



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, este documento y todas sus complicaciones. 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

Desarrollar un sistema de plugins en Rust no es una tarea tan sencilla como en el caso de C o C++. Esto no se debe únicamente a los problemas con el ABI, sino también con la inmadurez del lenguaje de programación en comparación.

TODO

ABSTRACT

TODO, en inglés

Índice

1. Introducción	1
1.1. Contexto	1
1.2. Objetivo	2
1.3. Motivación	2
1.3.1. Tiempos de compilación	2
1.3.2. Modularidad	3
1.3.3. Aprender de otros	3
1.4. Metodología	4
1.4.1. Organización	4
1.4.2. Desarrollo	4
1.4.3. Recursos públicos	5
2. Breve guía de Rust	7
2.1. ¿Qué es Rust?	8
2.2. Conceptos básicos	8
2.3. Genéricos y librería estándar	10
2.4. Gestión de errores	11
2.5. Macros	12
2.6. Lifetimes	13
2.7. Unsafe	14
2.8. Programación asíncrona	14
3. Entendiendo Tremor	17
3.1. Procesado de Eventos	17
3.2. Casos de uso	18
3.3. Conceptos básicos	18
3.4. Conectores	21
3.5. Arquitectura interna	22
3.6. Detalles de implementación	22

3.6.1. Registro	23
3.6.2. Inicialización	23
3.6.3. Configuración	24
3.6.4. Notas adicionales	24
4. Investigación previa	31
4.1. Seguridad	32
4.1.1. Código unsafe	32
4.1.2. Resiliencia a errores	32
4.1.3. Ejecución de código remota a través de plugins	33
4.2. Retro-compatibilidad	34
4.2.1. Posibles soluciones	34
4.2.2. Resiliencia a errores	35
4.3. Arquitectura	35
4.4. Plan de desarrollo	37
4.5. Tecnologías a considerar	38
4.5.1. Lenguajes interpretados	38
4.5.2. WebAssembly	39
4.5.3. eBPF	42
4.5.4. Comunicación Inter-Proceso	42
4.5.5. Cargado dinámico	45
4.6. Sistemas de plugins de referencia	49
4.7. Elección Final	50
5. Implementación	53
5.1. Metodología	53
5.2. <code>abi_stable</code>	55
5.2.1. Versionado	55
5.2.2. Cargado de plugins	56
5.2.3. Exportando un plugin	56
5.2.4. Gestión de pánicos	56
5.2.5. Programación asíncrona	57
5.2.6. Seguridad en hilos	57
5.2.7. Rendimiento	58
5.3. Conversión al ABI de C	58
5.3.1. Consecuencias del sistema de plugins	60
5.3.2. Problemas con tipos externos	61

<i>ÍNDICE</i>	IX
5.3.3. Problemas con varianza y subtipado	65
5.4. Estado final del proyecto	68
5.5. Despliegue en producción	69
6. Conclusiones	73
Lista de Figuras	81
Lista de Tablas	83
Anexos	84
A. Contribuciones de código abierto	87

Capítulo 1

Introducción

1.1. Contexto

Este proyecto se ha realizado en colaboración con *Tremor*¹, un sistema de procesamiento 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)”⁵⁶. 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 hogar⁸. 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/>

⁵Página 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>

⁷RFC 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 *NGINX* [4] o *Apache HTTP Server* [5], llevan beneficiándose de un sistema de plugins desde hace mucho. Informan mejoras 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 Tremor⁹ 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 GitHub¹⁰.

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*. Dispone de una serie con un total de 6 artículos, cada uno de unos 30 minutos de lectura. Gran parte del contenido de este documento se obtiene de ahí, omitiendo en lo posible detalles específicos al código, dado que aquí no se asume familiaridad con Rust. La organización también difiere considerablemente, puesto que los artículos se escribieron a medida que se realizaba el proyecto, resultando en una estructura menos estricta. 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 un sistema de plugins propio.

- 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/>.

TODO: debería ser más específico sobre las horas? Mucha cuenta no he llevado pero vamos sí que estoy segurísimo de que han sido 300 horas y mucho más también.

TODO: quizá hacer una tabla que relacione cada sección del índice con los capítulos relacionados de nullderef? Pero eso ya lo pondría en un anexo.

Capítulo 2

Breve guía de Rust

TODO: quizá mover esto a un anexo para que no cuente en la suma de palabras total?

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. *The Rust Programming Language* [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 *Rust for Professionals* [9], *A Gentle Introduction to Rust* [10] o *30 minutes of Introduction to Rust for C++ programmers* [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 *The Rustnomicon* [12] y *Asynchronous Programming in Rust* [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 seguridad, tanto en memoria como en concurrencia, ambos en tiempo de compilación. Proporciona control a bajo nivel, manteniendo una productividad cercana a lenguajes de alto nivel.

Comenzando por el clásico “Hola Mundo”, se incluyen algunos ejemplos de cómo es la sintaxis de Rust más básica. Un binario o una librería en Rust reciben el nombre de *crate*. Nuestra *crate* se podría ejecutar fácilmente con *Cargo*, el administrador de dependencias oficial, o 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 `!` al final.

2.2. Conceptos básicos

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

```
1 fn factorial(i: u64) -> u64 {  
2     match i {  
3         // Primer caso: i = 0  
4         0 => 1,  
5         // El resto de casos, asignado a una variable `n`  
6         n => n * factorial(n-1)  
7     }  
8 }
```

Uso de variables y métodos:

```
1 fn main() {  
2     // Declaración de una variable, cuyo tipo se infiere  
3     // automáticamente.  
4     let my_number = 1234;  
5     // Declaración de una variable con un tipo especificado  
6     // manualmente. Notar que se puede usar el mismo nombre, y la  
7     // variable anterior será destruida.  
8     let my_number: i32 = 4321;  
9     // Invocación de la función estática (constructor) `new` dentro  
10    // del tipo `String`. El uso de `mut` indica que la instancia del  
11    // tipo se puede modificar. Funciona de forma inversa a C++, que  
12    // por defecto es mutable y `const` indica que *no* se puede  
13    // modificar.  
14    let mut my_str = String::new();  
15    // Invocación del método `push` de `my_str`, que añade un  
16    // carácter al final de la cadena.  
17    my_str.push('a');  
18 }
```

Otros componentes principales de Rust son:

– Estructuras de datos:

```
1 struct MessageA {  
2     // Campo público con una cadena de caracteres  
3     pub text: String,  
4     // Campo privado con un entero  
5     user_id: i32,  
6 }
```

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

– Enumeraciones, que también permiten contener datos:

```
1 enum MessageC {  
2     Join,  
3     Text(String, i32),  
4     Leave(i32),  
5 }
```

- *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:

```

1  trait Sender {
2      // Los métodos requieren especificar `self` explícitamente,
3      // que es lo mismo que `this` en Java o C++. En este caso,
4      // `&send` tomará una referencia al tipo que implemente
5      // `Sender`. También podría ser una referencia mutable con
6      // `&mut self`, o el mismo tipo con `self`.
7      fn send(&self, msg: String);
8
9      // Implementación por defecto.
10     fn send_twice(&self, msg: String) {
11         self.send(msg);
12         self.send(msg.clone());
13     }
14 }

```

Y para implementar un trait para un tipo:

```

1  impl Sender for MessageC {
2      fn send(&self, msg: String) {
3          match self {
4              Join => println!("Joined"),
5              Text(txt, id) => println!("{}", sent: {}, id, txt),
6              // Las variables `_` son ignoradas
7              Leave(_) => println!("Left"),
8          }
9      }
10
11     // `send_twice` se implementará automáticamente.
12 }

```

Notar que, sin embargo, Rust no es un lenguaje orientado a objetos. Un *trait* puede heredar de otro *trait*, pero un *struct* no puede heredar de otro *struct*.

2.3. 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:

- 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 prelude, 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` no nulo en memoria. Además de un `*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.

2.4. 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:

```
1 match load_file(input) {  
2     Ok(data) => /* ... */,  
3     Err(e) => eprintln!("Error: {e}"),  
4 }
```

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.

2.5. 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 (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) [14, Procedural Macros]. Consecuentemente, su complejidad es mucho mayor, pero expanden las posibilidades de los macros enormemente.

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



```

1 // Sintaxis típica de invocación de un macro
2 some_macro! {
3     fn some_function() { /* ... */ }
4 }
5
6 // También permitido en el caso de los procedurales
7 #[some_macro]
8 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:

```

1 // Con un macro `derive` para el trait `Debug`, que sirve para
2 // mostrar variables por pantalla.
3 #[derive(Debug)]
4 struct X(i32);
5
6 // Sin ellos sería lo siguiente. Como es trivial se puede
7 // simplificar en un macro procedural de tipo `derive`.
8 impl fmt::Debug for X {
9     fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
10         write!(f, "{:?}", self.0)
11     }
12 }

```

2.6. Lifetimes

La seguridad que provee Rust en memoria se basa en un modelo a tiempo de compilación con *lifetimes*. Una *lifetime* es

TODO: esto depende de cómo se acaba incluyendo la sección de “problemas con varianza y subtipado”.

2.7. 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 algunas ocasiones es posible que rechace algunos programas correctos. El desarrollador puede indicar que es consciente de la situación y puede apagar este análisis 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 una función `unsafe fn name() { /* ... */ }`. Funciona igual que rust, 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`

2.8. Programación asíncrona

Como muchos lenguajes modernos, Rust da soporte a la programación asíncrona, un modelo de programación concurrente. Sin entrar en excesivo detalle, 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 [13].

```
1 // Con `async` se indica que la función es asíncrona.
2 async fn get_two_sites_async() {
3     // Creación de dos "futuros" que, al completarse, descargarán
4     // asíncronamente las páginas web. Similar a la creación de
5     // un nuevo hilo.
6     let future_one = download_async("https://www.foo.com");
7     let future_two = download_async("https://www.bar.com");
8
9     // Ejecutar las dos tareas. Similar a esperar la terminación de
```

```
10 // los hilos.  
11 join!(future_one, future_two);  
12  
13 // Con `.await` se puede esperar a la terminación de un futuro  
14 // individual.  
15 let future_three = download_async("https://www.bar.com").await;  
16 }
```


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)” [15]. Tremor fue creado como una alternativa de alto rendimiento a herramientas como *Logstash* [16] o *Telegraf* [17], 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).

Robins [18] y Cugola y Margara [19] 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 [20] y Tawsif y col. [21] 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.

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

3.2. Casos de uso

La Figura 3.1 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.

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 [22].
- 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 [23].

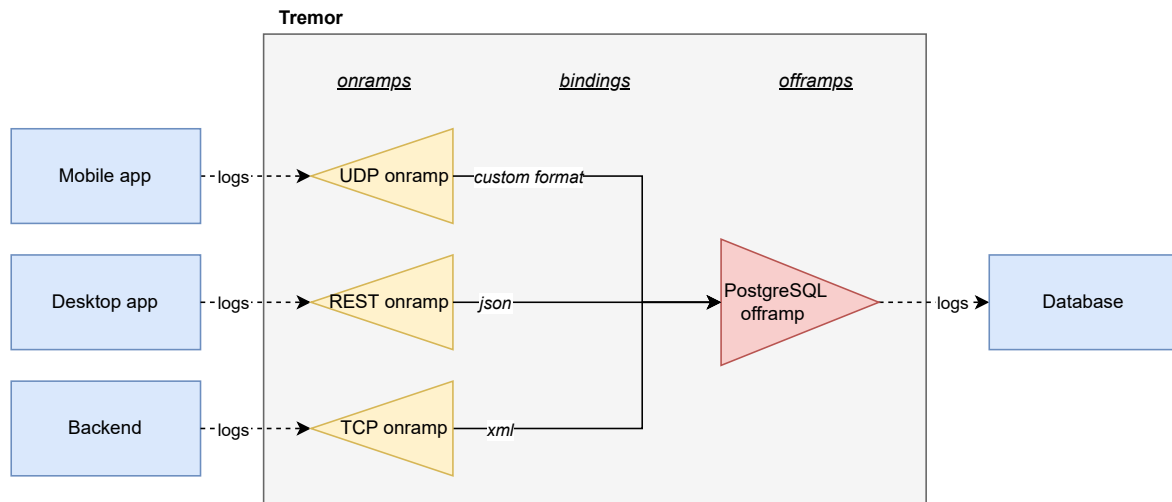


Figura 3.1: Ejemplo de uso básico de Tremor

- Una *pipeline* es una lista de operaciones (transformación, agregación, eliminación, etc) a través de la cual se pueden encaminar los eventos [24]. La Figura 3.2 muestra un ejemplo de una *pipeline*, definida con Troy, su propio lenguaje inspirado en SQL.

Estos *onramps* u *offramps* suelen contener una cantidad de información que es demasiado grande como para guardarla y debería tratarse en tiempo real. Su procesamiento 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 numérico) 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:

```

1 define pipeline main
2 # The exit port is not a default port, so we have to overwrite the
3 # built-in port selection
4 into out, exit
5 pipeline
6 # Use the `std::string` module
7 use std::string;
8 use lib::scripts;
9
10 # Create our script
11 create script punctuate from scripts::punctuate;
12
13 # Filter any event that just is `exit` and send it to the exit port
14 select {"graceful": false} from in where event == "exit" into exit;
15
16 # Wire our capitailized text to the script
17 select string::capitalize(event) from in where event != "exit"
18     into punctuate;
19 # Wire our script to the output
20 select event from punctuate into out;
21 end;

```

Figura 3.2: Ejemplo de una *pipeline* definida para 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 *Tremor Getting Started* [25], que introduce sus conceptos más básicos y sus posibles usos — o cuándo no usarlo, en *Tremor Constraints and Limitations* [26]. *Tremor Recipes* [27] 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.

```

1 pub trait Connector {
2     /// Crea la parte "source" del conector, si es aplicable.
3     async fn create_source(
4         &mut self,
5         _source_context: SourceContext,
6         _builder: source::SourceManagerBuilder,
7     ) -> Result<Option<source::SourceAddr>> {
8         Ok(None)
9     }
10
11     /// Crea la parte "sink" del conector, si es aplicable.
12     async fn create_sink(
13         &mut self,
14         _sink_context: SinkContext,
15         _builder: sink::SinkManagerBuilder,
16     ) -> Result<Option<sink::SinkAddr>> {
17         Ok(None)
18     }
19
20     /// Intenta conectarse con el mundo exterior. Por ejemplo, inicia la
21     /// conexión con una base de datos.
22     async fn connect(
23         &mut self,
24         _c: &ConnectorContext,
25         _attempt: &Attempt
26     ) -> Result<bool> {
27         Ok(true)
28     }
29
30     /// Llamado una vez cuando el conector inicia.
31     async fn on_start(&mut self, _c: &ConnectorContext) -> Result<()> {
32         Ok(())
33     }
34     /// Llamado cuando el conector pausa.
35     async fn on_pause(&mut self, _c: &ConnectorContext) -> Result<()> {
36         Ok(())
37     }
38     /// Llamado cuando el conector continúa.
39     async fn on_resume(&mut self, _c: &ConnectorContext) -> Result<()> {
40         Ok(())
41     }
42     /// Llamado ante un evento de "drain", que se asegura de que no
43     /// lleguen más eventos a este conector.
44     async fn on_drain(&mut self, _c: &ConnectorContext) -> Result<()> {
45         Ok(())
46     }
47     /// Llamado cuando el conector para.
48     async fn on_stop(&mut self, _c: &ConnectorContext) -> Result<()> {
49         Ok(())
50     }
51 }

```

Figura 3.3: Simplificación del *trait* Connector

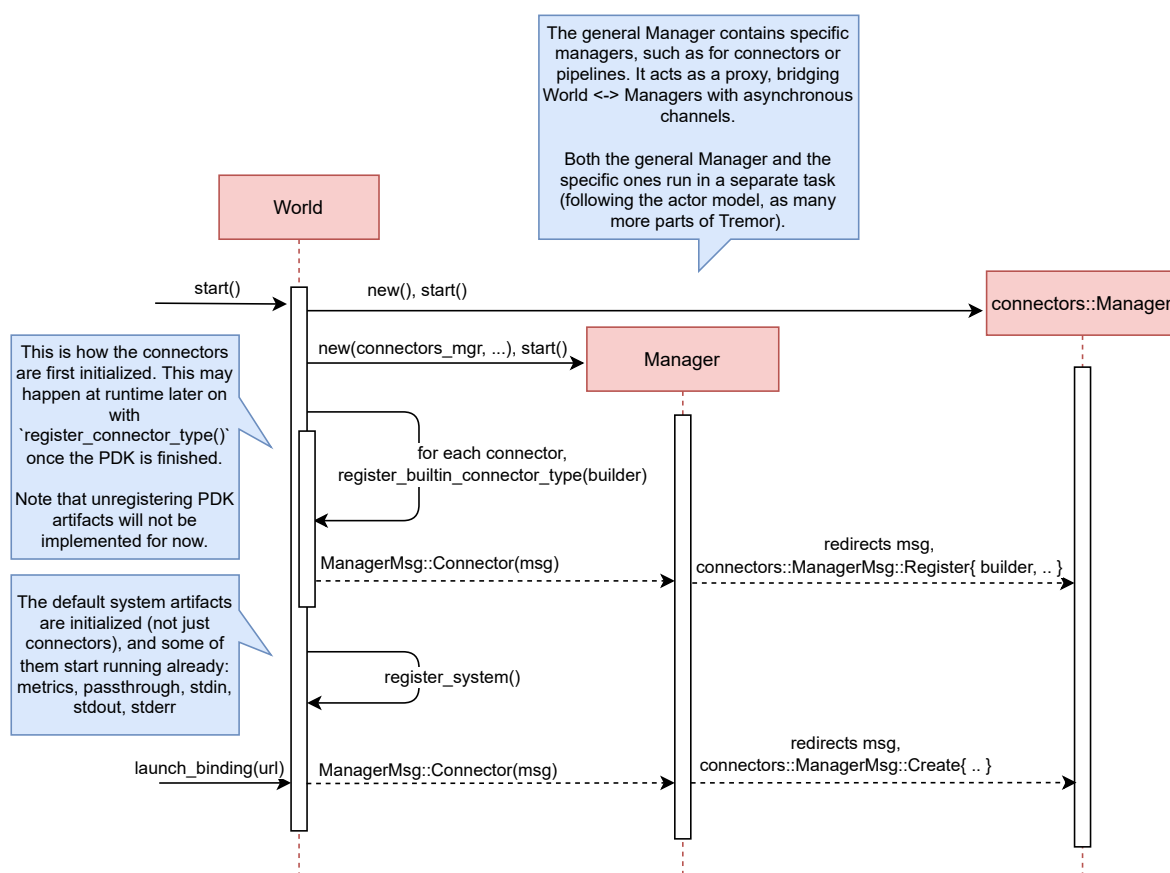


Figura 3.4: Registro de un conector en el programa

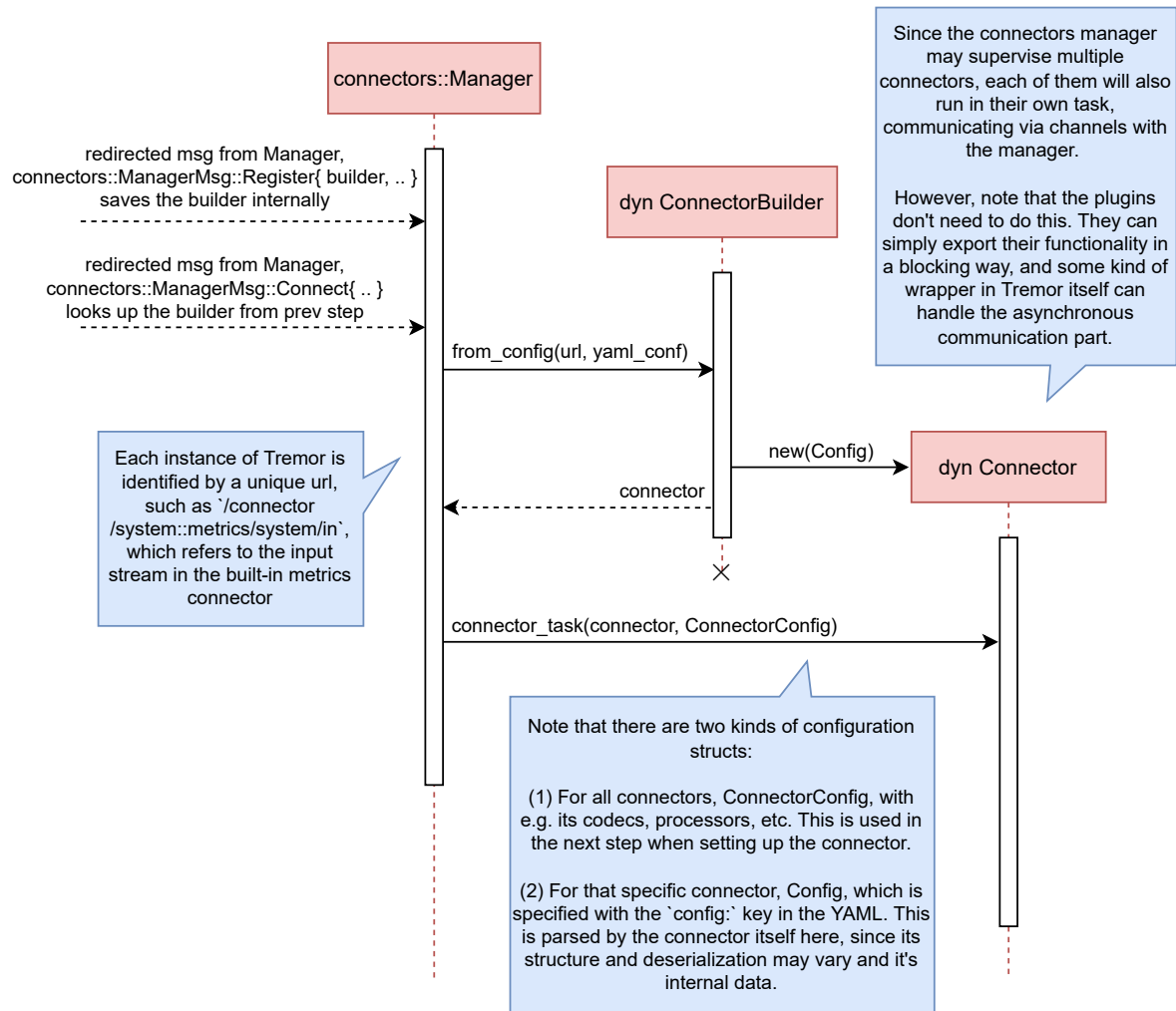


Figura 3.5: Inicialización de un conector en el programa

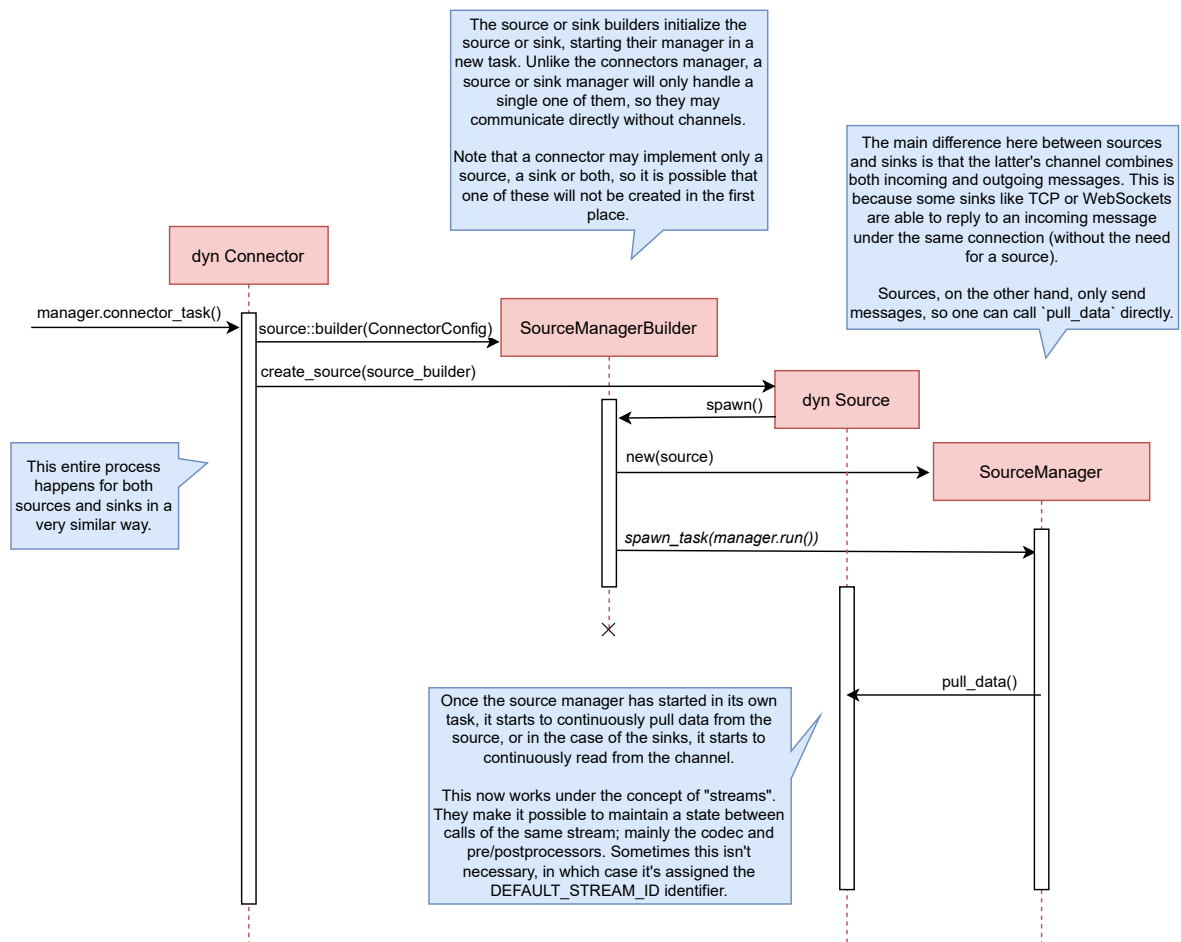


Figura 3.6: Configuración de un conector en el programa

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 MIRI¹ — 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 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.

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 aísla 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 [28]. 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 npm². 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 [29][30]. 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 Deno³, 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 aislado? 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 aislado, 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.

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 [31]:

- 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. Resiliencia a 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. Arquitectura

Para el sistema de plugins final, la estructura de Tremor debería ser la ilustrada en la Figura 5.3:

- 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* .

Esta estructura es esencial para el objetivo principal: mejorar los tiempos de

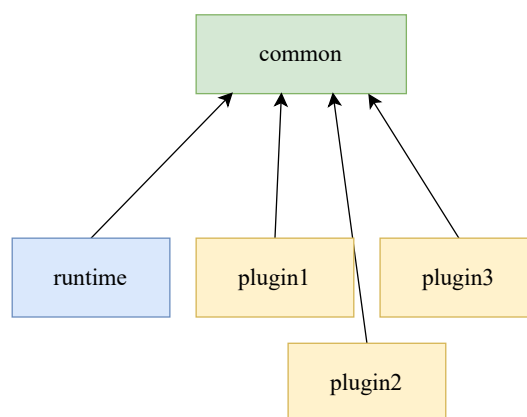


Figura 4.1: La estructura ideal para un sistema de plugins.

compilación. Existen dos maneras de entender los tiempos de compilación:

- Para el desarrollo de la **runtime**
- 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`, `plugins`, y `common`. El primer paso debería ser el que muestra la Figura 5.4: separar los plugins del mono-binario. La funcionalidad se encuentra en binarios diferentes, así que la runtime tendrá un tiempo de compilación considerablemente 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 código en pequeñas iteraciones, en vez de una única revisión grande.

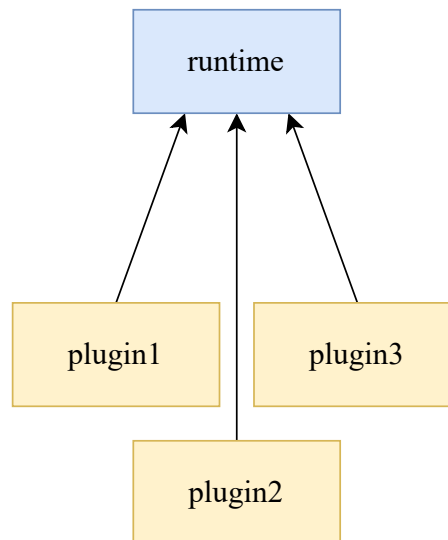


Figura 4.2: La estructura inicial del sistema de plugins para un desarrollo más rápido.

4.4. 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 todo lo mínimo 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 mergear con la rama principal, ya que el programa seguirá funcionando de la misma forma, simplemente con una interfaz distinta para los conectores internamente.

2. **Realmente 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, únicamente al final se separará la interfaz a una *crate common* nueva, para

una última mejora a 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.

4.5. 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.

4.5.1. 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 [32]. Ahora NeoVim, un fork más moderno, está esforzándose por tener Lua como lenguaje de primera clase para su configuración [33]. 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 [34]. Algunas implementaciones del lenguaje, como LuaJIT, son extremadamente eficientes y pueden ser viables hasta en escenarios de rendimiento crítico [35]. 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”) [36].

Rust dispone de librerías como `rlua`⁴, con bindings para interoperar con Lua.

⁴<https://crates.io/crates/rlua>

`r lua` 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: *Gluon* [37], *Rhai* [38] o *Rune* [39]. 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 *TIOBE Index* [40].

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.

4.5.2. WebAssembly

WebAssembly [41], 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 [42].

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

⁵<https://crates.io/crates/mlua>

de *scripting*).

Existen dos runtimes principales para Rust: *Wasmer* [43] y *Wasmtime* [44]. 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 para ambos casos [45].

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. [46] muestra resultados mezclados al realizar pruebas de rendimiento. Depende principalmente del compilador⁶ 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, Denis [47] 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 [48]. Existen algunas maneras de tratar tipos no triviales como estructuras o enumeraciones:

- A través de la *Interface Types Proposal for WebAssembly* [49]. 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

⁶Nos 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.

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 4.3.

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 [50][51]. 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 [52][53], 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 [54], 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.

```
1 (use "errno.witx")
2
3 ;;; Add two integers
4 (module $calculator
5   (@interface func (export "add")
6     (param $lh s32)
7     (param $rh s32)
8     (result $error $errno)
9     (result $res s32)
10  )
11 )
```

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

4.5.3. eBPF

eBPF es “a revolutionary technology with origins in the Linux kernel that can run sandboxed programs in an operating system kernel” [55]. 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.

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 soporte 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.

4.5.4. 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 Protocol*⁹, basado en JSON-RPC y usado por la mayoría de editores de texto para

⁷<https://crates.io/crates/rbpf>

⁸https://crates.io/crates/solana_rbpf

⁹<https://microsoft.github.io/language-server-protocol/>

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 4.4 y la Figura 4.5, 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 *Protocol Buffers* [57] 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 nnn¹⁰ 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.

¹⁰<https://github.com/jarun/nnn>

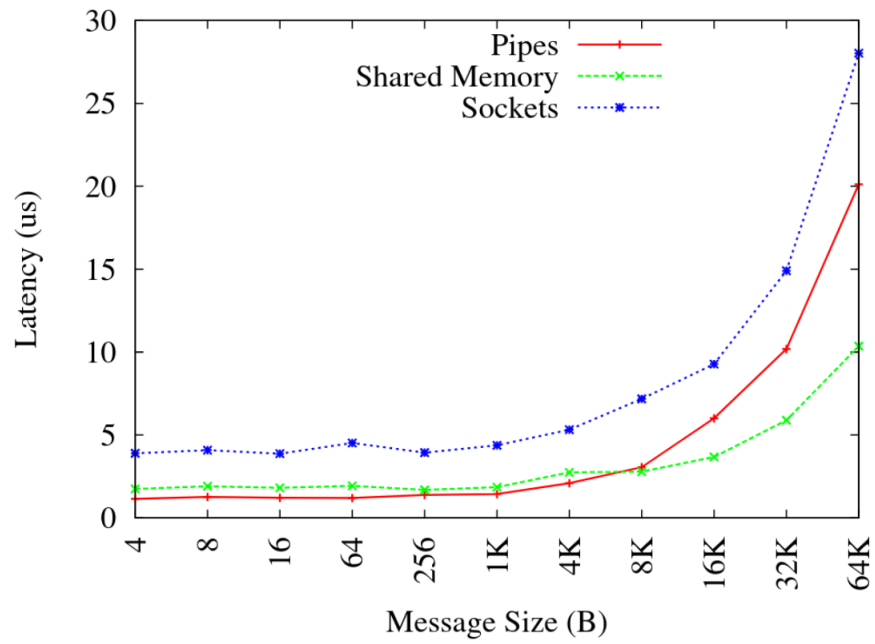


Figura 4.4: Latencia vs. Tamaño de Mensaje [56].

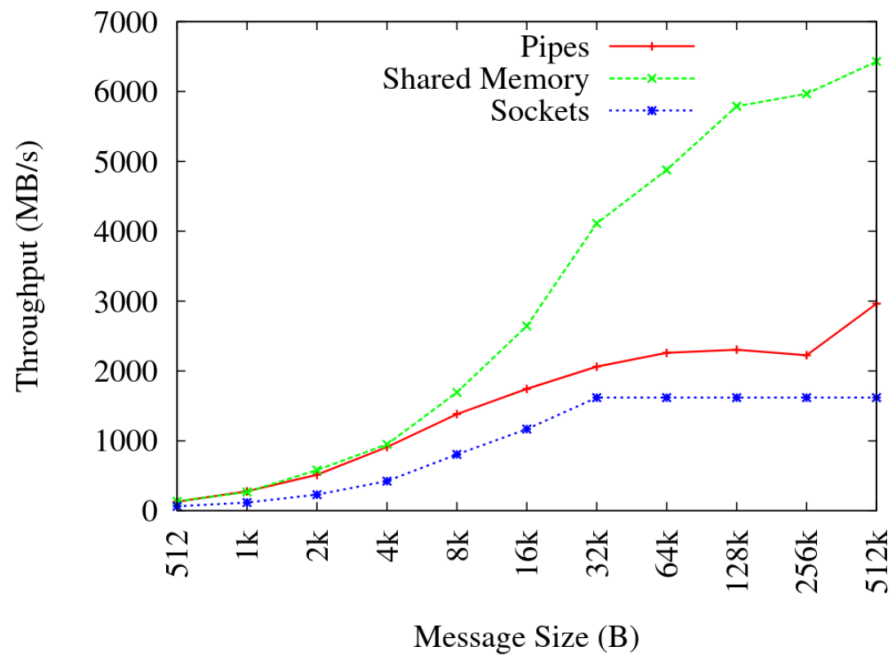


Figura 4.5: Rendimiento vs. Tamaño de Mensaje [56].

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* [58].

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 4.4 y 4.5, 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 [59].

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 4.5.5). No parece ofrecer nada mejor que el cargado dinámico y por tanto se descarta como opción.

4.5.5. 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>

¹³https://crates.io/crates/shared_memory

¹⁴https://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 adicional tras cargar la librería dinámica.

La *crate* principal para este método es `libloading`¹⁵, aunque también existen las menos conocidas `dlopen`¹⁶ y `cratesharedlib`, con pequeñas diferencias [60]. Todas ellas requieren de uso extensivo de `unsafe`, son complicadas de usar correctamente [61][62], incluyendo sutiles disparidades entre sistemas operativos [63], 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 [64][65][66][67]. 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 [14, 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” [14, Type Layout]

¹⁵<https://crates.io/crates/libloading>

¹⁶<https://crates.io/crates/dlopen>

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 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 [68], 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 4.6 y 4.7.

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 4.8 demuestra parte de la simplificación del código en la Figura 4.7.

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

¹⁷https://crates.io/crates/abi_stable

¹⁸<https://crates.io/crates/crossbeam>

¹⁹https://crates.io/crates/serde_json

²⁰<https://github.com/LightningCreations/lccc>

²¹https://crates.io/crates/safer_ffi

²²<https://crates.io/crates/cglue>

```

1 pub struct Event {
2     pub count: i32,
3     pub name: &'static str,
4 }
5
6 pub fn transform(x: Event) -> i32 {
7     println!("Received an event with count {}", x.count);
8     x.count
9 }
10
11 pub static cached: Event = Event {
12     count: 0,
13     name: "my data"
14 };

```

Figura 4.6: Ejemplo de cómo sería un plugin escrito con Rust.

```

1 // Using C's memory layout with `#[repr]`
2 #[repr(C)]
3 pub struct Event {
4     // We can't have types from the standard library anymore, only
5     // either basic ones...
6     pub count: i32,
7     // ...or types from C itself
8     pub name: *const std::os::raw::char_c,
9 }
10
11 // Using C's calling conventions with `extern "C"`
12 pub extern "C" fn transform(x: Event) -> i32 {
13     println!("Received an event with count {}", x.count);
14     x.count
15 }
16
17 // Disabling mangling so that the resource's name is known when
18 // loading the plugin.
19 #[no_mangle]
20 pub static cached: Event = Event {
21     count: 0,
22     name: "my data".as_ptr() as _
23 };

```

Figura 4.7: El mismo plugin que la Figura 4.6, pero usando el ABI de C.

en cuenta en la decisión final también.

4.6. 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 [69], método también usado en Visual Studio Code [70] o Eclipse [71].
- `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 4.5.5 y únicamente usa tipos definidos en Rust.

Otras fuentes como Amos [72] o Zicklag [73] también intentan usar cargado dinámico para funcionalidades similares. Este último se trata de `amethyst`²⁷, el predecesor de `bevy`, que acabó rindiéndose debido a la

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>

```
1 // Now we also "derive" the trait StableAbi, i.e., it's implemented
2 // automatically.
3 #[repr(C)]
4 #[derive(StableAbi)]
5 pub struct Event {
6     pub count: i32,
7     // We can use abi_stable's types!
8     pub name: RStr<'static>,
9 }
10
11 // Using C's calling conventions with `extern "C"`
12 #[sabi_extern_fn]
13 pub fn transform(x: Event) -> i32 {
14     println!("Received an event with count {}", x.count);
15     x.count
16 }
```

Figura 4.8: El mismo plugin que la Figura 4.6, pero con `abi_stable`. La declaración de la global `cached` se omite por simplicidad.

inestabilidad del ABI [74][75].

4.7. 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 4.9, 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.

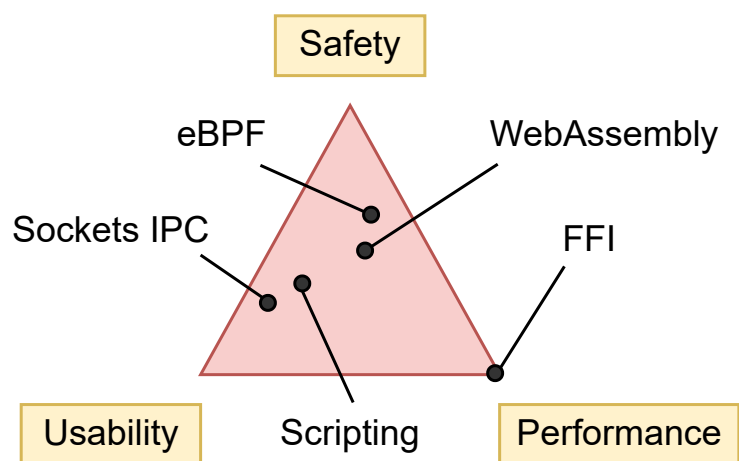


Figura 4.9: Comparación aproximada de los métodos investigados.

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.

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 inspirada por mis mentores, que lo denominaron el “Just Make it Work”, o “Primero 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:

TODO: alguna traducción de “just make it work” un poco más natural? “Solo que funcione” “No te líes, que funcione primero”

- Código “feo” (no idiomático, repetitivo o desordenado).
- Código de bajo rendimiento.
- Documentación pobre.
- No tener tests.
- No aplicar sugerencias recomendadas 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 este caso, al menos al principio. 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. 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. `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

`abi_stable` especifica lo siguiente respecto a su sistema de seguridad [76]:

- 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* `StableAbi`, indicador de que un tipo es seguro para FFI. Contiene información sobre la estructura en memoria y puede ser derivado automáticamente. Todos los tipos exportados por `abi_stable`, además de usar el ABI de C, implementan dicho *trait*. Por tanto, si queremos declarar nuestro propio tipo para usar con `abi_stable`, además de marcarlo con `#[repr(C)]`, tendremos que añadir `#[derive(StableAbi)]`.

5.2.2. Cargado de plugins

5.2.3. Exportando un plugin

5.2.4. Gestión de pánicos

Actualmente, lanzar pánicos a través del FFI es comportamiento no definido [12, 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 abortaría el programa, en lugar de dejar que se propague a la runtime, que sería indefinido.

También es posible configurar el programa 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)* [77] 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 de forma transparente; no hace falta que el desarrollador se preocupe en ningún momento por ello.

La solución de `abi_stable` no es perfecta por tener un pequeño coste de rendimiento y por imposibilitar 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 [78].

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.

TODO: podría poner un ejemplo de cómo se usa esta librería (está en un comentario aquí), pero creo que es demasiado detalle.

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 tuve que abrir un pull request para hacerlo yo mismo [79]. El uso de esta librería resulta en código más verboso, pero esto se podría mejorar en el futuro con un macro procedural [80].

5.2.6. Seguridad en hilos

`abi_stable` utiliza la `crate` `libloading`² internamente, cuya gestión de errores no es segura en hilos en algunas plataformas, como `dllerror` en FreeBSD [81][82]. Sí que lo es en Linux [83], macOS [84] y Windows [85], así que en el caso de Tremor no hace falta preocuparse por esto.

Será importante tener esto 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

¹https://crates.io/crates/async_ffi

²<https://crates.io/crates/libloading>

acceso con un *mútex* interno, como lo hace la `crate dlopen`³ [86]. Notar que esta librería es también mejorable, dado que usa el *mútex* siempre, incluso en sistemas operativos donde el sistema de errores sí es seguro en hilos [87].

5.2.7. Rendimiento

Un último 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 $O(1)$.

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

```
1 pub enum Value {  
2     /// Valores estáticos (enteros, booleanos, etc)  
3     Static(StaticNode),  
4     /// Tipo para cadenas de caracteres  
5     String(String),  
6     /// Tipo para listas  
7     Array(Vec<Value>),
```

³<https://crates.io/crates/dlopen>

```

8      /// Tipo para objetos (mapas clave-valor)
9      Object(Box<HashMap<String, Value>>),
10     /// Tipo para datos binarios
11     Bytes(Vec<u8>),
12 }

```

Para poder usar `Value` en la interfaz del sistema de plugins, se puede convertir a:

```

1  #[repr(C)] // La representación en memoria de Value seguirá el ABI de C
2  #[derive(StableAbi)] // Solo necesario cuando se usa abi_stable
3  pub enum Value {
4      Static(StaticNode),
5      /// Ahora usa `RString`, la alternativa a `String` de abi_stable
6      String(RString),
7      /// De forma similar, usa `RVec` en vez de `Vec`
8      Array(RVec<Value>),
9      /// Cambio de `Box`, `HashMap` y `String` por sus alternativas
10     Object(RBox<RHashMap<RString, Value>>),
11     /// Otro cambio de `Vec`
12     Bytes(RVec<u8>),
13 }

```

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`⁴, que lo declara tal que:

```

1  pub enum StaticNode {
2      I64(i64),
3      U64(u64),
4      F64(f64),
5      Bool(bool),
6      Null,
7  }

```

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 [88]. Será importante también que la estructura use `#[repr(C)]`

⁴https://crates.io/crates/value_trait

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 ofrece `#[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:

```
1 // No funcionará porque Value::Array contiene un RVec ahora
2 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:

```
1 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`:

```
1 impl Value {
2     // Código original
3     fn as_object(&self) -> Option<&HashMap<String, Value>> {
4         match self {
5             // Problema: `m` ahora es una `RHashMap`, pero la función
6             // devuelve un `HashMap`.
7             //
8             // Solución 1: cambiar el tipo devuelto a `RHashMap`
9             // Solución 2: convertir `m` a un `HashMap` con `m.into()`
10            Self::Object(m) => Some(m),
```



```

11         _ => None,
12     }
13 }
14 }

```

- 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 añadirlo⁵. 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;

⁵El anexo A lista todas las contribuciones de código abierto realizadas para este proyecto

realmente, Tremor usa la implementación de `halfbrown`⁶ 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`⁷ 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 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

⁶<https://crates.io/crates/halfbrown>

⁷<https://crates.io/crates/beef>

```
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.

posible su uso desde el ABI de C de forma segura. Sin embargo, estos ejemplos ya existentes son complejos [89] 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 la Figura 5.2.

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`):

```
1 pub enum PdkValue {  
2     /// Valores estáticos (enteros, booleanos, etc)  
3     Static(StaticNode),  
4     /// Tipo para cadenas de caracteres  
5     String(String),  
6     /// Tipo para listas  
7     Array(Vec<Value>),  
8     /// Tipo para objetos (mapas clave-valor)  
9     Object(Box<HashMap<String, PdkValue>>),  
10    /// Tipo para datos binarios  
11    Bytes(Vec<u8>),  
12 }
```

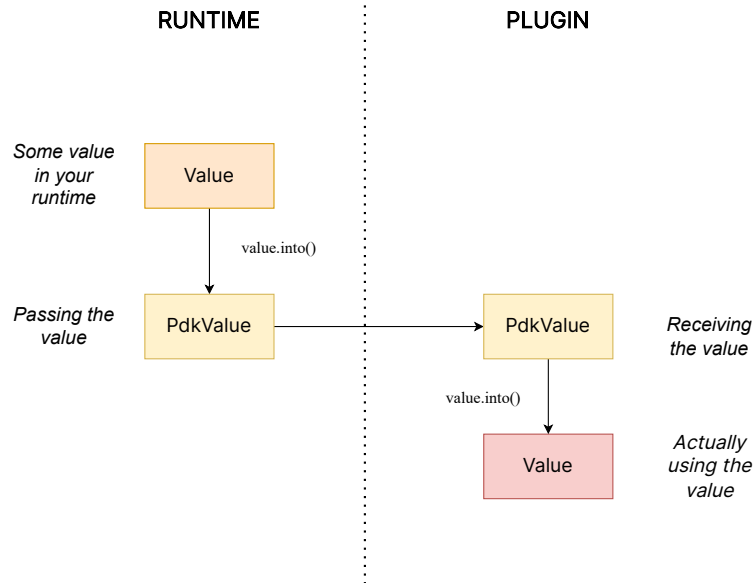


Figura 5.2: Comunicación entre runtime y plugins en el PDK.

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, reemplazarlo 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 siguiente snippet de código?

```

1 // Esta función es exportada por el plugin. Funcionará porque
2 // `PdkValue` está declarado con el ABI de C.
3 pub extern "C" fn plugin_func(value: PdkValue) {
4     let value = Value::from(value);
5     value.do_func()
6 }
7
8 // Esto se puede implementar en la runtime para facilitar su uso,
9 // convirtiendo al tipo original.
10 fn runtime_wrapper(value: Value) {
11     plugin_func(value.into());
12 }

```

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 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 errores intrincados y difíciles de identificar.

Este tema no se cubre en *The Rust Programming Language* [8], sino en *The Rustnomicon* [12, Subtyping and Variance] y *The Rust Reference* [14, Subtyping and Variance]. También es recomendable consultar el artículo *Diving Deep: implied bounds and variance #25860* [90] o a Gjengset [91] para un formato en vídeo.

Deriva también de los problemas encontrados con el tipo `Value`. La historia

completa se incluye en el issue *lifetimes with R* types break compared to non R* types* — [GitHub rodrimati1992/abi_stable_crates](#) [92]. Al cambiar los tipos de la librería estándar a los de `abi_stable`, se producían errores de *lifetimes inexplicables* (ver Figura 5.1). 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:

```

1 use abi_stable::std_types::RCow;
2 use std::borrow::Cow;
3
4 fn cmp_cow<'a, 'b>(left: &Cow<'a, ()>, right: &Cow<'b, ()>) -> bool {
5     left == right
6 }
7
8 // Este caso falla en compilación, pero es aparentemente igual
9 fn cmp_rcow<'a, 'b>(left: &RCow<'a, ()>, right: &RCow<'b, ()>) -> bool {
10     left == right
11 }

```

```

1 $ cargo build
2 error[E0623]: lifetime mismatch
3     --> src/lib.rs:10:10
4
5 9 | fn cmp_rcow<'a, 'b>(
6   |     left: &RCow<'a, ()>, right: &RCow<'b, ()>) -> bool {
7   |                                     -----
8   |                                     |
9   |                                     these two types are declared with
10  |                                     different lifetimes...
11 10 |         left == right
12   |         ^^ ...but data from `left` flows into `right` here
13
14 For more information about this error, try `rustc --explain E0623`.
15 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. Por ejemplo, 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`, que es imposible porque el compilador

no sabe cuál 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.

```

1 struct Foo<'a> {
2     x: &'a isize,
3 }
4
5 fn bar<'short, 'long>(c: Foo<'short>, l: &'long isize) {
6     // Equivalente a asignarle otra lifetime a c
7     let c: Foo<'long> = c; // error!
8 }

```

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

```

1 // Notar que ahora `short` se declara tal que `short: long`
2 fn bar<'short: 'long, 'long>(c: Foo<'short>, l: &'long isize) {
3     let c: Foo<'long> = c; // ok!
4 }

```

Por tanto, uno pensaría que tiene que ver con el operador `==`, que 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>`. La mención anterior a estos errores como *inexplicables* se debe a que en este caso únicamente existe una *lifetime* `'a`, así que no se podría arreglar indicando que `'short: 'long`. No obstante, existía alguna pequeña diferencia sin relación a este problema entre las implementaciones, y al usar *exactamente lo mismo* que en `Cow<'a, T>`, compilaba correctamente.

Tras cambiar `left == right` por `left.cmp(right)` en la reproducción inicial, se repetía el problema, incluso con aparentemente la misma implementación de `Ord` (el *trait* con el método `cmp`). No fue hasta que expliqué mi problema en un servidor de Discord con más desarrolladores de Rust

que descubrí que el verdadero problema era un término llamado la *varianza* de `RCow<'a, T>`.

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` o `T: BorrowOwned<'a>`. El problema era que al relacionar la *lifetime* `'a` de esta forma, estaba rompiendo una regla que hacía a `RCow` *invariante*, en vez de *covariante*.

TODO: es esta sección demasiado técnica? Debería continuar? Me falta explicar qué es invarianza y covarianza. La verdad que esto sí que me llevó mucho tiempo en su momento, así que me gustaría incluirlo de alguna forma.

5.4. Estado final del proyecto

Como se ha explicado, la complejidad del proyecto resultó ser mucho mayor que lo esperado por problemas con el ABI. Por tanto, resultó imposible desarrollar en el tiempo disponible un sistema de plugins tan completo y eficiente como se especificaba.

La última versión del sistema de plugins es funcional y, mediante contribuciones de código abierto, ha hecho posible su futura inclusión en una versión de Tremor. Muchas de las librerías usadas no disponían de la funcionalidad necesaria para este proyecto, así como `async_ffi`, `abi_stable`, `halfbrown` o `simd-json`.

El problema principal tiene que ver con el rendimiento. Dada la naturaleza de Tremor, es un requerimiento imprescindible para poderlo incluir en una futura versión. Tras unas mediciones iniciales, se calculó que su inclusión reducía el

rendimiento del programa un 30%, así que los siguientes pasos consistieron en reducir dicha cifra.

TODO: incluir todas las figuras de benchmarks con las explicaciones

5.5. Despliegue en producción

Preparing the release for the plugin system has always sounded like a quite complicated task. I've never wanted to introduce too many changes at once because otherwise it will become very hard to review and merge into the main branch. This is why I decided to schedule the PDK release in four separate iterations:

Define the new interface and use it internally: the plugin system should be as minimal as possible. The program can be converted to support plugins, but while keeping them in the same binary, for simplicity. The plugin loading functionality can be left as a proof-of-concept for now. Still, this release could maybe include a couple external plugins for testing purposes.

Separate the runtime from the interface: what I talked about in the previous section can actually be done now. Plugins should be able to depend only on the common crate, rather than on the runtime. We may also run into many Git conflicts while doing this, but it should be a much faster process, since the crate separation is all we have to worry about now.

Actually make the plugins external: since the in-tree plugins are implemented with the new interface, making them actually external should be trivial. It'll just require re-organizing the repository with the new crates, fixing the build system, and similars.

Polishing for deployment: last improvements before the release, extensive documentation, evaluating the final results, etc.

Step 2 and 3 can be interchanged as desired, but I thought that the sooner we separate the crates, the better. Plugin development should be much faster after

that step for everyone else.

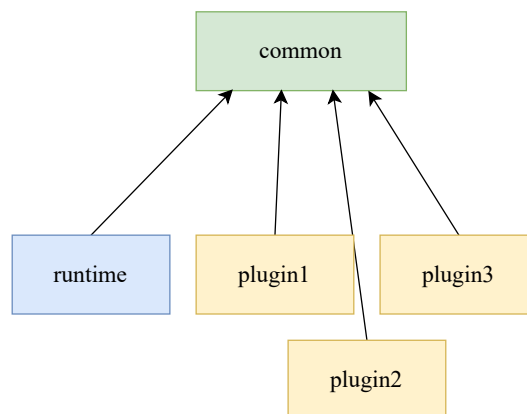


Figura 5.3: La estructura ideal para un sistema de plugins.

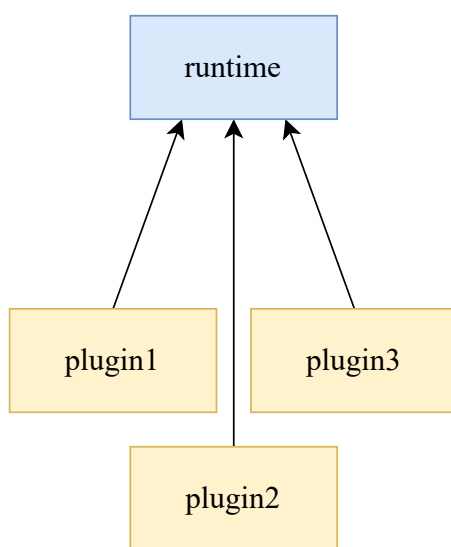


Figura 5.4: La estructura inicial del sistema de plugins para un desarrollo más rápido.

Capítulo 6

Conclusiones

TODO

Bibliografía

- [1] Chris Sweeney. *Inside Wayfair's Identity Crisis*. 2019. URL: <https://www.bostonmagazine.com/news/2019/10/01/inside-wayfair/>.
- [2] *Rust Survey Results*. 2021. URL: <https://blog.rust-lang.org/2022/02/15/Rust-Survey-2021.html#challenges-ahead>.
- [3] Carliss Young Baldwin y Kim B Clark. *Design rules: The power of modularity*. Vol. 1. MIT press, 2000.
- [4] NGINX. 2004. URL: <https://www.influxdata.com/time-series-platform/telegraf/>.
- [5] *Apache HTTP Server*. 1995. URL: <https://httpd.apache.org/>.
- [6] *Dynamic Modules Development*. 2016. URL: <https://www.nginx.com/blog/dynamic-modules-development/#demand>.
- [7] *Dynamic Shared Object (DSO) Support*. 2021. URL: <https://httpd.apache.org/docs/2.4/dso.html#advantages>.
- [8] *The Rust Programming Language*. 2022. URL: <https://doc.rust-lang.org/stable/book/>.
- [9] *Rust for Professionals*. 2020. URL: <https://overexact.com/rust-for-professionals/>.
- [10] *A Gentle Introduction to Rust*. 2018. URL: <https://stevedonovan.github.io/rust-gentle-intro/>.
- [11] *30 minutes of Introduction to Rust for C++ programmers*. 2017. URL: <https://vnduongthanhtung.gitbooks.io/migrate-from-c-to-rust/content/>.
- [12] *The Rustnomicon*. 2022. URL: <https://doc.rust-lang.org/nomicon/>.
- [13] *Asynchronous Programming in Rust*. 2022. URL: https://rust-lang.github.io/async-book/01_getting_started/01_chapter.html.
- [14] *The Rust Reference*. 2022. URL: <https://doc.rust-lang.org/reference/introduction.html>.
- [15] David C Luckham. *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons, 2011.
- [16] *Logstash*. 2011. URL: <https://www.elastic.co/logstash/>.
- [17] *Telegraf*. 2015. URL: <https://www.influxdata.com/time-series-platform/telegraf/>.

- [18] D Robins. “Complex event processing”. En: *Second International Workshop on Education Technology and Computer Science. Wuhan*. Citeseer. 2010, págs. 1-10.
- [19] 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>.
- [20] Miyuru Dayarathna y Srinath Perera. “Recent advancements in event processing”. En: *ACM Computing Surveys (CSUR)* 51.2 (2018), págs. 1-36.
- [21] 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.
- [22] *Tremor Onramps*. 2021. URL: <https://www.tremor.rs/docs/0.11/artefacts/onramps>.
- [23] *Tremor Offramps*. 2021. URL: <https://www.tremor.rs/docs/0.11/artefacts/offramps>.
- [24] *Tremor Pipelines*. 2021. URL: <https://www.tremor.rs/docs/next/language/#pipelines>.
- [25] *Tremor Getting Started*. 2021. URL: <https://www.tremor.rs/docs/next/getting-started/>.
- [26] *Tremor Constraints and Limitations*. 2021. URL: <https://www.tremor.rs/docs/0.11/ConstraintsLimitations>.
- [27] *Tremor Recipes*. 2021. URL: <https://www.tremor.rs/docs/0.11/recipes/README>.
- [28] Jill Steinberg. *Competing components make for prickly panelists*. 1997. URL: <https://www.infoworld.com/article/2077623/competing-components-make-for-prickly-panelists.html>.
- [29] Jamie Kyle. *How to build an npm worm*. URL: <https://jamie.build/how-to-build-an-npm-worm>.
- [30] 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/>.
- [31] Aria Beingessner. *How Swift Achieved Dynamic Linking Where Rust Couldn't*. 2019. URL: <https://gankra.github.io/blah/swift-abi/>.
- [32] *VimScript Documentation*. 2015. URL: http://vimdoc.sourceforge.net/html/doc/usr_41.html.
- [33] *Lua in NeoVim*. 2022. URL: <https://neovim.io/doc/user/lua.html>.
- [34] Roberto Ierusalimsky. *Programming in lua*. Roberto Ierusalimsky, 2006.
- [35] *LuaJIT Benchmarks*. 2008. URL: <https://luajit.org/performance.html>.

- [36] *Lua Sand Boxes*. 2015. URL: <http://lua-users.org/wiki/SandBoxes>.
- [37] *Gluon*. 2016. URL: <https://crates.io/crates/gluon>.
- [38] *Rhai*. 2016. URL: <https://crates.io/crates/rhai>.
- [39] *Rune*. 2018. URL: <https://crates.io/crates/rune>.
- [40] *TIOBE Index*. 2022. URL: <https://www.tiobe.com/tiobe-index/>.
- [41] *WebAssembly*. 2017. URL: <https://webassembly.org/>.
- [42] *WASI*. 2019. URL: <https://wasi.dev/>.
- [43] *Wasmer*. 2019. URL: <https://wasmer.io/>.
- [44] *Wasmtime*. 2019. URL: <https://wasmtime.dev/>.
- [45] *Actually Using Wasm*. 2020. URL: <https://wiki.alopez.li/ActuallyUsingWasm>.
- [46] 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.
- [47] Frank Denis. *Benchmark of WebAssembly runtimes*. 2021. URL: <https://00f.net/2021/02/22/webassembly-runtimes-benchmarks/>.
- [48] *WasmExternType* — *Wasmer v0.17.1*. 2021. URL: https://docs.rs/wasmer-runtime-core/0.17.1/wasmer_runtime_core/types/trait.WasmExternType.html.
- [49] *Interface Types Proposal for WebAssembly*. 2015. URL: <https://github.com/webassembly/interface-types>.
- [50] *Support for Interface Types in wasmtime API* — *GitHub bytecodealliance/wasmtime*. 2021. URL: <https://github.com/bytecodealliance/wasmtime/issues/677>.
- [51] *State of the art of the interface types* — *GitHub wasmerio/wasmer*. 2021. URL: <https://github.com/wasmerio/wasmer/issues/2480>.
- [52] *feather/quill* — *GitHub feather-rs/feather*. 2021. URL: <https://github.com/feather-rs/feather/tree/main/quill>.
- [53] *Plugins* — *Veloren Project Architecture*. 2021. URL: <https://book.veloren.net/contributors/developers/codebase-structure.html#plugins>.
- [54] *zellij-org/zellij* — *GitHub*. 2021. URL: <https://github.com/zellij-org/zellij>.
- [55] *eBPF*. 2014. URL: <https://ebpf.io/>.
- [56] Aditya Venkataraman y Kishore Kumar Jagadeesha. “Evaluation of inter-process communication mechanisms”. En: *Architecture* 86 (2015), pág. 64.
- [57] *Protocol Buffers*. 2008. URL: <https://developers.google.com/protocol-buffers>.
- [58] *Rust By Example*. 2021. URL: <https://doc.rust-lang.org/rust-by-example/index.html>.

- [59] Mats Petersson. *Performance difference between IPC shared memory and threads memory*. 2013. URL: <https://stackoverflow.com/a/14512554>.
- [60] *dlopen: Compare with other libraries*. 2021. URL: <https://docs.rs/dlopen/0.1.8/dlopen/#compare-with-other-libraries>.
- [61] *Idiomatic Rust plugin system — StackOverflow*. 2017. URL: <https://stackoverflow.com/questions/44708483/idiomatic-rust-plugin-system/46249019#46249019>.
- [62] *papyrus/src/compile/execute.rs — GitHub kurtlawrence/papyrus*. 2019. URL: <https://github.com/kurtlawrence/papyrus/blob/1c7f0a669fed59d220bdefb161c568072126d3d5/src/compile/execute.rs#L36>.
- [63] *Thread-safety — libloading v0.7.0*. 2021. URL: <https://docs.rs/libloading/0.7.0/libloading/struct.Library.html#thread-safety>.
- [64] Michael F. Bryan. *Plugins in Rust*. 2019. URL: <https://web.archive.org/web/20211202080304/https://adventures.michaelfbryan.com/posts/plugins-in-rust/>.
- [65] *Plugin system with API — r/Rust*. 2017. URL: https://www.reddit.com/r/rust/comments/6v29z0/plugin_system_with_api/dlx9w7v/.
- [66] *Add the -Z randomize-layout flag — GitHub rust-lang/compiler-team*. 2021. URL: <https://github.com/rust-lang/compiler-team/issues/457>.
- [67] Michael F. Bryan. *Dynamic Loading & Plugins*. 2020. URL: <https://fasterthanli.me/articles/so-you-want-to-live-reload-rust>.
- [68] *A Stable Modular ABI for Rust*. 2020. URL: <https://internals.rust-lang.org/t/a-stable-modular-abi-for-rust/12347/71>.
- [69] Raph Levien. *JSON — xi-editor retrospective*. 2020. URL: <https://raphlinus.github.io/xi/2020/06/27/xi-retrospective.html#json>.
- [70] Danny Tuppeny. *Extensions using the "typecommand (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>.
- [71] Azad Bolour. *Notes on the Eclipse Plug-in Architecture*. 2003. URL: http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.
- [72] Amos. *So you want to live-reload Rust*. 2020. URL: <https://fasterthanli.me/articles/so-you-want-to-live-reload-rust>.
- [73] Zicklag. *Rust Plugins*. 2019. URL: <https://zicklag.github.io/rust-tutorials/rust-plugins.html>.

- [74] 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>.
- [75] Zicklag. *[NEW FEATURE] WebAssembly scripting system — GitHub amethyst/amethyst*. 2019. URL: <https://github.com/amethyst/amethyst/issues/1729>.
- [76] *Safety* — GitHub [rodrimati1992/abi_stable_crates](https://github.com/rodrimati1992/abi_stable_crates). 2021. URL: https://github.com/rodrimati1992/abi_stable_crates#safety.
- [77] *Pluggable panic implementations (tracking issue for RFC 1513)*. 2021. URL: <https://github.com/rust-lang/rust/issues/32837>.
- [78] *"C-unwind" ABI* — *The Rust RFC Book*. 2021. URL: <https://rust-lang.github.io/rfcs/2945-c-unwind-abi.html>.
- [79] *Support for abi_stable* — GitHub [oxalica/async-ffi](https://github.com/oxalica/async-ffi). 2021. URL: <https://github.com/oxalica/async-ffi/pull/10>.
- [80] *Procedural macro for boilerplate* — GitHub [oxalica/async-ffi](https://github.com/oxalica/async-ffi). 2021. URL: <https://github.com/oxalica/async-ffi/issues/12>.
- [81] *Thread-safety — Libloading v0.7.1*. 2021. URL: <https://docs.rs/libloading/0.7.1/libloading/struct.Library.html#thread-safety>.
- [82] *dlerror* — *The Open Group Base Specifications*. 2021. URL: <https://pubs.opengroup.org/onlinepubs/009604499/functions/dlerror.html>.
- [83] *dlerror attributes* — *Linux Manual Page*. 2021. URL: <https://man7.org/linux/man-pages/man3/dlerror.3.html#ATTRIBUTES>.
- [84] *dlerror* — *Mac OS X Man Pages*. 2021. URL: https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/dlerror.3.html.
- [85] *SetThreadErrorMode* — *Microsoft Documentation*. 2021. URL: <https://docs.microsoft.com/en-us/windows/win32/api/errhandlingapi/nf-errhandlingapi-setthreaderrormode>.
- [86] *src/raw/unix.rs* — GitHub [szymonwieloch/rust-dlopen](https://github.com/szymonwieloch/rust-dlopen). 2021. URL: <https://github.com/szymonwieloch/rust-dlopen/blob/f9fed7168dfdcfd682a6c409595d4e2430b2dd2d/src/raw/unix.rs#L17>.
- [87] *dlerror is thread-safe on some platforms*. 2021. URL: <https://github.com/szymonwieloch/rust-dlopen/issues/42>.
- [88] *Add support for StableAbi* — GitHub [simd-lite/value-trait](https://github.com/simd-lite/value-trait). 2021. URL: <https://github.com/simd-lite/value-trait/pull/14>.
- [89] *abi_stable/src/external_types* — GitHub [rodrimati1992/abi_stable_crates](https://github.com/rodrimati1992/abi_stable_crates). 2021. URL: https://github.com/rodrimati1992/abi_stable_crates/tree/edfb2a97a7b5d7ecbc29c1f9f115f61e26f42da6/abi_stable/src/external_types.

- [90] *Diving Deep: implied bounds and variance* #25860. 2015. URL: <https://lcnr.de/blog/diving-deep-implied-bounds-and-variance/>.
- [91] Jon Gjengset. *Crust of Rust: Subtyping and Variance*. 2021. URL: <https://www.youtube.com/watch?v=iVYWDIW71jk>.
- [92] *lifetimes with R* types break compared to non R* types* — GitHub [rodrimati1992/abi_stable_crates](https://github.com/rodrimati1992/abi_stable_crates). 2021. URL: https://github.com/rodrimati1992/abi_stable_crates/issues/75.
- [93] *Tremor Linked Transports*. 2021. URL: <https://www.tremor.rs/docs/0.11/operations/linked-transports/>.
- [94] *Tremor Connectors*. 2021. URL: <https://www.tremor.rs/docs/next/reference/connectors/>.
- [95] *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](https://github.com/rust-lang/rust). 2021. URL: <https://github.com/rust-lang/rust/issues/90608>.

Lista de Figuras

3.1. Ejemplo de uso básico de Tremor	19
3.2. Ejemplo de una <i>pipeline</i> definida para Tremor	20
3.3. Simplificación del <i>trait</i> Connector	26
3.4. Registro de un conector en el programa	27
3.5. Inicialización de un conector en el programa	28
3.6. Configuración de un conector en el programa	29
4.1. La estructura ideal para un sistema de plugins.	35
4.2. La estructura inicial del sistema de plugins para un desarrollo más rápido.	37
4.3. Ejemplo de interfaz definida con <i>witx</i>	41
4.4. Latencia vs. Tamaño de Mensaje [56].	44
4.5. Rendimiento vs. Tamaño de Mensaje [56].	44
4.6. Ejemplo de cómo sería un plugin escrito con Rust.	48
4.7. El mismo plugin que la Figura 4.6, pero usando el ABI de C.	48
4.8. El mismo plugin que la Figura 4.6, pero con <i>abi_stable</i> . La declaración de la global <i>cached</i> se omite por simplicidad.	50
4.9. Comparación aproximada de los métodos investigados.	51
5.1. Al intentar evitar los tipos externos se produjeron más de 120 errores de compilación.	62
5.2. Comunicación entre runtime y plugins en el PDK.	64
5.3. La estructura ideal para un sistema de plugins.	70
5.4. La estructura inicial del sistema de plugins para un desarrollo más rápido.	71
A.1. Error de compilación de <i>rustc</i> , relacionado con la compilación incremental y arreglado ya en una futura versión [95].	89

Lista de Tablas

Anexos

Anexos A

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 (e.g., preguntas o ideas descartadas)
- Fueron repetitivas (e.g., tuve que realizar varios pull requests idénticos en Tremor cuando arreglaba problemas con Git)

Pull Requests:

- “Support for `abi_stable`” — GitHub oxalica/async-ffi #10
<https://github.com/oxalica/async-ffi/issues/10>
- TODO: copiar del blog

Issues:

- TODO: copiar del blog

Pull requests o issues relacionados directamente con el sistema de plugins de Tremor:

– TODO: copiar del blog

Adicionalmente, aunque no esté relacionado directamente con contribuciones de código abierto, otro logro del que me siento extrañamente orgulloso es de romper el mismo compilador de Rust una vez, como se ve en la Figura A.1.

TODO: a lo mejor sobra esto? A mi la verdad que me gusta

```

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 A.1: Error de compilación de `rustc`, relacionado con la compilación incremental y arreglado ya en una futura versión [95].