Universidad Zaragoza

Trabajo Fin de Grado

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

Autor:

Mario Ortiz Manero

Director:

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 dejarse la piel para que pueda estudiar.

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 a Irlanda y con este documento. Mis mentores de Tremor tomaron un papel fundamental, no solo dándome consejo para el proyecto, sino también para mi carrera profesional y mi vida.

Finalmente, agradecer a todas las organizaciones que han hecho esto posible. A la Fundación de Linux por ofrecer los medios. A Wayfair por apostar sus fondos en código abierto y 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

TODO, en castellano

ABSTRACT

TODO, en inglés

Índice

Capítulo 1

Introducción

1.1. Contexto

Este proyecto se ha realizado en colaboración con *Tremor*[1], un sistema de procesado de eventos de alto rendimiento, escrito en el lenguaje de programación *Rust*[2]. Tremor es un programa de código abierto bajo la fundación *Cloud Native Computing Foundation (CNCF)*[3], que es también parte de la organización *Linux Foundation (LFX)*[4].

Formalmente, el trabajo se ha llevado a cabo gracias a la iniciativa *LFX Mentorship*, con el título "CNCF – Tremor: Add plugin support for tremor (PDK)"[5][6][7]. Esta iniciativa promueve el aprendizaje de desarrolladores de código abierto, proporcionando una plataforma transparente y facilitando un sistema de pagos.

Finalmente, *Wayfair* es una empresa estadounidense de comercio digital de muebles y artículos del hogar[8]. Actualmente, ofrece 14 millones de ítems de más de

[1] `https://tremor.rs`
[2] `https://www.rust-lang.org`
[3] `https://www.cncf.io/`
[4] `https://www.linuxfoundation.org/`
[5] Página oficial de la iniciativa: `https://mentorship.lfx.linuxfoundation.org/project/b90f7174-fc53-40bc-b9e2-9905f88c38ff`
[6] *Tracking issue* en GitHub: `https://github.com/tremor-rs/tremor-runtime/issues/791`
[7] RFC en la documentación de Tremor: `https://www.tremor.rs/rfc/accepted/plugin-development-kit/`
[8] `https://www.wayfair.com/`

11.000 proveedores globales [1] y es el principal financiador tanto de Tremor como de este proyecto.

1.2. Objetivo

La tarea a llevar a cabo es la implementación de un sistema de plugins, denominado *Plugin Development Kit (PDK)*, para la base de código ya existente en Tremor.

Esto es una tarea no-trivial, dado que Rust no tiene un *Application Binary Interface (ABI)* estable. Es decir, que si se compila la *runtime* (el binario principal encargado de cargar funcionalidad externa) y los *plugins* (los binarios individuales con la funcionalidad) de forma separada, no hay garantía de que la representación binaria de los datos o la convención de llamada a funciones — entre otros — sea la misma.

Esto implica que *dynamic loading* es imposible de forma segura puramente con Rust, debiéndose recurrir a otro ABI que sí sea estable, como el del lenguaje de programación C. Por tanto, se deben escribir *bindings* (la definición de la interfaz compartida entre runtime y plugins) completas en C y transformar tipos de Rust a C y viceversa cuando se interactúe con plugins.

1.3. Motivación

1.3.1. Tiempos de compilación

Actualmente, el problema más importante en Tremor es sus tiempos de compilación. En un ordenador de gama media de ~600 € como el Dell Vostro 5481, compilar el binario `tremor` desde cero requiere de más de 7 minutos en modo debug. Incluso en el caso de cambios incrementales (una vez las dependencias ya han sido compiladas), hay que esperar unos 10 segundos. Esto no es una buena experiencia de desarrollo e impide que nuevos programadores se unan a la comunidad de Tremor.

Debido a la naturaleza del programa, este problema solo empeorará con el tiempo. Tremor debe tener soporte para un gran número de protocolos (e.g., TCP o UDP), software (e.g., Kafka o PostgreSQL) y codecs (e.g., JSON o YAML). El número de dependencias continuará incrementando hasta que imposibilite la creación de nuevas prestaciones en Tremor.

Los problemas relacionados con tiempos de compilación excesivamente largos no se limitan a Tremor. Es uno de las mayores críticas que recibe Rust y un 61 % de sus usuarios declaran que aún se necesita trabajo para mejorar la situación [2].

1.3.2. Modularidad

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 [3]. También hace posible el desacoplamiento del ejecutable y sus componentes; algunas dependencias tienen un ciclo de versionado más rápido que otras y generalmente es más conveniente actualizar únicamente un plugin, en lugar del programa por completo.

1.3.3. Aprender de otros

Otros proyectos maduros con características similares a las de Tremor, como [4] o [5], llevan beneficiándose de un sistema de plugins desde hace mucho. Informan mejorías en flexibilidad, extensibilidad y facilidad de desarrollo [6][7]. Aunque las desventajas también mencionen un pequeño impacto en el rendimiento y la posibilidad de caer en un *dependency hell*, 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 10 meses, comenzando en agosto de 2021 y terminando en mayo/junio de 2022. Su realización ha sido completamente remota y con horarios muy flexibles. Se usó el servidor de Discord de Tremor[9] como plataforma principal para comunicarse, tanto por texto como por videollamada. Se programó una llamada por semana, en la que explicaba mi progreso y recibía ayuda de mis mentores en caso de que me hubiera quedado atascado en algún momento.

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.

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[10].

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. El sistema de compilación usado es 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.

[9] `https://discord.com/invite/Wjqu5H9rhQ`
[10] `https://github.com/marioortizmanero/tremor-runtime/projects/1`

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*; gran parte de los contenidos de este documento se han obtenido de ahí. Tiene una serie con un total de 5 artículos que entran en gran detalle y suman unas 2 horas adicionales de lectura.

- 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/`.

Capítulo 2

Breve introducción a 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. Este proyecto requiere ser familiar con cómo funciona, por lo que en este capítulo se introducirán los conceptos más básicos necesarios. Sí que se asume conocimiento de lenguajes de propósito general, como C, C++, Python o Java.

Sin embargo, es posible que se omitan algunos conceptos o que algunas explicaciones no sean completamente precisas por razones de simplicidad. [8] es el libro oficial para aprender Rust por completo, pero es una lectura larga y posiblemente demasiado exhaustiva. Para mayor brevedad, se recomienda leer [9], [10] o [11].

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 [12] y [13], respectivamente.

TODO: revisar traducciones del libro o similares para asegurarse de que la terminología es la misma.

2.1. ¿Qué es Rust?

Rust es un lenguaje de programación de sistemas compilado y de propósito general. Su objetivo es maximizar rendimiento y usabilidad, esto último basándose en seguridad integrada en el lenguaje, en vez de en el

2.2. Primeros pasos

Comenzando por el clásico "Hola Mundo", se incluyen algunos ejemplos de cómo es la sintaxis de Rust más básica. El programa se podría ejecutar fácilmente con *Cargo*, el administrador de dependencias oficial, o específicamente con el comando `cargo run`.

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

`main` es nuestra función principal, que invoca al macro *println* para escribir por pantalla. Esto se sabe porque, a diferencia de una llamada a función, la invocación termina con una exclamación (`!`).

Los bloques básicos (`if`, `else`, `while`, `for`) son los mismos que en otros lenguajes, con la introducción de `match`, que permite extraer patrones.

```
fn factorial(i: u64) -> u64 {
    match i {
        0 => 1,
        n => n * factorial(n-1)
    }
}
```

2.3. Tipos de datos básicos

* Struct * Enum * Trait

2.4. 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:

- 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:

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

- `Box<T>`, usado para localizar un tipo `T` en memoria. Incluye el tamaño que ocupa `T`.

– `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.

2.5. 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 es un tipo nuevo, si el programador se olvidase de comprobar errores, el programa no compilaría. 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 puediera 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. Esto se puede invocar con el macro `panic!` o utilidades similares.

2.6. Macros

Rust da soporte para 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 mayor o menor.

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

Los macros procedurales se describen como extensiones del lenguaje. Esencialmente, ejecutan código en la compilación que consume y produce sintaxis de Rust. Consisten en directamente transformar el Árbol de Sintaxis Abstracta (AST) [**procmacrosref**]. 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 de invocación de un macro normal
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:

```
// 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)
    }
}
```

2.7. Programación asíncrona

Como muchos lenguajes modernos, Rust da soporte a la programación asíncrona, un modelo de programación concurrente. Sin entrar en excesivos detalles, esta permite

Capítulo 3

Entendiendo Tremor

3.1. Procesado de Eventos

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)" [14]. Tremor fue creado como una alternativa de alto rendimiento a herramientas como [15] o [16], 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).

[17] y [18] 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. [19] y [20] resumen los avances más recientes en el campo, analizan su evolución, y clasifican sus subáreas. La mayoría de la información teórica en esta sección se extrae de estas fuentes.

3.2. Casos de uso

La Figura 3.1 ilustra uno de los casos de uso más básicos de Tremor:

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

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.

3.3. 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 [21].

- 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 [22].

- Una *pipeline* es una lista de operaciones (transformación, agregación, eliminación, etc) a través de la cual se pueden encaminar los eventos [23]. La Figura 3.2 muestra un ejemplo de una *pipeline*, definida con Troy, su propio lenguaje inspirado en SQL.

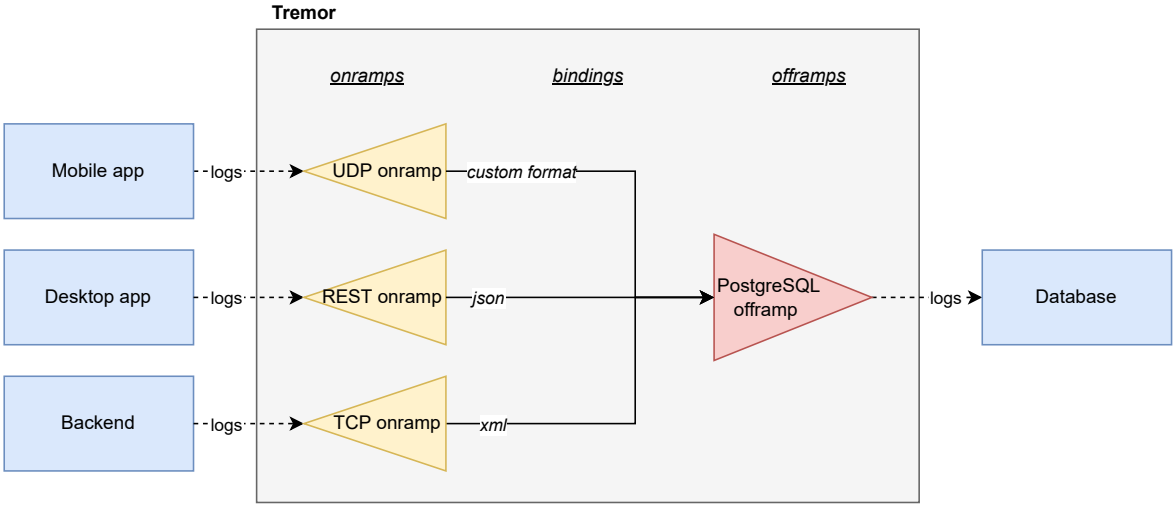

Figura 3.1: Ejemplo de uso básico de Tremor

```
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 3.2: Ejemplo de una *pipeline* definida para Tremor

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, así como incrementar un campo con un contador, reemplazar valores, o reorganizar su estructura.

- *Matching*: búsqueda de partes de los eventos que siguen un patrón en específico (e.g., 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 lo pueda entender Tremor.

- *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.

Para más información sobre Tremor se puede consultar [24], que introduce sus conceptos más básicos y sus posibles usos — o cuándo *no* usarlo, en [25]. [26] lista un total de 32 ejemplos de cómo configurar y emplear el software.

3.4. Conectores

Sin embargo, 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 conectores desde el principio, en lugar de *onramps* u *offramps*, que actualmente están en desuso.

A nivel de implementación, los conectores se definen con el *trait* `Connector`, incluido en la figura 3.3. 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.

Por tanto, 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 pasar tipos complejos (`async`, 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`[2], que implementa una capa de abstracción asíncrona sobre su interfaz de C en `rdkafka-sys`[3].

[2] `https://crates.io/crates/rdkafka`
[3] `https://crates.io/crates/rdkafka-sys`

3.5. Arquitectura interna

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

"[The actor model treats the] actor as the universal primitive of concurrent computation. In response to a message it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received. Actors may modify their own private state, but can only affect each other indirectly through messaging (removing the need for lock-based synchronization)."

No usa un lenguaje (e.g., Erlang) o framework (e.g., `bastion`[4], 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`[5], la runtime asíncrona que usa Tremor:

"An executing asynchronous Rust program consists of a collection of native OS threads, on top of which multiple stackless coroutines are multiplexed. We refer to these as "tasks". Tasks can be named, and provide some built-in support for synchronization."

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

3.6. Detalles de implementación

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

[4] `https://crates.io/crates/bastion`
[5] `https://crates.io/crates/async-std`

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.

3.6.1. Registro

La Figura 3.4 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.

3.6.2. 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 3.5.

3.6.3. Configuración

Una vez haya un conector corriendo, la Figura 3.6 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.

3.6.4. 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 3.3: Simplificación del *trait* `Connector`

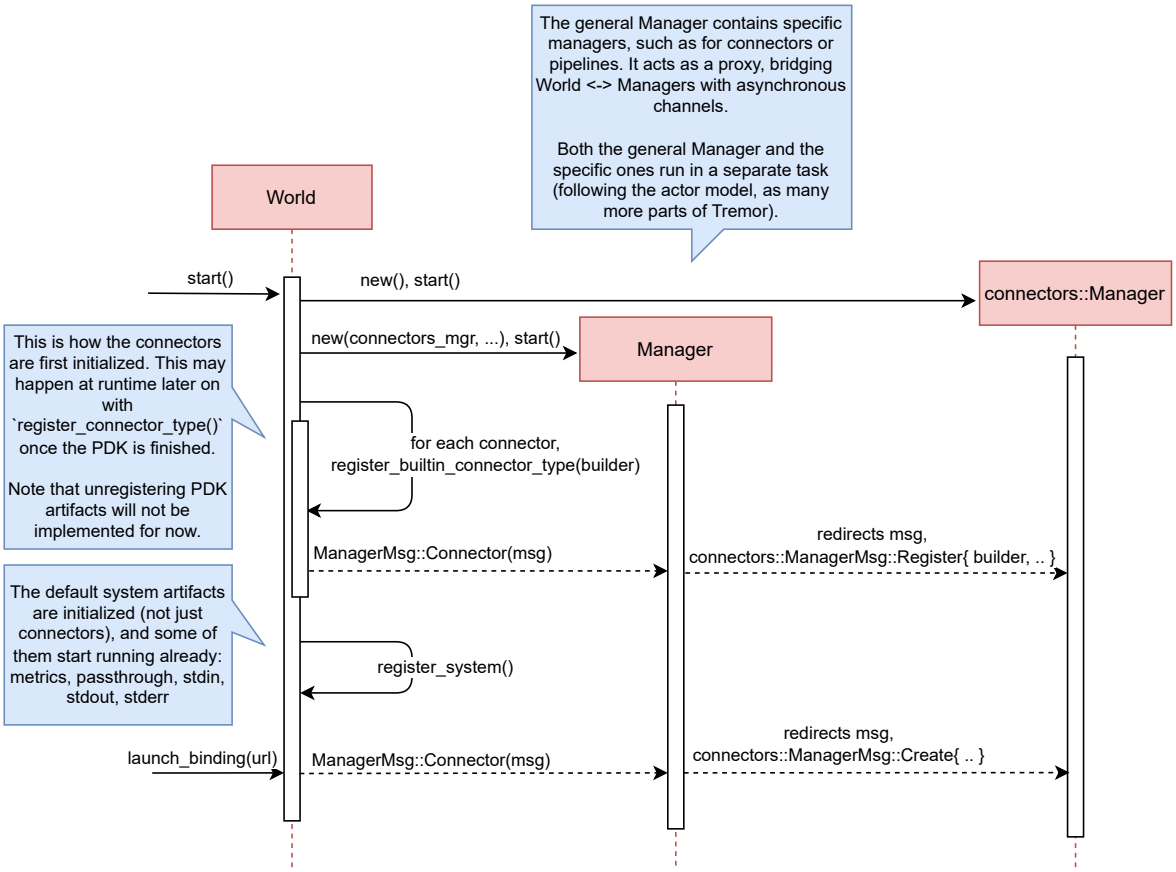

Figura 3.4: Registro de un conector en el programa

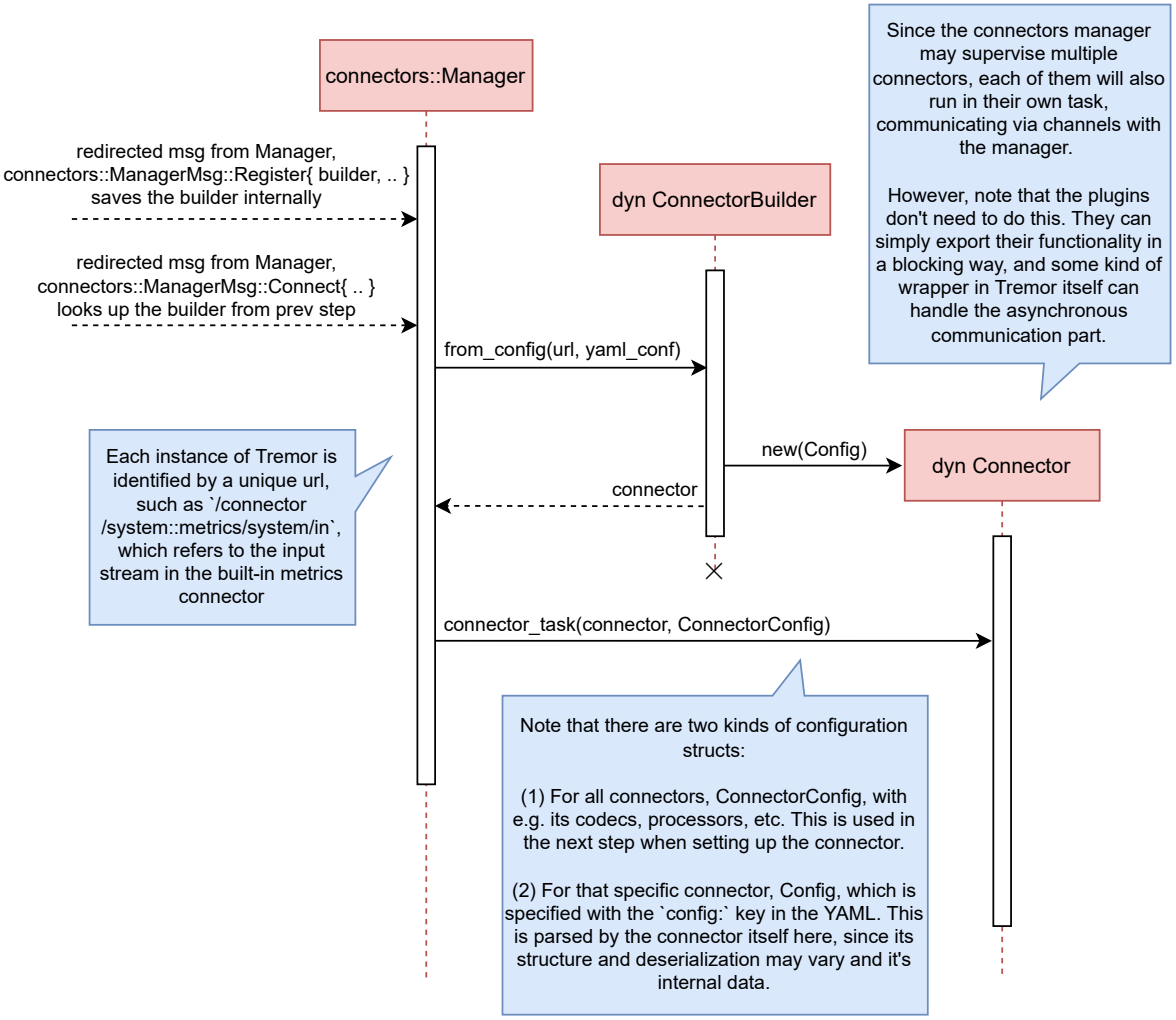

Figura 3.5: Inicialización de un conector en el programa

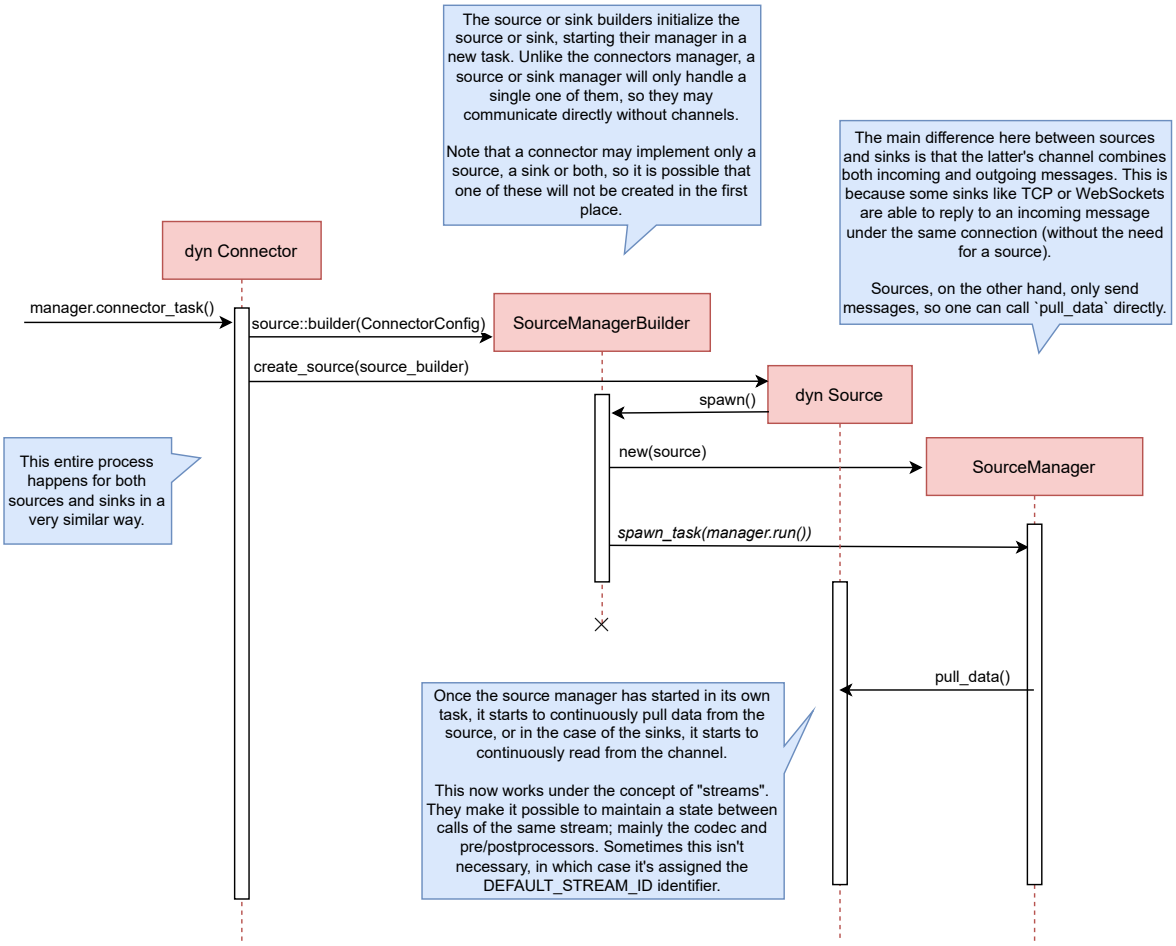

Figura 3.6: Configuración de un conector en el programa

Capítulo 4

Investigación previa

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.

Los requerimientos mínimos a tener en cuenta son los siguientes:

- Debe ser posible añadir y quitar plugins tanto en el inicio del programa como durante su ejecución.
- Disponibilidad y madurez en el ecosistema de Rust.
- Soporte multi-plataforma: Windows, MacOS y Linux.
- No debe tener un impacto excesivo en el rendimiento. Esto significa que los eventos no se pueden copiar en ningún momento.

Y opcionalmente:

- Maximizar la seguridad en lo posible, como se especifica en la sección 4.1.
- Debería ser retro-compatible con el código ya existente, como indica la sección 4.2.

– Minimizar el esfuerzo necesario para reescribir los conectores para el nuevo sistema de plugins.

4.1. Seguridad

4.1.1. Código `unsafe`

Muchas de las tecnologías que se pueden aplicar para un sistema de plugins usan código `unsafe`. Técnicamente, esto no es necesariamente un problema si la implementación está autocontenida y auditada exhaustivamente, pero se pierden algunas garantías que proporcionadas por Rust, incrementando 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[1] — que integraría en Tremor en caso de tener que recurrir a `unsafe`.

4.1.2. Resiliencia a errores

Rust no protege a sus usuarios de *leaks* de memoria. De hecho, es tan sencillo como llamar a `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 *panic*, 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.

[1] `https://github.com/rust-lang/miri`

4.1.3. 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 acceder a memoria externa (evitando acceder 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 [27]. 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[2]. 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 [28][29]. 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 alternativa mejorada a Node y npm sería algo como Deno[3], que es una runtime segura por defecto. Esto es posible gracias a *sandboxing*, y requiere que el desarrollador active manualmente, por ejemplo, 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.

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

[2] `https://www.npmjs.com/`
[3] `https://github.com/denoland/deno`

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 Apacha Kafka, que posiblemente tenga *logs* sensitivos.

4.2. 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.

4.2.1. 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 [30]:

- 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.

- Dar a la estructura un puntero a sus datos en la “segunda versión” (lo cual sería opaco en la “primera versión”).

4.2.2. Evitar errores

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/interfaz/etc para las que fue desarrollado. Después, cuando sean cargados por Tremor, se podrá comprobar su compatibilidad, en vez de romperse de formas misteriosas.

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

Todo tipo de proyectos usan lenguajes interpretados para extender su funcionalidad a tiempo de ejecución, como Python, Ruby, Perl, Bash, o JavaScript. Particularmente, el editor de texto Vim creó su propio lenguaje para poderlo personalizar por completo, Vimscript [31]. Ahora NeoVim, un fork más moderno, está esforzándose por tener Lua como lenguaje de primera clase para su configuración [32]. 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 y únicamente alrededor de 220KB [33]. Algunas implementaciones del lenguaje, como LuaJIT, son extremadamente eficientes y pueden ser viables hasta en escenarios de rendimiento crítico [34]. Adicionalmente, las garantías

de seguridad de Lua son más fuertes que otros lenguajes, dado que no requiere `unsafe` y que incluye una *sandbox* (aunque es "delicado y difícil de configurar correctamente") [35].

Rust dispone de librerías como `rlua`[4], 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. Por desgracia, parece estar semi-abandonada y fue reemplazada por `mlua`[5]. Por lo general, el ecosistema de Lua en Rust no parece lo suficientemente maduro para un proyecto como este; aún queda trabajo para mejorar la estabilidad.

También sería posible usar uno de los lenguajes interpretados creados específicamente para Rust: [36], [37] o [38]. Usarlos posiblemente resulte en código más limpio y simple. Sin embargo, su 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 [39].

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

[40], 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 [41].

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

[4] `https://crates.io/crates/rlua`
[5] `https://crates.io/crates/mlua`

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 de *scripting*).

Existen dos runtimes principales para Rust: [42] y [43]. Ambas son implementaciones competitivas que se enfocan 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 [44].

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, [45] muestra resultados mezclados al realizar pruebas de rendimiento. Depende principalmente del compilador[6] 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 contra código nativo, [46] describe una varianza similar, donde Wasmer es 2.47x más lento y con Wasmtime es 3.28x. En resumen, mientras que WebAssembly es una solución más eficiente que algunos lenguajes de *scripting*, 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 era excelente. Si fuera posible transferir datos 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 [47]. Existen algunas maneras de tratar tipos no triviales como estructuras o enumeraciones:

– A través de la [48]. Esta define un formato binario para codificar y decodificar los nuevos tipos que define: tipos de números más especializados, caracteres

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

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 lenguaje de programación que se esté usando, gracias al formato `witx` [**witx**], como muestra la Figura 4.1.

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 [49][50]. Tras fallar al intentar usarlo, esta opción fue descartada.

– La forma actualmente funcional pero imperfecta, con punteros y memoria compartida. El usuario debe construir y serializar el tipo complejo 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 [51][52], 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 complicado 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 usan programas como Zellij [53], que usa un ejecutable de Wasm en vez de usarlo como una librería. Para cargarlo, lo ejecuta y usa *stdin* y *stdout* para los flujos de datos. Por desgracia, esto también requiere copiar datos, y tiene que descartarse.

4.3.3. eBPF

eBPF es "a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in an operating system kernel" [54]. Sin embargo, de forma similar a WebAssembly, su uso se ha extendido a *user-space*. eBPF define una lista de instrucciones que pueden ejecutarse por una máquina virtual, también como WebAssembly funciona.

```
(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 4.1: Ejemplo de interfaz definida con `witx`.

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`[7] y derivados como `solana_rbpf`[8], 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. Comunicación Inter-Proceso

Otra opción popular para sistemas de plugins es la *Comunicación Inter-Proceso*, que divide el programa en un cliente y un servidor en procesos distintos. El cliente actuaría como runtime y estaría conectado a múltiples servidores que proporcionan la funcionalidad. Se podría comparar con el *Language Server*

[7] `https://crates.io/crates/rbpf`
[8] `https://crates.io/crates/solana_rbpf`

Protocol[9], 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 4.2 y la Figura 4.3, 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 deserialización, dado que los datos se envían en paquetes. Formatos como JSON son los más flexibles, pero otros como [56] son ligeros y tienen mejor rendimiento.

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[10] 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.

[9] `https://microsoft.github.io/language-server-protocol/`
[10] `https://github.com/jarun/nnn`

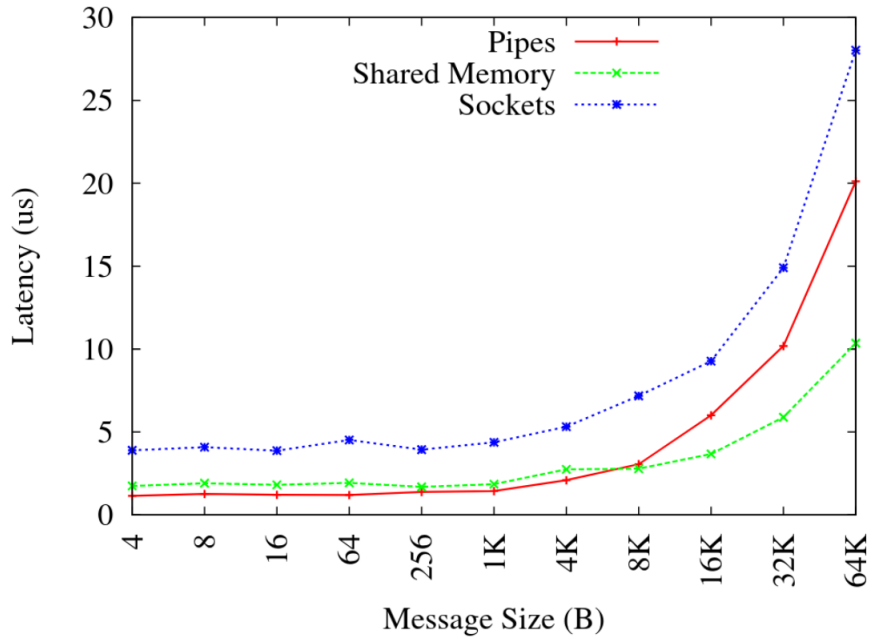

Figura 4.2: Latencia vs. Tamaño de Mensaje [55].

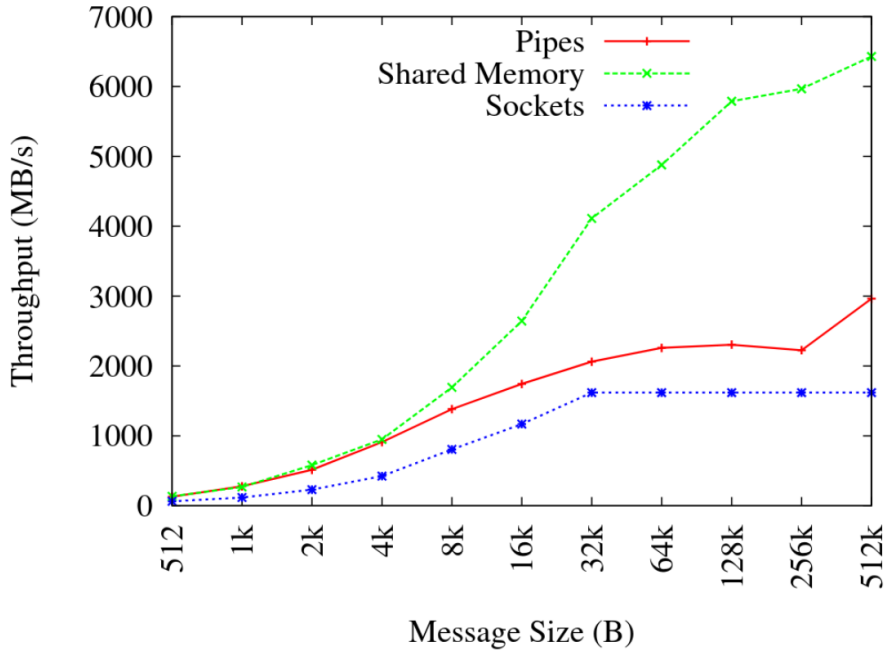

Figura 4.3: Rendimiento vs. Tamaño de Mensaje [55].

La única desventaja es que no parecen haber librerías populares para la funcionalidad genérica de *pipes* (quizá `interprocess`[11] o `ipipe`[12]). 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* [57].

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 posiblemente no sea tan portable como otras soluciones.

Tal y como indican las Figuras 4.2 y 4.3, es el método con mejor rendimiento, ya que no requiere copiar ni transformar datos entre procesos. El único coste adicional es la inicialización de las páginas compartidas en el sistema operativo, que se debe hacer únicamente al principio [58].

Desgraciadamente, el soporte para memoria compartida en Rust es casi inexistente. Las únicas *crates* disponibles son `shared_memory`[13] y `raw_sync`[14], 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 4.3.5). No parece ofrecer nada mejor que el cargado dinámico y por tanto se descarta como opción.

4.3.5. 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

[11] `https://crates.io/crates/interprocess`
[12] `https://crates.io/crates/ipipe`
[13] `https://crates.io/crates/shared_memory`
[14] `https://crates.io/crates/raw_sync`

directamente a recursos en objetos compilados separadamente, aun después de la fase de *linking*, gracias al cargado dinámico. Es una de las opciones más eficientes porque no implica casi ningún coste adicinal tras cargar la librería dinámica.

La *crate* principal para este método es `libloading`[15], aunque también existen las menos conocidas `dlopen`[16] y *crate*sharedlib, con pequeñas diferencias [59]. Todas ellas requieren de uso extensivo de `unsafe`, son complicadas de usar correctamente [60][61], incluyendo sutiles disparidades entre sistemas operativos [62], ni disponen de un mecanismo similar a una *sandbox*. La única manera de mejorar la usabilidad será a través de los macros y herramientas facilitados por algunas librerías.

Estabilidad del ABI

El problema principal con esta alternativa es que Rust carece de un *Application Binary Interface* (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, *estable*), 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 [63][64][65][66]. 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 la misma versión de compilador. Sin embargo, esto es incorrecto por varias razones y fuentes como la referencia oficial no entran en suficiente detalle [67]. 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” [68]

[15] `https://crates.io/crates/libloading`
[16] `https://crates.io/crates/dlopen`

La asumpción anterior incorrecta se basa en que, hasta el momento, Rust no ha implementado ninguna optimización que rompa el ABI entre ejecuciones de un mismo compilador. 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 [69], en cuyo caso el sistema de plugins tendría que reescribirse por completo.

Descubrir esto implicó un cambio de planes y un aumento en la complejidad del proyecto de órdenes de magnitud. 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`[17]. 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, generalmente precedidos por la letra `R`. Por tanto, en vez de `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`[18] o `serde_json`[19]. La Figura 4.6 demuestra parte de la simplificación del código en la Figura 4.5.

Existen algunas alternativas o extensiones de `abi_stable` como `lccc`[20], `safer_ffi`[21] o `cglue`[22] que se tuvieron en cuenta, pero no son soluciones tan completas ni maduras, por lo que no serán incluidas en este proyecto. `abi_stable` se trata de una librería compleja, con más de 50.000 líneas de código en Rust (como referencia, Tremor tiene unas 35.000 líneas), lo cual deberá tenerse

[17] `https://crates.io/crates/abi_stable`
[18] `https://crates.io/crates/crossbeam`
[19] `https://crates.io/crates/serde_json`
[20] `https://github.com/LightningCreations/lccc`
[21] `https://crates.io/crates/safer_ffi`
[22] `https://crates.io/crates/cglue`

```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
// Using C's memory layout with `#[repr]`
#[repr(C)]
pub struct Event {
    // We can't have types from the standard library anymore, only
    // either basic ones...
    pub count: i32,
    // ...or types from C itself
    pub name: *const std::os::raw::char_c,
}

// Using C's calling conventions with `extern "C"`
pub extern "C" fn transform(x: Event) -> i32 {
    println!("Received an event with count {}", x.count);
    x.count
}

// Disabling mangling so that the resource's name is known when
// loading the plugin.
#[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.

en cuenta en la decisión final también.

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 o `mdbook`[23] 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 (e.g., `cargo-expand`). Si está disponible en la variable de entorno `PATH` al ejecutar `cargo`, se podrá invocar al plugin con `cargo expand` también. Es una manera especialmente simple y creativa de usar *pipes* con IPC, dado que usa *stdin* y *stdout* para comunicarse con la runtime.

- `zellij`[24] 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 el ejecutable principal ejecuta el código en WebAssembly, comunicándose con *stdin* y *stdout*.

- `xi`[25] es un editor de texto moderno ahora abandonado. Usa RPC con mensajes JSON para comunicarse con plugins en procesos distintos [70], método también usado en Visual Studio Code [71] o Eclipse [72].

- `bevy`[26] 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 4.3.5 y únicamente usa tipos definidos en Rust.

  Otras fuentes como [73] o [74] también intentan usar cargado dinámico para funcionalidades similares. Este último se trata de `amethyst`[27], el

[23] `https://github.com/rust-lang/mdBook`
[24] `https://github.com/zellij-org/zellij`
[25] `https://github.com/xi-editor/xi-editor`
[26] `https://crates.io/crates/bevy`
[27] `https://crates.io/crates/amethyst`

```rust
// Now we also "derive" the trait StableAbi, i.e., it's implemented
// automatically.
#[repr(C)]
#[derive(StableAbi)]
pub struct Event {
    pub count: i32,
    // We can use abi_stable's types!
    pub name: RStr<'static>,
}

// Using C's calling conventions with `extern "C"`
#[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.

predecesor de `bevy`, que acabó rindiéndose debido a la inestabilidad del ABI [75][76].

4.5. Elección Final

Tras implementar varios sistemas de plugins en miniatura con las tecnologías más prometedoras mencionadas en este capítulo, 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 4.7, pero ninguna de las demás terminaron de cumplir los requisitos de rendimiento establecidos por el equipo de Tremor.

Todas las tecnologías excepto cargado dinámico, eBPF o memoria compartida requieren la 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.

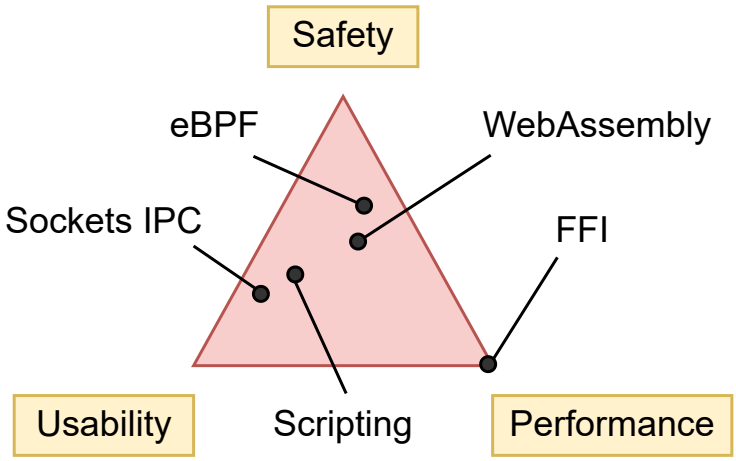

Figura 4.7: Comparación aproximada de los métodos investigados.

Capítulo 5

Implementación

5.1. Prototipado eficiente

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.

TODO: podría mencionar trucos relacionados con Rust en detalle (desactivar algunos warnings, o quitar statements `use`), pero no creo que sea tan importante en este caso y el documento ya es bastante largo.

La metodología fue insipirada por mis mentores, que lo denominaron el “Just Make it Work”, o “Primero que Funcione”. 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:

TODO: alguna traducción de ”just make it workün poco más natural? ”Solo que funcione¿ ”Primero que funcione?

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

TODO: el siguiente párrafo puede no gustarle a alguno de ingeniería del software porque rompe todas las metodologías de desarrollo que tienen, pero fue como sinceramente ocurrió.

El no trabajar con tests es discutible, dado que depende de si el programador prefiere seguir un desarrollo basado en tests. Sin embargo, personalmente no sentí la necesidad de escribir ningún test en mi caso; gracias al sistema fuertemente tipado de Rust, fue principalmente un desarrollo basado en el compilador. Mi progreso se basaba en realizar algunos cambios y posteriormente intentar que los aceptara el compilador, repetidamente. Únicamente avancé a la parte de tests cuando todo parecía funcionar manualmente y estaba lo suficientemente satisfecho con el resultado.

Adicionalmente, las optimizaciones prematuras son la fuente de todos los problemas. No es algo que sea importante aún. 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 que sí que debería preocuparse en escoger un método o tecnología que sea apropiado en términos de rendimiento; por ello se descartó WebAssembly o IPC en el paso anterior. Pero definitivamente el desarrollador debería rendirse en, por ejemplo, evitar una conversión de tipos posiblemente 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 las veces que esto ocurra.

5.2. `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:

5.2.1. Versionado

5.2.2. Cargado de plugins

5.2.3. Exportando un plugin

5.2.4. Gestión de pánicos

5.2.5. 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 `async` con el ABI de C se puede recurrir a la *crate* `async_ffi` [1], cuyo único problema era no tener soporte para `abi_stable`. Al no implementar el *trait* `StableAbi`, no se podía usar en la interfaz del PDK, por lo que

```
// 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
    }
}

// For FFI-safe interfaces there can't be generics involved, so the future is a
// concrete type instead of a trait. This conversion from `Future` to
// `FfiFuture` can be done with `into_ffi`.
fn example() -> FfiFuture<String> {
    async move {
        read_file().await
    }
    .into_ffi()
}
// `FfiFuture<T>` implements `Future<Output = T>`, so it can be awaited as usual
async fn user() {
    example().await
}
```

5.2.6. Seguridad en hilos

5.2.7. Rendimiento

5.3. Conversión al ABI de C

El primer paso en el proceso es declarar toda 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), pero el problema reside en que todos los tipos dentro suyo *también* tendrán que haber sido declarados con dicho atributo. Para ilustrarlo mejor, la estructura más complicada al respecto fue

[1] `https://crates.io/crates/async_ffi`

`Value`, usado para representar datos pseudo-JSON y definido a continuación de forma simplificada:

```
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 puede convertir a:

```
#[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` [2], que lo declara tal que:

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

[2] `https://crates.io/crates/value_trait`

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 [77] A:

```
#[cfg_attr(feature = "abi_stable", repr(C))]
#[cfg_attr(feature = "abi_stable", derive(abi_stable::StableAbi))]
pub enum StaticNode {
    I64(i64),
    U64(u64),
    F64(f64),
    Bool(bool),
    Null,
}
```

El atributo `cfg_attr` se asegura de que la estructura es `#[repr(C)]` únicamente cuando opcionalmente se configure a tiempo de compilación como necesario. De esta forma, el resto de usuarios podrán seguir aprovechándose de las ventajas de rendimiento que puede ofrecer `#[repr(Rust)]`.

5.3.1. Consecuencias del sistema de plugins

Por desgracia, este cambio 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` por `RVec`. La intención de los tipos de `abi_stable` es que sean un reemplazo directo de los de la librería estándar, i.e., su interfaz será la misma:

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

Es un poco más complicado cuando los tipos anteriores se exponían 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 la decisión de 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, e.g., en el bucle principal, sí que podría causar una degradación considerable.

Como indica la sección 5.2.7, las conversiones entre la librería estándar y `abi_stable` son $O(1)$. Esto es dónde la metodología "Primero 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.

5.3.2. 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 A. 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` [3] de `HashMap<K, V>`. 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 otro tipo `Cow`, cuya alternativa en la *crate* `beef` [4] ocupa menos espacio en memoria y ofrece un mejor rendimiento en Tremor.

Ninguno de estos dos tipos tienen soporte dentro de `abi_stable`, y aunque estos tipos estén basados en otros de la librería estándar, la conversión no es directa. Se pueden tomar cuatro posibles alternativas:

Evitar el tipo externo

Basándose en "Primero 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 5.1, especialmente

[3] `https://crates.io/crates/halfbrown`
[4] `https://crates.io/crates/beef`

cuando la funcionalidad extra del tipo externo no está disponible.

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 [78] y difíciles de mantener, ya que tendrán que actualizarme con cada nueva versión de `halfbrown`.

Reimplementar el tipo con el ABI de C desde cero

Similar a la solución anterior, pero con incluso más costoso, dado que también requeriría reimplementar la funcionalidad. 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.

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.

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 para comunicarse entre runtime y plugins, ilustrado en

```
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 5.1: Al intentar evitar los tipos externos se produjeron más de 120 errores de compilación.

la Figura 5.5.

Ya que es un tipo nuevo, no se romperá nada del código ya existente, y únicamente hará falta cambiarlo donde se use la interfaz. Su implementación es sencilla (notar el cambio de nombre a `PdkValue`):

```rust
pub enum PdkValue {
    /// 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, PdkValue>>),
    /// Tipo para datos binarios
    Bytes(Vec<u8>),
}
```

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:

TODO: mejor si omito el código aquí con la conversión, no?

```rust
// 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());
}
```

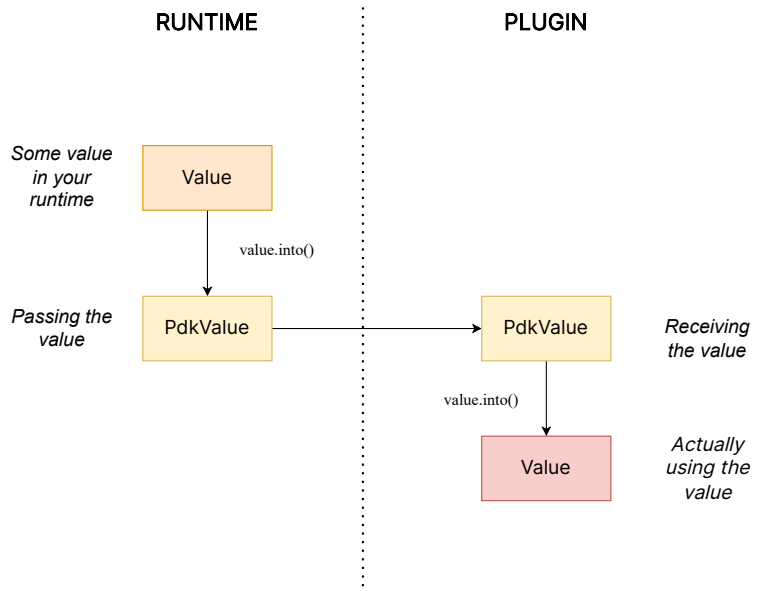

Figura 5.2: Comunicación entre runtime y plugins en el PDK.

Es la alternativa más sencilla, pero implica un coste de rendimiento; dos conversiones implican iterar los datos dos veces. Tras mediciones posteriores, se verificó que convertir los datos era un 5-10% de la ejecución del programa. Es menos de lo esperado, pero sigue sin ser suficiente para Tremor.

También tiene un coste de usabilidad; en comparación con tener un único `Value`, es necesario convertir los tipos y posiblemente crear 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.

En conclusión, esta alternativa es la más fácil de implementar en el corto plazo y por tanto la que mejor sigue "Primero que Funcione". Una vez esté terminado, se puede analizar el rendimiento y optar por una alternativa más eficiente, como reescribir los tipos para el caso de uso específico de Tremor.

5.3.3. Problemas con varianza y subtipado

Otro problema muy importante

5.4. Separación de runtime e interfaz

5.5. Despliegue en producción

5.6. Lecciones aprendidas

* Quizá incluir consejos de los mentores?

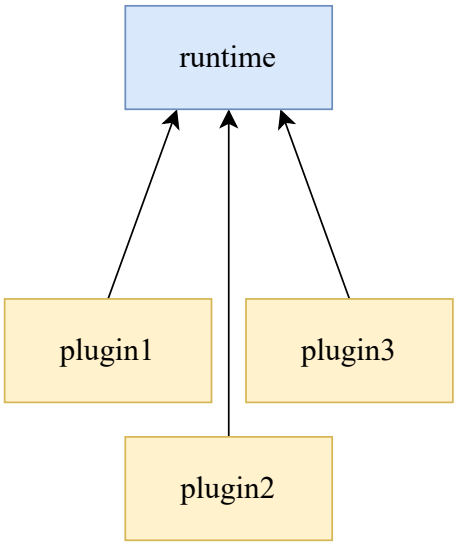

Figura 5.3: Ejemplo de uso de Tremor

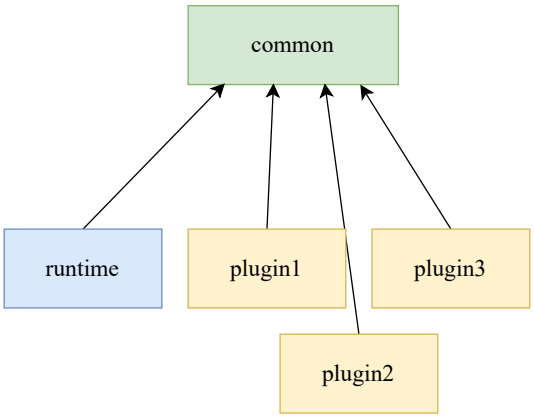

Figura 5.4: Ejemplo de uso de Tremor

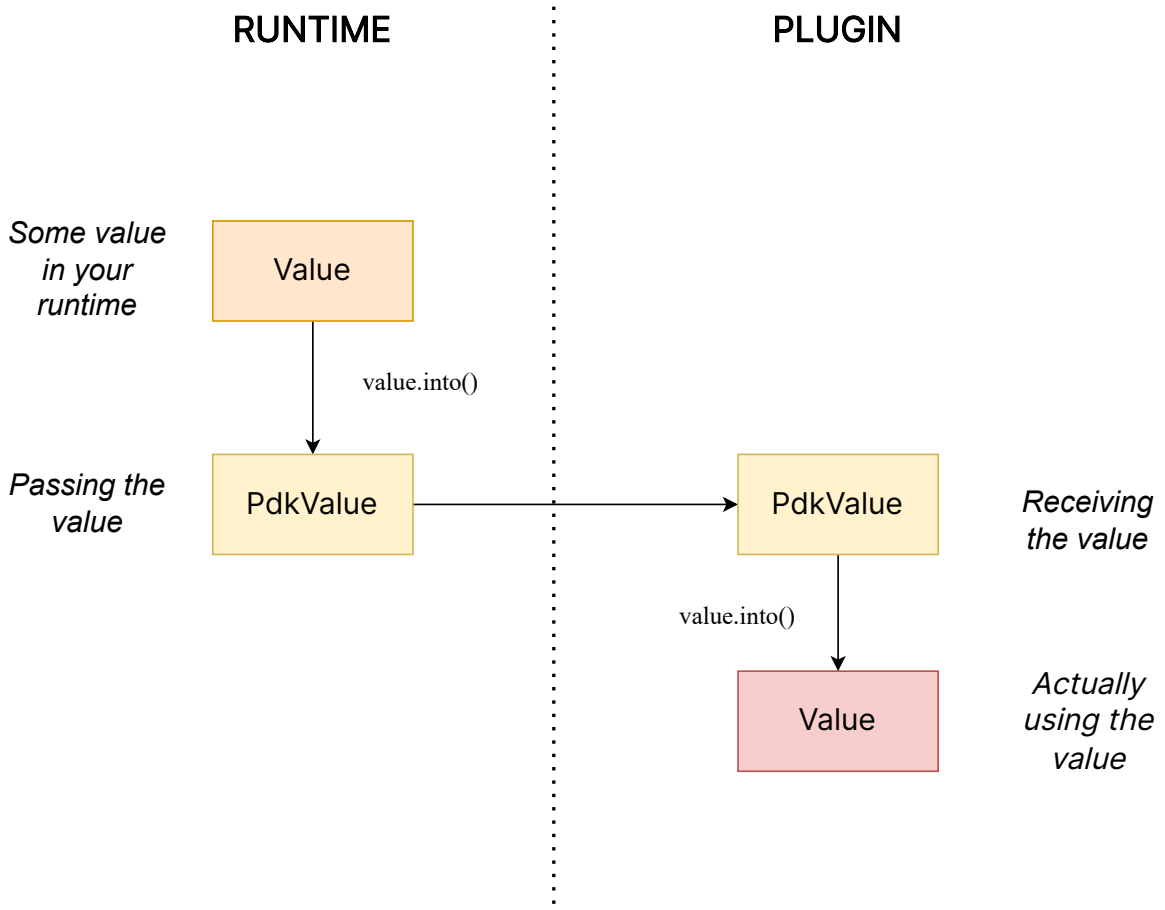

Figura 5.5: Ejemplo de uso de Tremor

Capítulo 6

Conclusiones

TODO

Bibliografía

[1] .

[2] .

[3] .

[4] .

[5] .

[6] .

[7] .

[8] .

[9] .

[10] .

[11] .

[12] .

[13] .

[14] .

[15] .

[16] .

[17] En:

[18] En: ().

[19] En: ().

[20] En:

[21] .

[22] .

[23] .

[24] .

[25] .

[26] .

[27] .

[28] .

[29] .

[30] .
[31] .
[32] .
[33] .
[34] .
[35] .
[36] .
[37] .
[38] .
[39] .
[40] .
[41] .
[42] .
[43] .
[44] .
[45] En:
[46] .
[47] .
[48] .
[49] .
[50] .
[51] .
[52] .
[53] .
[54] .
[55] En: ().
[56] .
[57] .
[58] .
[59] .
[60] .
[61] .
[62] .
[63] .
[64] .
[65] .

[66] .

[67] .

[68] .

[69] .

[70] .

[71] .

[72] .

[73] .

[74] .

[75] .

[76] .

[77] .

[78] .

[79] .

[80] .

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Contribuciones de código abierto

TODO: copiar de blog

–