan las capabilities con esa característica, `/hub/exec/in/` que lista los protocolos, servicios, etc., “importados”, es decir, provistos por el component manager durante para esa instancia o `/hub/exec/used/` que es donde se listan las capabilities requeridas por ese componente durante su tiempo de ejecución, de forma que, en caso de que un componente haya solicitado acceso a hub, puede también acceder a esos servicios desde estas rutas [44].

- Storage: mientras que en las directory capabilities se declara un directorio compartido, en esta se declara uno de forma “personal” para prevenir que otros accedan a él. Cada una de estas debe estar respaldada por una directory capability que sirva para alojar un subdirectorio personal para cada componente, de manera que, cuando el componente cliente intenta acceder a la storage capability, el component framework le genera su propio subdirectorio aislado dentro del directorio de respaldo [45].

Para declararla, además de indicar los parámetros correspondientes al nombre y a “from” como en el resto de capabilities, hay que indicar como ya vimos, la directory capability de respaldo (disponible en el mismo lugar al que se hace referencia en el parámetro from) y una id. Como cada uno tiene que gestionar su propio subdirectorio, es necesario identificar dicha instancia con un identificador, ya sea el moniker de la instancia de su respectivo componente, o usando una forma un poco más ‘segura’ mediante una instance ID.

El moniker de un componente es una identificación que se construye a partir del path que tiene ese componente en el árbol de componentes seguido de su id como instancia, de tal forma que, para un componente hijo su moniker seria “hijo:0” o si está dentro de una colección “colección:hijo:0” [46].

Una colección es un contenedor de instancias de componentes condicionales que son creados de forma dinámica durante el tiempo de ejecución y luego destruidos [47].

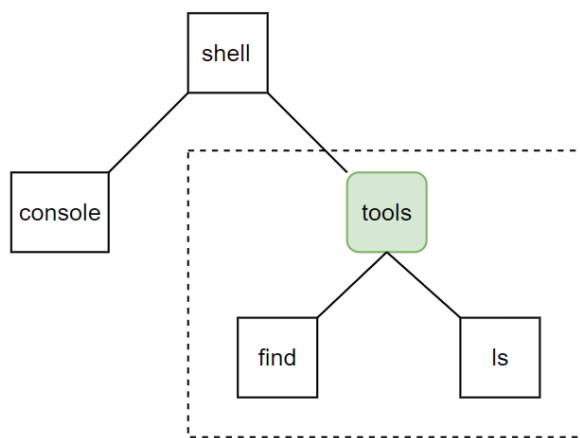


Figura 15. Esquema de un realm con una colección

Como se puede ver en la figura 15, el componente “shell” declara un componente “console” y una colección “tools”, la cual contiene dos instancias dinámicas (hijas de “shell”). La importancia de las colecciones se debe a que antes del tiempo de ejecución, hay situaciones en las que no se puede tener conocimiento de cuáles son las instancias que se tienen que crear. Como se ve en el ejemplo, “ls” y “grep” son instancias que se crean luego de que una persona usa esos respectivos comandos de la terminal, por lo que, ya en tiempo de ejecución, se crean y al finalizar sus respectivas funciones se destruyen.

Los moniker también pueden ser representados de forma relativa usando un “.” para indicar el origen, “/” para indicar que el nodo anterior en el moniker es un hijo o “\” para indicar que el nodo anterior es un parent, esto se puede ver representado en la figura 16.

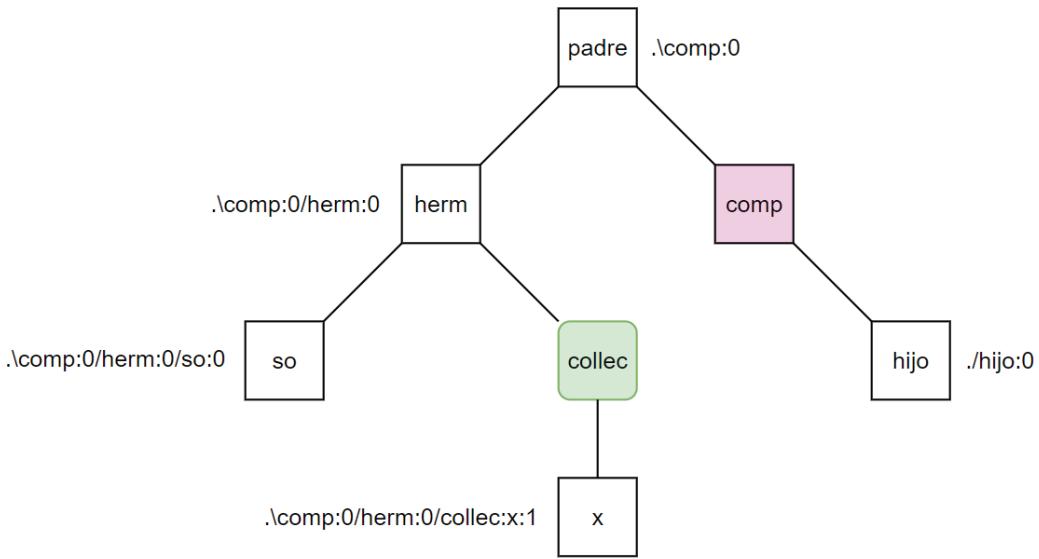


Figura 16. Monikers relativos

El uso de monikers para identificar una storage (usando la opción `storage_id: "static_instance_id_or_moniker"` [37]) como se puede ver es susceptible a dar problemas cuando hay cambios en la posición de los componentes o en sus nombres, ya que uno solo de estos cambios hace que el moniker pase a hacer referencia a un objeto que ya no existe en ese path.

En caso de que se tenga la intención de que la storage sea más duradera y persistente a cambios (como los que afectan a los monikers), es necesario usar la opción `storage_id: "static_instance_id"`, con esto se añadirá una nueva entrada en el “index file”, fichero con formato JSON5 que permite establecer relaciones entre una “`instance_id`” generada (única y estable) y un moniker, para que, si hay cambios en el moniker, se actualice dicha entrada y no se pierda rastro de la storage [48].

Para mantener la consistencia del index file, puede haber múltiples index files en un árbol de componentes, los cuales serán mezclados en un index file unificado que estará disponible para todos en tiempo de ejecución [48]

Un ejemplo de los componentes “Built-in” que actualmente trabajan en Fuchsia y que se pueden encontrar en su código fuente serían:

- “Cobalt”: herramienta de análisis y gestión de datos la cual permite entre otras cosas “preservar la privacidad y el anonimato del usuario mientras brinda a los propietarios de productos los datos que necesitan para mejorar sus productos” [49]
- “Fargo”: herramienta que permite el uso y la interacción con el gestor “Cargo” para compilar/correr aplicaciones escritas en Rust o para la administración de paquetes del mismo [50].

4.2.4 File System Layer

Otra de las características que hace a Fuchsia tan especial es el hecho de la independencia que tiene su sistema de ficheros respecto al kernel. A diferencia de otros sistemas operativos en los que hay que solicitar el acceso a los ficheros mediante las respectivas syscalls al núcleo, en Fuchsia, el sistema de ficheros es un proceso nativo del user space que implementa servidores que simulan ser sistemas de ficheros y que tienen la capacidad de responder a llamadas de procedimientos remotos (RPC) usando el mecanismo de comunicación ya explicado FIDL [51].

Es decir, cada uno de los sistemas de ficheros de Fuchsia son componentes que implementan servidores, y es a través de los handles en estos servidores que se tiene acceso a los ficheros.

Como ventajas de este diseño están el hecho de que como dice en la propia documentación, “El sistema de ficheros de Fuchsia en sí mismo puede ser cambiado con facilidad – las modificaciones no requieren que se recompile el kernel” [51], al funcionar como un proceso independiente del núcleo del sistema adquiere la capacidad no solo de poder ser modificado, reemplazado o eliminado como una aplicación más sin necesidad de reiniciar el sistema sino que también, por ejemplo, da la opción de tener más de un sistema de ficheros funcionando.

Otro ejemplo de sistemas de ficheros son los namespaces [51] de los que ya hablamos un poco y en los cuales están, por ejemplo, los nodos usados para acceder a los protocolos propios o a los protocolos declarados en otros componentes mediante la instrucción `open()` (`/svc`) o los objetos (ficheros) traídos de los file system (`/tmp`).

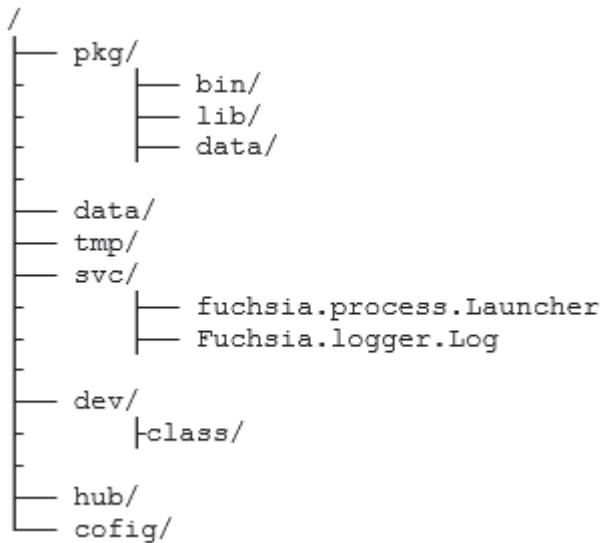


Figura 17. Esquema de un namespace [38]

Un componente, además de ser dotado del handle o los handles a sus namespaces también es dotado de un handle que representa la conexión con el CWD (siglas de Current Working Directory) [52] permitiéndole usar rutas relativas, de esta forma, cuando un componente quiere abrir el fichero “f”, inicialmente debería usar la llamada `open("f")` (instrucción ya vista en las capabilities y a la cual tienen acceso gracias a la librería Fuchsia.io), mediante la cual se creara los handles y un channel para luego pasar uno de los extremos (con el handle) a través de un mensaje FIDL.

La librería Fdio es la responsable de proveer de una interfaz unificada a todos los recursos que se encuentren en el namespace, ya sean archivos, servicios, protocolos, etc., haciendo que el cliente los vea a todos con simples descriptores de ficheros y permitiendo la interacción con ellos través de una serie de funciones tales como `read`, `write`, `open`, `close` o `seek` [52]

Seguidamente el servidor del sistema de ficheros interpretará el contenido de ese mensaje como un “open” por lo que recibirá ese handle y el path que viene como parámetro, que en este caso es “f”, y lo verificará comprobando la existencia de cada nodo hasta llegar al final de dicho path. Si el servidor no verifica como correcto el path recibido responderá con objeto FIDL describiendo el error, mientras que, si lo encuentra y además es accesible por parte del cliente, creara un objeto “iostate” y lo registrará en el “dispatcher” [52] (objeto que despacha mensajes entrantes a los handles apropiados, se encarga de redirigir estos mensajes a una función callback la cual previamente ya fue inicializada para representar una conexión abierta, dando apoyo tanto a la API’s asíncronas como a la gestión de eventos y errores), de esta forma, la próxima vez que se necesite acceder a ese fichero se podrá acceder directamente a él gracias al objeto iostate representando la conexión abierta.

Finalmente, el servidor enviará un objeto FIDL describiendo que todo ha ido correctamente al cliente, quien luego creará, mediante la librería Fuchsia.io, el descriptor de fichero que encapsulará a su handle y mediante el cual podrá realizar futuras llamadas al fichero sin necesidad de pasar por el canal CWD [52].

Actualmente Fuchsia cuenta con varios tipos de file system, entre las cuales están:

- MemFS: “usado para implementar sistemas de ficheros temporales como `/tmp` donde los ficheros existen únicamente en la RAM” [52] mediante el uso del C allocator.
- MinFS: a diferencia de MemFS, este es usado cuando se busca que los archivos perduren en el tiempo, esto lo logran mediante el uso de los block devices.

Un block device es un driver que recibe las llamadas a procedimientos remotos que les hacen los clientes, en este caso, los sistemas de ficheros MinFS, para realizar transacciones de entrada y salida mediante las cuales es capaz de escribir o leer grandes porciones de memoria de forma más eficiente usando un protocolo FIFO sobre virtual memory objects [53].

Estos VMO son objetos de memoria compartida usados para representar páginas de memoria de forma física, estos pueden ser mapeados dentro de los procesos o también pueden pasarse de un proceso a otro [54].

De esta forma, el cliente (un sistema de ficheros, un proceso, un componente, etc.) puede poner un pequeño mensaje en la cola indicándole al disco que escriba tantos bytes de un VMO específico (ya existente) en una determinada zona de memoria en lugar de enviar esa información mediante buffers de tamaño limitado mediante mensajes IPC que son.

- Blobfs: este tipo de sistema de fichero está optimizado para escribir y luego limitar a solo lectura de los ficheros por lo que es usado generalmente dentro de Fuchsia para trabajar con los paquetes software. Se diferencia de los otros sistemas de ficheros ya que, en él, además de que los ficheros (blobs) son inmutables, es decir, no pueden ser modificados, también garantiza su veracidad mediante algoritmos criptográficos para generar un hash que verifique su integridad [55].
- Thinfs: “es una colección de utilidades para la gestión de discos que pueden ser usada a la hora armar o compilar un file system” [56]

Por ejemplo, en el caso de que se tenga la intención de crear un nuevo tipo de sistema de ficheros, nos podemos fijar en cómo esta implementado MinFS [57], el cual basa su funcionamiento en la utilización de ficheros como fsck.h, bcache.h, mount.h, de modo que, en función de las intenciones que se tengan respecto a cómo tiene que actuar el nuevo sistema o que características se desea que este posea, se puede replicar parte de su lógica.

Bcache es un objeto en memoria creado a partir de los block devices explicados anteriormente y usado para construir el filesystem, dicho objeto sirve de base sobre la cual construir el sistema de ficheros usando el método `mkfs()` de la clase mimfs, y luego, a partir de él, mediante el método `MountAndServe()` definido en la clase mount.h, montar el sistema para que pueda recibir peticiones a través de un canal.

Fsck ayuda a comprobar, entre otras cosas, la integridad de la bcache recibida luego de haber creado la partición mediante `mkfs()` (se comprueba que todos los inodos estén conectados o se verifica si un inodo no está correctamente allocated, etc).

Un inodo es una estructura que cuenta con información respectiva a su tamaño, fecha de creación, enlaces a los inodos anterior y posterior, etc.

A partir de este punto, se puede usar la clase VnodeMinfs [57] que extiende de Vnode para dar lugar a un nuevo vnode, inicializando su respectivo inode en memoria (mediante el método `allocate()`) e inicializando el respectivo VMO que contiene el contenido del fichero leyéndolo del disco (mediante el método `InitVmo()`). Este vnode puede corresponder con un directorio o con un archivo, por esta razón también se ofrecen las interfaces `directory.h` y `file.h` [57] para poder realizar las operaciones respectivas de lectura, escritura, etc sobre dicho Vnode.

Finalmente, dicho VMO se puede enlazar a el block device mediante el método `BlockAttachVmo()` de la clase bcache. Cabe recalcar que esta implementación tiene una gran cantidad de secciones con la etiqueta “TODO” (no es de carácter final), dando a entender que todavía hay cosas por hacer tanto en MinFS como en general en todo el sistema y que está sujeta a cambios.

Cada una de estas clases estarían ubicadas dentro de la respectiva carpeta de nuestro nuevo tipo de sistema de ficheros en la ruta `/src/storage/newFS`, mientras que en `src/storage/bin/newFS` estaría el fichero main en el cual por ejemplo (como es el caso de mimFS), estaría un bucle `while(true)` esperando a que el usuario introduzca por la línea de comandos desde el shell los comandos respectivos de `mkfs`, `fsck` o `mount`.

4.2.5 Application Layer

Como ya hemos podido comprobar a lo largo del análisis de cada una de las capas que componen a la arquitectura de Fuchsia y en las cuales se basa su funcionamiento nos damos cuenta de que prácticamente todo el software dentro de este sistema operativo son componentes hechos para ser “composable” y “sandboxed”, logrando así que no solo un componente sea funcional por sí solo si no que también puedan agruparse varios componentes para formar un paquete.

Un componente solo podrá usar las librerías compartidas que estén dentro de su mismo paquete (en caso de que este dentro de uno), así como también esta librería compartida podrá ser usada por varios componentes a la vez dentro de ese paquete, a esto se le conoce como “ABI dependency” [58].

Por otro lado, también presentan el concepto de “API dependency” lo que permite que un componente defina una dependencia en otro sin importar la implementación de esta, es decir, se define una interfaz y es otro componente el que se encarga de comunicarse con ella a través de FIDL e implementarla [58].

En esta capa es donde se encontraría el código de más alto nivel y donde, mediante la unión de estos componentes en paquetes o por si solos, se puede dar origen a lo que serían los programas o aplicaciones del día a día, de forma que cuando estas se estén ejecutando, tengan por debajo a las instancias de dichos componentes realizando sus respectivas tareas y comunicándose entre sí.

5 Estudio de los aspectos específicos de la programación en Fuchsia

Actualmente, gran parte del código fuente de Fuchsia está escrito en C++ y en Rust por lo que este sistema operativo provee de soporte para la programación de código y ejecución de programas tanto en estos lenguajes como en otros como C, Dart o Go.

Una de las diferencias más grandes entre Fuchsia y otros sistemas POSIX es por ejemplo la no inclusión de las system calls `fork()` y `exec()` en su entorno (devolviendo el código de error “ENOSYS” al intentar usarlas), debido a la propia naturaleza del sistema operativo, la cual es dirigida por eventos que rigen o determinan cómo se comportan los objetos dentro de sistema operativo y los cuales, en su mayoría, como ya hemos visto, son accesibles a través de handles (objetos del user space que permiten referenciar objetos del kernel space).

Un job es un objeto del kernel formado por un grupo de procesos u otros jobs, es decir, permite agrupar procesos bajo un mismo objeto en función de sus privilegios para ejecutar operaciones sobre el kernel. En la fase de “boot” el kernel crea el root job (job del primer proceso, “userboot”) del cual cuelgan el resto de jobs [59].

Un proceso es una instancia de un programa formada por un conjunto de instrucciones ejecutadas por uno o varios threads. Para crear un nuevo proceso se usa `fdio_spawn()` [60] (perteneciente a la librería Fdio vista también en los file systems) incluyendo el respectivo handle al job del cual pasara a formar parte y permitiendo usar flags como “`FDIO SPAWN_CLONE_NAMESPACE`”, “`FDIO SPAWN_CLONE_STDIO`” o “`FDIO SPAWN_CLONE_ENVIRON`” para clonar en el nuevo proceso nuestro namespace, nuestros descriptores 0,1,2 (entrada, salida y salida de error estándar) o nuestro entorno.

En el caso de las señales, también presenta diferencias respecto a los sistemas POSIX, ya que ofrece una máscara de 32 bits para representar 32 señales diferentes las cuales pueden ser activadas (valor igual a 1) o desactivadas (valor igual a 0) [61], por ejemplo, al escribir un mensaje a través del extremo de un canal, se activa el bit que representa la señal “`ZX_CHANNEL_READABLE`” en el otro extremo, por lo que se puede usar la syscall `zx_object_wait_one()` para bloquear al proceso hasta que se active dicho bit y así tener una forma de saber que hay un mensaje disponible para leer.

Además de señales presentes en el sistema como “`ZX_CHANNEL_WRITABLE`”, “`ZX_CHANNEL_READABLE`” o “`ZX_PROCESS_TERMINATED`”, dentro de la máscara de 32 bits, también hay espacio para otras 8 señales similares a las “`SIGUSR1`” y “`SIGUSR2`” de POSIX (de “`ZX_USER_SIGNAL_0`” hasta “`ZX_USER_SIGNAL_7`”) para ser usadas según se estime conveniente en cualquier proceso del user space y las cuales pueden ser activadas o

desactivadas mediante el método `zx_object_signal()` indicando el handle del proceso, job, etc al cual se le va a activar dicho bit y la máscara [62].

En el caso de señales POSIX como “SIGKILL” que permiten terminar un proceso la cual está en el sistema de Fuchsia, se puede usar la syscall `zx_task_kill()` [63] se puede terminar un proceso, thread o job indicando su handle de referencia como parámetro.

Por otro lado, además de las diferencias con los sistemas POSIX mencionadas, también introduce un diseño y una forma de programar diferente, con ayuda de varias herramientas que facilitan esta tarea.

Para empezar a programar en Fuchsia, una vez descargado el código fuente a partir del cual se va a construir la imagen del sistema operativo, habría que configurar las variables de entorno y el perfil del shell para poder usar tanto la herramienta “jiri”, que brinda apoyo para una mejor integración con GIT y la gestión de los repositorios locales, y la herramienta “fx”, base para poder, entre otras cosas, generar la imagen de Fuchsia, iniciar el emulador, etc.

Para realizar esto se podrían incluir las siguientes líneas en el fichero `~/.bash_profile` (siempre y cuando se haya descargado el código fuente en la carpeta home) :

```
export PATH=~/fuchsia/.jiri_root/bin:$PATH  
source ~/fuchsia/scripts/fx-env.sh
```

Para configurar y construir la build se puede hacer uso de la ya disponible herramienta “fx” ejecutando, en función del tipo de arquitectura del dispositivo en el que se esté trabajando y desde el directorio de Fuchsia, los siguientes comandos:

```
fx set core.qemu-x64  
fx set workstation.x64  
fx build
```

A partir de este punto ya se tendría totalmente configurada la imagen de Fuchsia para poder ser emulada y trabajar en ella.

Para iniciar el emulador es necesario ejecutar en terminales diferentes (y ubicado en el directorio de Fuchsia) los comandos `fx server` (para iniciar el servidor de paquetes) y `fx vdl start -N` para iniciar el emulador, indicando que se va a hacer uso del “virtual device launcher” (o indicando en su lugar “emu” para usar la última versión en desarrollo del emulador) y “-N” para indicar que se va a usar una tarjeta de red (NIC) emulada. Otra opción también es incluir “-H” o no para indicar si se quiere o no que se inicie la interfaz gráfica de usuario (GUI).

Por otro lado, también se puede abrir un tercer terminal en el cual, luego de hacer uso del comando `fx set-device fuchsia-5254-0063-5e7a` (dependiendo del identificador que se haya generado para dicho emulador), se puede “controlar” el emulador mediante la herramienta “ffx” para, por ejemplo, iniciar o detener un componente, ver el LOG, ejecutar test, etc.

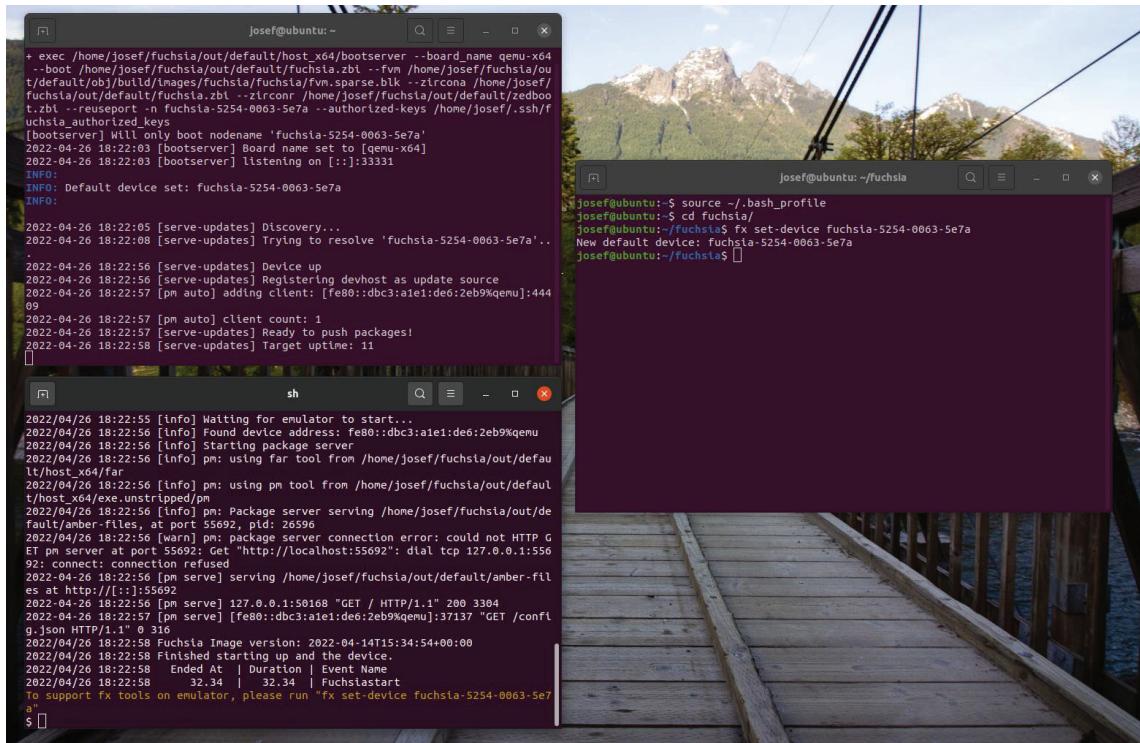


Figura 18. Inicio (modo headless) del emulador de Fuchsia

En la figura 18 se puede ver como a la izquierda en los terminales de arriba y abajo se usaron los comandos para iniciar el servidor de paquetes y para iniciar el emulador respectivamente, mientras que en el de la derecha se estableció el nuevo device correspondiente al emulador para así poder darle órdenes.

De esta forma, en el terminal de abajo a la izquierda se inicializa el shell del sistema operativo, el cual se pueden todos los archivos o ficheros que forman dicha build, permitiendo usar comandos típicos de un shell UNIX como `cd`, `ls`, `find`, etc.

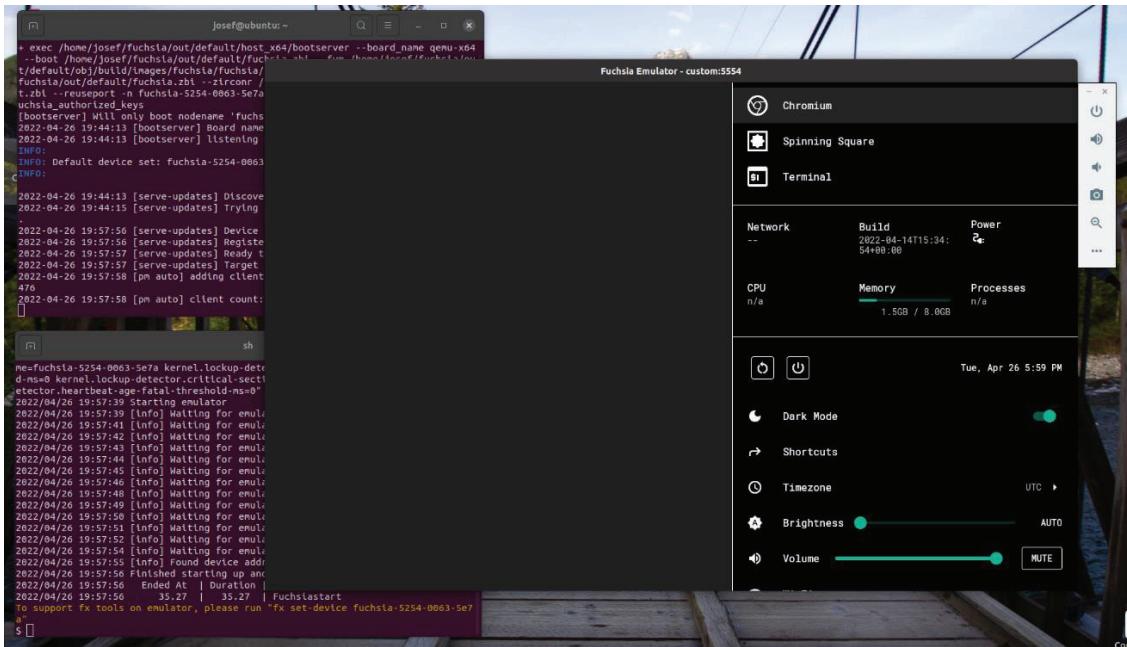


Figura 19. Inicio (con GUI) del emulador de Fuchsia

En la figura 19 se ve ilustrado el inicio del emulador esta vez incluyendo la interfaz gráfica, permitiéndole al usuario entre otras cosas controlar ajustes como el brillo o el volumen, ver el estado del emulador y los recursos que se tienen disponibles e iniciar tanto un explorador funcional de internet (Chromium) o el mismo terminal que vimos en la figura 18.

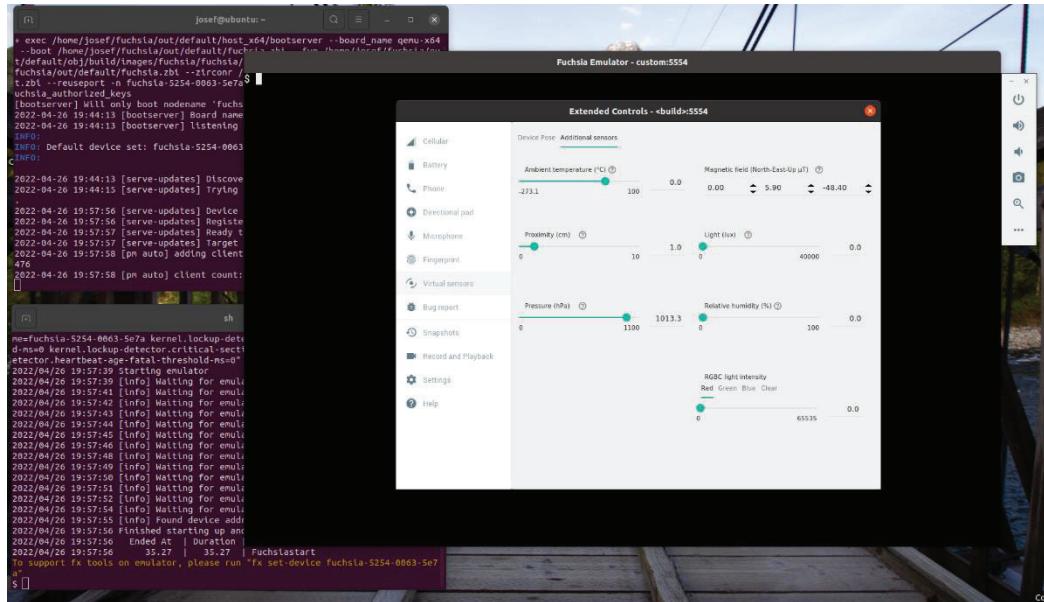


Figura 20. Interfaz de ajustes del emulador de Fuchsia

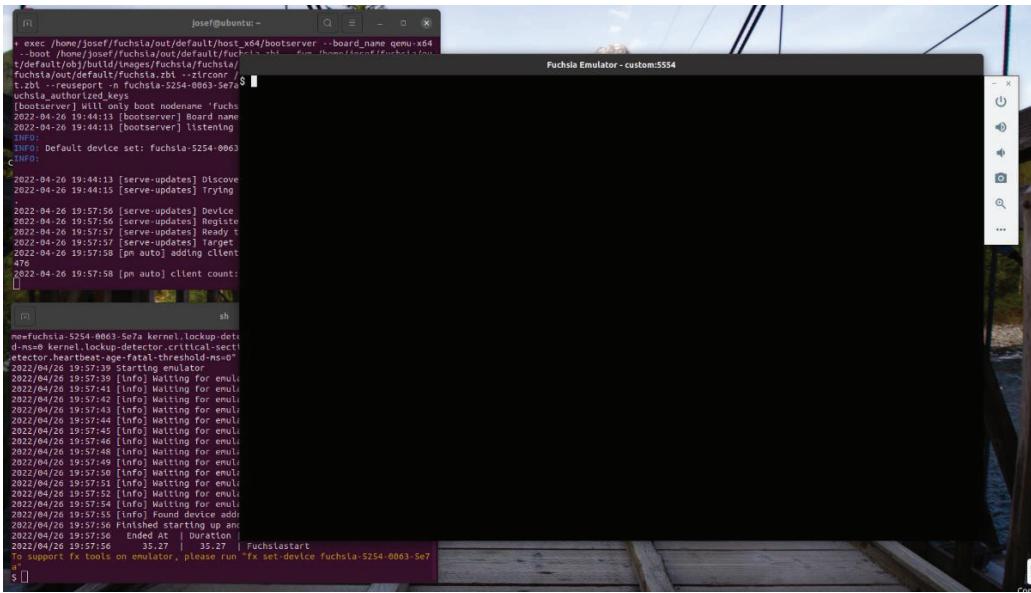


Figura 21. Shell en el emulador de Fuchsia

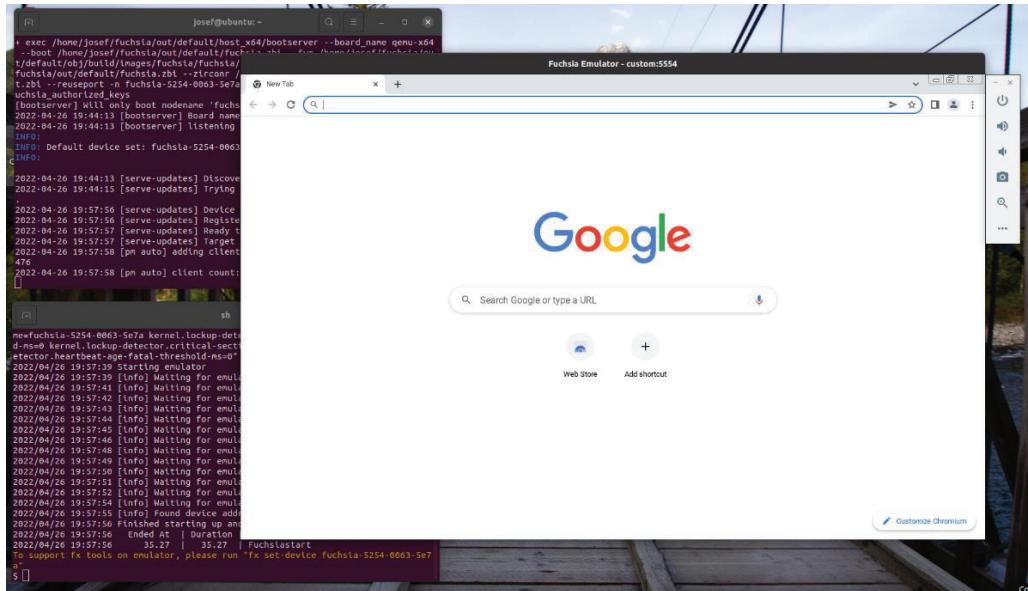


Figura 22. Chromium en el emulador de Fuchsia

Como se puede ver, en un entorno que ofrece bastantes posibilidades y opciones al usuario para probar el sistema operativo, sin embargo, y como consecuencia de su todavía estado experimental, no está exento de fallos o “bugs”, por ejemplo, a día de hoy, una vez que se abre la ventana de Chromium no es posible volver al inicio sin cerrar el emulador y abrirlo otra vez.

6 Selección de programas a desarrollar y diseño de los mismos

Para ilustrar cómo se puede llevar a cabo el desarrollo en este entorno se ha tomado la decisión de diseñar y realizar tanto un programa enfocado al funcionamiento de los drivers como un programa enfocado al funcionamiento de los componentes, los cuales estén encargados de realizar una serie de tareas simples que permitan aportar ejemplos de los temas tratados anteriormente en este trabajo.

En cuanto a la parte de los drivers realizará una implementación de `/dev/full` en C++, la cual, en escritura, devolverá el código “ZX_ERR_NO_SPACE” propio de Fuchsia para indicar que no hay espacio disponible en el destino (por lo que no escribirá ningún byte), mientras que, en lectura, devolverá caracteres “null”.

Además de esto se realizará un driver, esta vez en C, que inicialice en su contexto un unsigned char cualquiera para que, por un lado, en lectura, devuelva los n bytes sucesivos en función del parámetro “count” recibido, mientras que, en escritura, simplemente avance el puntero.

En el caso de la parte correspondiente a los componentes, se desarrollarán un componente padre y dos componentes hijos que juntos conformarán un realm, de forma que el primer hijo hará uso de una capability expuesta por el segundo para enviar un mensaje.

Cuando el primer componente quiera usar dicha capability (la cual tiene en la sección “use” de su manifiesto), el component manager se la facilitara poniéndolo en contacto con el segundo componente, lo que hará que esté se “despierte” al recibir dicha alerta y generando que, además, de responderla, esté realice tanto una serie de operaciones con ficheros y directorios, como una demostración de la creación y envío de un mensaje FIDL a través de un canal.

7 Desarrollo de los programas

7.1 Drivers

Un driver en Fuchsia se compone principalmente de cuatro ficheros principales que determinarán y permitirán su funcionamiento dentro del sistema, en este caso, el driver desarrollado se denomina “fuchsiadiversimple”, por lo que a continuación se explicarán los ficheros que lo componen.

- BUILD.gn: fichero que hace referencia a “generate ninja” y que permite al sistema ubicar de forma rápida cuales son los ficheros que se tienen que usar para generar el driver/componente, así como las dependencias, configuraciones y reglas de “binding” que necesita o tiene.
- fuchsiadiversimple.bind: aquí se definen las reglas booleanas que determinan qué devices pueden o no hacer bind con él.
- fuchsiadiversimple.c: código fuente del driver.
- fuchsiadiversimple-info.json: fichero donde se describe la información necesaria para clasificar al driver, indicando el “area”, “modelo” o la “familia”, en nuestro caso son “Misc”, “generic” y “generic” respectivamente.

En primer lugar, se darán detalles de cómo están implementados para fuchsiadiversimple cada uno de estos ficheros y luego se expondrán los resultados al intentar incluirlo en la build.

7.1.1 Implementación

El fichero `Build.gn` contiene la información sobre la composición del driver, este fichero está organizado por bloques donde cada uno indica el nombre mediante el cual se le puede hacer referencia, el nombre de la cabecera que genera, las dependencias y el o los ficheros de origen.

Entre las partes más importantes que se encuentran el correspondiente fichero de fuchsiadiversimple están:

Declaración correspondiente al fichero `.bind` del driver:

```
driver_bind_rules("fds_bind") {
    rules = "fuchsiadiversimple.bind"
    header_output = "fuchsiadiversimple-bind.h"
    bind_output = "fuchsiadiversimple.bindbc"
}
```

Declaración del grupo correspondiente al driver, donde se establece el nombre de referencia, el fichero fuente y sus dependencias:

```
fuchsia_driver("fds_driver") {
    output_name = "fuchsiadiversimple"
    configs += [
        "//build/config:all_source",
        "//build/config/fuchsia:enable_zircon_asserts",
    ]
    sources = [ "fuchsiadiversimple.c" ]
    deps = [
        ":fds_bind",
        "//src/devices/lib/driver",
        "//src/lib/ddktl",
        "//zircon/system/ulib/fbl",
        "//zircon/system/ulib/inspect",
        "//zircon/system/ulib/zx",
    ]
}
```

Declaración del componente que encapsulará al driver:

```
fuchsia_driver_component("fds_component") {
    component_name = "fuchsiadiversimple"
    deps = [ ":fds_driver" ]
    info = "fuchsiadiversimple-info.json"
    colocate = true
}
```

Y de forma análoga, declaración del paquete donde estará contenido dicho componente:

```
fuchsia_driver_package("pkg") {
    package_name = "fuchsiadiversimple"
    driver_components = [ ":fds_component" ]
}
```

Por último, creamos el grupo principal donde se hace referencia al paquete:

```
group("fuchsiadiversimple") {
    testonly = true
    deps = [ ":pkg", ]
```

El fichero `fuchsiadiversimple.bind`, que debe contener las reglas que al evaluarse determinarán si un dispositivo se puede conectar o no, contiene (para facilitar el testeo) un “true”, indicando que cualquier dispositivo puede hacer uso del driver, mientras que el fichero `fuchsiadiversimple.c` contiene la implementación del driver.

En cuanto a la implementación del código fuente del driver, estaría formada por las siguientes secciones de código:

Declaración de la estructura que conformará el contexto del driver:

```
typedef struct {
    zx_device_t* zxdev;
    void* val;
} fuchsiadriver_simple_device_t;
```

Implementación de los métodos read, write y release:

- Lectura: a partir del valor tomado del contexto del driver se devolverán en el buffer de salida, la cantidad de bytes sucesivos indicada por count:

```
static zx_status_t read(void* ctx, void* buf, size_t count, zx_off_t off, size_t* actual) {
    size_t i;

    fuchsiadriver_simple_device_t* device = ctx;
    void* n = &device->val;
    unsigned char v = (unsigned char)(long) n;
    for(i = 0; i < count; i++, buf++, v++){
        *(unsigned char*)(long)buf = v;
    }
    *(unsigned char*)(long)n = v;
    *actual = count;
    printf("El valor de count es %zu.\n", count);
    return ZX_OK;
}
```

- Escritura: método trivial donde simplemente se descartan los bytes recibidos y se avanza el puntero en función del valor de count:

```
static zx_status_t write(void* ctx, const void* buf, size_t count,
zx_off_t off, size_t* actual) {
    *actual = count;
    return ZX_OK;
}
```

- Release: libera el espacio ocupado por la estructura del contexto:

```
static void release(void* ctx) {
    free(ctx);
}
```

Especificación de cómo tiene que actuar el driver al recibir peticiones de lectura, escritura o bind por parte de un dispositivo:

```
static zx_protocol_device_t fuchsiadiversimple_device_proto = {
    .version = DEVICE_OPS_VERSION,
    .read = read,
    .write = write,
    .release = release,
};
```

Inicialización tanto del contexto del driver como de sus operaciones y conexión con el dispositivo:

```
zx_status_t bind(void* ctx, zx_device_t* parent) {
    fuchsiadiversimple_device_t* device = calloc(1, sizeof(*device));
    if (!device) {
        return ZX_ERR_NO_MEMORY;
    }

    unsigned char c = 44;
    device->val = (void*)(long)c;

    device_add_args_t args = {
        .version = DEVICE_ADD_ARGS_VERSION,
        .name = "fuchsiadiversimple",
        .ops = &fuchsiadiversimple_device_proto,
        .ctx = device,
    };

    zx_status_t rc = device_add(parent, &args, &device->zxdev);
    if (rc != ZX_OK) {
        free(device);
    }

    return rc;
}
```

Especificación del método de bind, `zx_driver_ops_t` es la ABI entre los módulos del driver y el device manager, se usa para que los drivers se puedan volver a reconstruir sin intervención del compilador.

```
static zx_driver_ops_t fuchsiadiversimple_driver_ops = {
    .version = DRIVER_OPS_VERSION,
    .bind = bind,
};
```

Mientras que con la siguiente instrucción el driver se declara a sí mismo, con su nombre, sus ops, su “vendor” id y la versión del driver:

```
ZIRCON_DRIVER(fuchsiadiversimple, fuchsiadiversimple_driver_ops,  
"zircon", "0.1");
```

En cuanto a los resultados, y debido a la inestabilidad que todavía presenta este entorno a la hora de desarrollar drivers, no fue posible incluir esta implementación (ni una implementación previa que fue realizada en C++) en la lista de drivers, a pesar de haberla enrutado correctamente en los respectivos ficheros dentro del source code del sistema:

- */build/drivers/all_drivers_list.txt*
- */bundles/BUILD.gn*
- */bundles/drivers/BUILD.gn*
- */src/devices/misc/BUILD.gn*

Por el contrario, si se logró incluir la implementación de */dev/full*, definiéndola manualmente dentro del device filesystem (“devfs”) para que fuera construida en la fase de boot del sistema.

Como esta implementación está creada manualmente dentro del directorio */dev* no es necesario que disponga de un fichero BUILD.gn, ya que ya viene enrutada desde el arranque en el “devfs”.

Para realizar esto, dentro del device filesystem, incluimos un puntero de tipo nodo mediante el cual acceder a nuestro device driver:

```
std::unique_ptr<Devnode> full_devnode;
```

Indicamos dentro del método *devfs_open()* (que recibe como parámetros el path que se quiere abrir, los flags y el dispatcher mediante el cual llama al device y se crea su instancia) un condicional que en función de si el parámetro path es igual a *kfullDevName* (variable que contiene el string “full”) realiza la apertura de nuestro device:

```
..if (path_view == kfullDevName){  
    fullDevices::Get(dispatcher)->HandleOpen(flags, std::move(ipc),  
    path_view);  
}
```

E inicializamos el nodo que va a tomar como padre a la raíz del directorio devfs y que tendrá como nombre “full”:

```
full_devnode = devfs_mkdir(root_devnode.get(), "full");
```

De forma que, una vez realizado este proceso, solo faltaría implementar el device.

En el fichero de cabecera definimos la clases correspondientes, indicando los métodos `read()` y `write()` que realizarán la funcionalidad del device e indicando los métodos `get()` y `handleOpen()` que permitirán crear la instancia del device y enlazarla con el puntero creado:

```
class fullDevVnode :  
public fs::Vnode, public fidl::WireServer<fuchsia_io::Directory> {  
public:  
    zx_status_t Read(void* data, size_t len, size_t off, size_t*  
        out_actual) override;  
    zx_status_t Write(const void* data, size_t len, size_t off, size_t*  
        out_actual) override;  
    ...  
  
class fullDevices {  
public:  
    static fullDevices* Get(async_dispatcher_t* dispatcher);  
    zx_status_t HandleOpen(fuchsia_io::OpenFlags flags,  
        fidl::ServerEnd<fuchsia_io::Node> request, std::string_view name);  
  
private:  
    explicit fullDevices(async_dispatcher_t* dispatcher):  
        vfs_(dispatcher) {}  
    fbl::RefPtr<fs::Vnode> full_dev_ = fbl::MakeRefCounted<fullDevVnode>();  
    fs::ManagedVfs vfs_;  
};
```

Mientras que en el fichero `.cc` implementamos los métodos:

- Lectura: llena el buffer destino de caracteres “null” y avanza el puntero en función del valor de “len”:

```
zx_status_t fullDevVnode::Read(void* data, size_t len, size_t off,  
size_t* out_actual) {  
    memset(data, 0, len);  
    *out_actual = len;  
    return ZX_OK;  
}
```

- Escritura: devuelve el código interno que indica que no queda espacio disponible, por lo cual no se realiza la escritura:

```
zx_status_t fullDevVnode::Write(const void* data, size_t len, size_t  
off, size_t* out_actual) {  
    return ZX_ERR_NO_SPACE;  
}
```

- Get: crea una instancia nueva del device si no existía previamente recibiendo como parámetro un objeto dispatcher:

```
fullDevices* fullDevices::Get(async_dispatcher_t* dispatcher) {
    if (instance == nullptr) {
        instance = new fullDevices(dispatcher);
    }
    return instance;
}
```

- HandleOpen: que permitirá crear la instancia del device y enlazarla con el puntero creado. Una vez creada la instancia se declara en su return el método serve mediante el cual se obtienen los protocolos (GetProtocols()) del vnode indicado en el primer parámetro para que sean servidos a través del endpoint indicado como segundo parámetro.:

```
zx_status_t fullDevices::HandleOpen(fio::OpenFlags flags,
                                    fidl::ServerEnd<fio::Node> request, std::string_view name) {

    auto options = fs::VnodeConnectionOptions::FromIoV1Flags(flags);

    fbl::RefPtr<fs::Vnode> vnode;
    if (name == kfullDevName) {
        vnode = full_dev_;
    } else {
        return ZX_ERR_INVALID_ARGS;
    }

    fbl::RefPtr<fs::Vnode> target;
    if (!options.flags.node_reference) {
        zx_status_t status = vnode->OpenValidating(options, &target);
        if (status != ZX_OK) {
            return status;
        }
    }
    if (target == nullptr) {
        target = vnode;
    }

    return vfs_.Serve(std::move(target), request.TakeChannel(), options);
}
```

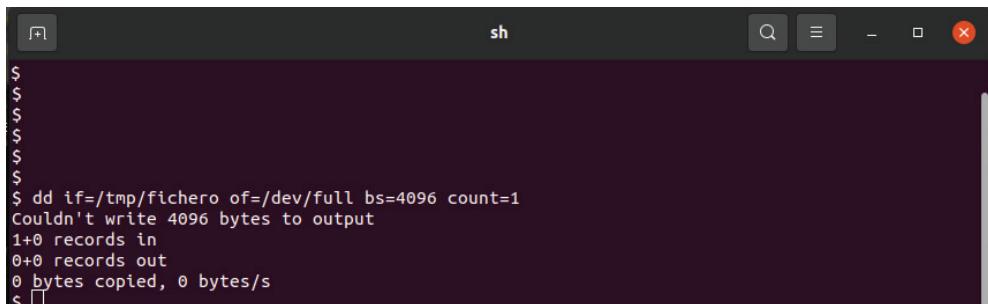
7.1.2 Resultados

Al ejecutar el device obtenemos los siguientes resultados:

- Lectura: para realizar el test modificamos el metodo para que en lugar de llenar el buffer con caracteres null lo llenara con el carater “A”

Figura 23. Test #1 /dev/full

- Escritura:



```
$
$ 
$ 
$ 
$ 
$ dd if=/tmp/fichero of=/dev/full bs=4096 count=1
Couldn't write 4096 bytes to output
1+0 records in
0+0 records out
0 bytes copied, 0 bytes/s
$
```

Figura 24. Test #2 /dev/full

7.2 Componentes

En cuanto al desarrollo de programas referentes a documentar el funcionamiento de los componentes de Fuchsia, se han implementado, de acuerdo a lo establecido en la fase de diseño, tres componentes formando un realm

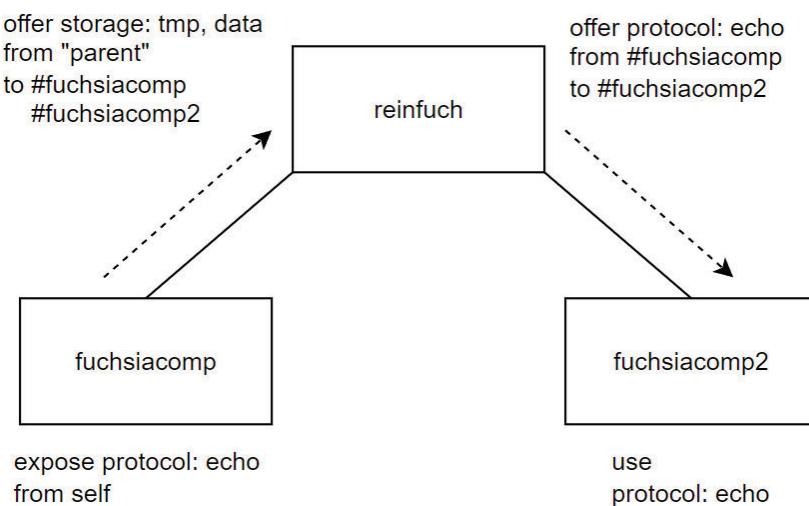


Figura 25. Esquema del realm

Como se puede ver en el esquema de la figura 25, el ciclo es el siguiente, fuchsiacomp tiene declarada la implementación de un protocolo echo y además la tiene en la sección expose de su manifiesto, por lo que un componente que la quiera usar puede solicitarla.

Reinfuch es el padre de ambos, por lo que hace de “puente” entre ellos para el intercambio de capabilities, de forma que, además de incluir en su manifiesto la oferta de la capability echo de fuchsiacomp a fuchsiacomp2 también ofrece, esta vez a ambos hijos, 2 capabilities storage llamadas “tmp” y “data” de parte de la colección “ffx-laboratory” hija del “core” del sistema.

Por último, fuchsiacomp2, para poder utilizar el protocolo echo, debe incluirlo en la sección use de su manifiesto, así, el component manager podrá seguir la “capability route” entre fuchsiacomp y fuchsiacomp2 para que este último pueda utilizar dicho protocolo (siguiendo un proceso similar para las dos storage capabilities).

7.2.1 Implementación

En cuanto a cómo están implementados, tanto el padre como cada hijo, están compuestos primeramente por su respectivo fichero `BUILD.gn` (que permite su correcta inclusión en la build del sistema al igual que en el caso de los drivers) y por su respectivo manifiesto (donde se declara como ya hemos visto anteriormente, la información referente a sus capabilities, entorno, runner, etc.), mientras que el caso de los hijos, además de estos dos ficheros, también presentan ficheros `.cc` y `.h` donde están contenidas las tareas que realizan.

El fichero `BUILD.gn` de fuchsiacomp presenta la siguiente información:

El grupo principal, donde se establece el nombre mediante el cual se le hace referencia y donde se incluyen, en este caso el paquete y los test (opcionales):

```
group("fuchsiacomp") {  
    testonly = true  
    deps = [  
        ":package", ":tests",  
    ]}
```

La declaración del componente y del paquete donde está contenido, como se puede ver, en el componente se incluye el grupo bin donde están las dependencias:

```
fuchsia_component("component") {  
    component_name = "fuchsiacomp"  
    manifest = "meta/fuchsiacomp.cml"  
    deps = [ ":bin" ]  
}  
fuchsia_package("package") {  
    package_name = "fuchsiacomp"  
    deps = [ ":component" ]  
}
```

Los grupos lib y bin, correspondientes a las dependencias que necesitaran los ficheros del componente para funcionar:

```
source_set("lib") {
    sources = [
        "fuchsiacomp.cc",
        "fuchsiacomp.h",
    ]

    public_deps = [
        "//sdk/lib/sys/inspect/cpp",
        "//sdk/lib/syslog/cpp",
        "//zircon/system/ulib/async",
        "//zircon/system/ulib/async:async-cpp",
    ]
}

executable("bin") {
    output_name = "fuchsiacomp"
    sources = [ "main.cc" ]
    deps = [
        ":lib",
        "//zircon/system/ulib/async-default",
        "//zircon/system/ulib/async-loop:async-loop-cpp",
        "//zircon/system/ulib/async-loop:async-loop-default",
        "//src/lib/files:files",
        "//zircon/system/ulib/fidl:fidl-llcpp",
        "//examples/components/routing/fidl:echo",
        "//zircon/system/ulib/fidl:fidl_base",
    ]
}
```

Los grupos correspondientes a los test son opcionales, y es donde se incluyen las pruebas desarrolladas por el programa para testear el componente/driver:

```
group("tests") {
    testonly = true
    deps = [ ":fuchsiacomp-unittests" ]
}
```

```

executable("unitests") {
  output_name = "fuchsiacomp_test"
  testonly = true
  sources = [ "fuchsiacomp_unittest.cc" ]
  deps = [
    ":lib",
    "//src/lib/fxl/test:gtest_main",
    "//third_party/googletest:gtest",
    "//zircon/system/ulib/async-default",
    "//zircon/system/ulib/async-loop:async-loop-cpp",
    "//zircon/system/ulib/async-loop:async-loop-default",
  ]
}

```

En cuanto al manifiesto de fuchsiacomp, fue desarrollado de la siguiente forma:

Los grupos “include” y “program” donde están, por ejemplo, la información del “inspect” usada para depurar, el tipo de runner, los argumentos con los que se llamará al componente cuando se inicie y las variables de su entorno:

```

{
  include: [
    "inspect/client.shard.cml",
    "syslog/client.shard.cml",
  ],

  program: {
    runner: "elf",
    binary: "bin/fuchsiacomp",
    args: [ "componentes", "drivers", "capas", "runtimes" ],
    environ: [ "VAL=usuario" ],
  },
}

```

Los grupos referentes a las capabilities, en este caso incluyendo “expose” y “use”:

```

capabilities: [
  { protocol: "fidl.examples.routing.echo.Echo" },
],

expose: [
  {
    protocol: "fidl.examples.routing.echo.Echo",
    from: "self",
  },
],

```

```

use: [
{
  storage: "tmp",
  path: "/tmp2",
}, {
  storage: "data",
  path: "/data2",
},
],
}

```

En los casos de Reinfuch y fuchsiacomp2, los ficheros `BUILD.gn` y sus manifiestos son bastante similares a excepción, claro está, de la sección correspondiente a las capabilities.

Fuchsiacomp2 declara en la sección “use” el protocolo echo:

```

use: [
  { protocol: "fidl.examples.routing.echo.Echo" },
],

```

Mientras que Reinfuch, además de las capabilities, también incluye la identificación de sus hijos:

```

{
  children: [
    {
      name: "fuchsiacomp",
      url: "#meta/fuchsiacomp.cm",
    }, {
      name: "fuchsiacomp2",
      url: "#meta/fuchsiacomp2.cm",
    },
  ],
}

offer: [
  {
    protocol: "fuchsia.logger.LogSink",
    from: "parent",
    to: [ "#fuchsiacomp",
      "#fuchsiacomp2", ],
  }, {
    protocol: "fidl.examples.routing.echo.Echo",
    from: "#fuchsiacomp",
    to: "#fuchsiacomp2",
  },
]

```

```
{
  storage: "tmp",
  from: "parent",
  to: [
    "#fuchsiacomp",
    "#fuchsiacomp2",
  ],
},
{
  storage: "data",
  from: "parent",
  to: [
    "#fuchsiacomp",
    "#fuchsiacomp2",
  ],
},
],
}
```

En cuanto a los ficheros .cc, por el lado de fuchsiacomp tenemos el siguiente código:

Por un lado, está la clase que implementa el protocolo echo, donde se declaran el echo propiamente dicho incluyendo un “callback” para obtener la respuesta y un “event sender” que es quien permite que su cliente envíe eventos, alertas, etc., a través de un canal:

```
class EchoImplementation : public fidl::examples::routing::echo::Echo {
public:
  void EchoString(fidl::StringPtr value, EchoStringCallback callback)
override {
  value->append(std::getenv("VAL"));
  callback(value);
}
  fidl::examples::routing::echo::Echo_EventSender* event_sender_;
};
```

Mientras que por otro lado está el main, donde se realizan tanto operaciones con ficheros y descriptores con el fin de ilustrar el comportamiento de Fuchsia en este ámbito, la creación de un mensaje FIDL para su posterior envío y recepción a través de un canal y la declaración del protocolo saliente echo en el Outgoing directory:

Hacemos uso de la clase “ScopedTempDirAt” propia de Fuchsia para crear un objeto que representa un directorio temporal, donde luego crearemos un fichero “file0” mediante el método `WriteFile()` (indicando el directorio, la ruta y el contenido), cabe recalcar que la ruta pasada como parámetro debe ser una ruta de acceso válida, es decir, debe indicarse como en este caso, la ruta de la storage capability (`/tmp2`):

```
files::ScopedTempDirAt temp_dir(AT_FDCWD);
bool aux = files::WriteFile(
    files::JoinPath(temp_dir.path(), "tmp2/file0"),
    "Lorem Ipsum is simply dummy text of the...");

FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Creamos el Fichero en tmp2/file0 = %d!", aux );
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Comprobamos que es un fichero valido = %d!",
files::IsFile("tmp2/file0"));
```

De modo similar al caso anterior, creamos de forma dinámica una serie de ficheros en la storage `/data2` en función de los argumentos definidos en el manifiesto del componente:

```
files::ScopedTempDir temp_dir2;
bool aux2;
for (int i = 1; i < argc; i++) {
    aux2 = 0;
    aux2 = files::WriteFile(
        files::JoinPath(temp_dir2.path(),
        "data2/file" + std::to_string(i) + "_" + argv[i]),
        "Lorem Ipsum is simply dummy text of the...");

    FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Creamos el Fichero en file%d = %d!", i, aux2 );
}
```

Hacemos uso del método `GetFileSize()` para establecer la capacidad del fichero indicado:

```
uint64_t n = 256;
aux = files::GetFileSize("tmp2/file0",&n);
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Establecemos la capacidad de file0 a 256 bytes = %d!", aux );
```

Y seguidamente realizamos la lectura de dicho fichero:

```
std::string result;
aux = files::ReadFileToString("tmp2/file0", &result);
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Leemos del fichero file0 = %d!", aux );
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] El contenido leido es = %s!", result.c_str());
```

Una vez finalizadas las operaciones con ficheros, creamos un descriptor y verificamos que se creó correctamente:

```
fbl::unique_fd fd(open("tmp2/file0", O_RDWR));
aux = fd.is_valid();
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Creamos un fd al fichero file0 = %d!", aux );
```

Hacemos uso de un método de Zircon para obtener el tamaño de una página de memoria:

```
long page_size = zx_system_get_page_size();
std::string texto = " El tamaño de una página en memoria es " +
std::to_string(page_size) + " ";
```

y para concluir trabajamos con dicho descriptor:

```
aux = lseek(fd.get(), 0, SEEK_END);
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Dejamos el fd apuntando a la última posicion del fichero
= %d!", aux );

aux = WriteFileDescriptor(fd.get(), texto.data(), texto.size());
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Escribimos a traves del fd = %d!", aux );

aux = lseek(fd.get(), 0, SEEK_SET);
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Dejamos el fd apuntando a la posicion %d del fichero!",
aux );
```

```

std::vector<char> buffer;
buffer.resize(256);
ssize_t leidos = ReadFileDescriptor(fd.get(), buffer.data(), 256);
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] Leemos %zd bytes del fichero del fd", leidos);
FX_LOGF(INFO, "fuchsiacomp",
"\n [Comp 1] El contenido leido es = %s!",
reinterpret_cast<char*>(buffer.data()));

```

Con respecto a la creación y envío del mensaje FIDL, primero creamos las variables que vamos a usar:

Declaramos un buffer de tamaño máximo para mensajes FIDL (“(uint32_t) 65536u”), dos estructuras (de tamaño “(uint32_t) 64u”) que usaremos para indicar por ejemplo los handles, el tipo de mensaje o los derechos tanto en el caso de la lectura como en el de escritura y el canal que se usará en la comunicación.

```

uint8_t byte_buffer[ZX_CHANNEL_MAX_MSG_BYTES];
zx_handle_info_t read_info [ZX_CHANNEL_MAX_MSG_HANDLES];
zx_handle_disposition_t write_info [ZX_CHANNEL_MAX_MSG_HANDLES];

// typedef struct zx_handle_info {
// zx_handle_t handle;
// zx_obj_type_t type;
// zx_rights_t rights;
// uint32_t unused;
// } zx_handle_info_t;

// typedef struct zx_handle_disposition {
// zx_handle_op_t operation;
// zx_handle_t handle;
// zx_obj_type_t type;
// zx_rights_t rights;
// zx_status_t result;
// } zx_handle_disposition_t;

zx::channel c1, c2;
zx::channel::create(0, &c1, &c2);

```

Posteriormente vamos a crear un “builder”, que es un objeto definido dentro de Fuchsia para almacenar de forma ordenada todo lo que se va a incluir para formar el Fidl message, se respalda en el `void*` que recibe como parámetro y que en nuestro caso denominamos `byte_buffer`:

```
fidl::Builder builder(byte_buffer, ZX_CHANNEL_MAX_MSG_BYTES);
```

Ya creado el “builder”, pasaremos a crear tanto el header como el contenido que tendrá el mensaje, donde, para el primero, usaremos la estructura de 16 bytes `fidl_message_header_t` en la que se incluyen los valores del formato del canal a usar en la comunicación (dos lados con diferente formato son incompatibles) y el ID de la transacción (`zx_txid_t txid`).

Como se mencionó anteriormente, el orden es relevante por lo que primero hay que crear el header dentro del “builder” y luego el contenido:

```
fidl_message_header_t* header = builder.New<fidl_message_header_t>();  
header->txid = 5u;  
header->ordinal = 42u;
```

```
fidl_string_t* view = builder.New<fidl_string_t>();  
  
// typedef struct fidl_string {  
//     uint64_t size;  
//     char* data;  
// } fidl_string_t;  
  
char* data = builder.NewArray<char>(10);  
strcpy(data, "hola");  
view->data = data;  
view->size = strlen(data);
```

Para crear el mensaje se usa la clase “HLCPPOutgoingMessage”, donde se especifica:

- El mensaje que se va a enviar, en este caso, el objeto “builder”, con el método `Finalize()` (para indicar que ya se concluyó la construcción del mensaje).
- Los handles de escritura

```
fidl::HLCPPOutgoingMessage outgoing_message(  
    builder.Finalize(),  
    fidl::HandleDispositionPart(  
        write_info, ZX_CHANNEL_MAX_MSG_HANDLES));
```

Y para enviarlo se escribe en el canal creado previamente:

```
outgoing_message.Write(c1.get(), 0u);
```

Una vez enviado el mensaje podemos recibirla a través del otro extremo del canal creado, por lo que, de forma similar a la escritura, usamos esta vez la clase “HLCPPIncomingMessage” e indicando el buffer y sus respectivos handles de lectura:

```
memset(byte_buffer, 0, ZX_CHANNEL_MAX_MSG_BYTES);
fidl::HLCPPIncomingMessage incoming_message(
    fidl::BytePart(byte_buffer, ZX_CHANNEL_MAX_MSG_BYTES),
    fidl::HandleInfoPart(read_info, ZX_CHANNEL_MAX_MSG_HANDLES));

incoming_message.Read(c2.get(), 0u);

fidl::BytePart bytes = fidl::BytePart(
    incoming_message.body_view().bytes(), 0);

fidl_string_t* view2 = reinterpret_cast<fidl_string_t*>(bytes.data());
char *d = new char[16];
strcpy(d, view2->data);
FX_LOGF(INFO, "fuchsiacomp", "\n [Comp 1] El msg es = %s!", d);
```

Por último, inicializamos el Outgoing directory, creamos una instancia de la implementación del protocolo echo y una lambda que realice la conexión (a través de la función `handler` creada) al recibir la petición:

```
auto component_context =
    sys::ComponentContext::CreateAndServeOutgoingDirectory();

EchoImplementation instancia;

fidl::Binding<fidl::examples::routing::echo::Echo> binding(&instancia);
instancia.event_sender_ = &binding.events();

fidl::InterfaceRequestHandler<fidl::examples::routing::echo::Echo>
handler = [&]()
{
    fidl::InterfaceRequest<fidl::examples::routing::echo::Echo> request
} {
    binding.Bind(std::move(request));
};

component_context->outgoing()->AddPublicService(std::move(handler));
```

En cuanto a la implementación de fuchsiacomp2, usamos de igual forma el método CreateAndServeOutgoingDirectory() pero esta vez con el objetivo de conectarnos a los servicios que nos han ofrecido y luego realizar la conexión con el protocolo echo para luego hacer uso de él:

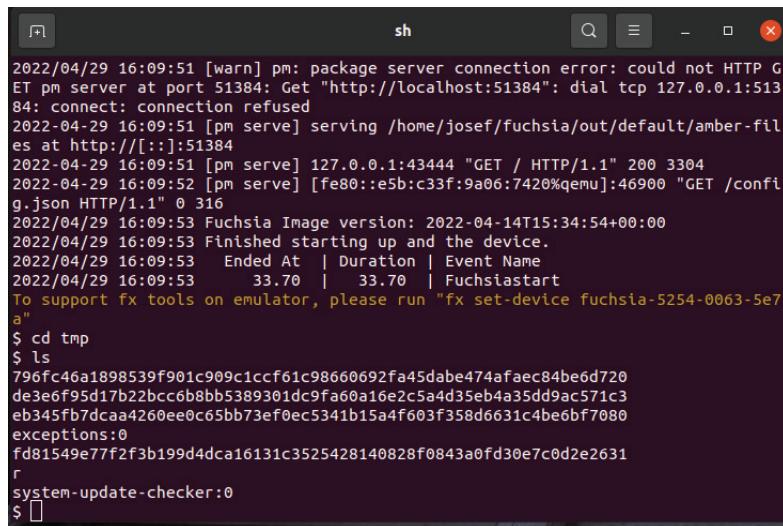
```
auto component_context =
sys::ComponentContext::CreateAndServeOutgoingDirectory();

fidl::examples::routing::echo::EchoSyncPtr conn_fuchsiacomp;
component_context->svc()->Connect(conn_fuchsiacomp.NewRequest());

fidl::StringPtr res = nullptr;
conn_fuchsiacomp->EchoString("Hola ", &res);
if (!res.has_value()) {
    FX_LOGF(INFO, "fuchsiacomp2",
    "\n [Comp 2] Envie un msg y no me respondieron");
} else {
    FX_LOGF(INFO, "fuchsiacomp2",
    "\n [Comp 2] Envie un msg y me respondieron = %s", res->c_str());
}
```

7.2.2 Resultados

Para exponer los resultados de la ejecución de estos componentes primero podemos comprobar el estado de los directorios del sistema:

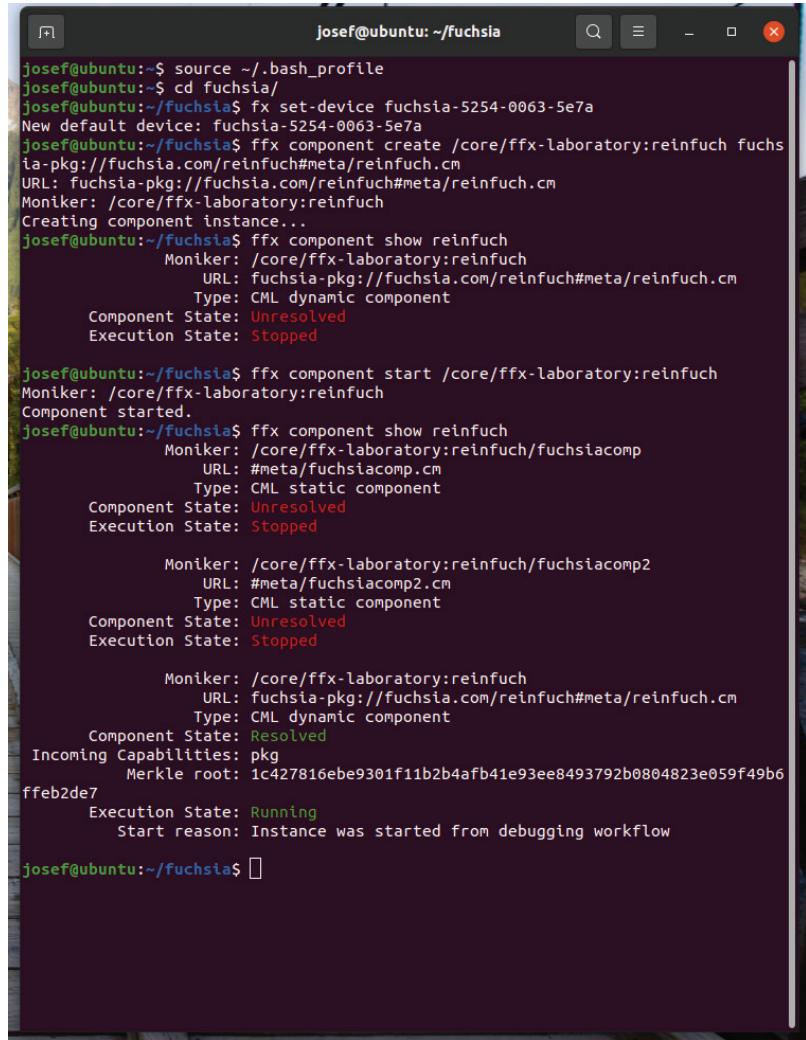


The screenshot shows a terminal window with the title 'sh'. The window displays a series of log messages from a Fuchsia system. The logs include warnings about package server connection errors, information about a device starting up, and details about a Fuchsia Image version. At the bottom, there is a command prompt with '\$' followed by several file paths and names, indicating a directory listing.

```
2022/04/29 16:09:51 [warn] pm: package server connection error: could not HTTP GET pm server at port 51384: Get "http://localhost:51384": dial tcp 127.0.0.1:51384: connect: connection refused
2022/04/29 16:09:51 [pm serve] serving /home/josef/fuchsia/out/default/amber-filles at http://[::]:51384
2022/04/29 16:09:51 [pm serve] 127.0.0.1:43444 "GET / HTTP/1.1" 200 3304
2022/04/29 16:09:52 [pm serve] [fe80::e5b:c33f:9a06:7420%qemu]:46900 "GET /config.json HTTP/1.1" 0 316
2022/04/29 16:09:53 Fuchsia Image version: 2022-04-14T15:34:54+00:00
2022/04/29 16:09:53 Finished starting up and the device.
2022/04/29 16:09:53 Ended At | Duration | Event Name
2022/04/29 16:09:53 33.70 | 33.70 | Fuchsiastart
To support fx tools on emulator, please run "fx set-device fuchsia-5254-0063-5e7a"
$ cd tmp
$ ls
796fc46a1898539f901c909c1ccf61c98660692fa45dabe474afaec84be6d720
de3e6f95d17b22bcc6b8bb5389301dc9fa0a16e2c5a4d35eb4a35dd9ac571c3
eb345fb7dcaa4260ee0c65bb73ef0ec5341b15a4f603f358d6631c4be6bf7080
exceptions:0
fd81549e77f2f3b199d4dca16131c3525428140828f0843a0fd30e7c0d2e2631
r
system-update-checker:0
$
```

Figura 25. Estado inicial del sistema

Como se puede ver en la figura 25, no hay ficheros ni directorios referentes a nuestros componentes.



```
josef@ubuntu:~/fuchsia$ source ~/.bash_profile
josef@ubuntu:~/fuchsia$ cd fuchsia/
josef@ubuntu:~/fuchsia$ fx set-device fuchsia-5254-0063-5e7a
New default device: fuchsia-5254-0063-5e7a
josef@ubuntu:~/fuchsia$ ffx component create /core/ffx-laboratory:reinfuch fuchs
ia-pkg://fuchsia.com/reinfuch#meta/reinfuch.cm
URL: fuchsia-pkg://fuchsia.com/reinfuch#meta/reinfuch.cm
Moniker: /core/ffx-laboratory:reinfuch
Creating component instance...
josef@ubuntu:~/fuchsia$ ffx component show reinfuch
    Moniker: /core/ffx-laboratory:reinfuch
        URL: fuchsia-pkg://fuchsia.com/reinfuch#meta/reinfuch.cm
        Type: CML dynamic component
    Component State: Unresolved
    Execution State: Stopped

josef@ubuntu:~/fuchsia$ ffx component start /core/ffx-laboratory:reinfuch
Moniker: /core/ffx-laboratory:reinfuch
Component started.
josef@ubuntu:~/fuchsia$ ffx component show reinfuch
    Moniker: /core/ffx-laboratory:reinfuch/fuchsiacomp
        URL: #meta/fuchsiacomp.cm
        Type: CML static component
    Component State: Unresolved
    Execution State: Stopped

        Moniker: /core/ffx-laboratory:reinfuch/fuchsiacomp2
        URL: #meta/fuchsiacomp2.cm
        Type: CML static component
    Component State: Unresolved
    Execution State: Stopped

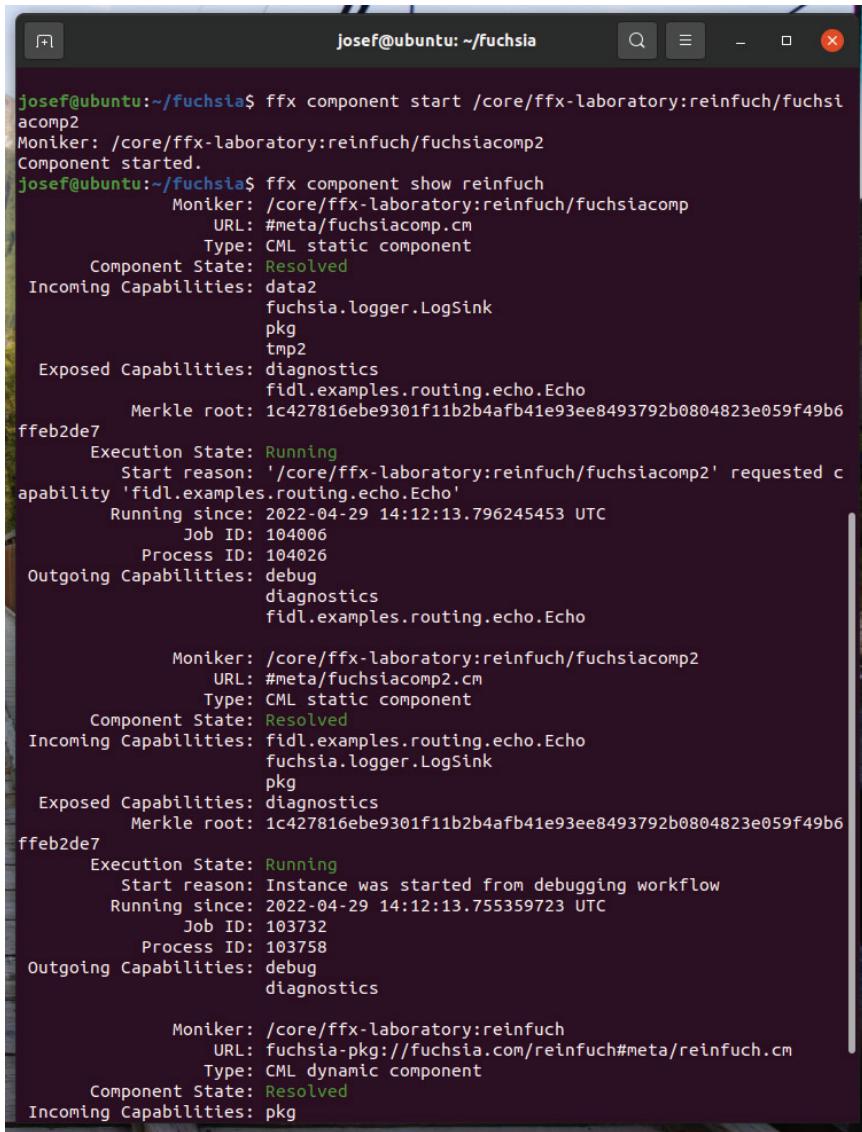
        Moniker: /core/ffx-laboratory:reinfuch
        URL: fuchsia-pkg://fuchsia.com/reinfuch#meta/reinfuch.cm
        Type: CML dynamic component
    Component State: Resolved
    Incoming Capabilities: pkg
        Merkle root: 1c427816ebe9301f11b2b4afb41e93ee8493792b0804823e059f49b6
ffeb2de7
    Execution State: Running
    Start reason: Instance was started from debugging workflow

josef@ubuntu:~/fuchsia$
```

Figura 26. Creación e inicialización de la instancia de reinfuch

En la figura 26, se ve como, mediante el comando `ffx component create`, creamos una instancia de nuestro componente padre (indicando el moniker y la URL), la cual, como se ve al ejecutar la siguiente instrucción “`ffx component show`”, está detenida ya que no le hemos dado ninguna orden.

Al ejecutar el comando `ffx component start` y el moniker correspondiente, iniciamos la ejecución de dicha instancia, por lo que, en ese momento, se crean además las instancias de sus dos hijos.



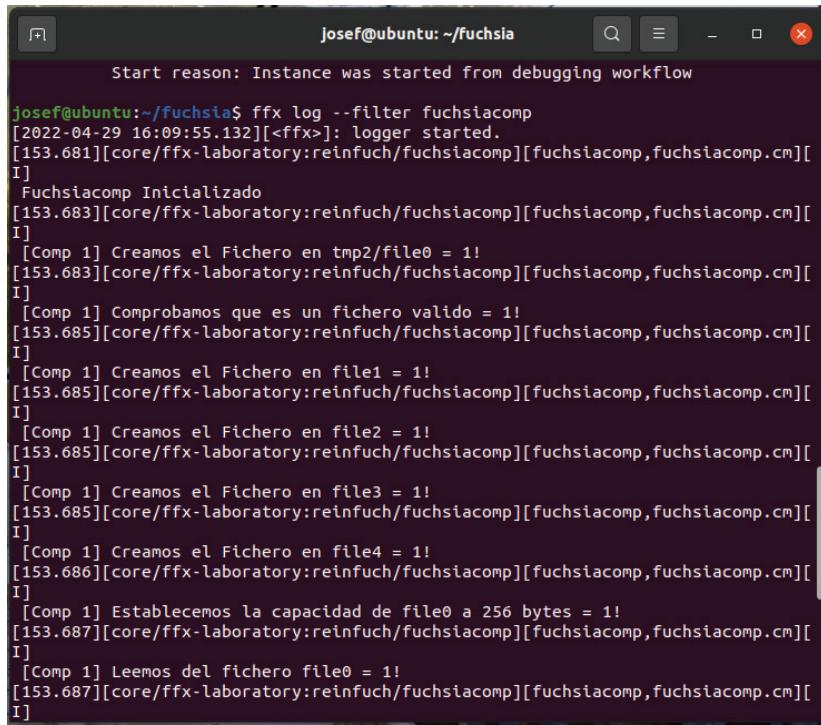
```
josef@ubuntu:~/fuchsia$ ffx component start /core/ffx-laboratory:reinfuch/fuchsiacomp2
Moniker: /core/ffx-laboratory:reinfuch/fuchsiacom2
Component started.
josef@ubuntu:~/fuchsia$ ffx component show reinfuch
    Moniker: /core/ffx-laboratory:reinfuch/fuchsiacom2
        URL: #meta/fuchsiacom2.cm
        Type: CML static component
    Component State: Resolved
    Incoming Capabilities: data2
        fuchsia.logger.LogSink
        pkg
        tmp2
    Exposed Capabilities: diagnostics
        fidl.examples.routing.echo.Echo
    Merkle root: 1c427816ebe9301f11b2b4afb41e93ee8493792b0804823e059f49b6
ffeb2de7
        Execution State: Running
            Start reason: '/core/ffx-laboratory:reinfuch/fuchsiacom2' requested capability 'fidl.examples.routing.echo.Echo'
            Running since: 2022-04-29 14:12:13.796245453 UTC
            Job ID: 104006
            Process ID: 104026
    Outgoing Capabilities: debug
        diagnostics
        fidl.examples.routing.echo.Echo

        Moniker: /core/ffx-laboratory:reinfuch/fuchsiacom2
            URL: #meta/fuchsiacom2.cm
            Type: CML static component
        Component State: Resolved
    Incoming Capabilities: fidl.examples.routing.echo.Echo
        fuchsia.logger.LogSink
        pkg
    Exposed Capabilities: diagnostics
        Merkle root: 1c427816ebe9301f11b2b4afb41e93ee8493792b0804823e059f49b6
ffeb2de7
        Execution State: Running
            Start reason: Instance was started from debugging workflow
            Running since: 2022-04-29 14:12:13.755359723 UTC
            Job ID: 103732
            Process ID: 103758
    Outgoing Capabilities: debug
        diagnostics

        Moniker: /core/ffx-laboratory:reinfuch
            URL: fuchsia-pkg://fuchsia.com/reinfuch#meta/reinfuch.cm
            Type: CML dynamic component
    Component State: Resolved
    Incoming Capabilities: pkg
```

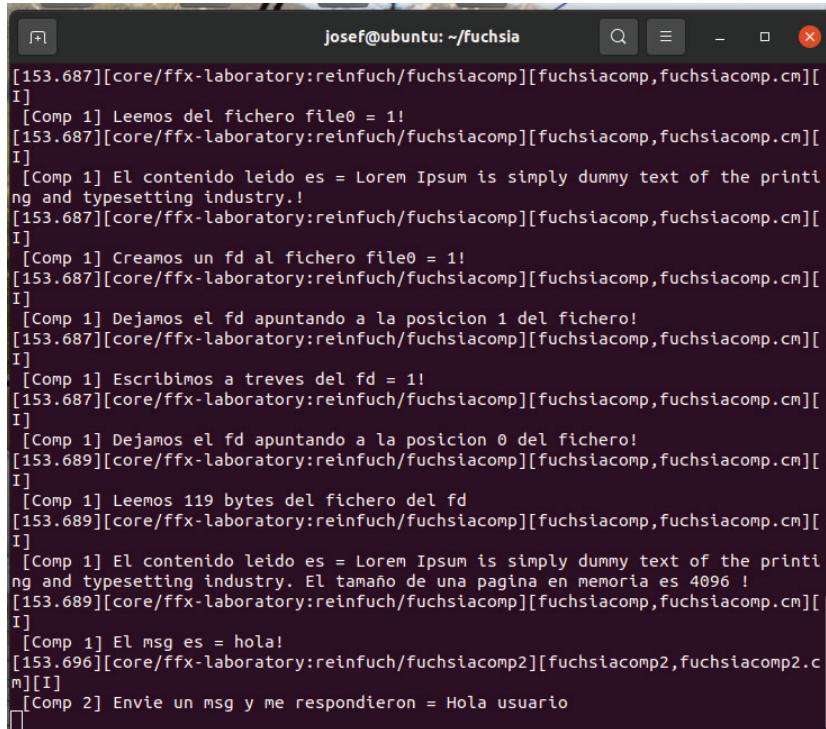
Figura 27. inicialización de la instancia de fuchsiacom2

De forma análoga a reinfuch, usamos el comando ffx component start para inicializar la instancia creada previamente de fuchsiacom2, y al realizar esto y como fue explicado en la implementación, al iniciarse dicha instancia se dará lugar al proceso de búsqueda de capabilities, por lo que “despertarán” a la instancia de fuchsiacom para que pueda resolver la petición de fuchsiacom2, de esta forma, todas las instancias de nuestro realm estarían inicializadas y ejecutándose.



```
josef@ubuntu:~/fuchsia$ ffx log --filter fuchsiacomp
[2022-04-29 16:09:55.132][<ffx>]: logger started.
[153.681][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
Fuchsiacomp Inicializado
[153.683][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Creamos el Fichero en tmp2/file0 = 1!
[153.683][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Comprobamos que es un fichero valido = 1!
[153.685][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Creamos el Fichero en file1 = 1!
[153.685][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Creamos el Fichero en file2 = 1!
[153.685][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Creamos el Fichero en file3 = 1!
[153.685][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Creamos el Fichero en file4 = 1!
[153.686][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Establecemos la capacidad de file0 a 256 bytes = 1!
[153.687][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Leemos del fichero file0 = 1!
[153.687][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
```

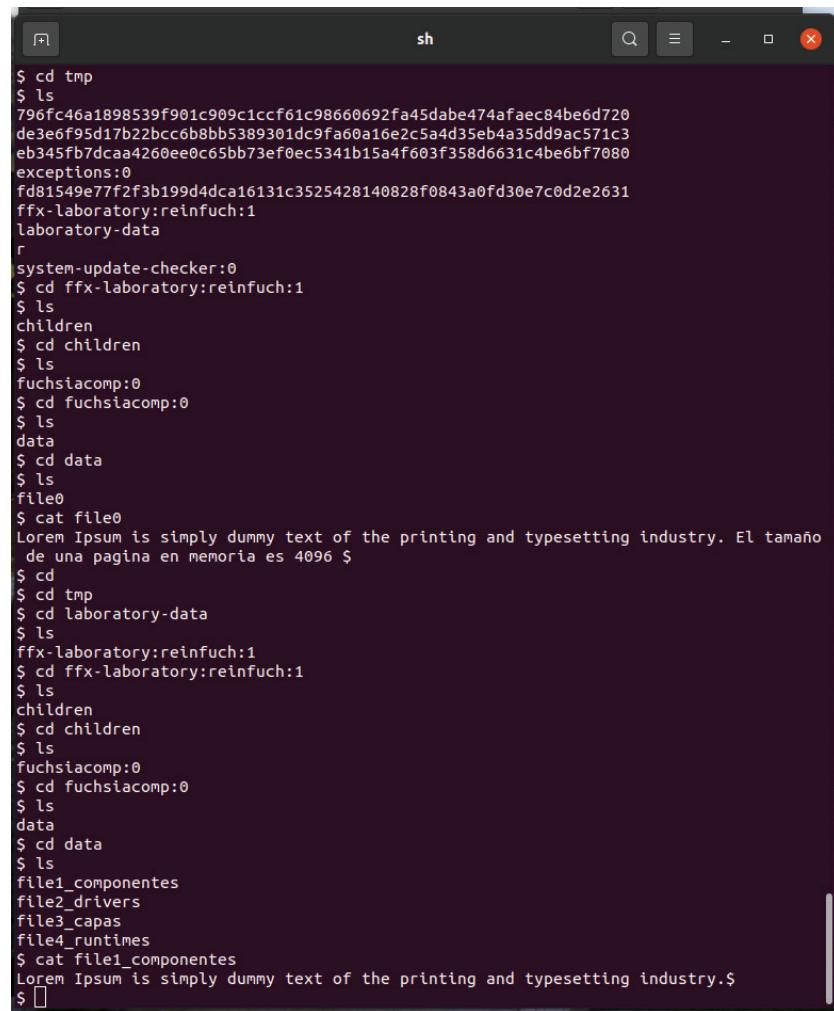
Figura 28. Comprobación de salidas #1



```
[153.687][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Leemos del fichero file0 = 1!
[153.687][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] El contenido leido es = Lorem Ipsum is simply dummy text of the printing and typesetting industry.!
[153.687][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Creamos un fd al fichero file0 = 1!
[153.687][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Dejamos el fd apuntando a la posicion 1 del fichero!
[153.687][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Escribimos a treves del fd = 1!
[153.687][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Dejamos el fd apuntando a la posicion 0 del fichero!
[153.689][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] Leemos 119 bytes del fichero del fd
[153.689][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] El contenido leido es = Lorem Ipsum is simply dummy text of the printing and typesetting industry. El tamaño de una pagina en memoria es 4096 !
[153.689][core/ffx-laboratory:reinfuch/fuchsiacomp][fuchsiacomp,fuchsiacomp.cm][I]
[Comp 1] El msg es = hola!
[153.696][core/ffx-laboratory:reinfuch/fuchsiacomp2][fuchsiacomp2,fuchsiacomp2.cm][I]
[Comp 2] Envie un msg y me respondieron = Hola usuario
```

Figura 29. Comprobación de salidas #2

En las figuras 28 y 29 se puede apreciar como gracias a ejecutar la instrucción `ffx log --filter fuchsiacomp` podemos acceder al log del sistema y ver cada una de las salidas que declaramos a lo largo de la implementación de los componentes, comprobando así, como se crean de forma correcta cada uno de los ficheros con sus respectivos contenidos, se leen y se escriben, como se recibe el mensaje FIDL enviado y como el componente `fuchsiacomp2` recibe la respuesta de `fuchsiacomp`.



```

$ cd tmp
$ ls
796fc46a1898539f901c909c1ccf61c98660692fa45dabe474afaec84be6d720
de3e6f95d17b22bcc6b8bb5389301dc9fa60a16e2c5a4d35eb4a35dd9ac571c3
eb345fb7dcaa4260ee0c65bb73ef0ec5341b15a4f603f358d6631c4be6bf7080
exceptions:0
fd81549e77f2f3b199d4dca16131c3525428140828f0843a0fd30e7c0d2e2631
ffx-laboratory:reinfuch:1
laboratory-data
r
system-update-checker:0
$ cd ffx-laboratory:reinfuch:1
$ ls
children
$ cd children
$ ls
fuchsiacomp:0
$ cd fuchsiacomp:0
$ ls
data
$ cd data
$ ls
file0
$ cat file0
Lorem Ipsum is simply dummy text of the printing and typesetting industry. El tamaño
de una pagina en memoria es 4096 $
$ cd
$ cd tmp
$ cd laboratory-data
$ ls
ffx-laboratory:reinfuch:1
$ cd ffx-laboratory:reinfuch:1
$ ls
children
$ cd children
$ ls
fuchsiacomp:0
$ cd fuchsiacomp:0
$ ls
data
$ cd data
$ ls
file1_componentes
file2_drivers
file3_capas
file4_runtimes
$ cat file1_componentes
Lorem Ipsum is simply dummy text of the printing and typesetting industry.$
$ []

```

Figura 30. Estado de los directorios después de la ejecución.

Comprobamos además en la figura 30 como todos los ficheros están creados correctamente cada uno con su contenido. “`file0`”, que fue creado en la storage “`tmp`” se encuentra en el directorio temporal `/ffx-laboratory:reinfuch:1` mientras que el resto de los ficheros fueron creados en la storage “`data`”, que se encuentra en el directorio `/laboratory-data`, ya que emula un comportamiento persistente.

8 Conclusiones

Una vez expuestas tanto la implantación de los programas desarrollados como los resultados obtenidos al ponerlos en ejecución podemos darnos cuenta de que a día de hoy todavía es un poco difícil poner a prueba la documentación y la información que se tiene sobre cómo, a priori, deberían de desarrollarse los componentes, drivers, y en general cualquier programa en este entorno, no solo por la falta de estabilidad del sistema (por su aun estado experimental y porque constantemente se realizan cambios tanto en el código fuente como en la documentación) y lo difícil que es probar el código que se está realizando (ya que, por ejemplo, cada vez que se desarrolla un programa o se realiza algún cambio en alguna sección de un programa ya desarrollado, hay que volver a generar el sistema con dicho programa modificado e iniciararlo, lo cual puede tardar 5, 10 minutos o más) sino también por la poca claridad en la documentación oficial sobre cuáles son las herramientas, métodos o librerías a usar en cada caso.

Fuchsia es un sistema operativo bastante llamativo que, solo por las premisas que plantea de ser un sistema universal, para cualquier dispositivo, y capaz de ejecutar código en una gran variedad de lenguajes, hace que sea difícil no apostar por él, pero eso no quita que todavía le quede desarrollo por delante y trabajo por realizar si tienen el objetivo de cumplir dichas premisas y de captar al gran público de desarrolladores en el mercado para que trabajen y se decanten por este entorno en el futuro.

En cuanto a la programación en este sistema, se plantea una metodología bastante interesante, sobre todo por el hecho de poder programar componentes y diseñar a partir de ellos un árbol de “parentescos” (usando los component manifest) mediante el cual se facilite el intercambio de capabilities entre padres e hijos y viceversa, de recursos, y además teniendo la opción de intercambiar mensajes con la ayuda de FIDL entre ellos.

FIDL es una herramienta bastante potente que, con la ayuda de una buena documentación, se podría lograr que cualquier desarrollador o usuario que esté interesado en incursionar en el mundo de Fuchsia y Zircon la use de forma eficiente y se logre explotar al máximo su potencial, lo que falta es precisamente un poco más de trabajo no solo ya a nivel del sistema operativo sino también a nivel de documentación y de guías.

Por otro lado, en cuanto a los drivers, creo que falta claridad en este ámbito, no tanto en la forma de implementarlos o desarrollarlos, si no principalmente en cómo lograr de forma satisfactoria que se incluyan en la lista de drivers del sistema ya que de momento no es una tarea sencilla.

En general, quitando el hecho de que tanto el código fuente como la documentación están en constante cambio (como es normal en el primero ya que todavía está en una etapa “experimental”), esta última la veo poco clara o ambigua a la hora de explicar cómo desarrollar código en Fuchsia, por ejemplo, en la página Fundamentals/Concepts/kernel/libc [64] explican que para crear un nuevo proceso dentro de un job hay que usar el método `Fdio_spawn()`, pero justo antes, en la página Fundamentals/Concepts/processes [65] explican que el método que hay que usar es `zx_process_create()`, y al revisar sus descripciones no dan ninguna información sobre cuándo usar un método en lugar del otro, o si es indiferente, etc.

Para incluir el driver por ejemplo, en Source/Drivers/Developer-Guide [28], indican que es necesario incluir su ruta en el fichero “relevante” dentro del directorio `//boards` (directorio en el que hay más de 15 ficheros) sin especificar en cual (cabe recalcar que intenté incluirlo en cada uno de ellos sin obtener resultados positivos), pero no mencionan que hay que incluir también dicha ruta dentro de los ficheros `BUILD.gn` desde la raíz hasta la carpeta donde está ubicada la implementación de dicho driver.

También se podría agregar alguna documentación o ejemplo sobre cómo crear una capability desde cero o sobre otros temas de utilidad que posiblemente generen dudas en los usuarios/ desarrolladores que empiecen a programar código en este entorno, como, por ejemplo, el uso de FIDL dentro de un componente, etc.

En cuanto a las líneas de trabajo futuras, y con el objetivo de lograr profundizar más en cada uno de los conceptos estudiados y aplicados a lo largo de este trabajo, habría que, primero y más importante, seguir al día las actualizaciones y modificaciones que se hacen tanto en las APIs, código fuente y documentación del sistema operativo para así estar siempre al día de cuáles son las mejores y más eficientes formas de desarrollar programas en este entorno, a partir de este punto, ya se podría ayudar al equipo de desarrollo de Fuchsia llevando a cabo las tareas que se consideren de utilidad para el futuro del sistema operativo y para su comunidad, como por ejemplo el desarrollo APIs, el desarrollo de nuevos módulos gráficos de Flutter o trabajar en la adición de un nuevo lenguaje de programación al sistema.

9 Análisis de Impacto

En cuanto a la relación de este trabajo y en general del desarrollo de Fuchsia como sistema operativo con los objetivos de desarrollo sostenible marcados por la Organización Mundial de la Salud para la agenda de 2030 se puede resaltar el hecho de que, por naturaleza, el ser humano siempre ha buscado nuevas formas de innovar y de expandir sus horizontes ya sea en el ámbito social, laboral, o personal, es por eso que el desarrollo de un sistema tan novedoso e inclusivo como Fuchsia, que puede ser usado en cualquier dispositivo y con un gran y variado soporte para lenguajes de programación precisamente motiva no solo a los desarrolladores y usuarios sino también a la competencia de Google en ese mismo sector y a ellos mismos a buscar esta innovación, de mantener la competencia y de lograr superarse cada vez más en pro de modernizar y alcanzar nuevos niveles en la tecnologización de la sociedad, dando así y como mencionan en el objetivo 9, “rienda suelta a las fuerzas económicas dinámicas y competitivas que generan el empleo y los ingresos”.

Otro punto para resaltar es que, por la forma en la que tanto de Fuchsia como de su núcleo Zircon funcionan o están diseñados, ambos buscan precisamente ese uso eficiente de los recursos que se menciona en la agenda de la ONU, para gestionar de forma efectiva, por ejemplo, la energía eléctrica que tan importante es en la sociedad en la que vivimos y que además brinda más opciones y facilita cada vez más la reducción de esa brecha digital que afecta a gran parte de la población.

10 Bibliografía

- [1] Google (08 Oct 2021). *Fidl Overview* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/fidl/overview>
- [2] littlekernel (20 Feb 2016). *Kernel APIs And Primitives* [Online]. Available: <https://github.com/littlekernel/lk/wiki/Kernel-APIs-And-Primitives>
- [3] Wikipedia (09 Feb 2022). *Fastboot* [Online]. Available: <https://en.wikipedia.org/wiki/Fastboot>
- [4] GNU (17 Feb 2015). *Mach* [Online]. Available: <https://www.gnu.org/software/hurd/microkernel/mach.html>
- [5] GNU (17 Feb 2015). *MIG* [Online]. Available: <https://www.gnu.org/software/hurd/microkernel/mach/mig.html>
- [6] Academic. *Mach (kernel)* [Online]. Available: <https://en-academic.com/dic.nsf/enwiki/12799>
- [7] Academic. *Tru64* [Online]. Available: <https://es-academic.com/dic.nsf/eswiki/351046>
- [8] Wikipedia (08 Aug 2021). *Micrónucleo* [Online]. Available: [https://es.wikipedia.org/wiki/Micr%C3%B3n%C3%BAcleo#:~:text=En%20computaci%C3%B3n%2C%20un%20micr%C3%B3n%C3%BAcleo%20\(en,entre%20procesos%20y%20planificaci%C3%B3n%20b%C3%A1sica.](https://es.wikipedia.org/wiki/Micr%C3%B3n%C3%BAcleo#:~:text=En%20computaci%C3%B3n%2C%20un%20micr%C3%B3n%C3%BAcleo%20(en,entre%20procesos%20y%20planificaci%C3%B3n%20b%C3%A1sica.)
- [9] Wikipedia (24 Feb 2008). *Núcleo monolítico* [Online]. Available: <https://www.classicistranieri.com/es/articles/n/%C3%BA/c/N%C3%BAcleo%20monol%C3%ADtico.html#:~:text=Un%20N%C3%BAcleo%20monol%C3%ADtico%20es%20el,todos%20los%20servicios%20del%20sistema.>
- [10] Google (10 Feb 2022). *Zircon* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/kernel>
- [11] Google (10 Feb 2022). *Zircon* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/kernel>
- [12] Anna-Lena Marx (5 May 2021). *A deep-dive into Fuchsia* [Online]. Available: <https://www.inovex.de/de/blog/a-deep-dive-into-fuchsia/>
- [13] Google (17 Feb 2022). *Fuchsia Interface Definition Language* [Online]. Available: <https://fuchsia.dev/fuchsia-src/get-started/learn/fidl/fidl>
- [14] Google (22 Nov 2021). *Fuchsia Interfaces* [Online]. Available: <https://fuchsia.dev/fuchsia-src/get-started/learn/fidl>
- [15] Google (25 Oct 2021). *Simple Drivers* [Online]. Available: https://fuchsia.dev/fuchsia-src/development/drivers/developer_guide/simple

- [16] Paul Emmerich, Simon Ellmann, (21 Abr 2019). *Drivers in High-Level Programming Languages* [Online]. Available: <https://github.com/ixy-languages/ixy-languages/blob/master/slides/2019-04-21%20Easterhegg.pdf>
- [17] CVE Details. *Linux Kernel : Vulnerability Statistics* [Online]. Available: https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33
- [18] Google. *The Fuchsia layer cake* [Online]. Available: <https://fuchsia.googlesource.com/docs/+/refs/changes/56/101356/4/layers.md>
- [19] Prithi Pal Thakur (20 May 2019). *Google Fuchsia OS – Everything you need to know* [Online]. Available: <https://www.vtnezwelt.com/insights/google-fuchsia-os/>
- [20] Google (18 Ene 2022), *Layer cake deprecation* [Online]. Available: https://fuchsia.dev/fuchsia-src/contribute/open_projects/srcs/layer_cake_deprecation
- [21] Google (25 nov 2019), *Remove Ledger from the component context* [Online]. Available: <https://fuchsia-review.googlesource.com/c/fuchsia/+/340928>
- [22] Google, *Git repositories on Fuchsia* [Online]. Available: <https://fuchsia.googlesource.com/?format=HTML>
- [23] Google, *Garnet* [Online]. Available: <https://cs.opensource.google/fuchsia/fuchsia/+/main:garnet/>
- [24] Francesco Pagano, Luca Verderame and Alessio Merlo, (9 Aug 2021). *Understanding Fuchsia Security* [Online]. Available: <https://arxiv.org/abs/2108.04183#:~:text=Fuchsia%20is%20a%20new%20open,focus%20on%20security%20and%20privacy.>
- [25] Google (22 nov 2021). *Fuchsia Driver Framework* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/drivers/fdf>
- [26] Wikipedia (19 Jun 2020). *Interfaz binaria de aplicaciones* [Online]. Available: https://es.wikipedia.org/wiki/Interfaz_binaria_de_aplicaciones
- [27] Google (10 Feb 2022). *Banjo in drivers* [Online]. Available: https://fuchsia.dev/fuchsia-src/development/drivers/concepts/device_driver_model/banjo
- [28] Google (18 Ene 2022). *Fuchsia driver development* [Online]. Available: https://fuchsia.dev/fuchsia-src/development/drivers/developer_guide/developer-development
- [29] Google (13 Jul 2021). *The Device ops* [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/drivers/device_driver_model/device-ops

- [30] Google (10 Mar 2022). *Environments* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/environments>
- [31] Dart. *Dart Overview* [Online]. Available: <https://dart.dev/overview>
- [32] Google (10 Mar 2022). *Component runners* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/runners>
- [33] Evans Data Corporation (5 Oct 2016). *Mobile Developer Population* [Online]. Available: <https://evansdata.com/press/viewRelease.php?pressID=244>
- [34] Google (10 Mar 2022). *Introduction to Fuchsia components* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/introduction>
- [35] Google (10 Mar 2022). *Component resolvers* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/resolvers>
- [36] Google (23 Mar 2022). *Glossary* [Online]. Available: <https://fuchsia.dev/fuchsia-src/glossary>
- [37] Google (25 Mar 2022). *Component manifest (.cml) reference* [Online]. Available: <https://fuchsia.dev/reference/cml>
- [38] Google (23 Mar 2022). *Fuchsia Namespaces* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/process/namespaces>
- [39] Google (13 Mar 2022). *Capabilities* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities>
- [40] Google (21 Mar 2022). *Zircon Handles* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/kernel/handles>
- [41] Google (27 Mar 2022). *Life of an open protocol* [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/life_of_a_protocol_open
- [42] Google (23 Mar 2022). *Service capabilities* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/service>
- [43] Google (23 Mar 2022). *Directory capabilities* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/directory>
- [44] Google (21 Feb 2022). *Hub* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/hub#view-hub>

- [45] Google (23 Mar 2022). *Storage capabilities* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/capabilities/storage>
- [46] Google (23 Mar 2022). *Component monikers* [Online]. Available: <https://fuchsia.dev/fuchsia-src/reference/components/moniker>
- [47] Google (23 Mar 2022). *Realms* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/components/v2/realms>
- [48] Google (23 Mar 2022). *Component storage index* [Online]. Available: https://fuchsia.dev/fuchsia-src/development/components/component_id_index
- [49] Google, Cobalt: *Telemetry with built-in privacy* [Online]. Available: <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/cobalt/>
- [50] Fuchsia, Fargo [Online]. Available: <https://fuchsia.googlesource.com/fargo/+/refs/heads/main>
- [51] Google (16 Feb 2022). *Filesystem Architecture* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/filesystems/filesystems>
- [52] Google (21 Feb 2022). *Life of an Open* [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/filesystems/life_of_an_open
- [53] Google (13 Jul 2021). *Block Devices* [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/filesystems/block_devices
- [54] Google (28 Mar 2022). *Zircon Kernel Concepts* [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/kernel/concepts#shared_memory_virtual_memory_objects_vmos
- [55] Google (13 Jul 2021). *BlobFS* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/filesystems/blobfs>
- [56] Google. *ThinFS* [Online]. Available: <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/lib/thinfs/>
- [57] Google. *Storage* [Online]. Available: <https://cs.opensource.google/fuchsia/fuchsia/+/main:src/storage/minfs/>
- [58] Google (30 Jul 2021). *Fuchsia's software model* [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/software_model
- [59] Google (03 Feb 2022). *Jobs* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/process/jobs>
- [60] Google (28 Abr 2022). *Process Creation* [Online]. Available: https://fuchsia.dev/fuchsia-src/concepts/process/process_creation
- [61] Google (13 Jul 2022). *Zircon Signals* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/kernel/signals?hl=en>

- [62] Google (24 Ago 2021). *zx_object_signal* [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/syscalls/object_signal
- [63] Google (24 Ago 2021). *zx_task_kill* [Online]. Available: https://fuchsia.dev/fuchsia-src/reference/syscalls/task_kill
- [64] Google (13 Dic 2021). *libc* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/kernel/libc>
- [65] Google (03 Feb 2022). *Processes* [Online]. Available: <https://fuchsia.dev/fuchsia-src/concepts/process/overview>

Este documento esta firmado por

	Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Wed Jun 01 21:36:59 CEST 2022
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)