Universidad Zaragoza

Trabajo Fin de Grado

Cargado dinámico de plugins en Rust en ausencia de estabilidad en la Interfaz Binaria de Aplicación

Dynamic loading of plugins in Rust in the absence of a stable Application Binary Interface

Autor:

Mario Ortiz Manero

Director:

Francisco Javier Fabra Caro

Grado en Ingeniería Informática

Departamento de Ingeniería e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura

Junio 2022

AGRADECIMIENTOS

Un primer gracias a toda mi familia por apoyarme en cualquier momento que lo necesitara. En especial, a mi padre y a mi madre por aguantarme siempre y por hacer posible que estudie y me gradúe.

A mis amigos que me han acompañado toda la vida con tan buenos momentos, y a los de la universidad, que fueron la verdadera motivación a ir a clase y seguir día a día. Las horas incontables juntos en la biblioteca, en el bar de la EINA o incluso en los montes más recónditos de Juslibol a las dos de la madrugada, han hecho que estos años merezcan la pena.

También a mis profesores por orientarme, particularmente a Javier Fabra por ayudarme con mi Erasmus, este documento y todas sus complicaciones. Heinz, Matthias y Darach, mis mentores de Tremor, han tomado un papel fundamental, no solo dándome consejo para el proyecto, sino también para toda mi carrera profesional.

Finalmente, agradecer a todas las organizaciones y empresas que han hecho esto posible. A la Fundación de Linux, por ofrecer los medios. A Wayfair, por apostar en talento joven. Y a la comunidad de código abierto, que me ha motivado a programar desde el principio y me ha guiado hasta donde estoy hoy.

RESUMEN

La toma de decisiones de muchas empresas modernas como Wayfair se basa en la recolección y análisis de datos de sus sistemas. Para llevarlo a cabo de forma eficiente en una escala masiva, es necesario el uso de herramientas de alto rendimiento como Tremor, escrito con Rust, programación asíncrona y SIMD.

A medida que Tremor evoluciona, sus tiempos de compilación crecen y, consecuentemente, se deteriora la experiencia de desarrollo. Esto se puede aliviar implementando un sistema de plugins que divide su único binario en componentes más pequeños compilables independientemente.

Existen múltiples tecnologías disponibles para su desarrollo: lenguajes interpretados, WebAssembly, eBPF, comunicación inter-proceso o cargado dinámico. Sin embargo, gran parte de ellas deben descartarse por no cumplir los estándares de eficiencia de Tremor. Entre las alternativas restantes, se escoge cargado dinámico por ser la más usable y popular.

El cargado dinámico es imposible con tipos y funciones declarados con Rust puro, ya que su Interfaz Binaria de Aplicación (ABI) no es estable. Será necesario convertir los tipos al ABI de C, que sí es estable, y viceversa. Para facilitar el proceso, se pueden aprovechar librerías existentes y herramientas del lenguaje como macros procedurales.

Dado que el cargado dinámico en Rust es un ecosistema muy nuevo, es necesario contribuir en código abierto a gran cantidad de sus dependencias para implementar la funcionalidad necesaria para un sistema de plugins.

La complejidad del proyecto incrementó significativamente respecto al plan original, por lo que aunque funcional, no alcanza alguno de los objetivos iniciales, principalmente relacionados con el rendimiento. Sin embargo, sirve como una buena base para futuras versiones de Tremor que sí que lo incluyan en producción, y continuará evolucionando con el programa.

ABSTRACT

Decision-making in many modern companies like Wayfair is based on the collection and analysis of data within their systems. To do this efficiently on a massive scale, it is essential to use high-performance tools such as Tremor, written with Rust, asynchronous programming, and SIMD.

As Tremor evolves, its compilation times grow and, consequently, its development experience deteriorates. This can be alleviated by implementing a plugin system that divides its single binary into smaller, independently-compilable components.

There are multiple available technologies for its development: interpreted languages, WebAssembly, eBPF, inter-process communication, or dynamic loading. However, many of them must be discarded for not meeting Tremor's efficiency standards. Among the remaining alternatives, dynamic loading is chosen for being the most usable and popular one.

Dynamic loading is impossible with types and functions declared with pure Rust, because its Application Binary Interface (ABI) is not stable. It will be necessary to convert its types to C's ABI, which is stable, and vice-versa. To facilitate the process, existing libraries and language tools such as procedural macros can be used.

Since dynamic loading is a very new ecosystem in Rust, it is necessary to contribute in open source to a great amount of dependencies in order to implement the functionality necessary for a plugin system.

The complexity of the project increased significantly compared to the original plan, so even if functional, it does not attain some of the initial objectives, mainly related to performance. However, it serves as a great base for future versions of tremor that do include it in production, and it will continue to evolve with the program.

Índice

Lista de Figuras

Capítulo 1

Introducción

1.1. Contexto

Este proyecto ha sido organizado y financiado por *Wayfair*, una empresa estadounidense de comercio digital de muebles y artículos del hogar [1]. Ingresó 13.700 millones de dólares en 2021 [2] y actualmente ofrece 14 millones de ítems de más de 11.000 proveedores globales [3].

Se trata de una entidad de gran tamaño, lo que se refleja en su cantidad disponible de datos o *logs* para describir el comportamiento de sus servicios. Estos son de vital importancia, dado que permiten encontrar errores en sus sistemas y ayudan a medir el rendimiento del negocio.

Los logs no suelen seguir un formato único ni necesariamente estructurado (es decir, texto plano en vez de JSON o XML). Por tanto, para poderlos usar es necesario algún tipo de procesado, que también puede incluir ignorar información redundante o transformar ciertos valores. Este paso se llevaba a cabo con el sistema de procesado de datos *Logstash* [4], entre otros.

A medida que crecía la empresa, estas herramientas de observabilidad dejaban de ser apropiadas. Cada día, Wayfair procesaba 100 *terabytes* de logs. Para visualizar mejor esta cantidad, si se escribieran en papel, la pila resultante sería de unos 5.400 kilómetros, o en otras palabras, 5 idas y vueltas a la Estación Espacial Internacional.

El coste mensual de esta infraestructura alcanzaba unos 40,000 dólares[1].

El problema principal residía en Logstash, escrito en Java y con un propósito de uso general en vez de especializado en rendimiento y escalabilidad. La solución de Wayfair fue crear la alternativa *Tremor*. Esta nueva herramienta está escrita con el lenguaje de programación Rust, que ofrece un rendimiento similar a C o C++ con una mayor seguridad. Adicionalmente, implementa técnicas para maximizar la eficiencia, como SIMD, asincronía con hilos o, en un futuro cercano, *clustering*. Una vez la primera versión de Tremor fue publicada, el coste de la infraestructura se redujo a 780 dólares mensuales, 50 veces menos que el valor original [5].

Tremor ha evolucionado desde entonces, y expandiendo sus posibilidades más allá de logs a algo más abstracto, los *eventos*. Ahora soporta una mayor cantidad de protocolos, formatos y software de donde recibir y enviar eventos, e incluso implementa su propio lenguaje de configuración y procesado. El proyecto se lanzó como código abierto y de forma independiente a Wayfair, y aunque el equipo de desarrollo sigue siendo principalmente suyo, cualquiera puede contribuir.

Posteriormente, Tremor se unió a la *Cloud Native Computing Foundation (CNCF)* [6], principalmente conocida por mantener y gestionar *Kubernetes* [7]. Asimismo, CNCF es parte de la organización *Linux Foundation (LFX)* [8], que además del famoso kernel también ayuda a todo tipo de proyectos de código abierto, como Node.js o Let's Encrypt [9].

Formalmente, el trabajo se ha llevado a cabo gracias a la iniciativa *LFX Mentorship*, que promueve el aprendizaje de desarrolladores de código abierto, proporcionando una plataforma transparente y facilitando un sistema de pagos [10]. En ella, proyectos de código abierto especifican una tarea concreta a realizar, proporcionando a cambio un mentor que le guíe durante el proceso y una ayuda monetaria.

El título de este proyecto en la plataforma de LFX es "CNCF – Tremor: Add plugin support for tremor (PDK)", es decir, desarrollar un sistema de plugins para Tremor. El plazo original era de tres meses comenzando en agosto, pero se acabó alargando a unos once meses. Disponía de tres mentores — los desarrolladores principales de

[1] Las cantidades mencionadas en esta sección son aproximadas, para dar una representación del tamaño de datos procesados y su coste sin comprometer la privacidad de Wayfair.

Tremor —, que me ayudaron a entender el funcionamiento interno del programa y dieron consejo cuando era necesario.

1.2. Objetivo

La tarea a llevar a cabo es la implementación de un sistema de plugins, también denominado *Plugin Development Kit (PDK)*, para la base de código ya existente de Tremor. El programa se dividirá en dos partes independientes: la *runtime* y los *plugins*. Los plugins son componentes que implementan cualquier tipo de funcionalidad, y la runtime es capaz de cargarlos y usarlos dinámicamente en el programa. Una comparación de más alto nivel se podría dar en un móvil: el hecho de que el sistema operativo (runtime) pueda instalar cualquier aplicación (plugin) lo hace mucho más flexible que un dispositivo que únicamente tuviera una serie de aplicaciones predefinidas.

En el caso de Tremor, un plugin podría dar soporte para recibir eventos de Apache Kafka o para enviarlos a Postgres, por ejemplo. La runtime debería ser capaz de cargar y usar esa funcionalidad mientras se está ejecutando el programa, en vez de a tiempo de compilación. Dividir el binario de Tremor en varios componentes más pequeños compilables independientemente implican una reducción en los tiempos de compilación de cada uno de ellos, que era el objetivo principal del proyecto.

Existen varias tecnologías disponibles para este sistema de plugins, como *WebAssembly*, *eBPF* o comunicación inter-proceso. Se investigarán las más importantes antes de implementarlo, pero el equipo de Tremor pensaba usar *cargado dinámico* desde el principio. El concepto es simple: tanto la runtime como los plugins son binarios nativos (es decir, código máquina). Para inicializar un plugin, la runtime lo cargará en su propia memoria RAM reservada. Posteriormente, mediante una interfaz establecida, denominada *Interfaz Binaria de Aplicación (ABI)*, es posible acceder a funciones y recursos en los plugins.

El *cargado dinámico* es un método especialmente popular en el lenguaje de programación C, dado que su interfaz de comunicación es relativamente sencilla y está muy bien definida en su estándar. Asimismo, se trata de una solución

especialmente eficiente, dado que la comunicación es binaria y directa. Las características de Tremor implican que el método a usar debería tener un alto rendimiento, así que el cargado dinámico es una de las mejores opciones disponibles.

Sin embargo, lenguajes modernos como C++ o Rust no especifican estrictamente un ABI, dado que implementan características más complejas como excepciones o tipos genéricos. En el caso concreto de Rust, si se compila la runtime y los plugins separadamente, no existe ninguna garantía de que la representación binaria de los datos o de que la convención de llamada a funciones — entre otros — sea la misma.

Por tanto, el cargado dinámico es imposible puramente con Rust. Se debe recurrir a otro ABI que sí sea estable, como el de C. Todas las funciones y tipos involucrados en la comunicación entre runtime y plugins deberán declararse siguiendo el estándar de C. Los tipos seguirán unas reglas que definen cómo se representan en memoria y las funciones tendrán que seguir una convención específica.

El problema principal con Tremor es que, al estar escrito puramente con Rust, todas sus funciones y tipos internos también se declaran con este lenguaje. Este proyecto tendrá que desarrollar un método para transformar tipos complejos de Rust a C y viceversa para poder interactuar con plugins.

1.3. Motivación

1.3.1. Tiempos de compilación

Uno de los problemas más importantes en Tremor es su tiempo de compilación. En mi portátil Dell Vostro 5481 con un Intel i5-8265U, 16GB de RAM a 2667 MHz, un SSD de 256GB y Arch Linux con el kernel 5.18 de 64 bits, compilar el binario `tremor` desde cero requiere más de 7 minutos en modo *debug*, y sobre 13 en modo *release* (con optimizaciones del compilador). Incluso en el caso de cambios incrementales (una vez las dependencias ya han sido compiladas), hay que esperar unos 10 segundos.

Puede que estas cifras no sean tan preocupantes en comparación con software de un tamaño mucho mayor. Pero debido a la naturaleza del programa, el problema solo empeorará con el tiempo. Tremor debe tener soporte para un gran número de protocolos (e.g., TCP o UDP), software (e.g., PostgreSQL o ElasticSearch) y códecs (e.g., JSON o YAML). El número de dependencias continuará incrementando hasta resultar en una pésima experiencia de desarrollo e imposibilitar la creación de nuevas prestaciones.

Los problemas relacionados con tiempos de compilación excesivamente largos no se limitan a Tremor. Es uno de las mayores críticas que recibe Rust: un 61 % de sus usuarios declaran que aún se necesita trabajo para mejorar la situación [11].

1.3.2. Modularidad y flexibilidad

Otra ventaja que provee un sistema de plugins es modularidad; ser capaz de tratar la runtime y los plugins de forma separada suele resultar en una arquitectura más limpia [12]. Hace posible el desacoplamiento del ejecutable y sus componentes; algunas dependencias tienen un ciclo de versionado más rápido que otras. Generalmente, es más conveniente desarrollar, realizar tests o actualizar únicamente un plugin, en lugar del programa por completo.

1.3.3. Aprender de otros

Existen proyectos maduros con características similares a las de Tremor, como *NGINX* [13] o *Apache HTTP Server* [14], que llevan beneficiándose de un sistema de plugins desde hace mucho. Informan mejorías en flexibilidad, extensibilidad y facilidad de desarrollo [15][16]. Aunque las desventajas también mencionen un pequeño impacto en el rendimiento y la posibilidad de caer en un “infierno de dependencias” (relacionado con plugins que dependen de otros), sigue siendo una buena idea al menos considerarlo para Tremor.

1.4. Metodología

1.4.1. Organización

El proyecto ha tenido una duración de unos once meses. La propuesta a Tremor se realizó de abril a mayo de 2021, pero no se comenzó el desarrollo hasta aceptarse en agosto del mismo año, y se terminó en junio de 2022. Su realización ha sido completamente remota y con horarios muy flexibles. Se usó el servidor de Discord de Tremor [17] como plataforma principal de comunicación, tanto por texto como por videollamada. Se programó una llamada por semana, en la que explicaba mi progreso y recibía ayuda en caso de que me hubiera quedado atascado o necesitara alguna opinión adicional.

Disponía de tres mentores, que me guiaban en el proceso de desarrollo: Darach Ennis (*Principal Engineer and Director of Tremor Project*), Matthias Wahl (*Staff Engineer*) y Heinz N. Gies (*Senior Staff Engineer*), todos empleados por Wayfair. En el plano académico, Francisco Javier Fabra Caro (*Doctor Ingeniero en Informática*) de la Universidad de Zaragoza fue el director del proyecto.

La organización de forma más estructurada para las tareas que tenía pendientes, en las que estaba trabajando en ese momento, y las que ya había realizado, se basó principalmente en un Kanban en GitHub.

1.4.2. Desarrollo

Para reducir el coste de desarrollo y asegurarse de que el proceso sea completamente seguro (en memoria y concurrencia), el sistema de plugins aprovecha librerías existentes en Rust y herramientas como *macros procedurales*. Los macros procedurales son mucho más potentes que los convencionales (*declarativos*), puesto que directamente consumen y generan sintaxis de Rust a tiempo de compilación para implementar cualquier tipo de código trivial o repetitivo.

El sistema de compilación usado es la solución oficial de Rust, Cargo, que también

incluye un *formatter*, *linter*, y extensiones instalables creadas por la comunidad. Adicionalmente, existe una gran cantidad de tests y *benchmarks* que se han de tener en cuenta para mantener el *Code Coverage* (la cantidad de código cubierta por los tests) y el rendimiento.

1.4.3. Recursos públicos

Este trabajo está disponible públicamente al completo. Además, a medida que he investigado e implementado el sistema de plugins, he ido escribiendo todo en mi blog personal, *NullDeref* [18]. Dispone de una serie con un total de seis[2] artículos, con más detalles sobre la implementación final. La organización también difiere considerablemente, puesto que los artículos se escribieron a medida que se realizaba el proyecto, resultando en una estructura menos rigurosa. Esto toma un formato de tesis, mientras que el blog cuenta la historia cronológicamente y sirve mejor como un tutorial para alguien que quiera implementar su propio sistema de plugins.

- El repositorio de GitHub para el binario de Tremor:
  `https://github.com/tremor-rs/tremor-runtime`
- Mi *fork*, con ramas adicionales usadas durante el desarrollo:
  `https://github.com/marioortizmanero/tremor-runtime`
- Mi repositorio con experimentos antes de implementar la versión definitiva:
  `https://github.com/marioortizmanero/pdk-experiments`
- La serie de artículos en mi blog personal:
  `https://nullderef.com/series/rust-plugins/`.
- Página oficial de la iniciativa:
  `https://mentorship.lfx.linuxfoundation.org/project/b90f7174-fc53-40bc-b9e2-9905f88c38ff`
- *Tracking issue* en GitHub:
  `https://github.com/tremor-rs/tremor-runtime/issues/791`
- RFC en la documentación de Tremor:
  `https://www.tremor.rs/rfc/accepted/plugin-development-kit/`

[2] El último artículo será publicado poco después de la entrega de esta memoria.

- Kanban con las tareas para el sistema de plugins: `https://github.com/marioortizmanero/tremor-runtime/projects/1`

Se recomienda también consultar el Anexo E, que lista todas las contribuciones de código abierto realizadas durante este proyecto. El Anexo G especifica cada fase de desarrollo junto a sus horas dedicadas y enlaces relacionados.

Capítulo 2

Rust y Tremor

2.1. ¿Qué es Rust?

Dado que Rust es un lenguaje de programación que tan solo anunció su primera versión en 2015, aún no es conocido por muchos desarrolladores. Esta memoria no requiere conocimientos previos sobre Rust. Sin embargo, la implementación en sí y otras partes más avanzadas, como el Anexo C o el Anexo D, asumen una familiaridad con el lenguaje más extensiva. Para esos casos, se recomienda consultar el Anexo A, que entra en mayor detalle sobre el lenguaje.

Rust es un lenguaje de programación de sistemas compilado y de propósito general. Su objetivo es maximizar rendimiento y seguridad, tanto en memoria como en concurrencia y sin necesidad de un recolector de basura. Las garantías de seguridad son comprobadas en tiempo de compilación gracias a un modelo estricto de programación y a un sistema fuertemente y estáticamente tipado. No permite punteros nulos, referencias colgantes, ni condiciones de carrera — aunque sí fugas de memoria.

Proporciona control a bajo nivel, manteniendo una productividad cercana a lenguajes de alto nivel. Dispone de programación funcional, genéricos, inferencia de tipos y macros, entre otros. También tiene soporte integrado para la programación asíncrona, un modelo de concurrencia que puede ejecutar eficientemente una gran cantidad de tareas de entrada y salida. No obstante, es posible ignorar explícitamente el modelo de memoria y concurrencia para casos

avanzados en los que se necesite control completo.

Los programas en Rust se diseñan basándose en la composición, en lugar del polimorfismo convencional de Python o C++. Se puede conseguir la misma flexibilidad mediante tipos estructurados y un mecanismo llamado *traits* — similar a las interfaces de Java, pero más potentes.

El ecosistema básico de Rust es altamente cohesivo: incluye el sistema de compilado y administrador de paquetes *Cargo*, el *formatter* de código *Rustfmt* y el *linter Clippy*. La instalación de estas herramientas se suele gestionar con el programa *rustup*.

2.2. ¿Qué es Tremor?

Tremor es un *Sistema de Procesado de Eventos*, que consiste en "el monitorizado y análisis (procesado) de flujos de información (datos) sobre cosas que pasan (eventos)" [20]. Tremor fue creado como una alternativa de alto rendimiento a herramientas como *Logstash* o *Telegraf*, pero ha evolucionado para soportar casos de uso más complejos. Al contrario que esos programas, Tremor también tiene soporte para *agregación* y *rollups*, e incluye un lenguaje *ad hoc* para *Extract, Transform, and Load* (ETL).

Para más información sobre Tremor se puede consultar *Tremor Getting Started*, que introduce sus conceptos más básicos y sus posibles usos — o cuándo *no* usarlo, en *Tremor Constraints and Limitations* [23]. *Tremor Recipes* [24] lista un total de 32 ejemplos de cómo configurar y emplear el software. El Anexo B entra en más detalles sobre su implementación interna.

Robins [25] y Cugola y Margara [26] introducen en detalle los dos campos contenidos en Procesado de Eventos: *Procesado de Eventos Complejos* y *Procesado de Flujos de Eventos*[1], ambos relevantes a Tremor. Dayarathna y Perera [27] y Tawsif y col. [28] resumen los avances más recientes en el campo, analizan su evolución, y clasifican sus subáreas. La mayoría de la información teórica tanto en esta sección

[1] *Complex Event Processing* y *Event Stream Processing* respectivamente, siguiendo la terminología anglosajona.

como en el Anexo B se extrae de estas fuentes.

2.2.1. Casos de uso

La Figura 2.1 ilustra uno de los casos de uso más básicos de Tremor:

1. Recibir *logs* (eventos) de aplicaciones en diferentes protocolos o formatos. Es posible que esta heterogeneidad se deba a que algunas aplicaciones son legadas y no se puedan reducir a un único protocolo o formato, o que esta tarea es demasiado compleja como para gestionarse a nivel de aplicación.
2. Filtrar los eventos redundantes, añadir campos nuevos o eliminar aquellos innecesarios y transformar todo a un mismo formato. El uso de una herramienta ineficiente o *ad hoc* por la empresa podría ser inviable dada una cantidad de datos suficientemente grande o demasiados protocolos y formatos como para implementarlos todos.
3. Enviar todos los logs estructurados a una base de datos para analizarlos posteriormente.

Sin embargo, este caso subestima el potencial de Tremor. La entrada y salida del sistema se pueden abstraer más, por ejemplo implementando un chatbot que reproduce música. Este podría tomar mensajes de Discord como su entrada, y enviar comandos con el API de Spotify como salida.

2.2.2. Conceptos básicos

Tremor se basa en los términos de *onramps* o *sources* y *offramps* o *sinks*:

- Una *onramp* especifica cómo Tremor se conecta con el mundo exterior (o una *pipeline*) para **recibir** de sistemas externos. Por ejemplo TCP, periódicamente o PostgreSQL [29].
- Una *offramp* especifica cómo Tremor se conecta con el mundo exterior (o una *pipeline*) para **enviar** a sistemas externos. Por ejemplo, *stdout*, Kafka o ElasticSearch [30].

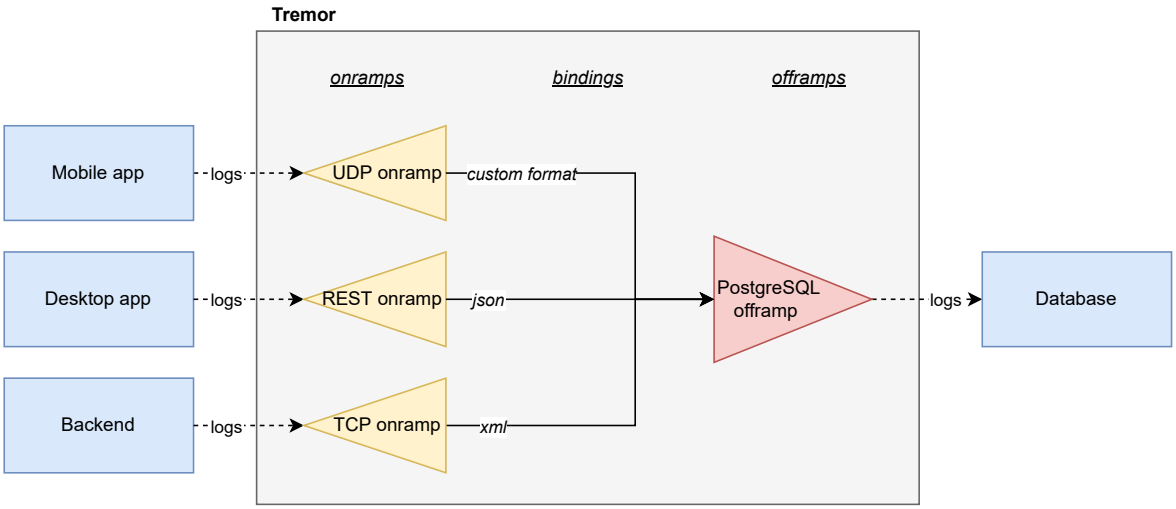

Figura 2.1: Ejemplo de uso básico de Tremor

– Una *pipeline* es una lista de operaciones (transformación, agregación, eliminación, etc) a través de la cual se pueden encaminar los eventos [31].

Estos *onramps* u *offramps* suelen contener una cantidad de información que es demasiado grande como para guardarla y debería tratarse en tiempo real. Su procesado se basa en las siguientes operaciones:

– *Filtros*: descarte de eventos completos a partir de reglas configuradas, con el objetivo de eliminar información de la *pipeline* que no se considera relevante.

– *Transformaciones*: conversión de los datos de un formato a otro, como por ejemplo incrementar un campo con un contador, reemplazar valores, o reorganizar su estructura.

– *Matching*: búsqueda de partes de eventos que siguen un patrón en específico (por ejemplo, un campo `"id"` con un valor númerico) para transformarlo o descartarlo.

– *Agregación* o *rollups*: recolección de múltiples eventos para producir otros nuevos (e.g., la media o máximo de un campo), de forma que la información útil se reduzca en tamaño.

Finalmente, otros términos misceláneos sobre Tremor:

– *Códec*: describen cómo decodificar los datos del flujo y como volverlos a

codificar. Por ejemplo, si los eventos de entrada usan JSON, tendrá que especificarse ese códec para que Tremor lo pueda entender.

– *Preprocesador* o *postprocesador*: operadores sobre flujos de datos brutos. Un preprocesador aplicará esta operación antes del códec y un postprocesador después. Por ejemplo, `base64` codifica o decodifica la información con ese protocolo.

– *Artefacto*: término genérico para hablar de *sinks*, *sources*, códecs, preprocesadores y postprocesadores.

2.2.3. Conectores

Es posible que algunas *onramps* no solo quieran recibir de sistemas externos, sino también responderles directamente, actuando como una *offramp* y viceversa. Esto es especialmente útil para casos como REST y *websockets*, donde el protocolo da la posibilidad de responder a eventos, por ejemplo con un ACK, usando la misma conexión. En la versión 0.11 — la presente cuando me uní al proyecto — este problema se solucionaba con el concepto de *linked transports*.

El término *conector* se introdujo en mayo de 2022 con la versión 0.12. Solucionan el problema desde el inicio, abstrayendo tanto los *onramps* como los *offramps* bajo el mismo concepto, incluyendo los *linked transports*. Dado que estos ya estaban siendo desarrollados mientras 0.11 era la última versión, el sistema de plugins se enfocó a los conectores desde el principio, en lugar de *onramps* u *offramps*, que ahora han quedado en desuso.

Capítulo 3

Contexto

La propuesta para el sistema de plugins asumía que se iba a implementar con un método que cubriré posteriormente, denominado *cargado dinámico*. Esto se debe a razones de rendimiento, pero el método también incluye otros problemas importantes, principalmente relacionados con seguridad. Por ello, es una buena idea considerar las alternativas existentes para el PDK, en caso de que hubiera alguno con la misma eficiencia pero menos vulnerabilidades.

3.1. Requisitos

Los requerimientos principales para el proyecto son los siguientes:

- Debe ser posible añadir y quitar plugins tanto en el inicio del programa como durante su ejecución.
- Disponibilidad y madurez de la tecnología en el ecosistema de Rust. Debería ser viable en el largo plazo y no quedar obsoleto.
- Soporte multi-plataforma. Tremor actualmente da soporte a Windows, MacOS y Linux, así que debería estar disponible para al menos todas ellas.
- No debe tener un impacto excesivo en el rendimiento. El equipo de Tremor está dispuesto a sufrir una degradación de rendimiento siempre y cuando esta sea menor. La definición de 'menor' es flexible; aproximadamente

un máximo de 10-20% de reducción en eficiencia para poderse incluir en producción. Copiar y serializar eventos, por ejemplo, implicaría la rotura de este requerimiento.

También existe una lista de tareas opcionales que se pueden tener en cuenta al finalizar el proyecto:

- Maximizar la seguridad en lo posible, como se especifica en la Sección 3.2.
- Debería ser retro-compatible e implementar algún tipo de versionado para evitar roturas inesperadas, como indica la Sección 3.3.
- Minimizar el esfuerzo necesario de reescribir los conectores para el nuevo sistema de plugins.
- Gestionar correctamente plugins con identificadores conflictivos.
- Desarrollar un sistema de tests que compruebe las invariantes de los plugins y proporcione mecanismos de testing para desarrolladores de plugins.
- Considerar la generación de documentación para plugins y herramientas para mejorar la experiencia de desarrollo.

3.2. Seguridad

3.2.1. Código unsafe

Muchas de las tecnologías que se pueden aplicar para un sistema de plugins usan código `unsafe`. El código `unsafe` consiste en una sección que ignora el modelo de memoria y concurrencia de Rust, permitiendo por ejemplo el uso de punteros nulos o referencias colgantes. Se suele usar para maximizar el control y rendimiento del programa.

```
let p = std::ptr::null::<i32>();
unsafe {
    // Imposible sin un bloque `unsafe`:
    println!("Mi número: {}", *p); // Segmentation fault!
}
```

Técnicamente, esto no es necesariamente un problema si la implementación está autocontenida y su corrección es auditada exhaustivamente. Pero al perderse garantías importantes proporcionadas por Rust, incrementa el coste de mantenimiento de la librería.

Asegurarse de que la implementación es segura implica una cantidad considerablemente mayor de trabajo, aun cuando existen herramientas como *MIRI* [32] — que sería integrado en Tremor en caso de tenerse que recurrir a `unsafe`.

3.2.2. Resiliencia a errores

Rust no protege a sus usuarios de *leaks* de memoria. De hecho, es tan sencillo como llamar a la función `mem::forget`. Si un plugin tuviera un *leak*, el proceso entero también se vería afectado; el rendimiento de Tremor se degradaría con plugins no desarrollados incorrectamente. Algo similar podría suceder en caso de que un plugin abortase o sufriese de un pánico, lo que terminaría la ejecución del programa por completo.

Idealmente, Tremor debería poder detectar plugins que no rinden óptimamente y pararlos antes de que sea demasiado tarde. La runtime debería poder continuar corriendo aun cuando falle un plugin, posiblemente avisando al usuario o reiniciándolo para seguir funcionando.

3.2.3. Ejecución de código remota a través de plugins

Uno de los casos más notorios se dio con Internet Explorer, que usaba COM y ActiveX, los cuales no disponían de una *sandbox*. Dicho mecanismo aisla por completo parte del programa, de forma que no pueda leer ni escribir a memoria externa (prohibiendo el acceso a información que no es suya), ni a recursos del sistema (como ficheros). Por tanto, extensiones maliciosas para el navegador podían ejecutar código arbitrario en la máquina en la que estuviera instalado [33]. Este problema puede ser menos grave si solo se instalan extensiones de confianza con firmas digitales, pero sigue siendo un vector de ataque importante.

Se podría aplicar lo mismo a Tremor. El usuario del producto — aquellos que añadan plugins a su configuración —, es un desarrollador, que debería ser más consciente sobre lo que incluye en sus proyectos. Sin embargo, en la práctica esto no es cierto.

Podría compararse con cómo funcionan los administradores de paquetes como *npm* [34]. Su infraestructura se suele basar por completo en cadenas de confianza; no hay nadie que te impida crear un paquete malicioso para ejecutar código remoto o robar credenciales [35][36]. Los plugins son como dependencias en este caso; tienen acceso completo a la máquina donde se ejecutan, y por tanto no deberían ser de confianza por defecto.

Una mejora respecto a Node y npm sería algo como *Deno* [37], una runtime segura por defecto. Esto es posible gracias a *sandboxing*, y requiere que el desarrollador configure manualmente, por ejemplo, el acceso al sistema de ficheros o a la red. No es una solución infalible porque puede que los desarrolladores acaben activando los permisos que necesitan sin pensarlo, pero es un mecanismo similar a `unsafe`: al menos te hace consciente de que estás en terreno pantanoso y facilita la búsqueda de vulnerabilidades.

Se podría discutir que, realísticamente, el programa va a ejecutarse la mayoría de los casos en una máquina virtual o un contenedor, donde este problema no es tan peligroso. Pero, ¿debería la seguridad del usuario recaer en el hecho de que el kernel está isolado? Por no mencionar que un contenedor afecta mucho más al rendimiento que algunos métodos de *sandboxing*. Aunque el sistema por completo estuviera isolado, seguiría habiendo una posibilidad de *leaks* internos: el plugin de Postgres tiene acceso a todo lo que esté usando el plugin de Apache Kafka, que posiblemente tenga *logs* sensitivos.

3.3. Retro-compatibilidad

Será necesario incluir algún tipo de gestión de versiones en el proyecto. Es probable que la interfaz de Tremor cambie con frecuencia, lo que romperá plugins basados en versiones previas. Si un plugin recibe una estructura de la runtime, pero esta estructura perdiese uno de sus campos en una nueva versión, se estará invocando

comportamiento no definido por intentar acceder a un campo ahora inexistente.

3.3.1. Posibles soluciones

La idea más sencilla para arreglar problemas con retrocompatibilidad es serializar y deserializar los datos con un protocolo flexible, en vez de usando su representación binaria directamente. Si se usara un protocolo como JSON para comunicarse entre la runtime y los plugins, añadir un campo no rompería nada, y eliminar uno puede ocurrir mediante un proceso de deprecación. Por desgracia, esto implicaría una degradación en el rendimiento que posiblemente no interese en la aplicación. Otros arreglos más elaborados para representaciones binarias incluyen [38]:

- Reservar espacio en la estructura para uso futuro.
- Hacer la estructura un tipo opaco, es decir, que sólo se puede acceder a sus campos con llamadas a funciones, en lugar de directamente en memoria.
- Declarar la estructura con un puntero con sus datos de la “segunda versión” (lo cual sería opaco en la “primera versión”).

3.3.2. Control de versiones

Hay casos donde un error inevitable. Es posible que Tremor quiera reescribir parte de su interfaz o finalmente eliminar una funcionalidad deprecada sin tener que preocuparse por romper todos los plugins desarrollados previamente.

Para ello, los plugins deben incluir metadatos sobre las diferentes versiones de rustc o de la interfaz para la que fue desarrollado. Después, cuando sean cargados por Tremor, se podrá comprobar su compatibilidad, en vez de romperse de formas misteriosas.

Capítulo 4

Diseño

4.1. Arquitectura

Para el sistema de plugins final, la estructura de Tremor debería ser la ilustrada en la Figura 4.1. Notar que una *crate* es el nombre que recibe un binario o librería en Rust, tanto local como externo (en cuyo caso sería una dependencia).

- La *crate* `runtime`, que carga los plugins.
- Las *crates* `plugin`, con las implementaciones de los componentes del sistema.
- La *crate* `common`, con la interfaz compartida entre la runtime y los plugins. Por tanto, ambos tipos de *crate* dependen de `common`.

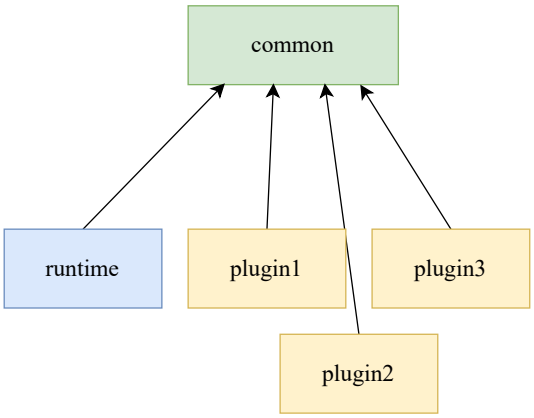

Figura 4.1: La estructura ideal para un sistema de plugins.

Esta estructura es esencial para el objetivo principal: mejorar los tiempos de compilación. Existen dos maneras de entender los tiempos de compilación: para el desarrollo de la runtime y para el desarrollo de los plugins.

En ambos casos, se quiere compilar únicamente *uno* de los componentes. En caso de desarrollar un plugin, no debería hacer falta recompilar también la runtime, porque no está siendo modificada. Y si se está trabajando sobre la runtime, no debería recompilarse la funcionalidad de los plugins.

El problema reside en que, inicialmente, únicamente existe una *crate* con todo: `runtime`, `plugin`s, y `common`. El primer paso debería ser el que muestra la Figura 4.2: separar los plugins del mono-binario. La funcionalidad se encuentra en binarios diferentes, así que la runtime tendrá un tiempo de compilación significativamente menor. Desarrollar un plugin también será menos costoso, ya que no hará falta compilar los demás.

Para un tiempo de compilación óptimo también es necesario un segundo paso. La interfaz del PDK sigue encontrándose en la misma *crate* que la runtime, así que un plugin tendrá que compilar también la runtime, aun cuando no es necesario. Esto no será una mejora tan grande sobre los tiempos de compilación como en el primer paso, dado que compilar la runtime es mucho menos costoso que compilar todos los plugins.

Este segundo paso se puede omitir durante el inicio del desarrollo, ya que la interfaz está muy fuertemente relacionada con la runtime. Los tipos usados en la interfaz tendrán que moverse a una *crate* distinta, pero todos estos tipos provienen de la runtime, así que habrá que modularizar una gran parte del código.

Cuantos menos cambios se produzcan al principio del proyecto, mejor. Habrán menos conflictos y resultará más fácil y seguro revisar el ćodigo en pequeñas en iteraciones, en vez de una única revisión grande.

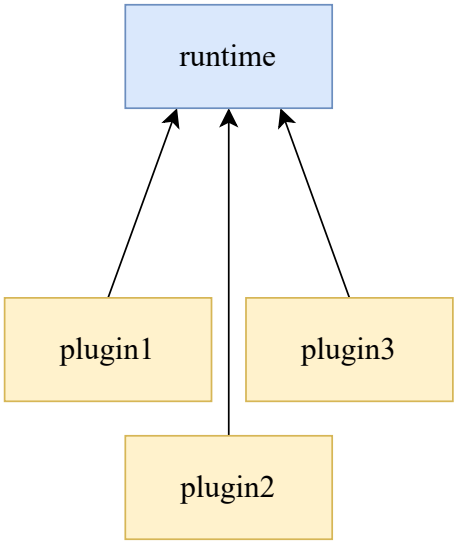

Figura 4.2: La estructura inicial del sistema de plugins para un desarrollo más rápido.

4.2. Plan de desarrollo

A partir de la sección anterior, se puede elaborar un plan aproximado para el sistema de plugins basado en iteraciones (o pull requests):

1. **Definir una nueva interfaz y usarla internamente**: el sistema de plugins debería ser lo más minimalista posible. La interfaz principal puede convertirse de forma que soporte plugins, pero se debería mantener todo en un mismo binario por simplicidad. La parte de cargado de plugins se puede dejar como una prueba de concepto por el momento y se pueden incluir algunos plugins externos para demostrar su funcionamiento.

   Esta iteración se podrá fusionar con la rama principal, ya que el programa seguirá funcionando de la misma forma, simplemente con una interfaz distinta para los conectores internamente.

2. **Hacer los plugins externos**: dado que los plugins desarrollados usan la nueva interfaz, este paso debería ser sencillo. Únicamente implicará una gran cantidad de cambios, ya que implica reorganizar el repositorio con *crates* nuevas, arreglar el sistema de compilación y similares.

3. **Separar la runtime de la interfaz**: como se ha explicado, esta parte es menos importante pero puede implicar un gran número de cambios. Por tanto, solamente al final se separará la interfaz a una crate `common` nueva, para

una última reducción de los tiempos de compilación.

4. **Otras mejoras para el despliegue**: últimos cambios antes de incluir el sistema de plugins en la nueva versión, incluyendo su documentación o la evaluación de los resultados finales.

4.3. Tecnologías a considerar

Esta sección describe las tecnologías que se han considerado más viables como base para el PDK. Algunas de ellas no cumplirán los requerimientos mencionados al principio del capítulo, pero es necesario aprender sobre ellas primero antes de escribir ninguna línea de código.

4.3.1. Lenguajes interpretados

Gran cantidad de proyectos usan lenguajes interpretados para extender su funcionalidad a tiempo de ejecución, con Python, Ruby, Perl, Bash, o JavaScript, entre otros. Particularmente, el editor de texto Vim creó su propio lenguaje, *VimScript* [39], para poderlo personalizar por completo. Ahora NeoVim, un fork más moderno, está esforzándose por tener Lua como lenguaje de primera clase para su configuración [40]. Incluso Tremor tiene su propio lenguaje para configurarlo, Troy.

De todos los lenguajes disponibles, Lua sería una de las mejores opciones para este sistema de plugins en específico. Está hecho con *embedding* en mente: es simple de usar y únicamente ocupa alrededor de 220KB [41]. Algunas implementaciones del lenguaje, como LuaJIT, son extremadamente eficientes y pueden ser viables hasta en escenarios de rendimiento crítico [42]. Adicionalmente, las garantías de seguridad de Lua son más fuertes que otros lenguajes, dado que no requiere `unsafe` y por incluir una *sandbox* (aunque es "delicado y difícil de configurar correctamente") [43].

Rust dispone de librerías como `rlua` [44], con bindings para interoperar con Lua. `rlua` en particular parece enfocar su interfaz en ser idiomática y

segura, que es un punto positivo para una librería fuertemente relacionada con C. Desgraciadamente, parece estar semi-abandonada y fue reemplazada por `mlua` [45]. Por lo general, el ecosistema de Lua en Rust no parece lo suficientemente maduro para un proyecto como este; aún queda trabajo para mejorar su estabilidad.

También sería posible usar uno de los lenguajes interpretados creados específicamente para Rust: *Gluon* [46], *Rhai* [47] o *Rune* [48]. Usarlos posiblemente resulte en código más limpio y simple. Sin embargo, el ecosistema todavía está en su infancia y ninguna de las opciones son tan estables o seguras como lenguajes de programación de propósito general. Rhai, el más usado, anunció su versión v1.0 en julio de 2021 y no sobrepasa las 200.000 descargas, mientras que Lua fue creado en 1993 y es uno de los 20 lenguajes más famosos, según el *TIOBE Index* [49].

De cualquier manera, portar el código a este sistema de plugins sería un trabajo excesivamente laborioso. Todos los conectores tendrían que reescribirse por completo a un lenguaje distinto. Para un proyecto nuevo sería una alternativa interesante, pero ciertamente no lo es en el caso de Tremor.

4.3.2. WebAssembly

WebAssembly [50], también conocido como Wasm, es esencialmente un formato binario abierto y portable. A diferencia de binarios normales, el mismo ejecutable Wasm puede correr en cualquier plataforma, siempre y cuando exista una runtime que lo soporte. Comenzó como una alternativa a JavaScript exclusiva a la web, pero ha evolucionado con el tiempo y ahora es posible usarlo en el escritorio gracias a *WASI* [51].

Los objetivos de Wasm son maximizar la portabilidad y seguridad, sin un coste de rendimiento excesivo. Su diseño incluye una *sandbox* para lidiar con programas no fiables, como es el caso en sistemas de plugins, y apenas no requiere usar `unsafe`. Ya que puede ser compilado desde otros lenguajes como Rust o C, el código existente en Tremor podría ser reusado (lo cual era imposible con lenguajes interpretados).

Existen dos runtimes principales para Rust: *Wasmer* [52] y *Wasmtime* [53]. Ambas son implementaciones competitivas enfocadas a unos u otros casos de uso. Por lo general, Wasmer es más adecuado para embebirlo en programas nativos, mientras que Wasmtime se centra en programas individuales, aunque los dos se pueden usar apara ambos casos [54].

WebAssembly todavía es una tecnología relativamente nueva, así que algunas partes siguen bajo desarrollo continuo y necesitan mejoras, como en rendimiento. En comparación a JavaScript, Jangda y col. [55] muestra resultados mezclados al realizar pruebas de rendimiento. Depende principalmente del compilador[1] y del entorno que se esté usando, variando desde mejoras en velocidad de 1.67x en Chrome, a 11.71x con Firefox. Cuando se compara con código nativo, Denis [56] describe una varianza similar, donde Wasmer es 2.47x más lento y con Wasmtime es 3.28x. En resumen, aunque WebAssembly es una solución más eficiente que algunos lenguajes interpretados, sigue sin llegar al nivel de binarios nativos, y posiblemente no sea lo suficiente para este caso.

Esta tecnología es de las más adecuadas encontradas por el momento; su único problema es el rendimiento. Tras implementar algún sistema de plugins en miniatura, su usabilidad y seguridad era excelente. Si fuera posible transferir datos (tipos no triviales) entre la runtime y el plugin sin tener que copiarlos, sería definitivamente la mejor alternativa. La especificación de WebAssembly define únicamente enteros y decimales como sus tipos disponibles [57]. Existen algunas maneras de tratar tipos compuestos, como estructuras o enumeraciones:

– A través de la *Interface Types Proposal for WebAssembly* [58]. Esta define un formato binario para codificar y decodificar los nuevos tipos que define: tipos de números más especializados, caracteres individuales, listas, estructuras y enumeraciones. También especifica una lista de instrucciones para transformar los datos entre WebAssembly y el mundo exterior. Notar que esta propuesta no intenta definir una representación fija de, por ejemplo, una cadena de caracteres en Wasm; intenta permitir tipos de alto nivel agnósticos a su representación.

  Adicionalmente, las interfaces se pueden definir independientemente del

[1] Nos referimos también a la runtime como un *compilador*, dado que las implementaciones más eficientes y populares son compiladores *Just-In-Time* (JIT), que transforman y ejecutan partes del código fuente como código máquina.

lenguaje de programación que se esté usando, gracias al formato `witx` [59], como muestra la Figura 4.3.

El mayor problema de esta solución es que aún está en "Fase 1": aún necesita mucho trabajo y su especificación no es estable. Ninguna de las runtimes tienen soporte para esta propuesta aún [60][61]. Tras fallar al intentar usarlo, esta opción fue descartada.

– El método más popular actualmente, que es funcional pero imperfecto, con punteros y memoria compartida. El usuario debe construir y serializar el tipo compuesto y después guardarlo en la memoria reservada para Wasm, a la que la runtime puede acceder directamente con punteros. Esto es lo que otros sistemas de plugins como Feather o Veloren hacen [62][63], así que es garantizado que funciona.

  No sólo requiere esto un paso de serialización y otro de deserialización y escribir y leer todos los datos de una memoria, sino que también es una tarea ardua y complicada de hacer correctamente. A nivel de rendimiento, esto implicaría copiar los datos, así que no es algo que Tremor se pueda permitir.

– Otra opción que emplean programas como Zellij [64], que usa un ejecutable de Wasm en vez de usarlo como una librería importable. Para cargarlo, ejecuta el binario independiente y usa *stdin* y *stdout* para los flujos de datos. Desgraciadamente, esto también requiere serializar y copiar datos, y tiene que descartarse.

```
(use "errno.witx")

;;; Add two integers
(module $calculator
  (@interface func (export "add")
    (param $lh s32)
    (param $rh s32)
    (result $error $errno)
    (result $res s32)
  )
)
```

Figura 4.3: Ejemplo de interfaz definida con `witx`.

4.3.3. eBPF

eBPF es "una tecnología revolucionaria con orígenes en el kernel de Linux que puede ejecutar programas en *sandbox* en el kernel de un sistema operativo" [65]. Sin embargo, de forma similar a WebAssembly, su uso se ha extendido al espacio de usuario. eBPF define una lista de instrucciones que pueden ejecutarse por una máquina virtual, también como WebAssembly funciona.

Esta tecnología es prometedora, ya que a diferencia de WebAssembly, no es necesario serializar o deserializar los datos o escribirlos a una memoria intermedia. Ya que existe control completo sobre la máquina virtual, la runtime podría implementar una *sandbox* personalizada para comprobar las direcciones de memoria de donde se lee o escribe para asegurarse de que se encuentran en el rango permitiendo, siendo posible compartir una única memoria. La única penalización en el rendimiento sería interpretar las instrucciones en vez de ejecutar código nativo, pero técnicamente Tremor sí que podría usarlo.

El problema principal con eBPF es que su suporte es carente. La mayoría de sus usuarios usan C y lo muestra la poca cantidad de tutoriales, guías, artículos o incluso librerías disponibles para Rust. No es posible compilar Rust a instrucciones eBPF de forma oficial y la única runtime disponible es `rbpf` [66] y derivados como `solana_rbpf` [67], ya que este primero parece estar obsoleto. Además, supondría un esfuerzo mucho mayor que WebAssembly, ya que también requeriría implementar una *sandbox* personalizada.

4.3.4. Cargado dinámico

Esta es la manera más popular para implementar un sistema de plugins, al menos fuera de Rust. Una *Foreign Function Interface* (FFI) nos permite acceder directamente a recursos en objetos compilados separadamente, aun después de la fase de *linking*, mediante el cargado dinámico. Es una de las opciones más eficientes porque no impone casi ningún coste adicional tras el cargado de la librería dinámica, y no requiere crear procesos nuevos en el sistema operativo.

La *crate* principal para este método es `libloading` [68], aunque también

existen las menos conocidas `dlopen` [69] y `sharedlib` [70], con pequeñas diferencias [71]. Todas ellas requieren de uso extensivo de `unsafe`, son complicadas de usar correctamente [72][73], incluyendo sutiles disparidades entre sistemas operativos [74], y no disponen de un mecanismo similar a una *sandbox*. La única manera de mejorar la usabilidad será a través de los macros, herramientas y abstracciones facilitadas por ellas mismas y otras librerías.

Estabilidad del ABI

El problema principal con esta alternativa es que Rust carece de una *Interfaz Binaria de Aplicación (ABI)* estable. El ABI es una interfaz entre dos módulos binarios, en nuestro caso entre un ejecutable (runtime) y una librería dinámica (plugin). Este se encarga de definir, entre otros, la estructura que siguen los tipos en memoria y la convención usada para llamar funciones. Sin un ABI definido (y por tanto, sin *estabilidad*), sería imposible saber cómo acceder a los recursos de otros binarios.

En el comienzo este proyecto, gran cantidad de fuentes en la comunidad confundían cómo funciona este concepto en Rust, y lo explicaban de forma incorrecta [75][76][77][78]. Este popular malentendido también se dio en la propuesta del PDK y no nos dimos cuenta del verdadero significado hasta haber invertido numerosas horas. El equipo de Tremor — yo incluido — creía que el ABI de Rust es estable, siempre que los dos binarios se compilen con exactamente la misma versión de compilador. Sin embargo, esto es incorrecto por varias razones y fuentes como la referencia oficial no entran en suficiente detalle [79, Application Binary Interface (ABI)]. Debe recurrirse a otra sección, que menciona brevemente lo siguiente:

“La estructura de tipos en memoria puede cambiar con cada compilación. En vez de intentar documentar exactamente qué se hace, se documenta solo lo que se garantiza hoy” [79, Type Layout]

La asunción anterior incorrecta se basa en que, hasta el momento, Rust no ha implementado ninguna optimización que rompa el ABI entre ejecuciones de un compilador en la misma versión. Pero no existe absolutamente ninguna garantía de que esto sea así, y es un detalle de implementación del compilador del que

no debería confiarse. Es posible que este comportamiento sí que se rompa en el futuro [80], en cuyo caso el sistema de plugins tendría que reescribirse por completo.

Descubrir esto implicó un cambio de planes y un aumento muy notable en la complejidad del proyecto. Ahora tendría que recurrirse a una ABI que sí que tuviera garantías de estabilidad, y traducir entre la de Rust y esta para comunicarse entre runtime y plugins. La ABI más conocida es la del lenguaje de programación C, que se puede acceder desde Rust como indican las Figuras 4.4 y 4.5.

Herramientas disponibles

Todo este proyecto va a ser posible gracias a una *crate* de más alto nivel, `abi_stable` [81]. Esta usa `libloading` internamente y exporta una gran cantidad de macros y herramientas para facilitar el desarrollo. Incluye una copia de la librería estándar de Rust declarada con el ABI de C, con nombres generalmente precedidos por la letra `R`. Por tanto, en vez del vector `Vec<T>`, podremos usar `RVec<T>`; en caso contrario habría que recurrir a punteros (`*const T`) o tendríamos que reescribir los tipos desde cero nosotros. También da soporte para librerías externas muy conocidas en la comunidad, como `crossbeam` [82] o `serde_json` [83]. La Figura 4.6 demuestra parte de la simplificación respecto al código en la Figura 4.5.

Existen más alternativas o extensiones para el cargado dinámico que se tuvieron en cuenta, como *LCCC* [84], `safer_ffi` [85] o `cglue` [86], pero no son soluciones tan completas ni maduras, por lo que no se usarán en este proyecto. `abi_stable` se trata de una librería grande, con más de 50.000 líneas de código en Rust (como referencia, Tremor tiene unas 35.000 líneas), por lo que el aumento de complejidad deberá tenerse en cuenta en la decisión de la tecnología final a usar.

4.3.5. Comunicación Inter-Proceso

Otra opción popular para sistemas de plugins es la *Comunicación Inter-Proceso*, que divide el programa en procesos distintos de tipo cliente y servidor. El

```rust
pub struct Event {
    pub count: i32,
    pub name: &'static str,
}

pub fn transform(x: Event) -> i32 {
    println!("Received an event with count {}", x.count);
    x.count
}

pub static cached: Event = Event {
    count: 0,
    name: "my data"
};
```

Figura 4.4: Ejemplo de cómo sería un plugin escrito con Rust.

```rust
#[repr(C)]
pub struct Event {
    pub count: i32,
    pub name: *const std::os::raw::char_c,
}

pub extern "C" fn transform(x: Event) -> i32 {
    println!("Received an event with count {}", x.count);
    x.count
}

#[no_mangle]
pub static cached: Event = Event {
    count: 0,
    name: "my data".as_ptr() as _
};
```

Figura 4.5: El mismo plugin que la Figura 4.4, pero usando el ABI de C.

```rust
#[repr(C)]
#[derive(StableAbi)]
pub struct Event {
    pub count: i32,
    // We can use abi_stable's types!
    pub name: RStr<'static>,
}

#[sabi_extern_fn]
pub fn transform(x: Event) -> i32 {
    println!("Received an event with count {}", x.count);
    x.count
}
```

Figura 4.6: El mismo plugin que la Figura 4.4, pero con `abi_stable`. La declaración de la global `cached` se omite por simplicidad.

cliente actuaría como runtime y estaría conectado a múltiples servidores que proporcionan la funcionalidad. Se podría comparar con el *Language Server Protocol* [87], basado en JSON-RPC y usado por la mayoría de editores de texto para tener soporte especializado para cualquier lenguaje de programación.

Una ventaja común para todos los métodos de esta familia es que, de forma similar a WebAssembly, los plugins se podrán escribir en Rust, así que el código existente se podría reusar. Además, ya que el cliente y servidor se dividirían en múltiples procesos, serían más seguros por lo general; plugins defectuosos no afectarían a la runtime de Tremor.

Sockets

Son los que peor rendimiento tienen de acuerdo a la Figura 4.7 y la Figura 4.8, pero también son los más famosos, y consecuentemente, los más fáciles de usar. Los *sockets* son la misma tecnología usada en cualquier servidor para comunicarse con un cliente y viceversa, por lo que hay una cantidad enorme de implementaciones disponibles.

Usar *sockets* también requiere un paso de serialización, dado que los datos se envían en paquetes. Formatos como JSON son los más flexibles, pero otros como *Protocol Buffers* [89] son ligeros y tienen mejor rendimiento.

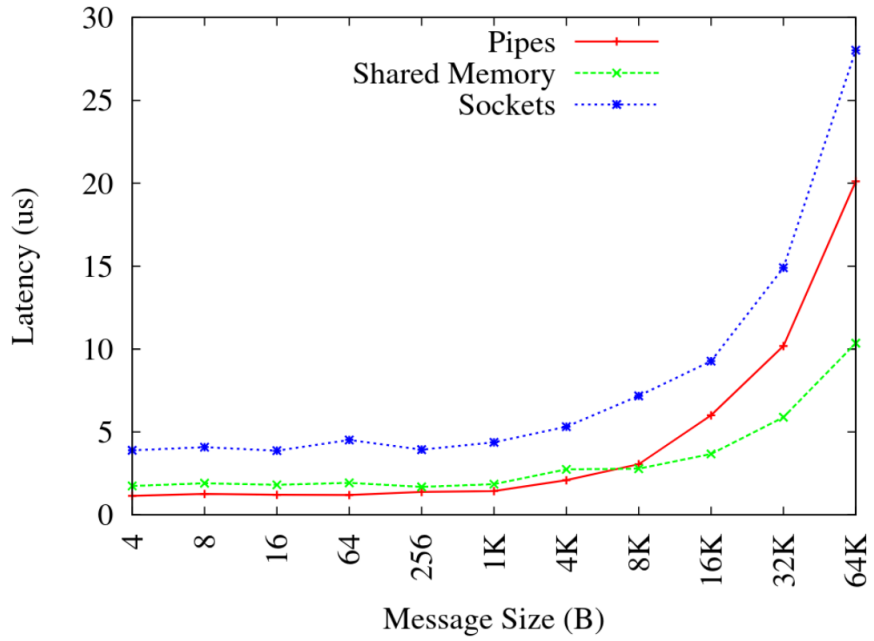

Figura 4.7: Latencia vs. Tamaño de Mensaje [88].

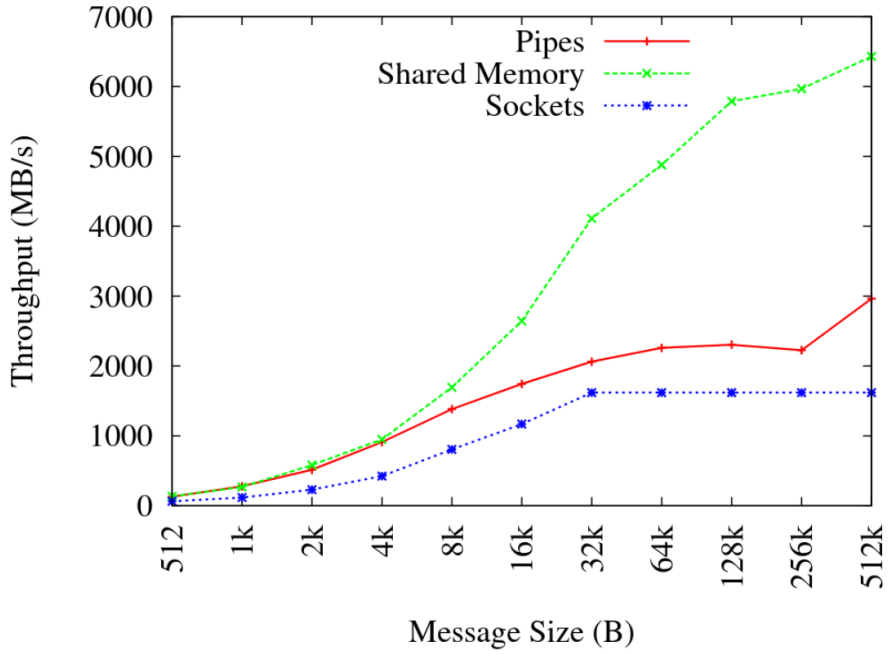

Figura 4.8: Rendimiento vs. Tamaño de Mensaje [88].

Pipes

Para un sistema de plugins, las *pipes* son muy similares a los *sockets*, con la única diferencia siendo que las *pipes* solo se pueden usar en una misma máquina. Con *sockets*, técnicamente podrías usar TCP o UDP y tener la runtime y los plugins en ordenadores distintos. Esto no es algo necesario para el caso de Tremor, y ya que las *pipes* ofrecen un mejor rendimiento, posiblemente sean una mejor opción por lo general.

Por ejemplo, el gestor de archivos nnn [**nnn**] usa este método: los plugins pueden leer de una FIFO (una *pipe* con nombre) para recibir las selecciones de archivos o directorios que realice el usuario e implementar su funcionalidad adicional.

La única desventaja es que no parecen haber librerías populares para la funcionalidad genérica de *pipes* (quizá `interprocess` [90] o `ipipe` [91]). Sin embargo, esto podría ser innecesario si se usaran las *pipes* de *stdin*, *stdout* o *stderr* implícitamente, ya que tienen soporte en la librería estándar al ejecutar comandos *shell* [92, Pipes].

Memoria compartida

Como el nombre indica, la memoria compartida consiste en inicializar un buffer del que se puede leer y escribir desde dos o más procesos al mismo tiempo para comunicarse. El API de memoria compartida se implementa a nivel del kernel, por lo que depende mucho del sistema operativo y quizá no sea tan portable como otras soluciones.

Tal y como indican las Figuras 4.7 y 4.8, es el método con mejor rendimiento, ya que no requiere copiar ni transformar datos entre procesos. El único coste adicional es el tener múltiples procesos, y la inicialización de las páginas compartidas en el sistema operativo, que se debe hacer únicamente al principio [93].

Desgraciadamente, el soporte para memoria compartida en Rust es casi inexistente. Las únicas *crates* disponibles son `shared_memory` [94] y `raw_sync` [95], que no superan las 150.000 descargas en total y usan gran

cantidad de `unsafe`. Esto probablemente tenga que ver con el hecho de que comparte los mismos problemas que cargado dinámico respecto a estabilidad de ABI (explicado en la sección 4.3.4). No parece ofrecer nada mejor que el cargado dinámico y por tanto se descarta como opción.

4.4. Sistemas de plugins de referencia

Otro punto de estudio importante es qué plugins ya hay existentes en Rust y cómo se han realizado:

- El mismo *Cargo* [96] o *mdBook* [97] implementan un sistema de extensiones a través de la línea de comandos. Añadir un subcomando nuevo es tan sencillo como crear un binario con un prefijo establecido (por ejemplo, `cargo-expand`). Si este binario está disponible en la variable de entorno `PATH` al ejecutar `cargo`, se podrá invocar al plugin con `cargo expand` también. Es una implementación especialmente simple con *pipes* e IPC, dado que usa *stdin* y *stdout* para comunicarse con la runtime.

- *Zellij* es un entorno de trabajo en el terminal con "un sistema de plugins que permite crear plugins en cualquier lenguaje que compile a WebAssembly". De forma similar al caso anterior, funciona un binario distinto para cada plugin y la runtime ejecuta el código en WebAssembly, comunicándose con *stdin* y *stdout*.

- *Xi* es un editor de texto moderno ahora abandonado. Usaba RPC con mensajes JSON para comunicarse con plugins en procesos distintos [100], método también usado en Visual Studio Code [101] o Eclipse [102].

- *Bevy* es un motor de videojuegos prometedor cuyas prestaciones se implementan como plugins. En la mayoría de los casos, se cargan en tiempo de compilación, pero `bevy::dynamic_plugin` da la posibilidad de hacerlo dinámicamente. `bevy` se basa en la falsa estabilidad del ABI explicado en la Sección 4.3.4, así que podría romperse en un futuro.

  Otras fuentes como Amos [104] o Zicklag [105] también intentan usar cargado dinámico para funcionalidades similares. Este último se trata de

Amethyst [106], el predecesor de `bevy`, que acabó rindiéndose debido a la inestabilidad del ABI [107][108].

4.5. Elección Final

Tras implementar varios sistemas de plugins en miniatura con las tecnologías más prometedoras mencionadas en esta sección, se tomó la decisión de usar cargado dinámico con `abi_stable`. Cada alternativa tiene sus puntos fuertes y sus puntos flojos, como ilustra la Figura 4.9, pero pocas realmente cumplen los requisitos de rendimiento establecidos por el equipo de Tremor.

Todas las tecnologías excepto cargado dinámico, eBPF o memoria compartida requieren la serialización y copia de los datos en algún momento, algo que Tremor no se puede permitir. De esas tres posibles soluciones, todas tienen que lidiar con problemas con el ABI. La que mejor soporte tiene es cargado dinámico, así que ese será el camino tomado.

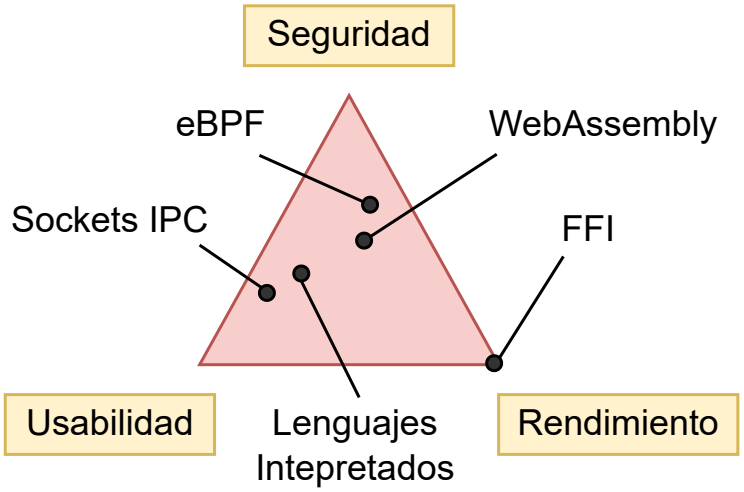

Figura 4.9: Comparación aproximada de los métodos investigados.

Capítulo 5

Implementación

5.1. Metodología

Antes de nada, es importante aprender un poco sobre cómo realizar cambios en el código de Tremor eficientemente. Este proyecto modificará gran cantidad de líneas y cuanto más rápido sea el desarrollo, menos problemas habrán. Esto se puede cubrir de forma específica al lenguaje Rust, con trucos o consejos que puedan facilitar el desarrollo, o de forma más general, con la estrategia de trabajo a seguir. En esta sección se cubrirá lo último, dado que es menos un detalle de implementación.

La metodología fue insipirada por mis mentores, que lo denominaron el "Just Make it Work", o "Primero haz que funcione". Se basa en que, inicialmente, con lo que más problemas tenía era el perderme en los detalles. Pero ciertamente, primero de todo lo importante es que "funcione". Siempre y cuando el sistema de plugins se pueda compilar y ejecutar, lo siguiente es secundario:

- Código "feo" (no idiomático, repetitivo o desordenado).
- Código de bajo rendimiento.
- Documentación pobre.
- No tener tests prematuros.
- No aplicar sugerencias recomendadas por *linters* (en el caso de Rust, *Clippy*).

El propio desarrollo invitaba a aprovechar las ventajas de un lenguaje fuertemente tipado como Rust, evitando realizar testing posiblemente prematuro. Esto consistía en realizar cambios y posteriormente trabajar en que los aceptara el compilador, repetidamente. Únicamente se procedió al testing exhaustivo una vez la interfaz final del sistema de plugins compilaba, pasadas las fases tempranas o incluso medias.

Adicionalmente, las optimizaciones prematuras son la fuente de todos los problemas. No es algo que sea importante aún. Solo una vez terminada la primera iteración se puede dedicar más tiempo a medir el rendimiento para saber cuáles optimizaciones merecen la pena. Notar que sí es importante escoger un *método o tecnología* que sea apropiado en términos de rendimiento; fue por ello por lo que se descartó WebAssembly o IPC en el capítulo anterior. Pero definitivamente el desarrollador debería rendirse en, por ejemplo, evitar una conversión entre dos tipos innecesaria que posiblemente no afecte al rendimiento al fin y al cabo.

Lo que quería dejar claro el equipo de Tremor es que todos los tests, limpiezas u optimizaciones que intentes realizar en este momento acabará muy probablemente siendo en vano. Se llegará a un punto en el que no se pueda continuar y que requiera repensar y reescribir gran parte del trabajo. Cuando todo compile y aparentemente funcione correctamente, se puede dedicar esfuerzo a trabajar en estos temas secundarios. Si algo no importante está llevando demasiado tiempo, se debería marcar como TODO o FIXME y dejarlo para otro momento.

Notar que no hay problema con "gastar" el tiempo con métodos que acaban siendo incorrectos, porque realmente no se está "gastando" nada; son un paso necesario para llegar a la solución final. Pero es doloroso tener que eliminar código al que le has dedicado tiempo, así que al menos debería intentarse minimizar el impacto que esto tenga.

5.2. Versionado en `abi_stable`

Dado que `abi_stable` va a ser la librería principal en la que se basará el sistema de plugins, es importante entender cómo funciona al completo.

Además de conocer los detalles de implementación, es importante conocer cómo `abi_stable` soluciona los problemas a tener en cuenta para implementar un sistema de plugins. La librería especifica lo siguiente respecto a su sistema de compatibilidad de tipos [109]:

- El ABI de `abi_stable` se comprueba siempre. Cada versión `0.y.0` y `x.0.0` de `abi_stable` define su propio ABI, que es incompatible con versiones anteriores.
- Los tipos se comprueban recursivamente cuando se carga una librería dinámica, antes de llamar ninguna función.

Todo esto se basa en el *trait* (equivalente en este caso a una clase abstracta de Java) `StableAbi`, indicador de que un tipo es seguro para cargado dinámico. Contiene información como su disposición en memoria y puede ser implementado automáticamente con un macro. Al cargar un plugin con `abi_stable`, se comprobará que sus tipos sean compatibles con aquellos de la runtime.

Sin este mecanismo, sería posible cargar un plugin con una interfaz diferente a la runtime, resultando en violaciones de acceso a memoria. Pongamos un caso en el que una estructura en la interfaz de la runtime tuviese un campo adicional. El plugin exportará esa estructura sin el campo nuevo, puesto que usa una interfaz anticuada. Cuando la runtime intente acceder a la estructura del plugin, se leerá un campo que no existe y que por tanto es parte de memoria no definida.

5.3. Conversión de la interfaz a C

Es importante mantener la interfaz de plugins lo más simple posible. Los detalles de comunicación deberían dejarse a la runtime, de forma que los plugins se limiten a exportar una lista de funciones síncronas. De esta forma, se podrá evitar en lo posible pasar tipos complejos (programación asíncrona, canales de comunicación, etc) entre la runtime y los plugins, que implicaría una carga de trabajo mucho más alta.

Una vez esta interfaz de bajo nivel se defina, se puede crear un *wrapper* de más alto

nivel en la runtime que se encargue de la comunicación y de mejorar su usabilidad dentro de Tremor. Esto mismo lo hacen otras *crates* como `rdkafka` [110], que implementa una capa de abstracción asíncrona sobre su interfaz de C en `rdkafka-sys` [111].

El primer paso consiste en declarar la interfaz del PDK de forma que use el ABI de C, en vez del de Rust. Esto se puede hacer con el atributo `#[repr(C)]` (en lugar del `#[repr(Rust)]` implícito). Para usar `abi_stable` también tendrá que implementarse el *trait* `StableAbi`.

La dificultad reside en que todos los tipos dentro de la interfaz *también* tendrán que haber sido declarados con tanto `#[repr(C)]` como `StableAbi`, recursivamente. Esto puede convertirse en un problema si alguno de los tipos a convertir es externo y no se tiene acceso directo. Tendrá que abrirse un pull request para añadir soporte, o crear un *wrapper* que envuelva la funcionalidad de forma opaca a su distribución en memoria. Se incluye la explicación completa respecto a Tremor en el Anexo C, con más detalles sobre la implementación.

5.4. Cargado de plugins

Afortunadamente, `abi_stable` ya se encarga de la mayoría del trabajo en este aspecto. Lo único que hace falta implementar es una manera en la que *encontrar* los plugins en el sistema. Para ello, se introduce una nueva variable de entorno `TREMOR_PLUGIN_PATH` que liste todos los directorios que pueden contener plugins. De forma similar a `PATH`, los directorios se separan con dos puntos.

Una vez se tiene la lista de directorios, se comprobarán también sus subdirectorios, recursivamente. Es importante añadir un límite de profundidad aquí para evitar que el programa se quede atascado, por ejemplo si el usuario incluyese el directorio raíz (`/`) accidentalmente. Tampoco deberían seguirse enlaces simbólicos por simplicidad.

Todos aquellos archivos encontrados cuya extensión sea la de una librería dinámica tendrán que añadirse a la lista de posibles plugins. Esta extensión varía según el sistema operativo: en Linux es `.so`, en Windows es `.dll`, y en MacOS

es `.dylib`.

Finalmente, si al intentar cargar uno de los plugins encontrados la operación falla, se mostrará una advertencia y se continuará hasta probar con todas las ocurrencias. Un método más robusto para evitar cargar ficheros que no sean plugins de Tremor sería usar una extensión personalizada, o algun tipo de convención para el nombre. Sin embargo, al personalizarse manualmente los directorios, se asume que la gran mayoría de las librerías dinámicas serán plugins.

5.5. Gestión de pánicos

Los pánicos en Rust se usan para expresar errores de los que un programa no se puede recuperar. Por ejemplo, acceder a un índice inexistente en una lista, o quedarse sin memoria. Su funcionamiento es similar a una excepción de C++ o Java: se propagarán hasta llegar a la función principal y parar la ejecución del programa.

Actualmente, lanzar pánicos a través de la interfaz C es comportamiento no definido [112, FFI and Panics]. Aunque el programa aborte en la mayoría de los casos, no existe ninguna garantía de que vaya a suceder así; podría continuar en un estado inválido, con cualquier tipo de consecuencia.

La solución más directa es usar la función `std::panic::catch_unwind`, que, para casos excepcionales como este, puede parar la propagación de pánicos cuando sea llamada. Se podría usar en todas las funciones exportadas por el plugin internamente, y en caso de producirse un pánico se terminaría el programa manualmente, en lugar de dejar que se propague a la runtime, que sería indefinido.

También es posible configurar el programa por completo para que aborte cuando se produzca un pánico, en vez de propagarlo. De esta forma, no se llegaría a invocar comportamiento no definido y se mantendría un rendimiento máximo — capturar pánicos tiene un coste. Sin embargo, implica varias desventajas importantes: al abortar, no se tendrá acceso a la información de *debug* que dan los pánicos, tampoco se limpiará el estado del programa, y desde la runtime es imposible saber si el plugin ha configurado los pánicos para que aborten. Esto último será

posible en el futuro, una vez *Pluggable panic implementations (tracking issue for RFC 1513)* [113] llegue a una versión estable.

Esto es algo que `abi_stable` ha tenido en cuenta desde el principio. Antes de que un pánico se vaya a propagar de plugin a runtime, la librería abortará el programa por completo. Esta parte se realiza transparentemente; no hace falta que el desarrollador se preocupe en ningún momento por ello.

La solución de `abi_stable` no es perfecta, ya que tiene un pequeño coste de rendimiento e imposibilita el recuperarse de errores en los plugins. En una futura versión de Tremor, podría ser posible reiniciar plugins en caso de que dejen de funcionar para mejorar la resiliencia a fallos.

El equipo de Rust conoce esta limitación y está trabajando en mejorar la situación. En una futura versión, planea definir cuándo se puede propagar un pánico de forma más precisa [114].

5.6. Programación asíncrona

El objetivo inicial era simplificar la interfaz lo suficiente como para que no sea necesario tratar aspectos como programación asíncrona en el PDK. Esto terminó siendo inevitable, dado que la asincronía es uno de los pilares de Tremor.

Para poder usar programación asíncrona con el ABI de C se puede recurrir a la *crate* `async_ffi` [115], cuyo único problema era no tener soporte para `abi_stable` en concreto. Al no implementar el *trait* `StableAbi` , no se podía usar en la interfaz del PDK, por lo que tuve que abrir un pull request para hacerlo yo mismo [116]. El uso de esta librería resulta en código más verboso, pero esto se podría mejorar en el futuro con macros [117].

5.7. Seguridad en hilos

`abi_stable` utiliza la *crate* `libloading` [68] internamente, cuya gestión

de errores no es segura en hilos en algunas plataformas, como `dlerror` en FreeBSD [118][119]. Sí que lo es en Linux [120], macOS [121] y Windows [122], así que en el caso de Tremor no es un problema.

Será importante tenerlo en cuenta en el futuro; al añadir soporte para un nuevo sistema operativo habrá que asegurarse de que su gestión de errores sea segura en hilos. En caso contrario, deberá actualizarse `libloading` para sincronizar el acceso con un mútex interno, como lo hace la *crate* `dlopen` [69] [123]. Notar que `dlopen` es también mejorable, dado que usa el mútex *siempre*, incluso en sistemas operativos donde la gestión de errores sí es segura en hilos [124].

5.8. Complejidad de las conversiones

Un punto vital a tener en cuenta es el coste de realizar conversiones entre tipos de la librería estándar y tipos de `abi_stable`. Esto se dará en numerosas ocasiones, dado que usar `abi_stable` cuando no es necesario es subóptimo para tanto el rendimiento como la usabilidad. Y si, por ejemplo, convertir un `Vec<T>` a un `RVec<T>` tuviese complejidad $O(n)$, probablemente `abi_stable` tendría que ser descartado como la solución escogida.

Afortunadamente, tras analizar la implementación de tipos como `RVec<T>`, `RSlice<T>`, `RStr` o `RString`, estas conversiones únicamente consisten en transferir un puntero con los datos, sin necesidad de copiar nada. Es decir, las conversiones que realizaremos serán $O(1)$.

5.9. Problemas con varianza y subtipado

Una complicación a la que se dedicó una cantidad considerable de tiempo tiene que ver con el concepto de *varianza* y *subtipado*. En resumen, si un tipo es *covariante*, el modelo de memoria de Rust será mucho más flexible al usarlo. Todo este mecanismo es implícito, es decir, en ningún momento el desarrollador especifica manualmente la varianza del tipo; el compilador de Rust lo infiere automáticamente.

Sin embargo, un tipo puede dejar de ser covariante si incumple una serie de reglas preestablecidas. En ese caso, la flexibilidad adicional se pierde y se producen errores casi imposibles de entender si uno no es familiar con el término *varianza*. Los tipos definidos en `abi_stable`, a diferencia de aquellos en la librería estándar, no eran covariantes, lo cual fue especialmente complicado de descubrir y requirió reescribir algunas partes suyas por completo.

Afortunadamente, una futura versión de Rust hará el debugging de estos errores mucho más intuitivo [125]. Es un tema que requiere conocimientos más avanzados de Rust, por lo que se incluye en el Anexo D al completo para más información.

5.10. Optimizaciones

Una vez implementada la primera versión del sistema de plugins, se realizan optimizaciones iterativamente hasta alcanzar un rendimiento lo suficientemente bueno. Antes de aplicarlas, sin embargo, es importante realizar mediciones para asegurarse de que la mejora ofrecida es apreciable. Para ello, debe elaborarse un entorno de *benchmarking* riguroso y pruebas que tengan en cuenta casos de uso variados y realistas. Los experimentos iniciales indican una degradación del 35 % del rendimiento, por lo que tendrán que realizarse mejoras hasta llegar al rango de 10-20 % propuesto por Tremor. Este paso se describe en detalle en el Anexo F.

Las primeras optimizaciones incluyen una limpieza general del código, la restauración de optimizaciones que se eliminaron temporalmente (mencionado en el Anexo C) o la simplificación de la interfaz del PDK. Sin embargo, no son suficiente para el objetivo marcado, únicamente alcanzándose un 30 % de degradación.

Capítulo 6

Conclusiones y trabajo futuro

6.1. Concusiones

La complejidad del proyecto ha resultado ser mucho mayor de lo esperado, principalmente por el malentendido sobre la estabilidad del ABI de Rust. Por tanto, ha resultado imposible desarrollar en el tiempo disponible un sistema de plugins tan completo y eficiente como se especificaba inicialmente.

El problema principal tiene que ver con el rendimiento. Dada la naturaleza de Tremor, es un requerimiento imprescindible para poderlo incluir en producción. Tras las pruebas realizadas en el Anexo F, se ha calculado que el sistema de plugins reduce el rendimiento un 30 %.

No obstante, la última versión del sistema de plugins es perfectamente funcional y, mediante su investigación, el diseño de su arquitectura y contribuciones de código abierto, se ha hecho posible su inclusión en una futura versión de Tremor.

Parte de esta ralentización en el desarrollo se debe también a Rust. Al ser un lenguaje tan inmaduro es frecuente encontrar documentación pobre o librerías incompletas. Muchas de las *crates* usadas no disponían inicialmente de la funcionalidad necesaria para un sistema de plugins, como `async_ffi`, `abi_stable`, `halfbrown` o `simd-json`. Se han resuelto problemas importantes en el entorno, extendiendo el soporte para el ABI de C, resolviendo tipos con varianzas inflexibles y elaborando conversiones de tipos no triviales,

todo ello manteniendo la máxima seguridad y eficiencia posible. Con esfuerzos como estos, el desarrollo de proyectos similares en el futuro resultará mucho más accesible.

6.2. Futuro

Se ha documentado tanto el proceso seguido como lo que queda pendiente, de forma que el equipo de Tremor pueda continuar trabajando en el sistema de plugins para su futuro lanzamiento. Sin embargo, aun después de esto el PDK nunca parará de evolucionar: su uso se extenderá en la base de código y se perfeccionarán otras características con el tiempo. Algunas ideas son las siguientes:

- **Mejoras de rendimiento**: el enfoque principal para el primer lanzamiento del PDK. Esto incluye la realización de *benchmarks* más variados y realistas, y el soporte de `abi_stable` en más librerías.
- **Soporte de otros componentes de Tremor**: el PDK únicamente se implementa para los conectores, pero también podría funcionar con códecs, preprocesadores, postprocesadores, operadores, funciones, extractores, etc.
- **Refinamiento de la experiencia de usuario**: creación de proyectos modelo como base para plugins nuevos, ejemplos de uso, macros, documentación exhaustiva y de más alto nivel, frameworks de testing, etc.
- **Carga de plugins a petición del usuario**: además de poder cargar los plugins al inicio del programa, sería especialmente útil solicitar su carga durante la ejecución. Se podría elaborar un nuevo método de configuración iterativo, en el que se cargan y configuran los plugins uno a uno, y finalmente se exporta la composición final.
- **Paquetes de plugins**: en ciertos casos, sería más conveniente exportar un plugin que implemente más de un componente. Por ejemplo, podrían juntarse los conectores de TCP y UDP en único plugin, dado que probablemente compartan partes de su código y dependencias.
- **Gestión de versiones alternativas**: la implementación de `abi_stable` para comprobar las versiones es rudimentaria y poco eficiente; se limita

a comprobarlos todos recursivamente. Otra opción más simple sería únicamente comprobar una cadena con el versionado global para la interfaz, por ejemplo.

- **Registro centralizado de plugins**: en el futuro a largo plazo, se podría desarrollar una funcionalidad similar a los repositorios de Maven o Cargo. Allí se podrían guardar todos los plugins de la comunidad para gestionarlos automáticamente.

- **Eliminación de plugins en tiempo de ejecución**: es especialmente complejo de implementar, dado que `abi_stable` explícitamente no lo soporta. Sin embargo, esto mejoraría considerablemente la resiliencia a errores, siendo posible reiniciar plugins completamente.

6.3. Valoración personal

Pese a las situaciones de frustración frente a todos los errores y bloqueos que he encontrado en el camino, ha sido una experiencia extraordinaria. Matthias bromeó una vez con que "El infierno de debugging es importante para el desarrollo de personaje", y creo que tiene toda la razón. Enfrentarme a errores que no sabía ni cómo abordar me ha enseñado mucho sobre Rust, y lo que es más importante, sobre desarrollo de software en general.

Estoy muy satisfecho con haber conseguido lo que he conseguido, y aún más por haberlo poder hecho junto al increíble equipo que es el de Tremor. Trabajar con ellos me ha ayudado a descubrir qué quiero hacer tras la graduación, y con qué tipo de empresa y personas quiero trabajar. Me mantendré en contacto con ellos para seguir el progreso del sistema de plugins.

Bibliografía

[1] *Wayfair*. 2008. URL: https://www.wayfair.com/.

[2] *Wayfair Announces Fourth Quarter and Full Year 2021 Results*. 2021. URL: https://investor.wayfair.com/news/news-details/2022/Wayfair-Announces-Fourth-Quarter-and-Full-Year-2021-Results/default.aspx.

[3] Chris Sweeney. *Inside Wayfair's Identity Crisis*. 2019. URL: https://www.bostonmagazine.com/news/2019/10/01/inside-wayfair/.

[4] Jordan Sissel. *Logstash*. 2011. URL: https://www.elastic.co/logstash/.

[5] Gary White. *Live, Laugh, Log: Save G's on GB/s with Tremor and K8s — TremorCon 2021*. 2021. URL: https://youtu.be/xsowS5hEKRg?list=PLNTN4J6tdf20vy14FVOazLTdou_8xyvfe.

[6] *Cloud Native Computing Foundation*. 2015. URL: https://www.cncf.io/.

[7] *Kubernetes*. 2014. URL: https://kubernetes.io/.

[8] *Linux Foundation*. URL: https://www.linuxfoundation.org/.

[9] *Projects — Linux Foundation*. URL: https://www.linuxfoundation.org/projects/.

[10] *LFX Mentorship — Linux Foundation*. URL: https://lfx.linuxfoundation.org/tools/mentorship/.

[11] *Rust Survey Results*. 2021. URL: https://blog.rust-lang.org/2022/02/15/Rust-Survey-2021.html#challenges-ahead.

[12] Carliss Young Baldwin y Kim B Clark. *Design rules: The power of modularity*. Vol. 1. MIT press, 2000.

[13] Igor Sysoev. *NGINX*. 2004. URL: https://www.influxdata.com/time-series-platform/telegraf/.

[14] Robert McCool. *Apache HTTP Server*. 1995. URL: https://httpd.apache.org/.

[15] Ruslan Ermilov. *Dynamic Modules Development*. 2016. URL: https://www.nginx.com/blog/dynamic-modules-development/#demand.

[16] *Dynamic Shared Object (DSO) Support*. 2021. URL: https://httpd.apache.org/docs/2.4/dso.html#advantages.

[17] *Tremor Discord*. 2020. URL: https://discord.com/invite/Wjqu5H9rhQ.

[18] Mario Ortiz Manero. *NullDeref*. 2021. URL: https://nullderef.com/.

[19] *Tremor*. 2020. URL: https://www.tremor.rs/.

[20] David C Luckham. *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons, 2011.

[21] *Telegraf*. 2015. URL: https://www.influxdata.com/time-series-platform/telegraf/.

[22] *Tremor Getting Started*. 2021. URL: https://www.tremor.rs/docs/0.12/getting-started/.

[23] *Tremor Constraints and Limitations*. 2021. URL: https://www.tremor.rs/docs/0.11/ConstraintsLimitations.

[24] *Tremor Recipes*. 2021. URL: https://www.tremor.rs/docs/0.11/recipes/.

[25] D Robins. "Complex event processing". En: *Second International Workshop on Education Technology and Computer Science. Wuhan*. Citeseer. 2010, págs. 1-10.

[26] Gianpaolo Cugola y Alessandro Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing". En: *ACM Comput. Surv.* 44.3 (2012). ISSN: 0360-0300. DOI: 10.1145/2187671.2187677. URL: https://doi.org/10.1145/2187671.2187677.

[27] Miyuru Dayarathna y Srinath Perera. "Recent advancements in event processing". En: *ACM Computing Surveys (CSUR)* 51.2 (2018), págs. 1-36.

[28] K. Tawsif y col. "A Review on Complex Event Processing Systems for Big Data". En: *2018 Fourth International Conference on Information Retrieval and Knowledge Management (CAMP)*. 2018, págs. 1-6. DOI: 10.1109/INFRKM.2018.8464787.

[29] *Tremor Onramps*. 2021. URL: https://www.tremor.rs/docs/0.11/artefacts/onramps.

[30] *Tremor Offramps*. 2021. URL: https://www.tremor.rs/docs/0.11/artefacts/offramps.

[31] *Pipeline Model*. 2021. URL: https://www.tremor.rs/docs/0.11/overview#pipeline-model.

[32] *MIRI*. 2016. URL: https://github.com/rust-lang/miri.

[33] Jill Steinberg. *Competing components make for prickly panelists*. 1997. URL: https://www.infoworld.com/article/2077623/competing-components-make-for-prickly-panelists.html.

[34] Isaac Z. Schlueter. *npm*. 2010. URL: https://www.npmjs.com/.

[35] Jamie Kyle. *How to build an npm worm*. URL: https://jamie.build/how-to-build-an-npm-worm.

[36] Simon Maple. *Yet another malicious package found in npm, targeting cryptocurrency wallets*. 2019. URL: https://snyk.io/blog/yet-another-malicious-package-found-in-npm-targeting-cryptocurrency-wallets/.

[37] Ryan Dahl. *Deno*. 2018. URL: https://deno.land/.

[38] Aria Beingessner. *How Swift Achieved Dynamic Linking Where Rust Couldn't*. 2019. URL: https://gankra.github.io/blah/swift-abi/.

[39] Bram Moolenaar. *VimScript*. 2015. URL: http://vimdoc.sourceforge.net/htmldoc/usr_41.html.

[40] *Lua in NeoVim*. 2022. URL: https://neovim.io/doc/user/lua.html.

[41] Roberto Ierusalimschy. *Programming in lua*. Roberto Ierusalimschy, 2006.

[42] *LuaJIT Benchmarks*. 2008. URL: https://luajit.org/performance.html.

[43] *Lua Sand Boxes*. 2015. URL: http://lua-users.org/wiki/SandBoxes.

[44] Simonas Kazlauskas. *rlua*. 2017. URL: https://crates.io/crates/rlua.

[45] Alex Orlenko. *mlua*. 2019. URL: https://crates.io/crates/mlua.

[46] Markus Westerlind. *Gluon*. 2016. URL: https://gluon-lang.org/.

[47] *Rhai*. 2016. URL: https://rhai.rs/.

[48] John-John Tedro. *Rune*. 2018. URL: https://rune-rs.github.io/.

[49] *TIOBE Index*. 2022. URL: https://www.tiobe.com/tiobe-index/.

[50] *WebAssembly*. 2017. URL: https://webassembly.org/.

[51] *WASI*. 2019. URL: https://wasi.dev/.

[52] *Wasmer*. 2019. URL: https://wasmer.io/.

[53] *Wasmtime*. 2019. URL: https://wasmtime.dev/.

[54] *Actually Using Wasm*. 2020. URL: https://wiki.alopex.li/ActuallyUsingWasm.

[55] Abhinav Jangda y col. "Not So Fast: Analyzing the Performance of {WebAssembly} vs. Native Code". En: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 2019, págs. 107-120.

[56] Frank Denis. *Benchmark of WebAssembly runtimes*. 2021. URL: https://00f.net/2021/02/22/webassembly-runtimes-benchmarks/.

[57] *WasmExternType — Wasmer v0.17.1*. 2021. URL: https://docs.rs/wasmer-runtime-core/0.17.1/wasmer_runtime_core/types/trait.WasmExternType.html.

[58] *Interface Types Proposal for WebAssembly*. 2015. URL: https://github.com/webassembly/interface-types.

[59] *witx*. 2022. URL: https://github.com/WebAssembly/WASI/blob/main/tools/witx-docs.md.

[60] Alex Crichton. *Support for Interface Types in wasmtime API — GitHub bytecodealliance/wasmtime*. 2021. URL: https://github.com/bytecodealliance/wasmtime/issues/677.

[61] Antonin Peronnet. *State of the art of the interface types — GitHub wasmerio/wasmer*. 2021. URL: https://github.com/wasmerio/wasmer/issues/2480.

[62] Caelum van Ispelen. `feather/quill` — *GitHub feather-rs/feather*. 2021. URL: https://github.com/feather-rs/feather/tree/main/quill.

[63] *Plugins — Veloren Project Architecture*. 2021. URL: https://book.veloren.net/contributors/developers/codebase-structure.html#plugins.

[64] Aram Drevekenin. *zellij-org/zellij — GitHub*. 2021. URL: https://github.com/zellij-org/zellij.

[65] Steven McCanne y Van Jacobson. *eBPF*. 2014. URL: https://ebpf.io/.

[66] Quentin Monnet. `rbpf`. 2017. URL: https://crates.io/crates/rbpf.

[67] Jack May, Alexander Meißner y Dmitri Makarov. `solana_rbpf`. 2018. URL: https://crates.io/crates/solana_rbpf.

[68] Simonas Kazlauskas. `libloading`. 2015. URL: https://crates.io/crates/libloading.

[69] Szymon Wieloch. `dlopen`. 2017. URL: https://crates.io/crates/dlopen.

[70] Tyler Wolf Leonhardt. `sharedlib`. 2016. URL: https://crates.io/crates/sharedlib.

[71] Szymon Wieloch. `dlopen`*: Compare with other libraries*. 2021. URL: https://docs.rs/dlopen/0.1.8/dlopen/#compare-with-other-libraries.

[72] Linear. *Idiomatic Rust plugin system — StackOverflow*. 2017. URL: https://stackoverflow.com/questions/44708483/idiomatic-rust-plugin-system/46249019#46249019.

[73] Kurt Lawrence. `papyrus/src/compile/execute.rs` — *GitHub kurtlawrence/papyrus*. 2019. URL: https://github.com/kurtlawrence/papyrus/blob/1c7f0a669fed59d220bdefb161c568072126d3d5/src/compile/execute.rs#L36.

[74] Simonas Kazlauskas. *Thread-safety — libloading v0.7.0*. 2021. URL: https://docs.rs/libloading/0.7.0/libloading/struct.Library.html#thread-safety.

[75] Michael F. Bryan. *Plugins in Rust*. 2019. URL: https://web.archive.org/web/20211202080304/https://adventures.michaelfbryan.com/posts/plugins-in-rust/.

[76] roblabla. *Plugin system with API — r/Rust*. 2017. URL: https://www.reddit.com/r/rust/comments/6v29z0/plugin_system_with_api/dlx9w7v/.

[77] Chase Wilson. *Add the* `-Z randomize-layout` *flag — GitHub rust-lang/compiler-team*. 2021. URL: https://github.com/rust-lang/compiler-team/issues/457.

[78] Michael F. Bryan. *Dynamic Loading & Plugins*. 2020. URL: https://fasterthanli.me/articles/so-you-want-to-live-reload-rust.

[79] *The Rust Reference*. 2022. URL: https://doc.rust-lang.org/reference/introduction.html.

[80] comex. *A Stable Modular ABI for Rust*. 2020. URL: https://internals.rust-lang.org/t/a-stable-modular-abi-for-rust/12347/71.

[81] Rodrimati1992. `abi_stable`. 2019. URL: https://crates.io/crates/abi_stable.

[82] Taiki Endo. `crossbeam`. 2015. URL: https://crates.io/crates/crossbeam.

[83] David Tolnay. `serde_json`. 2015. URL: https://crates.io/crates/serde_json.

[84] Connor Horman y Ray Redondo. *LCCC*. 2021. URL: https://github.com/LightningCreations/lccc.

[85] Daniel Henry-Mantilla. `safer_ffi`. 2020. URL: https://crates.io/crates/safer_ffi.

[86] Auri. `cglue`. 2021. URL: https://crates.io/crates/cglue.

[87] *Language Server Protocol*. 2016. URL: https://microsoft.github.io/language-server-protocol/.

[88] Aditya Venkataraman y Kishore Kumar Jagadeesha. "Evaluation of inter-process communication mechanisms". En: *Architecture* 86 (2015), pág. 64.

[89] *Protocol Buffers*. 2008. URL: https://developers.google.com/protocol-buffers.

[90] kotauskas. `interprocess`. 2020. URL: https://crates.io/crates/interprocess.

[91] Griffin O'Neill. `ipipe`. 2021. URL: https://crates.io/crates/ipipe.

[92] *Rust By Example*. 2021. URL: https://doc.rust-lang.org/rust-by-example/index.html.

[93] Mats Petersson. *Performance difference between IPC shared memory and threads memory*. 2013. URL: https://stackoverflow.com/a/14512554.

[94] elast0ny. `shared_memory`. 2018. URL: https://crates.io/crates/shared_memory.

[95] elast0ny. `raw_sync`. 2020. URL: https://crates.io/crates/raw_sync.

[96] *Cargo*. 2015. URL: https://github.com/rust-lang/cargo.

[97] *mdBook*. 2015. URL: https://rust-lang.github.io/mdBook/.

[98] Aram Drevekenin. *Zellij*. 2021. URL: https://zellij.dev/.

[99] *Xi*. 2016. URL: https://xi-editor.io/.

[100] Raph Levien. *JSON — xi-editor retrospective*. 2020. URL: https://raphlinus.github.io/xi/2020/06/27/xi-retrospective.html#json.

[101] Danny Tuppeny. *Extensions using the "typeçommand (for ex. Vim) have poor performance due to being single-threaded with other extensions — GitHub microsoft/vscode*. 2019. URL: https://github.com/microsoft/vscode/issues/75627.

[102] Azad Bolour. *Notes on the Eclipse Plug-in Architecture*. 2003. URL: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.

[103] Carter Anderson. *Bevy*. 2020. URL: https://bevyengine.org/.

[104] Amos. *So you want to live-reload Rust*. 2020. URL: https://fasterthanli.me/articles/so-you-want-to-live-reload-rust.

[105] Zicklag. *Rust Plugins*. 2019. URL: https://zicklag.github.io/rust-tutorials/rust-plugins.html.

[106] Eyal Kalderon. *Amethyst*. 2016. URL: https://amethyst.rs/.

[107] Zicklag. *Creating Rust Apps With Dynamically Loaded Rust Plugins*. 2019. URL: https://users.rust-lang.org/t/creating-rust-apps-with-dynamically-loaded-rust-plugins/28814/111092.

[108] Zicklag. *[NEW FEATURE] WebAssembly scripting system — GitHub amethyst/amethyst*. 2019. URL: https://github.com/amethyst/amethyst/issues/1729.

[109] Rodrimati1992. *Safety — GitHub rodrimati1992/abi_stable_crates*. 2021. URL: https://github.com/rodrimati1992/abi_stable_crates#safety.

[110] David Blewett y col. `rdkafka`. 2016. URL: https://crates.io/crates/rdkafka.

[111] David Blewett y col. `rdkafka-sys`. 20216. URL: https://crates.io/crates/rdkafka-sys.

[112] *The Rustnomicon*. 2022. URL: https://doc.rust-lang.org/nomicon/.

[113] Niko Matsakis. *Pluggable panic implementations (tracking issue for RFC 1513)*. 2021. URL: https://github.com/rust-lang/rust/issues/32837.

[114] Kyle J Strand. `"C-unwind" ABI` — *The Rust RFC Book*. 2021. URL: https://rust-lang.github.io/rfcs/2945-c-unwind-abi.html.

[115] oxalica. `async_ffi`. 2021. URL: https://crates.io/crates/async_ffi.

[116] Mario Ortiz Manero. *Support for* `abi_stable` *— GitHub oxalica/async-ffi*. 2021. URL: https://github.com/oxalica/async-ffi/pull/10.

[117] Mario Ortiz Manero. *Procedural macro for boilerplate — GitHub oxalica/async-ffi*. 2021. URL: https://github.com/oxalica/async-ffi/issues/12.

[118] Simonas Kazlauskas. *Thread-safety — Libloading v0.7.1*. 2021. URL: https://docs.rs/libloading/0.7.1/libloading/struct.Library.html#thread-safety.

[119] `dlerror` — *The Open Group Base Specifications*. 2021. URL: https://pubs.opengroup.org/onlinepubs/009604499/functions/dlerror.html.

[120] `dlerror` *attributes — Linux Manual Page*. 2021. URL: https://man7.org/linux/man-pages/man3/dlerror.3.html#ATTRIBUTES.

[121] `dlerror` — *Mac OS X Man Pages*. 2021. URL: https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/dlerror.3.html.

[122] `SetThreadErrorMode` — *Microsoft Documentation*. 2021. URL: https://docs.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-setthreaderrormode.

[123] Szymon Wieloch. `src/raw/unix.rs` — *GitHub szymonwieloch/rust-dlopen*. 2021. URL: https://github.com/szymonwieloch/rust-dlopen/blob/f9fed7168dfdcfd682a6c409595d4e2430b2dd2d/src/raw/unix.rs#L17.

[124] Mario Ortiz Manero. `dlerror` is *thread-safe on some platforms*. 2021. URL: https://github.com/szymonwieloch/rust-dlopen/issues/42.

[125] Jack Huey. *The Rust borrow checker just got (a little bit) smarter*. 2022. URL: https://jackh726.github.io/rust/2022/06/10/nll-stabilization.html.

[126] *Tremor Linked Transports*. 2021. URL: https://www.tremor.rs/docs/0.11/operations/linked-transports/.

[127] *Tremor Connectors*. 2021. URL: https://www.tremor.rs/docs/0.12/reference/connectors/.

[128] Steve J. Donovan. *A Gentle Introduction to Rust*. 2018. URL: https://stevedonovan.github.io/rust-gentle-intro/.

[129] vnduongthanhtung. *30 minutes of Introduction to Rust for C++ programmers*. 2017. URL: https://vnduongthanhtung.gitbooks.io/migrate-from-c-to-rust/content/.

[130] Pascal. *Rust for Professionals*. 2020. URL: https://overexact.com/rust-for-professionals/.

[131] *The Rust Programming Language*. 2022. URL: https://doc.rust-lang.org/stable/book/.

[132] *Asynchronous Programming in Rust*. 2022. URL: https://rust-lang.github.io/async-book/01_getting_started/01_chapter.html.

[133] Mario Ortiz Manero. *Add support for StableAbi — GitHub simd-lite/value-trait*. 2021. URL: https://github.com/simd-lite/value-trait/pull/14.

[134] Rodrimati1992. `abi_stable/src/external_types` — *GitHub rodrimati1992/abi_stable_crates*. 2021. URL: https://github.com/rodrimati1992/abi_stable_crates/tree/edfb2a97a7b5d7ecbc29c1f9f115f61e26f42da6/abi_stable/src/external_types.

[135] Bastian Johannes Kauschke. *Diving Deep: implied bounds and variance #25860*. 2015. URL: https://lcnr.de/blog/diving-deep-implied-bounds-and-variance/.

[136] Jon Gjengset. *Crust of Rust: Subtyping and Variance*. 2021. URL: https://www.youtube.com/watch?v=iVYWDIW71jk.

[137] Heinz N. Gies. *lifetimes with R* types break compared to non R* types — GitHub rodrimati1992/abi_stable_crates*. 2021. URL: https://github.com/rodrimati1992/abi_stable_crates/issues/75.

[138] Rafael Bachmann. *Compile crash after performing module rename via rust analyzer, but only when compiling for test with nightly, workaround by running* `cargo clean` *— GitHub rust-lang/rust*. 2021. URL: https://github.com/rust-lang/rust/issues/90608.

[139] *Module* `async_std::task` *—* `async_std` *documentation*. 2021. URL: https://docs.rs/async-std/1.10.0/async_std/task/index.html.

[140] *Actor model — Wikipedia*. URL: https://en.wikipedia.org/wiki/Actor_model.

[141] *LFX Mentorship Showcase*. 2022. URL: https://events.linuxfoundation.org/lfx-mentorship-showcase/.

[142] Heinz N. Gies. `halfbrown`. 2020. URL: https://crates.io/crates/halfbrown.

[143] Maciej Hirsz. `beef`. 2020. URL: https://crates.io/crates/value_trait.

[144] Heinz N. Gies. `value_trait`. 2020. URL: https://crates.io/crates/value_trait.

[145] *async-std*. 2019. URL: https://async.rs/.

[146] Theo M. Bulut. *Bastion*. 2019. URL: https://www.bastion-rs.com/.

[147] *Variance and associated types — Guide to Rustc Development*. 2021. URL: https://rustc-dev-guide.rust-lang.org/variance.html#variance-and-associated-types.

[148] Mario Ortiz Manero. *Subtyping and Variance - Trait variance not covered — GitHub rust-lang/nomicon*. 2022. URL: https://github.com/rust-lang/nomicon/issues/338.

[149] Mario Ortiz Manero. *Making abi_stable's types covariant — GitHub rodrimati1992/abi_stable_crates*. 2022. URL: https://github.com/rodrimati1992/abi_stable_crates/issues/81.

[150] Mario Ortiz Manero. *Fix `R*` lifetimes — GitHub rodrimati1992/abi_stable_crates*. 2022. URL: https://github.com/rodrimati1992/abi_stable_crates/pull/76.

[151] *Rustlang Community Discord Server*. URL: https://discord.gg/rust-lang-community.

[152] Mario Ortiz Manero. *PDK support — GitHub tremor-rs/tremor-runtime*. URL: https://github.com/tremor-rs/tremor-runtime/pull/1434.

[153] Mario Ortiz Manero. *PDK with a single value — GitHub tremor-rs/tremor-runtime*. URL: https://github.com/marioortizmanero/tremor-runtime/pull/11.

Anexos

Anexos A

Guía de Rust

Es posible que en este anexo se omitan algunos conceptos o que algunas explicaciones no sean completamente precisas por razones de simplicidad. *The Rust Programming Language* [131] es el libro oficial para aprender Rust por completo, pero es una lectura larga y posiblemente demasiado exhaustiva. Para mayor brevedad, se recomienda leer *Rust for Professionals* [130], *A Gentle Introduction to Rust* [128] o *30 minutes of Introduction to Rust for C++ programmers* [129]. Partes de este capítulo se obtienen de estas fuentes.

La comunidad dispone de otros libros que explican aspectos más avanzados del lenguaje en específico, como `unsafe` o la programación asíncrona. En esos casos, se recomienda leer *The Rustnomicon* [112] y *Asynchronous Programming in Rust* [132], respectivamente.

A.1. Primeros pasos

Comenzando por el clásico "Hola Mundo", se incluyen algunos ejemplos de cómo es la sintaxis de Rust más básica. Los binarios o librerías en Rust reciben el nombre de *crate*. Nuestra *crate* se podría ejecutar fácilmente con *Cargo*, el administrador de dependencias oficial, específicamente con el comando `cargo run`.

```
1 fn main() {
2     println!("Hello World!");
3 }
```

`main` es nuestra función principal, que invoca al macro `println!` para escribir por pantalla. Notar que la invocación de macros, a diferencia de funciones, requiere un una exclamación al final del identificador.

A.2. Conceptos principales

Los bloques básicos (`if`, `else`, `while`, `for`) son muy similares a los de otros lenguajes. También existe `match`, que permite extraer patrones de variables:

```
fn factorial(i: u64) -> u64 {
    match i {
        // Primer caso: i = 0
        0 => 1,
        // El resto de casos, asignado a una variable `n`
        n => n * factorial(n-1)
    }
}
```

Uso de variables y métodos:

```
fn main() {
    // Declaración de una variable, cuyo tipo se infiere
    // automáticamente.
    let my_number = 1234;
    // Declaración de una variable con un tipo especificado
    // manualmente. Notar que se puede usar el mismo nombre, y la
    // variable anterior será destruida.
    let my_number: i32 = 4321;
    // Invocación de la función estática (constructor) `new` dentro
    // del tipo `String`. El uso de `mut` indica que la instancia del
    // tipo se puede modificar. Funciona de forma inversa a C++, que
    // por defecto es mutable y `const` indica que *no* se puede
    // modificar.
    let mut my_str = String::new();
    // Invocación del método `push` de `my_str`, que añade un
    // carácter al final de la cadena.
    my_str.push('a');
}
```

Otros componentes principales de Rust son:

– Estructuras de datos:

```rust
struct MessageA {
    // Campo público con una cadena de caracteres
    pub text: String,
    // Campo privado con un entero
    user_id: i32,
}
```

```rust
// Sin nombres de campos; se pueden acceder con `msg.0` y `msg.1`,
// respectivamente.
struct MessageB(pub String, i32);
```

– Enumeraciones, que también permiten contener datos:

```rust
enum MessageC {
    Join,
    Text(String, i32),
    Leave(i32),
}
```

– *Traits*, similares a las interfaces de Java en el sentido de que son una serie de requerimientos y que un tipo puede implementar múltiples *traits*, pero también permiten implementaciones por defecto:

```rust
trait Sender {
    // Los métodos requieren especificar `self` explícitamente,
    // que es lo mismo que `this` en Java o C++. En este caso,
    // `&send` tomará una referencia al tipo que implemente
    // `Sender`. También podría ser una referencia mutable con
    // `&mut self`, o el tipo en sí con `self`.
    fn send(&self, msg: String);

    // Implementación por defecto.
    fn send_twice(&self, msg: String) {
        self.send(msg.clone());
        self.send(msg);
    }
}
```

Y para implementar un *trait* para un tipo:

```rust
impl Sender for MessageC {
    fn send(&self, msg: String) {
        match self {
            Join => println!("Joined"),
            Text(txt, id) => println!("{id} sent: {txt}"),
            // Las variables `_` son ignoradas
            Leave(_) => println!("Left"),
        }
    }

    // `send_twice` se implementará automáticamente.
}
```

Notar que, aunque Rust no sea un lenguaje orientado a objetos, un *trait* puede heredar de otro *trait*. Al contrario, un *struct* no puede heredar de otro *struct*.

A.3. Genéricos y librería estándar

De forma similar a C++, Rust posee tipos genéricos. Esto permite la implementación de una librería estándar flexible, con varias estructuras de datos importantes a conocer:

```rust
// Función genérica, donde `ToString` es un trait. El tipo del
// parámetro `T` tendrá que implementar `ToString`.
fn print<T: ToString>(t: T) {}

// Otra manera de especificar genéricos con diferencias menores
// que no se explicarán en esta introducción.
fn print(t: impl ToString) {}
```

- Tipos primitivos:
  - Carácteres con `char`.
  - Punto flotante con `f32` y `f64`.
  - Booleanos con `bool`.
  - Enteros: `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, e incluso `i128` y `u128` en las arquitecturas que lo soportan.
  - Vectores de tamaño fijo: por ejemplo `[1, 2, 3, 4, 5]`.

- N-tuplas como `(1, true, 9.2)`.
- El tipo "unidad", `()`, equivalente a `void` en C o C++.
- Punteros básicos con `*const T` o `*mut T`.

– `Vec<T>` representa un vector contiguo y redimensionable.

– `HashMap<K, V>` es una tabla hash, genérica respecto a su clave `K` y su valor `V`. No se encuentra en el preludio, por lo que requeriría la siguiente declaración, similar a un `import` de Java:

```
use std::collections::HashMap;
```

– `Box<T>`, usado para localizar un tipo `T` no nulo en memoria. Además de un puntero `*const T`, incluye el tamaño que ocupa `T` y tiene una interfaz limitada para que su uso sea siempre seguro.

– `str` es una cadena UTF-8 de solo lectura, típicamente usada con una referencia `&str`. Va acompañada por su longitud, por lo que no hace falta terminarla con `\0`, a diferencia de C. `String` es su versión modificable asignada en memoria.

A.4. Gestión de errores

En Rust, los errores se indican con el tipo `Result<T, E>`. Este se trata de una enumeración cuyo valor puede ser `Ok(T)`, con el resultado obtenido satisfactoriamente, o `Err(E)`, con el tipo de error que ha sucedido. Dado que el resultado está contenido dentro suyo, es imposible olvidar comprobar si se ha producido algún error. Se puede usar `match` para comprobar el resultado, o una serie de funciones disponibles para hacer el proceso más ergonómico:

```
match load_file(input) {
    Ok(data) => /* ... */,
    Err(e) => eprintln!("Error: {e}"),
}
```

En caso de que se produjera un error del que el programa no se pudiera recuperar,

como quedarse sin memoria o un fallo inesperado en la implementación, se usa la funcionalidad de *pánicos*. Un pánico se propaga de forma similar a una excepción de C++ o Java, y terminará la ejecución por completo. Se puede invocar con el macro `panic!` o utilidades similares.

A.5. Macros

Rust cuenta con dos tipos de macros: *declarativos* y *procedurales*. Ambos permiten generar código a tiempo de compilación, pero se diferencian principalmente en la flexibilidad que ofrecen, a coste de un coste de desarrollo menor o mayor, respectivamente.

Los macros declarativos se crean con una sintaxis especializada, similar a un `match` con patrones de tokens (identificadores, tipos, etc) como entrada, y los tokens nuevos como salida. Son similares a los macros de C o C++, pero más potentes e higiénicos (en el sentido de que su expansión no captura identificadores accidentalmente).

Los macros procedurales se describen como extensiones del lenguaje y en una *crate* independiente. Esencialmente, ejecutan código en la compilación que consume y produce sintaxis de Rust; consisten en directamente transformar el Árbol de Sintaxis Abstracta (AST) [79, Procedural Macros]. Consecuentemente, su complejidad es mucho mayor, pero expanden las posibilidades de los macros enormemente.

```
some_macro!(1, 2, 3); // Puede ser tanto declarativo como procedural
```

```
// Sintaxis típica de invocación de un macro
some_macro! {
    fn some_function() { /* ... */ }
}

// También permitido en el caso de los procedurales
#[some_macro]
fn some_function() { /* ... */ }
```

Finalmente, los macros procedurales se pueden declarar de forma que *deriven* (implementen automáticamente) un *trait*. Esto evita escribir código repetitivo de forma muy sencilla:

```rust
// Con un macro `derive` para el trait `Debug`, que sirve para
// mostrar variables por pantalla.
#[derive(Debug)]
struct X(i32);

// Sin ellos sería lo siguiente. Como es trivial se puede
// simplificar en un macro procedural de tipo `derive`.
impl fmt::Debug for X {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{:?}", self.0)
    }
}
```

A.6. Lifetimes

La seguridad que provee Rust en memoria se basa en un modelo a tiempo de compilación, que usa *lifetimes*. Una *lifetime* inicia cuando se crea una referencia, y termina cuando se destruye, por lo que indica la longitud de su vida [92, Lifetimes].

```rust
// Se anotan las lifetimes a continuación con líneas marcando la
// creación y destrucción de cada variable.
// `i` tiene la lifetime más larga porque su ámbito (scope)
// encierra completamente a tanto `ref1` como `ref2`. La duración
// de `ref1` comparada con `ref2` es irrelevante, puesto que son
// disjuntas.
fn main() {
    let i = 3; // La lifetime de `i` comienza ──────────────┐
    //                                                      │
    { //                                                    │
        let ref1 = &i; // La lifetime `ref1` comienza ──┐   │
        //                                              │   │
        println!("ref1: {ref1}");   //                 │   │
    } // `ref1 termina ──────────────────────────────────┘   │
    //                                                      │
    //                                                      │
    { //                                                    │
        let ref2 = &i; // La lifetime `ref2` comienza ──┐   │
        //                                              │   │
        println!("ref2: {ref2}");   //                 │   │
    } // `ref2` termina ──────────────────────────────────┘   │
    //                                                      │
}   // Lifetime termina──────────────────────────────────────┘
```

Las *lifetimes* se anotan de forma similar a los tipos genéricos, pero en minúsculas y precedidos por un apóstrofo: `Foo<'a>` tiene una *lifetime* `'a` como parámetro. La *lifetime* `'static` se reserva para aquellas referencias estáticas, es decir, que existen durante toda la ejecución del programa. Las *lifetimes* en referencias se indican con `&'a T`. Por ejemplo, las siguientes funciones expanden las *lifetimes* manualmente, que no es necesario en casos sencillos porque lo puede inferir el compilador:

```
// Una referencia de entrada con lifetime `'a`, que debe vivir al
// menos lo mismo que la función.
fn print_one<'a>(x: &'a i32) {
    println!("`print_one`: x es {x}");
}

// Múltiples elementos con lifetimes diferentes. En este caso, ambas
// podrían tener también la lifetime `'a`, pero en casos más
// complejos es posible que se necesiten lifetimes distintas.
fn print_multi<'a, 'b>(x: &'a i32, y: &'b i32) {
    println!("`print_multi`: x es {x}, y es {y}");
}
```

A.7. Unsafe

Para poder mantener control completo a bajo nivel, es posible ignorar sus garantías de seguridad con el sub-lenguaje llamado *unsafe Rust*. El análisis estático de Rust es conservativo; en ocasiones es posible que rechace algunos programas correctos. El desarrollador puede indicar que es consciente de la situación y puede apagar este análisis en el bloque para corregirlo por sí mismo, arriesgándose a cometer un error en su código.

Se puede acceder a *unsafe Rust* conteniendo el código dentro de un bloque `unsafe { /* ... */ }` o mediante una función `unsafe fn name() { /* ... */ }`. El código dentro de bloques `unsafe` funciona igual que fuera de ellos, pero incluye varias nuevas habilidades, entre otras:

- Leer un puntero bruto en memoria
- Acceder o modificar una variable estática mutable

– Llamar a una función `unsafe`

A.8. Programación asíncrona

Como muchos lenguajes modernos, Rust da soporte a la programación asíncrona, un modelo de programación concurrente. Esta permite tener una gran cantidad de *tareas* concurrentes ejecutándose sobre unos pocos hilos del Sistema Operativo. Su caso de uso principal es programas cuyo rendimiento está limitado por operaciones de entrada y salida, como servidores o bases de datos [132].

```
// Con `async` se indica que la función es asíncrona.
async fn get_two_sites_async() {
    // Creación de dos "futuros" que, al completarse, descargarán
    // asíncronamente las páginas web. Similar a la creación de
    // un nuevo hilo.
    let future_one = download_async("https://www.foo.com");
    let future_two = download_async("https://www.bar.com");

    // Ejecutar las dos tareas. Similar a esperar la terminación de
    // los hilos.
    let (website_one, website_two) = join!(future_one, future_two);

    // Con `.await` se puede esperar a la terminación de un futuro
    // individual.
    let website_three = download_async("https://www.bar.com").await;
}
```

Anexos B

Funcionamiento interno de Tremor

B.1. Arquitectura

Antes de comenzar a modificar el código existente en Tremor, era importante conocer cómo funciona para evitar perder el tiempo. Tremor se basa en el modelo actor. Citando Wikipedia:

"[El modelo actor trata al] actor como el componente universal de computación concurrente. En respuesta a un mensaje que recibe, un actor puede: tomar decisiones locales, crear más actores, enviar más mensajes y determinar cómo responder al siguiente mensaje recibido. Los actores pueden modificar su propio estado privado, pero solo pueden afectarse entre sí indirectamente a través de mensajería (eliminando la necesidad de sincronización con *locks*)." [140]

No usa un lenguaje (e.g., Erlang) o framework (e.g., *Bastion* [146], quizá en el futuro) que siga estrictamente este modelo, pero re-implementa los mismos patrones frecuentemente de forma manual. Tremor se basa en *programación asíncrona*, es decir, que en vez de hilos trabaja con *tareas*, un concepto de nivel más alto y especializado para entrada/salida. De la documentación de *async-std* [145], la runtime asíncrona que usa Tremor:

"La ejecución de un programa asíncrono en Rust consiste en una recopilación de hilos nativos del Sistema Operativo, sobre los cuales múltiples corutinas no apilables (*stackless*) son multiplexadas. Nos referimos a ellas como "tareas". Las

tareas pueden tener nombre e incluir soporte para sincronización.” [139]

Podríamos resumir su arquitectura con la frase “Tremor se basa en actores corriendo en tareas diferentes, que se comunican asíncronamente con canales”.

B.2. Detalles de implementación

A nivel de implementación, los conectores se definen con el *trait* `Connector`, incluido en la figura B.1. Esencialmente, los plugins de tipo conector exportarán públicamente esta interfaz en su binario, y la runtime deberá ser capaz de cargarlo dinámicamente. Actualmente, todos los conectores disponibles se listan y cargan de forma estática al inicio del programa.

El actor principal se llama `World`. Contiene el estado del programa, como los artefactos disponibles (*repositorios*) y los que se están ejecutando (*registros*) y se usa para inicializar y controlar el programa.

Los *managers* o *gestores* son simplemente actores en el sistema que envuelven una funcionalidad. Ayudan a desacoplar la comunicación y la implementación de la funcionalidad interna. De esta forma, se puede eliminar código repetitivo al inicializar los componentes, así como la creación de canales de comunicación o el lanzamiento del componente en una tarea nueva. Generalmente, hay un gestor por cada tipo de artefacto para facilitar su inicialización y también uno por cada instancia que se esté ejecutando, para controlar su comunicación.

Notar que la inicialización de los conectores ocurre en dos pasos. Primero se *registran*, es decir, se indica su disponibilidad para cargarlo (añadiéndolo al repositorio). Posteriormente, no se ejecutará hasta conectarse con otro artefacto con `launch_binding`, lo cual lo movería del repositorio al registro, junto al resto de artefactos ejecutándose.

B.2.1. Registro

La Figura B.2 detalla todos los pasos seguidos en el código. Primero han de inicializarse los gestores, y después registrar los artefactos. Actualmente, esta parte se realiza de forma estática con `register_builtin_types`, pero después de implementar el PDK, debería ser dinámicamente. Tremor buscaría automáticamente plugins en sus directorios configurados e intentaría registrar todos los que encuentre. En una futura versión, el usuario podría solicitar manualmente el cargado de un plugin nuevo mientras se está ejecutando Tremor.

B.2.2. Inicialización

Ya que es un proceso en múltiples pasos (en la implementación es más complicado que registro + creación), la primera parte provee las herramientas para inicializar el conector (el *builder*). Cuando el conector necesite comenzar a ejecutarse porque se haya añadido a una *pipeline*, el *builder* ayuda a construir y configurarlo de forma genérica. Finalmente, se añade a una tarea propia para que se pueda comunicar con otras partes de Tremor. El gestor `connectors::Manager` contiene todos los conectores ejecutándose en Tremor, como se muestra en la Figura B.3.

La Figura B.4 muestra un ejemplo de una *pipeline*, definida con Troy, su propio lenguaje inspirado en SQL.

B.2.3. Configuración

Una vez haya un conector corriendo, la Figura B.5 visualiza cómo se divide en una parte *sink* y otra *source*. Estas son opcionales, pero no exclusivas, así que se puede tener cualquiera de las dos o ambas. De forma similar, un *builder* se usa para inicializar las partes y a continuación inicia una nueva tarea para ellos.

También se crea un gestor por cada instancia de *sink* o *source*, que se encargará de la comunicación con otros actores. De esta forma, sus interfaces pueden mantenerse lo más simple posible. Esos gestores recibirán peticiones de conexión de la *pipeline* y posteriormente leerán o enviarán eventos en ella.

La diferencia principal entre *sources* y *sinks* a nivel de implementación es que este último también puede responder a mensajes usando la misma conexión. Esto es útil para notificar que el paquete ha llegado (`Ack`) o que algo ha fallado (`Fail` para un evento específico, `CircuitBreaker` para dejar de recibir datos por completo).

Los códecs y preprocesadores se involucran aquí tanto para los *sources* como para los *sinks*. En la parte de *source*, los datos son transformados a través de una cadena de preprocesadores y posteriormente se aplica un códec. Para los *sinks*, se sigue el proceso inverso: los datos se codifican primero a bytes con el códec, y posteriormente una serie de postprocesadores se aplican a los datos binarios.

B.2.4. Notas adicionales

Algunos conectores se basan en *flujos*. Son equivalentes a los flujos de TCP, que ayudan a agrupar mensajes para evitar mezclarlos. Se inician y finalizan mediante mensajes, y el gestor se guarda el estado del flujo en un campo llamado `states` (ya que, por ejemplo, algunos preprocesadores puedan querer guardar un estado). Si un conector no necesita flujos, como `metronome` (que únicamente envía eventos periódicamente), puede especificar su identificador de flujo como `DEFAULT_STREAM_ID` siempre.

Tras implementar la interfaz de los conectores para el sistema de plugins, los primeros conectores a desarrollar deberían ser:

- *Blackhole*, usado para medir el rendimiento. Realiza mediciones de tiempos de final a final para cada evento pasando por la *pipeline*, y al final guarda un histograma HDR (*High Dynamic Range*).
- *Blaster*, usado para repetir una serie de eventos de un archivo, que es especialmente útil para pruebas de rendimiento.

Ambos son relativamente simples y serán de gran ayuda para medir el efecto de los cambios sobre el rendimiento. De todos modos, el equipo de Tremor insistía que lo más importante primero es que funcione, y después me podría preocupar sobre eficiencia.

```rust
pub trait Connector {
    /// Crea la parte "source" del conector, si es aplicable.
    async fn create_source(
        &mut self,
        _source_context: SourceContext,
        _builder: source::SourceManagerBuilder,
    ) -> Result<Option<source::SourceAddr>> {
        Ok(None)
    }

    /// Crea la parte "sink" del conector, si es aplicable.
    async fn create_sink(
        &mut self,
        _sink_context: SinkContext,
        _builder: sink::SinkManagerBuilder,
    ) -> Result<Option<sink::SinkAddr>> {
        Ok(None)
    }

    /// Intenta conectarse con el mundo exterior. Por ejemplo, inicia la
    /// conexión con una base de datos.
    async fn connect(
        &mut self,
        _c: &ConnectorContext,
        _attempt: &Attempt
    ) -> Result<bool> {
        Ok(true)
    }

    /// Llamado una vez cuando el conector inicia.
    async fn on_start(&mut self, _c: &ConnectorContext) -> Result<()> {
        Ok(())
    }
    /// Llamado cuando el conector pausa.
    async fn on_pause(&mut self, _c: &ConnectorContext) -> Result<()> {
        Ok(())
    }
    /// Llamado cuando el conector continúa.
    async fn on_resume(&mut self, _c: &ConnectorContext) -> Result<()> {
        Ok(())
    }
    /// Llamado ante un evento de "drain", que se asegura de que no
    /// lleguen más eventos a este conector.
    async fn on_drain(&mut self, _c: &ConnectorContext) -> Result<()> {
        Ok(())
    }
    /// Llamado cuando el conector para.
    async fn on_stop(&mut self, _c: &ConnectorContext) -> Result<()> {
        Ok(())
    }
}
```

Figura B.1: Simplificación del *trait* `Connector`

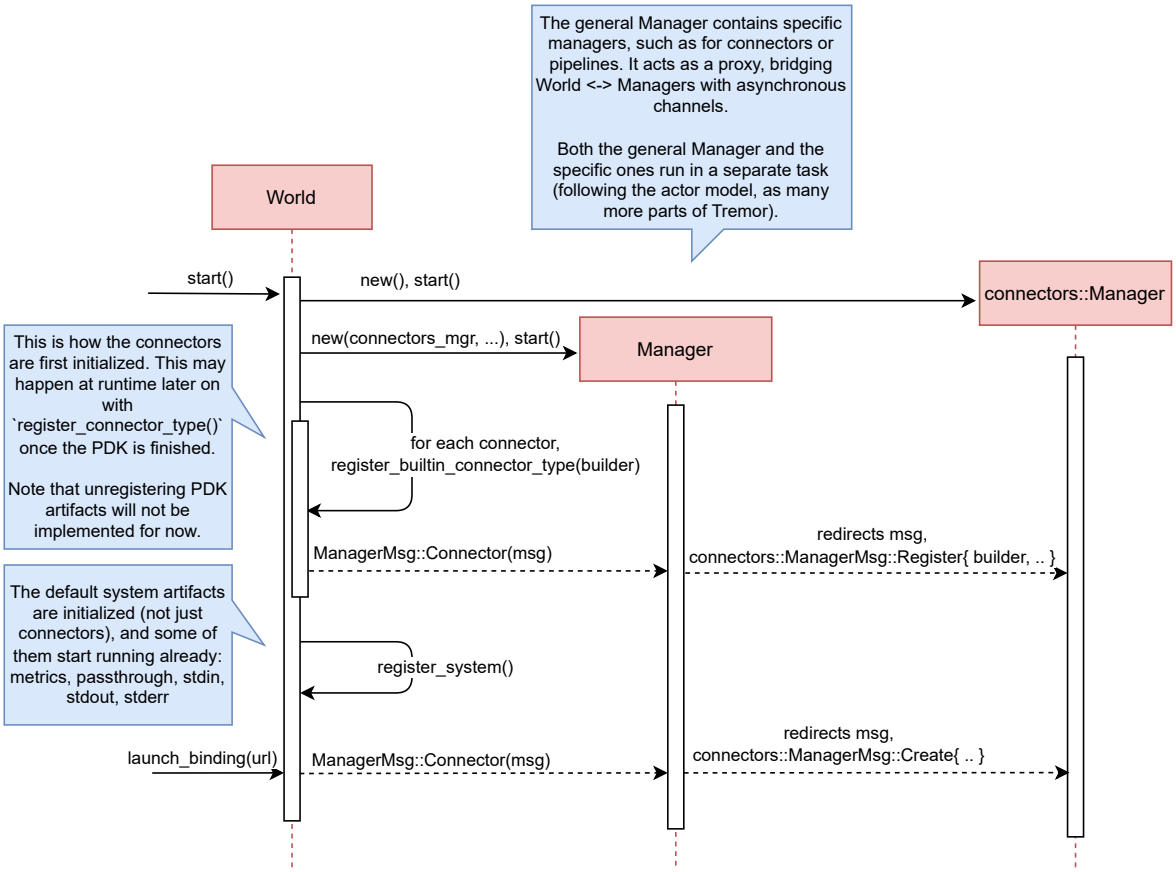

Figura B.2: Registro de un conector en el programa

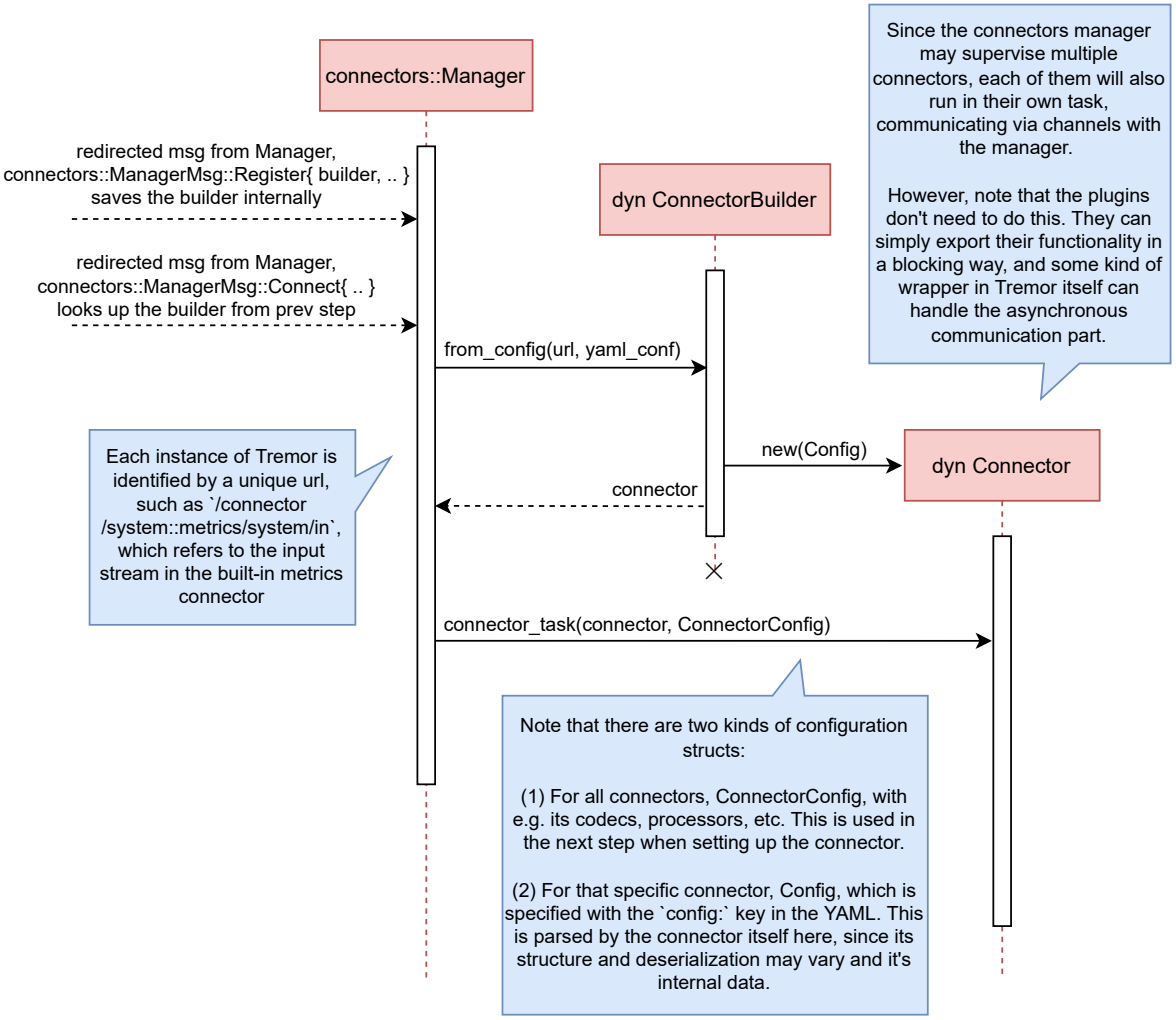

Figura B.3: Inicialización de un conector en el programa

```
define pipeline main
# The exit port is not a default port, so we have to overwrite the
# built-in port selection
into out, exit
pipeline
  # Use the `std::string` module
  use std::string;
  use lib::scripts;

  # Create our script
  create script punctuate from scripts::punctuate;

  # Filter any event that just is `"exit"` and send it to the exit port
  select {"graceful": false} from in where event == "exit" into exit;

  # Wire our capitalized text to the script
  select string::capitalize(event) from in where event != "exit"
    into punctuate;
  # Wire our script to the output
  select event from punctuate into out;
end;
```

Figura B.4: Ejemplo de una *pipeline* definida para Tremor

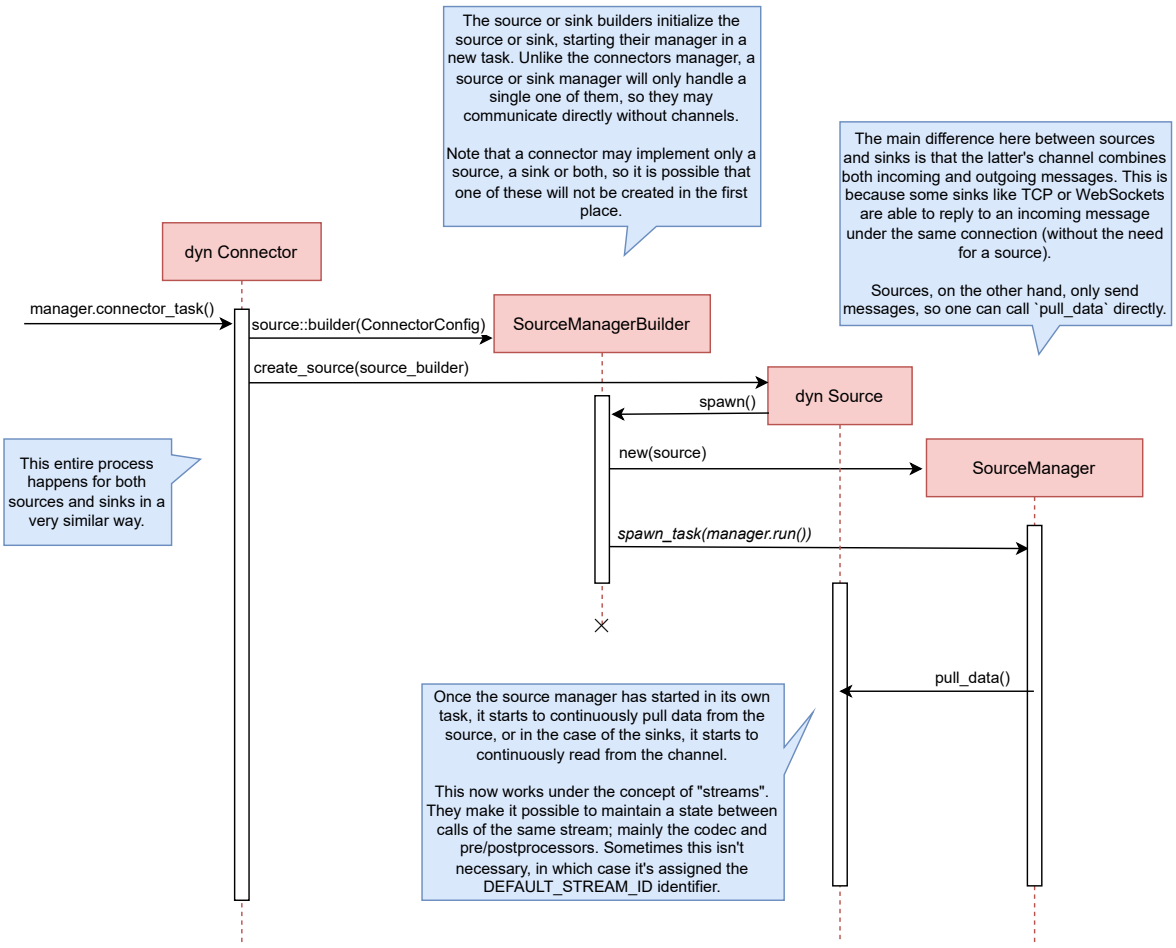

Figura B.5: Configuración de un conector en el programa

Anexos C

Conversión del ABI de Rust al ABI de C

Para ilustrar mejor la conversión de ABIs, se introduce la estructura más problemática al respecto, `Value`. Esta enumeración sirve para representar datos pseudo-JSON y es definido a continuación de forma simplificada:

```rust
pub enum Value {
    /// Valores estáticos (enteros, booleanos, etc)
    Static(StaticNode),
    /// Tipo para cadenas de caracteres
    String(String),
    /// Tipo para listas
    Array(Vec<Value>),
    /// Tipo para objetos (mapas clave-valor)
    Object(Box<HashMap<String, Value>>),
    /// Tipo para datos binarios
    Bytes(Vec<u8>),
}
```

Para poder usar `Value` en la interfaz del sistema de plugins, se pueden modificar sus tipos internos tal que:

```rust
#[repr(C)] // La representación en memoria de Value seguirá el ABI de C
#[derive(StableAbi)] // Solo necesario cuando se usa abi_stable
pub enum Value {
    Static(StaticNode),
    /// Ahora usa `RString`, la altenativa a `String` de abi_stable
    String(RString),
    /// De forma similar, usa `RVec` en vez de `Vec`
    Array(RVec<Value>),
    /// Cambio de `Box`, `HashMap` y `String` por sus alternativas
    Object(RBox<RHashMap<RString, Value>>),
```

```
    /// Otro cambio de `Vec`
    Bytes(RVec<u8>),
}
```

El primer problema surge en la variante `Static`. Su tipo contenido internamente, `StaticNode`, es externo y usa `#[repr(Rust)]`. Se declara en el *crate* `value_trait` [144] así:

```
pub enum StaticNode {
    I64(i64),
    U64(u64),
    F64(f64),
    Bool(bool),
    Null,
}
```

Esto se podría arreglar siguiendo el mismo procedimiento recursivamente, hasta que todo sea `#[repr(C)]`. Pero como se trata de una librería externa, tendrá que abrirse un nuevo pull request y esperar que al autor le parezcan bien los cambios [133]. Será importante también que la estructura use `#[repr(C)]` únicamente cuando se configure explícitamente a tiempo de compilación. De esta forma, el resto de usuarios podrán seguir aprovechándose de las ventajas de rendimiento que ofrece `#[repr(Rust)]`.

C.1. Consecuencias del sistema de plugins

Desgraciadamente, el procedimiento no termina ahí; cambiar las variantes de `Value` implica que el código que lo usaba se romperá de numerosas formas:

```
// No funcionará porque Value::Array contiene un RVec ahora
let value = Value::Array(Vec::new());
```

Este caso es el más sencillo: simplemente hace falta cambiar `Vec`, de la librería

estándar, por `RVec`, de `abi_stable`. La intención de los tipos de `abi_stable` es que sean un reemplazo directo de los de la librería estándar, es decir, su uso será exactamente el mismo:

```
let value = Value::Array(RVec::new());
```

Es un poco más complicado cuando los tipos anteriores se exponen en métodos, porque requiere tomar una decisión entre expandir el límite de FFI del *funcionamiento interno* de `Value` a los *usuarios* de `Value`. Por ejemplo, la variante `Value::Object` contiene un `RHashMap` ahora, pero el método `Value::as_object` solía devolver una referencia a `HashMap`. Se producirá un error nuevo ahí y tendrá que tomarse una decisión entre devolver `RHashMap` o añadir una conversión interna a `HashMap`:

```
impl Value {
    // Código original
    fn as_object(&self) -> Option<&HashMap<String, Value>> {
        match self {
            // Problema: `m` ahora es una `RHashMap`, pero la función
            // devuelve un `HashMap`.
            //
            // Solución 1: cambiar el tipo devuelto a `RHashMap`
            // Solución 2: convertir `m` a un `HashMap` con `m.into()`
            Self::Object(m) => Some(m),
            _ => None,
        }
    }
}
```

- Si se cambia el tipo devuelto a `RHashMap`, casi todas las veces que se llamaba a `as_object` ahora dejarán de compilar porque se esperan un `HashMap`.

  Esto puede ser complicado porque, para evitar realizar conversiones, el sistema de plugins *infectaría* la base de código por completo. Tendría que propagarse el uso de `RHashMap` por todo el programa, incluso cuando el PDK no es importante. Por ejemplo, `Value` también se usaba en la implementación del lenguaje de Tremor, Troy. Tener que usar un `RHashMap` en esa situación sería confuso y acabarían modificándose gran cantidad de ficheros sin relación al sistema de plugins.

- Si se realiza una conversión interna a `HashMap` en `as_object`, evitaremos

todos esos errores, con un pequeño coste de rendimiento. Es la opción más fácil, pero si `Value::as_object` se usara frecuentemente, como en el bucle principal, sí que podría causar una degradación considerable.

Como indica la sección 5.8, las conversiones entre la librería estándar y `abi_stable` son $O(1)$. Esto es dónde la metodología "Primero haz que funcione" es relevante: simplemente dejaremos el límite del FFI en su mínimo y añadiremos conversiones cuanto antes sea posible. Al terminar, si se detectan problemas de rendimiento en un caso en concreto, se puede reconsiderar.

C.2. Problemas con tipos externos

En algunos casos, los tipos de `abi_stable` no habían sido actualizados para incluir métodos nuevos de la librería estándar, por lo que era necesario un pull request para añadirlo[1]. Pero por lo general, convertir los tipos *de la librería a estándar a* `abi_stable` es una tarea trivial, simplemente un tanto tedioso.

Los problemas surgen cuando es necesario convertir *tipos externos a* `abi_stable`. La declaración anterior de `Value` era una simplificación; realmente, Tremor usa la implementación de `halfbrown` [142] de `HashMap`. Esto se debe a que es más eficiente para su caso de uso, y que posee algunas funcionalidades adicionales necesarias. El mismo caso se da para el tipo `Cow`, cuya alternativa en la *crate* `beef` [143] ocupa menos espacio en memoria y ofrece un mejor rendimiento en Tremor.

Ninguna de estas dos librerías tienen soporte dentro de `abi_stable`, y aunque sus tipos estén basados en otros de la librería estándar, la conversión no es directa porque su implementación es distinta. Se pueden tomar cuatro posibles alternativas:

[1] El Anexo E lista todas las contribuciones de código abierto realizadas para este proyecto.

C.2.1. Evitar el tipo externo

Basándose en "Primero haz que funcione", una solución perfectamente válida es eliminar las optimizaciones temporalmente y dejar un `TODO` para que se pueda revisar posteriormente. Es posible que el sistema de plugins tenga excesiva complejidad, y limitarse a usar tipos de la librería estándar podría ser suficiente.

En el caso específico de `Value`, eliminar las optimizaciones problemáticas parece la manera más fácil de arreglar el problema. Y lo sería, si no fuera porque eliminar código también puede ser complicado, como muestra la Figura C.1, especialmente cuando la funcionalidad extra del tipo externo no está disponible.

C.2.2. Encapsular el tipo externo

Otra opción es crear un *wrapper* para `halfbrown`, de la misma forma que lo hace ya `abi_stable` con otras librerías más conocidas. Este encapsulamiento hace posible su uso desde el ABI de C de forma segura. Sin embargo, estos ejemplos ya existentes son complejos [134] y difíciles de mantener, ya que tendrán que actualizarse con cada nueva versión de `halfbrown`.

C.2.3. Reimplementar el tipo con el ABI de C desde cero

Similar a la solución anterior, pero incluso más costoso, dado que también requeriría reimplementar la funcionalidad desde cero. Puede parecer indeseable, pero es la mejor forma de asegurar un rendimiento máximo. Los tipos externos mencionados son parte de optimizaciones; encapsularlos podría tener un impacto en su rendimiento y hacerlos inútiles.

```
error: aborting due to 120 previous errors

Some errors have detailed explanations: E0277, E0308, E0412, E0432, E0433, E0495, E0621, E0623, E0631...
For more information about an error, try `rustc --explain E0277`.
error: could not compile `tremor-script`

To learn more, run the command again with --verbose.
```

Figura C.1: Al intentar evitar los tipos externos se produjeron más de 120 errores de compilación.

```rust
// Así funciona la programación asíncrona en Rust; la primera
// función es prácticamente equivalente a la segunda.
async fn example() -> String {
    read_file().await
}
fn example() -> impl Future<Output = String> {
    async {
        read_file().await
    }
}

// No pueden haber genéricos en FFI, por lo que ahora `Future`
// es un tipo concreto `FfiFuture` en vez de un trait. La
// conversión de `Future` a `FfiFuture` se puede realizar con
// `into_ffi`.
fn example() -> FfiFuture<String> {
    async move {
        read_file().await
    }
    .into_ffi()
}
// `FfiFuture<T>` implementa `Future<Output = T>`, por lo que
// su uso es el mismo.
async fn user() {
    example().await
}
```

Figura C.2: Interfaz modificada para la programación asíncrona en el sistema de plugins con la *crate* `async_ffi`. Este caso sigue el método de encapsular `Future` con el tipo `FfiFuture`.

Si esta parte del proyecto es lo suficientemente importante y existen los recursos, debería considerarse. De hecho, el mismo tipo `Value` en Tremor surgió por esta razón: ya existía `simd_json::Value` de otra librería, pero carecía de la suficiente flexibilidad y el equipo implementó uno personalizado.

C.2.4. Simplificar el tipo para la interfaz

Esta última opción resultó ser la más sencilla de implementar: crear una copia de `Value` cuyo único uso es comunicarse entre runtime y plugins, ilustrado en la Figura C.3.

Ya que es un tipo nuevo, no se romperá nada del código existente, y únicamente hará falta cambiarlo donde se use la interfaz. Su implementación es exactamente igual que el `Value` modificado anteriormente, pero bajo el nuevo nombre `PdkValue`.

No es necesario escribir métodos adicionales para el nuevo `PdkValue`, solo sus conversiones desde y hasta el tipo original, `Value`. Esto sería equivalente a, en vez de pasar un `Vec<T>` al PDK, reemplzarlo con un `*const u8` para los datos y un `u32` para la longitud. Simplemente consiste en simplificar los tipos en la interfaz, y convertirlos de vuelta para usar la funcionalidad completa.

El problema principal es que la conversión entre tipos es ahora $O(n)$ en vez de $O(1)$, dado que es necesario iterar los datos en los objetos y vectores para la conversión. Su uso sería el siguiente:

```
// Esta función es exportada por el plugin. Funcionará porque
// `PdkValue` está declarado con el ABI de C.
pub extern "C" fn plugin_funfuncue: PdkValue) {
    let value = Value::from(value);
    value.do_func()
}

// Esto se puede implementar en la runtime para facilitar su uso,
// convirtiendo al tipo original.
fn runtime_wrapper(value: Value) {
    plugin_func(value.into());
}
```

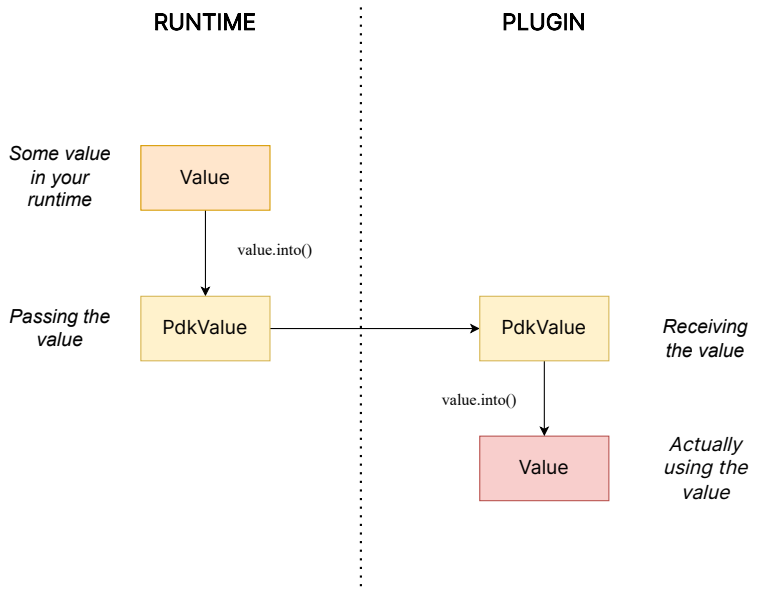

Figura C.3: Comunicación entre runtime y plugins en el PDK.

Es la alternativa más sencilla, pero implica un coste de rendimiento; dos conversiones implican iterar los datos dos veces. Las mediciones de rendimiento del Anexo F indicaron posteriormente que convertir los datos era un 5-10 % de la ejecución del programa.

También tiene un coste de usabilidad; en comparación con tener un único `Value`, es necesario convertir los tipos y posiblemente encapsularlo con una función de más alto nivel (`runtime_wrapper`). Es una tarea relativamente trivial, por lo que se podría automatizar con macros procedurales en Rust, pero esto debería dejarse para el final del proyecto.

C.3. Progreso

Esta última alternativa es la más fácil de implementar para la primera versión [152] y por tanto la que mejor sigue la metodología de "Primero haz que funcione". La segunda versión [153] añadió una versión encapsulada de `halbrown` en `abi_stable` para mejorar el rendimiento, y en el futuro se podría dar soporte también a `beef`.

Anexos D

Problemas con varianza y subtipado

D.1. Entendiendo el problema

Otro problema inesperado tuvo que ver con la *varianza* y *subtipado*. Son dos conceptos de teoría de sistemas de tipos, especialmente conocidos por desarrolladores de lenguajes orientados a objetos como Java o C#. En el caso de Rust solo se da en las *lifetimes*, así que no es tan popular. Lo que lo hace más complicado de tratar es que es completamente implícito: mejora la usabilidad del lenguaje cuando *funciona*; en caso contrario, resulta en intricados errores.

Este tema no se cubre en *The Rust Programming Language* [131], sino en *The Rustnomicon* [112, Subtyping and Variance] y *The Rust Reference* [79, Subtyping and Variance]. También es recomendable consultar el artículo *Diving Deep: implied bounds and variance #25860* [135] o a Gjengset [136] para un formato en vídeo.

Este anexo deriva de los problemas encontrados con el tipo `Value` del Anexo C.Al cambiar los tipos de la librería estándar a los de `abi_stable`, se producían errores de *lifetimes* inexplicables (ver Figura C.1). Estuve bloqueado con dicho problema durante mucho tiempo, así que tras comentárselo a mis mentores, Heinz me ayudó a reproducir el problema de forma mínima. Por alguna razón que todavía desconocíamos, dos tipos supuestamente equivalentes diferían a la hora de compilar:

```rust
use abi_stable::std_types::RCow;
use std::borrow::Cow;

fn cmp_cow<'a, 'b>(left: &Cow<'a, ()>, right: &Cow<'b, ()>) -> bool {
    left == right
}

// Este caso falla en compilación, pero es aparentemente igual
fn cmp_rcow<'a, 'b>(left: &RCow<'a, ()>, right: &RCow<'b, ()>) -> bool {
    left == right
}
```

```
> cargo build
error[E0623]: lifetime mismatch
  --> src/lib.rs:10:10
   |
9  | fn cmp_rcow<'a, 'b>(
   |        left: &RCow<'a, ()>, right: &RCow<'b, ()>) -> bool {
   |              ------------          ------------
   |              |
   |              these two types are declared with
   |              different lifetimes...
10 |     left == right
   |          ^^ ...but data from `left` flows into `right` here

For more information about this error, try `rustc --explain E0623`.
error: could not compile `repro` due to previous error
```

Este tipo de error suele darse en caso de que la *lifetime* de un valor no viva lo suficiente. En particular, el ejemplo de `rustc --explain E0623` es el siguiente. Se tienen dos *lifetimes sin relación entre sí*, `'short` y `'long`. La estructura `Foo` que se pasa como parámetro tiene la *lifetime* `'short`, pero dentro de la función se le intenta asignar una *lifetime* `'long`. Esto es imposible porque el compilador no sabe cuál de los dos tiene un tiempo de vida mayor. Asignarle una *lifetime* que viva más de lo que debe significaría que se podría seguir usando `Foo` después de que `'short` acabe, es decir, después de que `Foo` haya sido destruido. Finalmente, esto causaría inconsistencias en memoria porque nuestra variable de tipo `Foo` ya no existe, pero se está intentando acceder a ella.

```rust
struct Foo<'a> {
    x: &'a isize,
}

```

```
fn bar<'short, 'long>(c: Foo<'short>, l: &'long isize) {
    // Equivalente a asignarle otra lifetime a c
    let c: Foo<'long> = c; // error!
}
```

Solucionarlo es tan simple como indicar que `'short` tiene al menos el mismo tiempo de vida que `'long`. Ahora el compilador tiene garantizado que no se podría dar el caso de que `Foo` es usado después de destruirse:

```
// Notar que ahora `'short` se declara tal que `'short: 'long`
// ('short contiene a 'long, o 'short tiene un tiempo de
// vida mayor que 'long)
fn bar<'short: 'long, 'long>(c: Foo<'short>, l: &'long isize) {
    let c: Foo<'long> = c; // ok!
}
```

Por tanto, uno pensaría que, en el caso de `Value`, el error tiene que ver con el operador `==`. He descrito anteriormente estos errores como *inexplicables* porque, en un principio, una comparación binaria no modifica la *lifetime* de `RCow<'a, T>`. Su *lifetime* no debería importar porque simplemente es una función que compara dos estructuras en un momento dado — no es necesario que una tenga un tiempo de vida mayor que otra.

De todos modos, `==` se delega al *trait* `PartialEq`, así que dediqué tiempo intentando encontrar la diferencia entre su implementación en `Cow<'a, T>` y la de `RCow<'a, T>`. Aparentemente, `RCow<'a, T>` declaraba las *lifetimes* en su implementación de una forma distinta (aunque igualmente válida). Al usar *exactamente lo mismo* que en `Cow<'a, T>`, compilaba correctamente. Alegrado por haber arreglado el error, pero sin saber muy bien aún cómo, abrí un nuevo pull request en `abi_stable` y realicé alguna prueba más [150].

Sin embargo, tras cambiar `left == right` por `left.cmp(right)` en la reproducción inicial, se repetía el mismo problema. Incluso con aparentemente la misma implementación de `Ord` (el *trait* con el método `cmp`). Parecía que, al haber arreglado `PartialEq` el problema había pasado a estar en `Ord`, pero esta vez no había manera de "arreglarlo" porque ambas implementaciones eran

iguales.

No fue hasta que compartí mi problema en un servidor de Discord con expertos en el lenguaje [151] que descubrí que el verdadero problema era un concepto llamado la *varianza* de `RCow<'a, T>`.

D.2. Breve introducción a varianza y subtipado

La *varianza* y *subtipado* son un mecanismo que aumentan la flexibilidad del modelo de memoria de Rust implícitamente. Si `'a: 'b` ("`'a` contiene a `'b`" o "`'a` tiene un tiempo de vida mayor que `'b`"), entonces `'a` es un *subtipo* de `'b`. Por ejemplo, si una función espera un `&'a u8`, también se le podrá pasar un `&'static u8`, porque sabemos que la referencia estática `'static` siempre va a vivir más que `'a`. Asimismo, según una serie de reglas relacionadas con el subtipado, el compilador determinará si un tipo es *covariante*, *invariante* o *contravariante* respecto a sus parámetros (tipos genéricos y *lifetimes*).

La referencia inmutable `&'a T` es covariante respecto a tanto `'a` como `T` y la mutable `&'a mut T` es covariante en `T`, pero es invariante en `'a`, entre otros. Lo importante es que la varianza de un tipo compuesto como un `struct` o `enum` se "hereda", dependiendo de los campos con los que se define. Si todos sus campos tienen el mismo tipo de varianza respecto a un parámetro, entonces la estructura entera también tendrá la misma respecto a ese parámetro. En caso de discrepancias, será invariante. Por ejemplo, si un campo fuera covariante en `'a`, pero otro fuera contravariante o invariante en `'a`, entonces la estructura entera sería invariante en `'a`.

```
use std::cell::UnsafeCell;

struct MiTipo<'a, 'b, T, U: 'a> {
    x: &'a U,                // Esto hace `MiTipo` covariante en 'a, y
                             // lo haría covariante en U también, pero
                             // U se usa más tarde.
    y: *const T,             // Covariante en T
    z: UnsafeCell<&'b f64>, // Invariante en 'b
    w: *mut U,               // Invariante en U, hace que el struct
                             // entero sea invariante.
}
```

`&'a T` tiene sentido que sea covariante en `'a` porque siempre se podría pasar un `&'static T` en su lugar. Sin embargo, esto no es el caso con `&'a mut T`, o podrían producirse ciertos errores de memoria. Por tanto, un tipo invariante será menos flexible que uno covariante porque asume que podrían suceder los mismos errores que con `&'a mut T`. Estas conversiones de *lifetimes* implícitas no podrán ocurrir si el tipo es invariante.

El compilador no es lo suficientemente avanzado como para mencionar la varianza en la descripción de sus errores. Únicamente sabe que si un tipo es invariante, entonces algunos usos de sus *lifetimes* son imposibles. Esto se está mejorando en las futuras versiones [125], pero durante el desarrollo del sistema plugins resultó muy complicado entender qué estaba sucediendo.

Se recomienda consultar los recursos adicionales listados al inicio del anexo, que explican el concepto en detalle y con más ejemplos, dado que es un tema especialmente complicado de entender.

D.3. Resolviendo el problema

Todo acabó reduciéndose a la única diferencia en la implementación del *trait* `Ord`. `RCow<'a, T>` implementa un *trait* llamado `BorrowOwned<'a>` y `Cow<'a, T>` implementa otro llamado `ToOwned`. Ambos *traits* son iguales, excepto que en `BorrowOwned<'a>` se incluye funcionalidad adicional para `abi_stable`. El problema no tiene que ver con esta diferencia en funcionalidad, sino que `BorrowOwned<'a>` es genérico respecto a la *lifetime* `'a`, lo cual no es el caso de `ToOwned`.

Al implementar `Ord`, se tenía que indicar que `T: ToOwned` en `Cow` o `T: BorrowOwned<'a>` en `RCow`. El problema era que al relacionar la *lifetime* `'a` de esta forma, estaba rompiendo una regla que hacía a `RCow` invariante en `'a`, en vez de covariante. Tenemos que `Cow<'a, T>` es covariante en tanto `T` como `'a`, pero nuestro `RCow<'a, T>` es covariante en `T` e invariante en `'a`.

Adicionalmente, esta regla de covarianza específica tampoco está documentada apropiadamente en Rust. Las guías únicamente explican el sistema de herencia

de varianzas, pero no que por asignar una *lifetime* a un *trait* su implementador pasará a ser invariante. Para saber esto, uno tiene que referirse a la guía de desarrollo del compilador, que sí que lo menciona [147]. Esto se ha reportado en el repositorio de *The Rustnomicon* [112], el libro oficial de Rust donde debería haberse incluido [148].

Tras algunos experimentos míos [149], discusión con el autor de `abi_stable` y con ayuda de Heinz, llegamos a un nuevo diseño para `RCow<'a, T>` que no involucraba `BorrowOwned<'a>` y que por tanto era covariante en `'a` [137]. Resultó que realmente, la mayoría de tipos en `abi_stable` eran invariantes, así que esos también tendrían que arreglarse de formas distintas. La implementación final la llevó a cabo el autor de `abi_stable` y el arreglo se incluyó en la versión 0.11 de la librería.

Anexos E

Contribuciones de código abierto

Una de mis partes favoritas del proyecto ha sido poder contribuir tanto a diferentes dependencias de código abierto, así que he mantenido una lista de todas las ocurrencias. Algunas colaboraciones son más importantes que otras, pero sigue siendo una buena métrica de los resultados obtenidos. Esto no incluye aquellos issues o pull requests que:

- No contribuyeron nada (por ejemplo, preguntas o ideas descartadas).
- Fueron repetitivos (por ejemplo, tuve que realizar varios pull requests idénticos en Tremor para lidiar con problemas con Git).

E.1. Contribuciones externas

En esta sección se incluyen todas aquellas contribuciones a repositorios que no tengan relación directa con Tremor.

1. ⊙ *Subtyping and Variance — Trait variance not covered*
   `github.com/rust-lang/nomicon/issues/338`
2. ⊙ `dlerror` **is* thread-safe on some platforms*
   `github.com/szymonwieloch/rust-dlopen/issues/42`
3. ⊙ *Add deprecation notice to the crate* `wasmer-runtime`
   `github.com/wasmerio/wasmer/issues/2539`

4. *Support for* `abi_stable`
   `github.com/oxalica/async-ffi/pull/10`
5. *Cbindgen support*
   `github.com/oxalica/async-ffi/pull/11`
6. *Procedural macro for boilerplate*
   `github.com/oxalica/async-ffi/issues/12`
7. *Generating C bindings*
   `github.com/rodrimati1992/abi_stable_crates/issues/52`
8. *Fix 'carte' typo*
   `github.com/rodrimati1992/abi_stable_crates/pull/55`
9. *Fix some more typos*
   `github.com/rodrimati1992/abi_stable_crates/pull/57`
10. *Add support for* `.keys()` *and* `.values()` *in RHashMap*
    `github.com/rodrimati1992/abi_stable_crates/pull/58`
11. *Implement* `Index` *for slices and vectors*
    `github.com/rodrimati1992/abi_stable_crates/pull/59`
12. *Stable ABI for floating point numbers*
    `github.com/rodrimati1992/abi_stable_crates/issues/60`
13. *Support for* `f32` *and* `f64`
    `github.com/rodrimati1992/abi_stable_crates/pull/61`
14. *Implement* `ROption::as_deref`
    `github.com/rodrimati1992/abi_stable_crates/pull/68`
15. *Implement* `RVec::append`
    `github.com/rodrimati1992/abi_stable_crates/pull/70`
16. *Fix* `R*` *lifetimes*
    `github.com/rodrimati1992/abi_stable_crates/pull/76`
17. *Fix inconsistencies with* `RVec` *in respect to* `Vec`
    `github.com/rodrimati1992/abi_stable_crates/pull/77`
18. *Implement* `ROption::{ok_or,ok_or_else}`
    `github.com/rodrimati1992/abi_stable_crates/pull/82`

19. ⎇ *`RHashMap::raw_entry[_mut]` support*
    `github.com/rodrimati1992/abi_stable_crates/pull/83`

20. ⎇ *Fix hasher*
    `github.com/rodrimati1992/abi_stable_crates/pull/85`

21. ⎇ *Only implement `Default` once*
    `github.com/rodrimati1992/abi_stable_crates/pull/88`

22. ⎇ *Support for `abi_stable`*
    `github.com/simd-lite/simd-json-derive/pull/9`

23. ⊙ *No docs for v0.3.0*
    `github.com/simd-lite/simd-json-derive/issues/10`

24. ⎇ *Add support for StableAbi*
    `github.com/simd-lite/value-trait/pull/14`

25. ⎇ *User friendliness for the win! (close #15)*
    `github.com/simd-lite/value-trait/pull/16`

26. ⎇ *Update `abi_stable` after upstreamed changes*
    `github.com/simd-lite/value-trait/pull/18`

27. ⎇ *Small typo*
    `github.com/nagisa/rust_libloading/pull/94`

28. ⎇ *Fix typo*
    `github.com/szymonwieloch/rust-dlopen/pull/40`

29. ⎇ *Implement `remove_entry`*
    `github.com/Licenser/halfbrown/pull/13`

30. ⎇ *Implement `Clone` and `Debug` for `Iter`*
    `github.com/Licenser/halfbrown/pull/14`

31. ⎇ *Relax constraints*
    `github.com/Licenser/halfbrown/pull/16`

32. ⎇ *Same `Default` constraints*
    `github.com/Licenser/halfbrown/pull/17`

33. ⎇ *Fix `Clone` requirements for `Iter`*
    `github.com/Licenser/halfbrown/pull/18`

E.2. Contribuciones internas

Esta sección lista los pull requests o issues realizadas dentro de los repositorios de Tremor, tanto para el sistema de plugins, como para otras mejoras no relacionadas.

1. *PDK support*
   `github.com/tremor-rs/tremor-runtime/pull/1434`
2. *PDK with a single value*
   `github.com/marioortizmanero/tremor-runtime/pull/11`
3. *Fix* `makefile bench`
   `github.com/tremor-rs/tremor-runtime/pull/1447`
4. *Adding* `abi_stable` *support for* `tremor-script`
   `github.com/marioortizmanero/tremor-runtime/pull/2`
5. *Adding* `abi_stable` *support for* `tremor-runtime`
   `github.com/marioortizmanero/tremor-runtime/pull/1`
6. *Adding* `abi_stable` *support for* `tremor-value`
   `github.com/tremor-rs/tremor-runtime/pull/1303`
7. *Plugin Development Kit: Connectors*
   `github.com/tremor-rs/tremor-runtime/pull/1287`
8. `deny` *statememements in* `lib.rs` *should be enforced in the CI rather than in the code*
   `github.com/tremor-rs/tremor-runtime/issues/1353`
9. *Fix wrong links in getting started*
   `github.com/tremor-rs/tremor-www/pull/72`
10. *Redirect* `docs.tremor.rs` *to* `www.tremor.rs/docs`
    `github.com/tremor-rs/tremor-www/issues/73`
11. *Links pinned to 0.12 don't work*
    `github.com/tremor-rs/tremor-www/pull/186`
12. *Small fix in code snippet*
    `github.com/tremor-rs/tremor-www/pull/187`

13. ⊙ *No margins in benchmark page*
`github.com/tremor-rs/tremor-www/issues/195`

E.3. Otras contribuciones

Otro logro del que me siento extrañamente orgulloso es de accidentalmente romper el mismo compilador de Rust, como se ve en la Figura E.1. El error ya había sido reportado hace unos meses, pero los intentos de arreglarlo parecían haber fallado, así que dejé un comentario indicando cómo me había ocurrido a mí. Debería estar arreglado en la siguiente versión del compilador, y no me ha vuelto a ocurrir desde entonces [138]

```
thread 'rustc' panicked at 'called `Option::unwrap()` on a `None` value', /rustc/02072b482a8b5357f7fb5e563
7444ae30e423c40/compiler/rustc_hir/src/definitions.rs:452:14
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace

error: internal compiler error: unexpected panic

note: the compiler unexpectedly panicked. this is a bug.

note: we would appreciate a bug report: https://github.com/rust-lang/rust/issues/new?labels=C-bug%2C+I-ICE
%2C+T-compiler&template=ice.md

note: rustc 1.58.0 (02072b482 2022-01-11) running on x86_64-unknown-linux-gnu

note: compiler flags: -C embed-bitcode=no -C debuginfo=2 -C incremental -C target-feature=+avx,+avx2,+sse4
.2 --crate-type staticlib --crate-type cdylib --crate-type rlib

note: some of the compiler flags provided by cargo are hidden

query stack during panic:
#0 [evaluate_obligation] evaluating trait selection obligation `for<'r> registry::custom_fn::CustomFn<'r>:
 core::marker::Sync`
#1 [typeck] type-checking `ast::raw::<impl at tremor-script/src/ast/raw.rs:2373:1: 2429:2>::up`
end of query stack
```

Figura E.1: Error de compilación de `rustc`, relacionado con la compilación incremental y arreglado ya en una futura versión.

Adicionalmente, el programa *LFX Mentorship* finaliza con un evento en enero en el que los participantes que quieran pueden mostrar su trabajo una vez terminado [141]. Consiste en realizar una presentación de 15 minutos en la que explican su experiencia, lo cual es especialmente útil para aquellos que quieran unirse al programa en el futuro. No es lo suficientemente larga como para entrar en detalles de implementación, lo que la hace una buena introducción al proyecto: `https://youtu.be/htLCyqY0kt0?t=3166`.

Anexos F

Pruebas de rendimiento

Para medir y analizar la degradación de rendimiento introducida por el sistema de plugins, fue necesario realizar diversas pruebas. Se ejecutó Tremor con la configuración `passthrough`, que simplemente reenvía todos los eventos que recibe; no es necesario el envío de paquetes ni transformaciones más complejas. Aunque esto simplifica el proceso considerablemente, una posible mejora sería probar con casos de uso más cercanos a lo real, involucrando el envío de paquetes TCP, y filtros y transformaciones de los eventos, por ejemplo. Sin embargo, esto no fue posible por no haberse implementado aún ningún conector más avanzado al realizarse las pruebas.

Inicialmente, los experimentos se ejecutaban sobre el mismo portátil donde desarrollaba el proyecto, un Dell Vostro 5481 con un Intel i5-8265U, 16GB de RAM a 2667 MHz, un SSD de 256GB y Arch Linux con el kernel 5.18 de 64 bits. Sin embargo, el equipo de Tremor ofreció una máquina dedicada a *benchmarking* con un Intel Xeon E-2278G, 32GB de RAM a 2667 MHz y Ubuntu 20.04 con el kernel 5.4 de 64 bits, que permitía ejecutarlos más rápidamente y con mayor estabilidad. Se puede observar la diferencia entre las máquinas en la Figura F.2. También se incluye el script final usado para las pruebas, en la Figura F.6 y la Figura F.7.

Este anexo es autocontenido e incluye las pruebas más relevantes para la memoria, pero existen más detalles en el último artículo de *NullDeref*, todavía no publicado: `https://github.com/marioortizmanero/nullderef.com/tree/plugin-end/content/blog/plugin-end`.

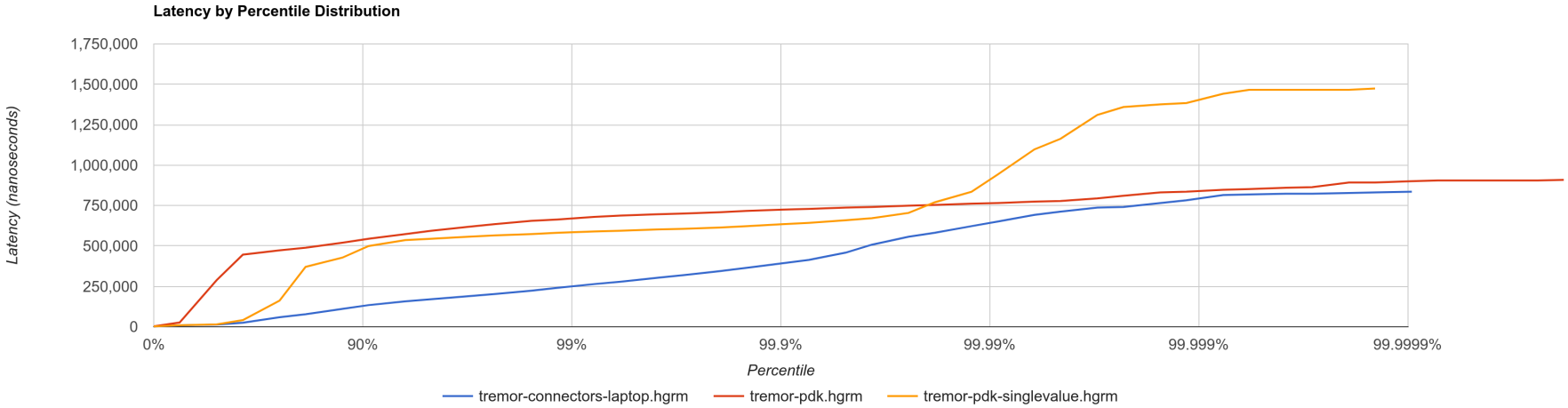

Figura F.1: Histograma con dos diferentes versiones del sistema de plugins, mejorando la latencia iterativamente. Concretamente, la mejora consiste en usar un único `Value`, en lugar de la copia `PdkValue` (ver Anexo C). La línea marcada como `tremor-connectors-laptop` es la rama original de Tremor. Basándose en el rendimiento puro (*throughput*), la primera versión reduce el rendimiento un 35 %, y la segunda un 30 %.

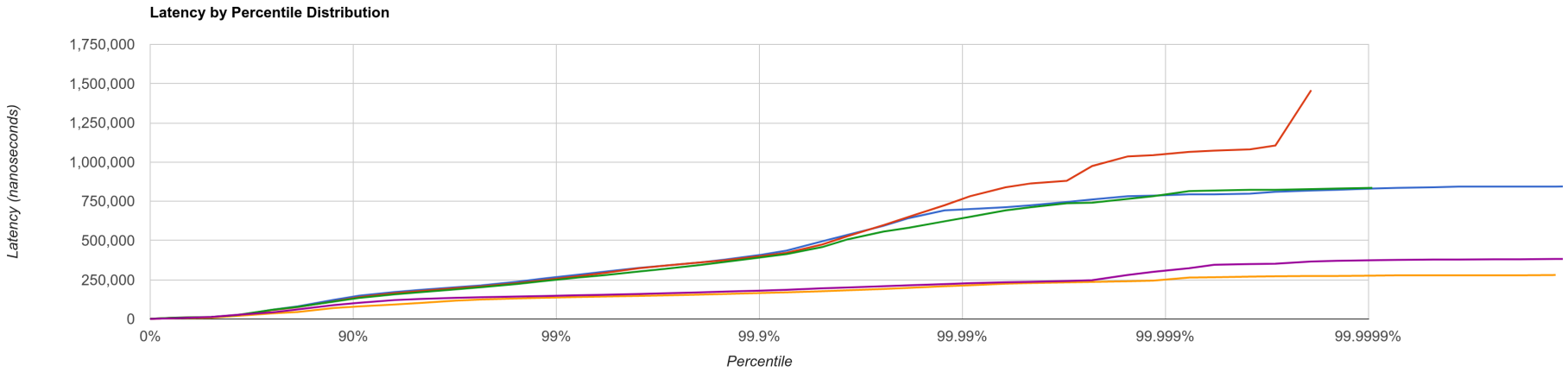

Figura F.2: Las dos líneas inferiores son el mismo programa ejecutado en el servidor dedicado para medir su varianza. Las siguientes dos sobre estas es lo mismo con el portátil; la diferencia en varianza no es tan perceptible como se esperaba. Una mejora importante en la fiabilidad se consiguió realizando varias ejecuciones de calentamiento previas a las mediciones, como se puede ver en la diferencia de la línea superior, el portátil sin calentamiento, con las otras dos del portátil.

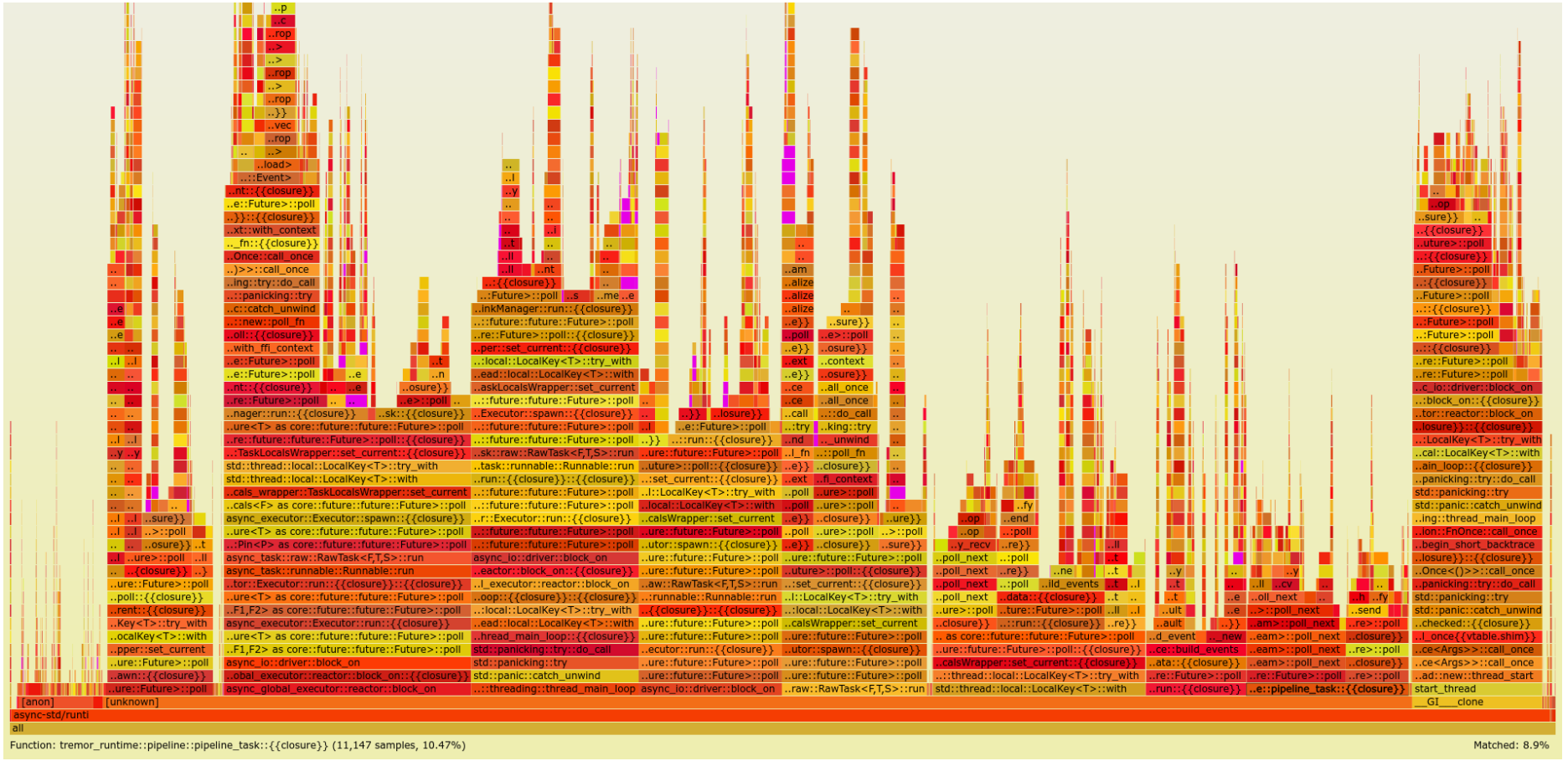

Figura F.3: *Flamegraph* con los porcentajes de ejecución de Tremor original, resaltando en rosa las conversiones de tipos realizadas.

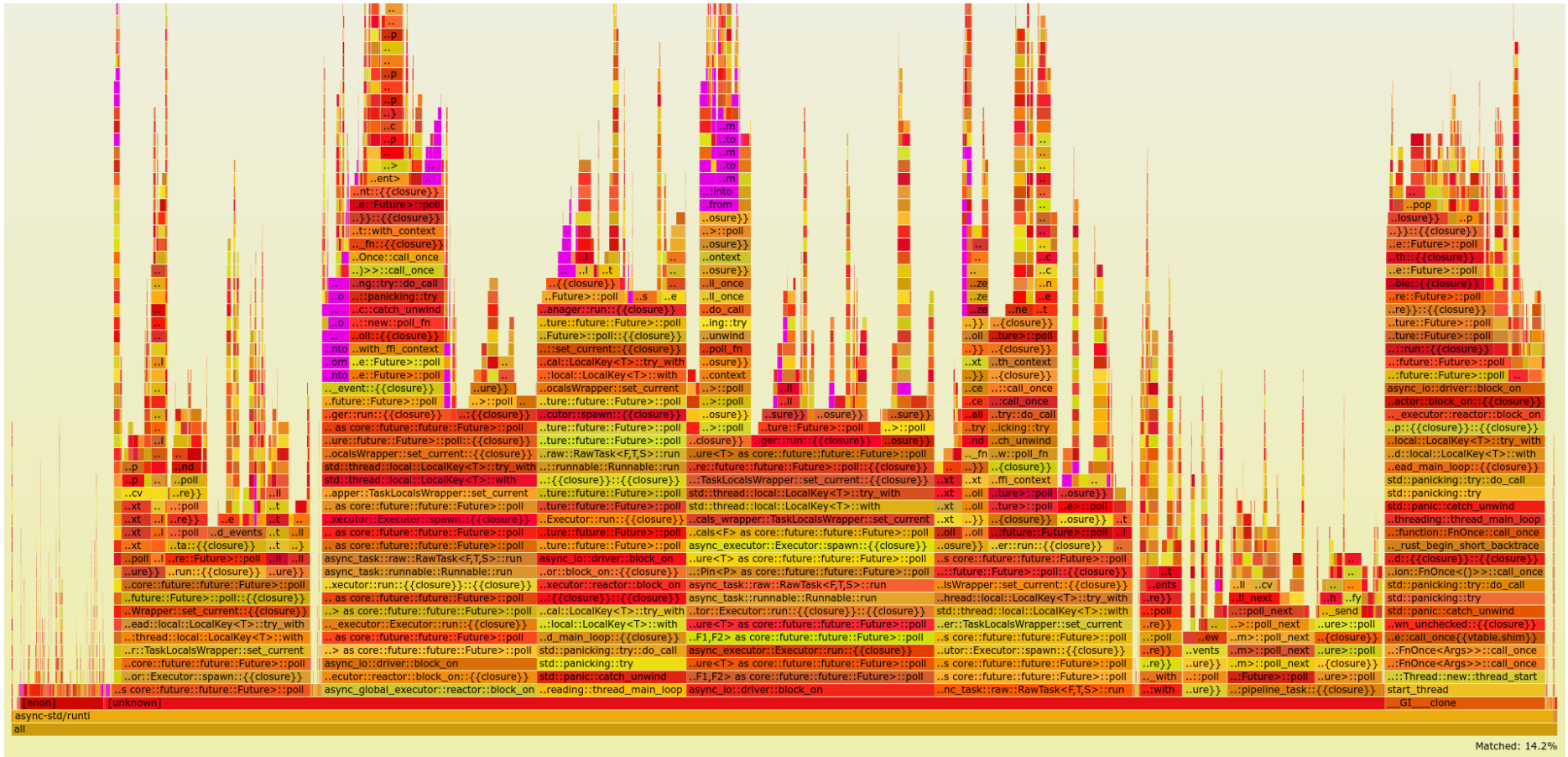

Figura F.4: *Flamegraph* una vez implementado el sistema de plugins. Las conversiones son más visibles ahora, puesto que es necesario para comunicarse entre runtime y plugins. Esta gráfica visualiza la degradación de rendimiento causada por conversiones de la librería estándar a `abi_stable` y viceversa.

```
Samples: 115K of event 'cycles', Event count (approx.): 225865701211
  Children      Self  Command          Shared Object         Symbol
+   72.14%     0.00%  async-std/runti  [unknown]             [k] 0xffffffffffffffff
+   51.04%     0.10%  async-std/runti  tremor-pdk            [.] async_io::driver::block_on
+   42.08%     0.00%  async-std/runti  tremor-pdk            [.] <futures_lite::future::Or<F1,F2> as core::future::future::Future>::poll (inlined)
+   41.63%     0.08%  async-std/runti  tremor-pdk            [.] <core::future::from_generator::GenFuture<T> as core::future::future::Future>::poll
+   41.61%     0.00%  async-std/runti  tremor-pdk            [.] async_executor::Executor::run::{{closure}} (inlined)
+   41.59%     0.00%  async-std/runti  tremor-pdk            [.] <futures_lite::future::Or<F1,F2> as core::future::future::Future>::poll (inlined)
+   41.59%     0.00%  async-std/runti  tremor-pdk            [.] async_executor::Executor::run::{{closure}}::{{closure}} (inlined)
+   38.03%     0.00%  async-std/runti  tremor-pdk            [.] async_global_executor::reactor::block_on
+   38.03%     0.00%  async-std/runti  tremor-pdk            [.] async_global_executor::reactor::block_on::{{closure}} (inlined)
+   33.40%     0.00%  async-std/runti  tremor-pdk            [.] async_task::runnable::Runnable::run (inlined)
+   26.65%     0.00%  async-std/runti  tremor-pdk            [.] async_global_executor::threading::thread_main_loop
+   26.64%     0.00%  async-std/runti  tremor-pdk            [.] async_global_executor::threading::thread_main_loop::{{closure}} (inlined)
+   26.64%     0.00%  async-std/runti  tremor-pdk            [.] async_global_executor::threading::thread_main_loop::{{closure}}::{{closure}} (inlined)
+   26.64%     0.00%  async-std/runti  tremor-pdk            [.] std::panic::catch_unwind (inlined)
+   26.64%     0.00%  async-std/runti  tremor-pdk            [.] std::panicking::try (inlined)
+   26.64%     0.00%  async-std/runti  tremor-pdk            [.] std::panicking::try::do_call (inlined)
+   26.64%     0.00%  async-std/runti  tremor-pdk            [.] std::thread::local::LocalKey<T>::with (inlined)
+   26.64%     0.00%  async-std/runti  tremor-pdk            [.] std::thread::local::LocalKey<T>::try_with (inlined)
+   26.07%     0.00%  async-std/runti  tremor-pdk            [.] tremor_runtime::connectors::sink::SinkManager::run::{{closure}} (inlined)
+   26.00%     0.01%  async-std/runti  tremor-pdk            [.] <core::future::from_generator::GenFuture<T> as core::future::future::Future>::poll
+   26.00%     0.00%  async-std/runti  tremor-pdk            [.] async_executor::Executor::spawn::{{closure}} (inlined)
+   25.99%     0.00%  async-std/runti  tremor-pdk            [.] std::thread::local::LocalKey<T>::with
+   25.99%     0.00%  async-std/runti  tremor-pdk            [.] <async_std::task::builder::SupportTaskLocals<F> as core::future::future::Future>::poll (inlined)
+   25.99%     0.00%  async-std/runti  tremor-pdk            [.] async_std::task::task_locals_wrapper::TaskLocalsWrapper::set_current (inlined)
+   25.99%     0.00%  async-std/runti  tremor-pdk            [.] std::thread::local::LocalKey<T>::try_with (inlined)
+   25.99%     0.00%  async-std/runti  tremor-pdk            [.] async_std::task::task_locals_wrapper::TaskLocalsWrapper::set_current::{{closure}} (inlined)
+   25.99%     0.00%  async-std/runti  tremor-pdk            [.] <async_std::task::builder::SupportTaskLocals<F> as core::future::future::Future>::poll::{{closure}} (inlined)
+   25.99%     0.00%  async-std/runti  tremor-pdk            [.] <core::future::from_generator::GenFuture<T> as core::future::future::Future>::poll (inlined)
+   25.07%     0.00%  async-std/runti  tremor-pdk            [.] tremor_runtime::connectors::source::SourceManager::run::{{closure}} (inlined)
+   23.96%     0.00%  async-std/runti  tremor-pdk            [.] <core::future::from_generator::GenFuture<T> as core::future::future::Future>::poll (inlined)
+   22.73%     0.01%  async-std/runti  tremor-pdk            [.] std::thread::local::LocalKey<T>::with
+   22.71%     0.00%  async-std/runti  tremor-pdk            [.] std::thread::local::LocalKey<T>::try_with (inlined)
+   22.70%     0.00%  async-std/runti  tremor-pdk            [.] async_std::task::task_locals_wrapper::TaskLocalsWrapper::set_current::{{closure}} (inlined)
+   22.70%     0.00%  async-std/runti  tremor-pdk            [.] <async_std::task::builder::SupportTaskLocals<F> as core::future::future::Future>::poll::{{closure}} (inlined)
+   22.70%     0.00%  async-std/runti  tremor-pdk            [.] <core::future::from_generator::GenFuture<T> as core::future::future::Future>::poll (inlined)
+   22.50%     0.01%  async-std/runti  tremor-pdk            [.] async_task::raw::RawTask<F,T,S>::run
+   22.49%     0.00%  async-std/runti  tremor-pdk            [.] <core::pin::Pin<P> as core::future::future::Future>::poll (inlined)
+   20.31%     0.00%  async-std/runti  tremor-pdk            [.] tremor_runtime::pipeline::pipeline_task::{{closure}} (inlined)
+   19.67%     0.02%  async-std/runti  tremor-pdk            [.] <core::future::from_generator::GenFuture<T> as core::future::future::Future>::poll
+   19.67%     0.00%  async-std/runti  tremor-pdk            [.] async_executor::Executor::spawn::{{closure}} (inlined)
+   19.65%     0.00%  async-std/runti  tremor-pdk            [.] <async_std::task::builder::SupportTaskLocals<F> as core::future::future::Future>::poll (inlined)
+   19.65%     0.00%  async-std/runti  tremor-pdk            [.] async_std::task::task_locals_wrapper::TaskLocalsWrapper::set_current (inlined)
+   18.70%     0.00%  async-std/runti  libc.so.6             [.] __GI___clone (inlined)
+   18.70%     0.00%  async-std/runti  libc.so.6             [.] start_thread
+   18.70%     0.00%  async-std/runti  tremor-pdk            [.] std::sys::unix::thread::Thread::new::thread_start
Tip: To add Node.js USDT(User-Level Statically Defined Tracing): perf buildid-cache --add `which node`
```

Figura F.5: También se utilizó la herramienta `perf` para visualizar las secciones del programa más frecuentes en su ejecución. El comando completo es `perf report --no-children -i FILE`. El stack de programación asíncrona introduce gran cantidad de ruido, pero sigue pudiéndose distinguir por ejemplo `SinkManager::run` (ver Anexo B), o el uso de `panic::catch_unwind` para pánicos seguros (ver Sección 5.5).

```
#!/usr/bin/env bash

set -e

# Both paths must be absolute, and not use `~` or similars. Otherwise
# it won't work when ran as root.
TREMOR_DIR=/<fill-yours>/tremor-runtime
OUTPUT_DIR=/<fill-yours>/results
WARMUP_ROUNDS=0
BINARIES="tremor-connectors tremor-pdk"
BENCHMARKS="passthrough"

# Working with paths easily
get_dir() {
    dir="$TREMOR_DIR/tremor-cli/tests/bench/$1"
}
get_bin() {
    bin="../../../../target/release/$1"
}

# Validation step to avoid waiting for the benchmarks only for one of
# them to be incorrectly configured.
for bin in flamegraph; do
    if ! command -v "$bin" > /dev/null; then
        echo "ERROR: Binary $bin not available"
        exit 1
    fi
done
for name in $VALUES; do
    get_bin "$name"

    if ! [ -f "$bin" ] || ! [ -x "$bin" ]; then
        echo "ERROR: Binary $bin doesn't exist"
        exit 1
    fi
done
for name in $BENCHMARKS; do
    get_dir "$name"

    if ! [ -d "$dir" ]; then
        echo "ERROR: Benchmark $dir doesn't exist"
        exit 1
    fi
done
echo ">> Verification step passed"

# Setup
mkdir -p "$OUTPUT_DIR"
export TREMOR_PATH=../../../../tremor-script/lib:../../lib/
```

Figura F.6: Script usado para realizar las pruebas de rendimiento (parte 1/1).

```
# Finally running the benchmarks
for bench in $BENCHMARKS; do
    # Benchmarks only work if you're in the same directory, and then
    # always use relative paths.
    get_dir "$name"
    cd "$dir"

    # Warmup round, ran alternately for accuracy
    for i in $(seq $WARMUP_ROUNDS); do
        for name in $BINARIES; do
            get_bin "$name"

            echo ">> Warming up with $name ($i/$WARMUP_ROUNDS)"
            "$bin" test bench . > /dev/null
        done
    done

    for name in $BINARIES; do
        get_bin "$name"

        echo ">> Benchmarking $name"
        "$bin" test bench . > "$OUTPUT_DIR/${name}.hgrm"

        echo ">> Creating flamegraph for $name"
        flamegraph "$bin" test bench . \
            > "$OUTPUT_DIR/${name}-flamegraph.hgrm"
        mv flamegraph.svg "$OUTPUT_DIR/$name-flamegraph.svg"
        mv perf.data "$OUTPUT_DIR/$name-perf.data"
    done
done
```

Figura F.7: Script usado para realizar las pruebas de rendimiento (parte 2/2).

Anexos G

Fases de desarrollo

Este anexo lista las fases en las que se llevó a cabo el proyecto y las horas invertidas en cada una de ellas. Se incluye el periodo en el que se realizaron, comenzando en abril de 2021 con la propuesta a Tremor. Una vez aceptado, el inicio oficial se dio en agosto. Adicionalmente, junto a sus descripciones se añaden uno o más enlaces relacionados con la tarea.

Tabla G.1: Fases de desarrollo del proyecto

Fase	**Horas**	**Periodo**
Propuesta a Tremor: incluye una introducción sobre quién soy, qué proyecto quiero hacer y una breve planificación de la metodología a seguir. *https://nullderef.com/blog/gsoc-proposal/*	6	Abr. '21
Investigación inicial de las tecnologías disponibles para el sistema de plugins y discusión con el equipo. Esto también formó parte de la propuesta, aunque de forma no oficial. *https://nullderef.com/blog/plugin-tech/*	36	Abr. '21 – May. '21
Aceptación del proyecto. Introducción a Tremor y a su equipo. *https://nullderef.com/blog/plugin-start/*	8	Ago. '21

Tabla G.1: Fases de desarrollo del proyecto

Fase	**Horas**	**Periodo**
Implementación de prototipos para el sistema de plugins, y medidas de rendimiento iniciales. También incluye otros experimentos menores para encontrar el mejor método para tener programación asíncrona o genéricos. *https://github.com/marioortizmanero/pdk-experiments* *https://nullderef.com/blog/plugin-start/* *https://nullderef.com/blog/plugin-dynload/*	51	Ago. '21 – Feb. '22
Investigación del funcionamiento interno de Tremor. Diseño de diagramas de secuencia. *https://nullderef.com/blog/plugin-dynload/*	13	Sep. '21
Investigación en detalle de cargado dinámico en Rust. *https://nullderef.com/blog/plugin-dynload/*	22	Sep.'21 – Oct. '21
Aprendizaje de la librería `abi_stable`. *https://nullderef.com/blog/plugin-abi-stable/*	17	Oct. '21 – Nov. '21
Soporte de `abi_stable` para programación asíncrona con `async_ffi`. *https://github.com/oxalica/async-ffi/pull/10*	15	Nov. '21 – Ene. '22
Primer diseño e implementación del sistema de plugins. *https://github.com/tremor-rs/tremor-runtime/pull/1434*	64	Oct. '21 – Abr. '22

Tabla G.1: Fases de desarrollo del proyecto

Fase	**Horas**	**Periodo**
Mediciones de rendimiento. *https://github.com/marioortizmanero/nullderef.com/pull/54 (artículo aún no publicado)*	14	Ene. '22 – Jun. '22
Resolución de los problemas de covarianza y subtipado. *https://github.com/rodrimati1992/abi_stable_crates/issues/75*	26	Ene. '22 – Mar. '22
Añadir soporte de la librería `halfbrown` para `abi_stable`. *https://github.com/rodrimati1992/abi_stable_crates/pull/83*	19	Mar. '22 – Jun. '22
Segunda versión del sistema de plugins con las dos mejoras anteriores de rendimiento. *https://github.com/tremor-rs/tremor-runtime/pull/1597*	26	May. '22 – Jun. '22
Documentación final para que Tremor pueda continuar con el desarrollo del proyecto. *https://github.com/tremor-rs/tremor-runtime/pull/1597* *https://github.com/marioortizmanero/tremor-runtime/projects/1*	6	Jun. '22
Memoria del Trabajo de Fin de Grado.	54	May. '22 – Jun. '22
Total	**379**	**Abr. '21 – Jun. '22**