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

Padre y madre o familia

Amigos, de uni y de siempre

Profesores, en especial Fabra

Mentores de Tremor, comunidad open source, fundación de linux, wayfair?

RESUMEN

TODO, en castellano

ABSTRACT

TODO, en inglés

Índice

Capítulo 1

Introducción

1.1. Contexto

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

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

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

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

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

1.2. Objetivo

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

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

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

1.3. Motivación

1.3.1. Tiempos de compilación

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

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

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

1.3.2. Modularidad

Otra ventaja que provee un sistema de plugins es modularidad; ser capaz de tratar la runtime y los plugins de forma separada suele resultar en una arquitectura más limpia [3]. También hace posible el desacoplamiento del ejecutable y sus componentes; algunas dependencias tienen un ciclo de versionado más rápido que otras y generalmente es más conveniente actualizar únicamente un plugin, en lugar del programa por completo.

1.3.3. Aprender de otros

Otros proyectos maduros con características similares a las de Tremor, como *NGINX* [4] o *Apache HTTP Server* [5], llevan beneficiándose de un sistema de plugins desde hace mucho. Informan mejorías en flexibilidad, extensibilidad y facilidad de desarrollo [6][7]. Aunque las desventajas también mencionen un pequeño impacto en el rendimiento y la posibilidad de caer en un *dependency hell*, sigue siendo una buena idea al menos considerarlo para Tremor.

1.4. Metodología

1.4.1. Organización

El proyecto ha tenido una duración de unos 10 meses, comenzando en agosto de 2021 y terminando en mayo/junio de 2022. Su realización ha sido completamente remota y con horarios muy flexibles. Se usó el servidor de Discord de Tremor[9] como plataforma principal para comunicarse, tanto por texto como por videollamada. Se programó una llamada por semana, en la que explicaba mi progreso y recibía ayuda de mis mentores en caso de que me hubiera quedado atascado en algún momento.

Disponía de tres mentores, que me guiaban en el proceso de desarrollo: Darach Ennis (*Principal Engineer and Director of Tremor Project*), Matthias Wahl (*Staff Engineer*) y Heinz N. Gies (*Senior Staff Engineer*), todos empleados por Wayfair.

La organización de forma más estructurada para las tareas que tenía pendientes, en las que estaba trabajando en ese momento, y las que ya había realizado, se basó principalmente en un Kanban en GitHub[10].

1.4.2. Desarrollo

Para reducir el coste de desarrollo y asegurarse de que el proceso sea completamente seguro (en memoria y concurrencia), el sistema de plugins aprovecha librerías existentes en Rust y herramientas como macros procedurales. El sistema de compilación usado es solución oficial de Rust: Cargo, que también incluye un *formatter*, *linter*, y extensiones instalables creadas por la comunidad. Adicionalmente, existe una gran cantidad de tests y *benchmarks* que se han de tener en cuenta para mantener el *Code Coverage* (la cantidad de código cubierta por los tests) y el rendimiento.

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

1.4.3. Recursos públicos

Este trabajo está disponible públicamente al completo. Además, a medida que he investigado e implementado el sistema de plugins, he ido escribiendo todo en mi blog personal, *NullDeref*; gran parte de los contenidos de este documento se han obtenido de ahí. Tiene una serie con un total de 5 artículos que entran en gran detalle y suman unas 2 horas adicionales de lectura.

- El repositorio de GitHub para el binario de Tremor:
  `https://github.com/tremor-rs/tremor-runtime`
- Mi *fork*, con ramas adicionales usadas durante el desarrollo:
  `https://github.com/marioortizmanero/tremor-runtime`
- Mi repositorio con experimentos antes de implementar la versión definitiva:
  `https://github.com/marioortizmanero/pdk-experiments`
- La serie de artículos en mi blog personal:
  `https://nullderef.com/series/rust-plugins/`.

Capítulo 2

Entendiendo Tremor

2.1. 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)" [8]. Tremor fue creado como una alternativa de alto rendimiento a herramientas como *Logstash* [9] o *Telegraf* [10], 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 [11] y Cugola y Margara [12] 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 [13] y Tawsif y col. [14] 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.

2.2. Casos de uso

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

1. Recibir *logs* (eventos) de aplicaciones en diferentes protocolos o formatos. Es posible que esta heterogeneidad se deba a que algunas aplicaciones son legadas y no se puedan reducir a un único protocolo o formato, o que esta tarea es demasiado compleja como para gestionarse a nivel de aplicación.

2. Filtrar los eventos redundantes, añadir campos nuevos o eliminar aquellos innecesarios y transformar todo a un mismo formato. El uso de una herramienta ineficiente o *ad hoc* por la empresa podría ser inviable dada una cantidad de datos suficientemente grande o demasiados protocolos y formatos como para implementarlos todos.

3. Enviar todos los logs estructurados a una base de datos para analizarlos posteriormente.

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

2.3. 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 [15].

- 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 [16].

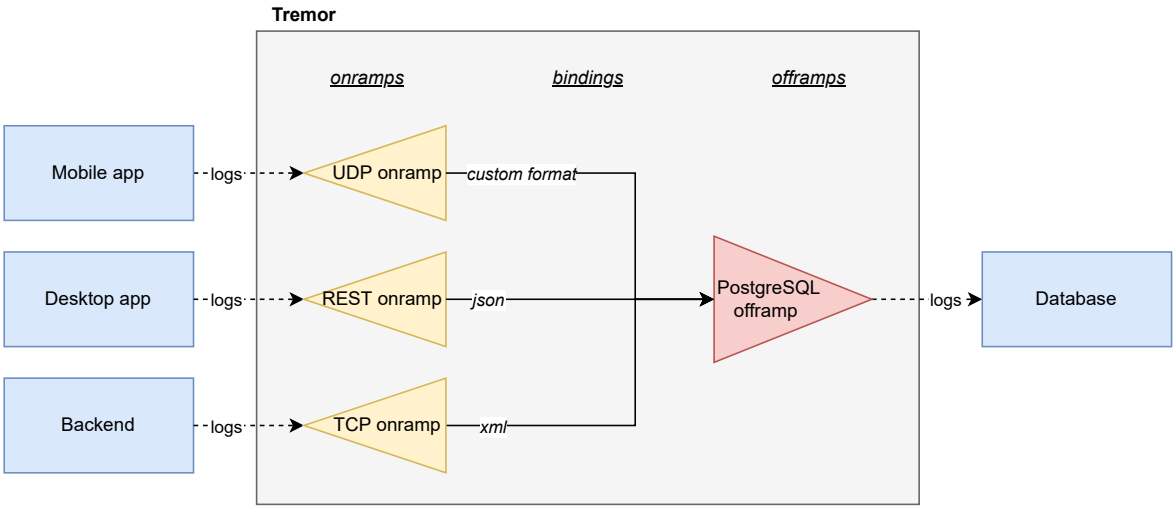

Figura 2.1: Ejemplo de uso básico de Tremor

– Una *pipeline* es una lista de operaciones (transformación, agregación, eliminación, etc) a través de la cual se pueden encaminar los eventos [17]. La Figura 2.2 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 procesado se basa en las siguientes operaciones:

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

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

– *Matching*: búsqueda de partes de los eventos que siguen un patrón en específico (e.g., un campo `"id"` con un valor 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:

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

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

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

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

Figura 2.2: 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* [18], que introduce sus conceptos más básicos y sus posibles usos — o cuándo *no* usarlo, en *Tremor Constraints and Limitations* [19]. *Tremor Recipes* [20] lista un total de 32 ejemplos de cómo configurar y emplear el software.

2.4. 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 2.3. 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`

2.5. 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".

2.6. 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.

2.6.1. Registro

La Figura 2.4 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.

2.6.2. 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 2.5.

2.6.3. Configuración

Una vez haya un conector corriendo, la Figura 2.6 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.

2.6.4. Notas adicionales

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

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

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

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

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

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

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

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

Figura 2.3: Simplificación del *trait* `Connector`

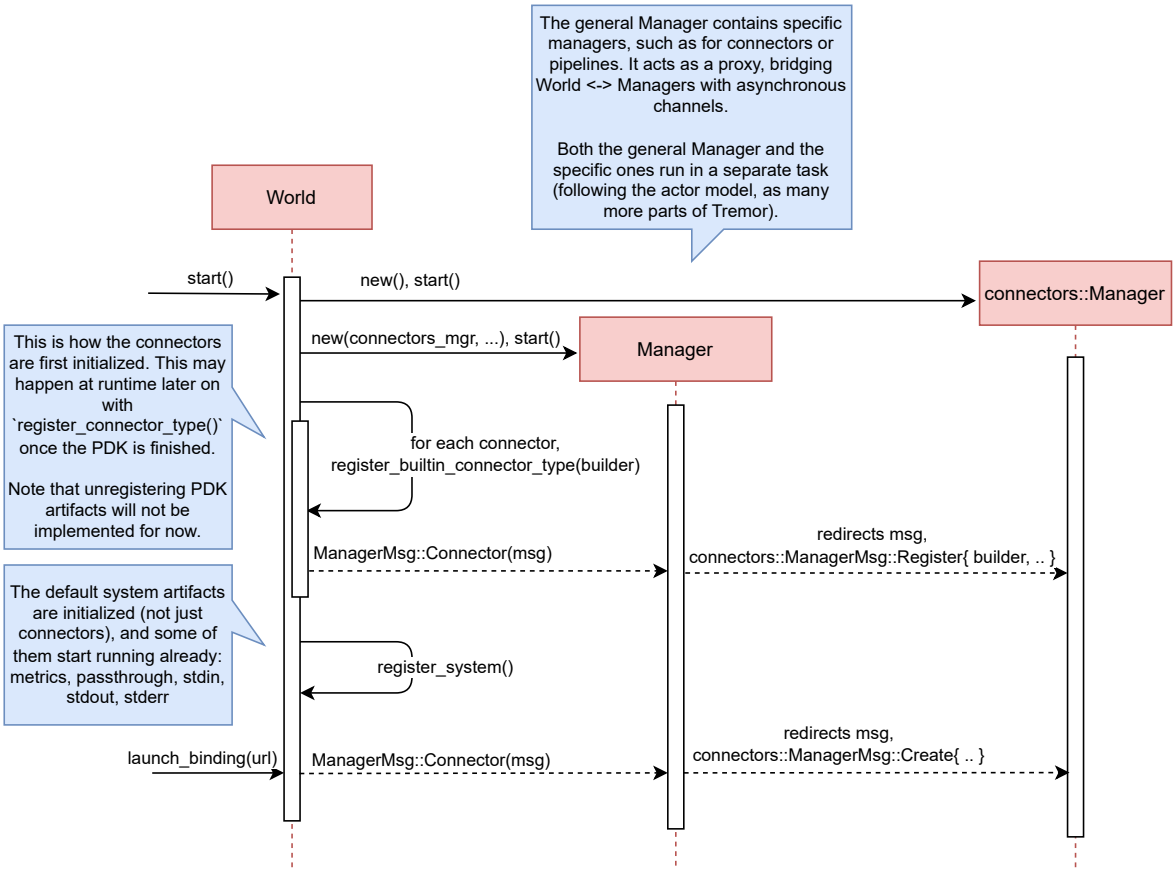

Figura 2.4: Registro de un conector en el programa

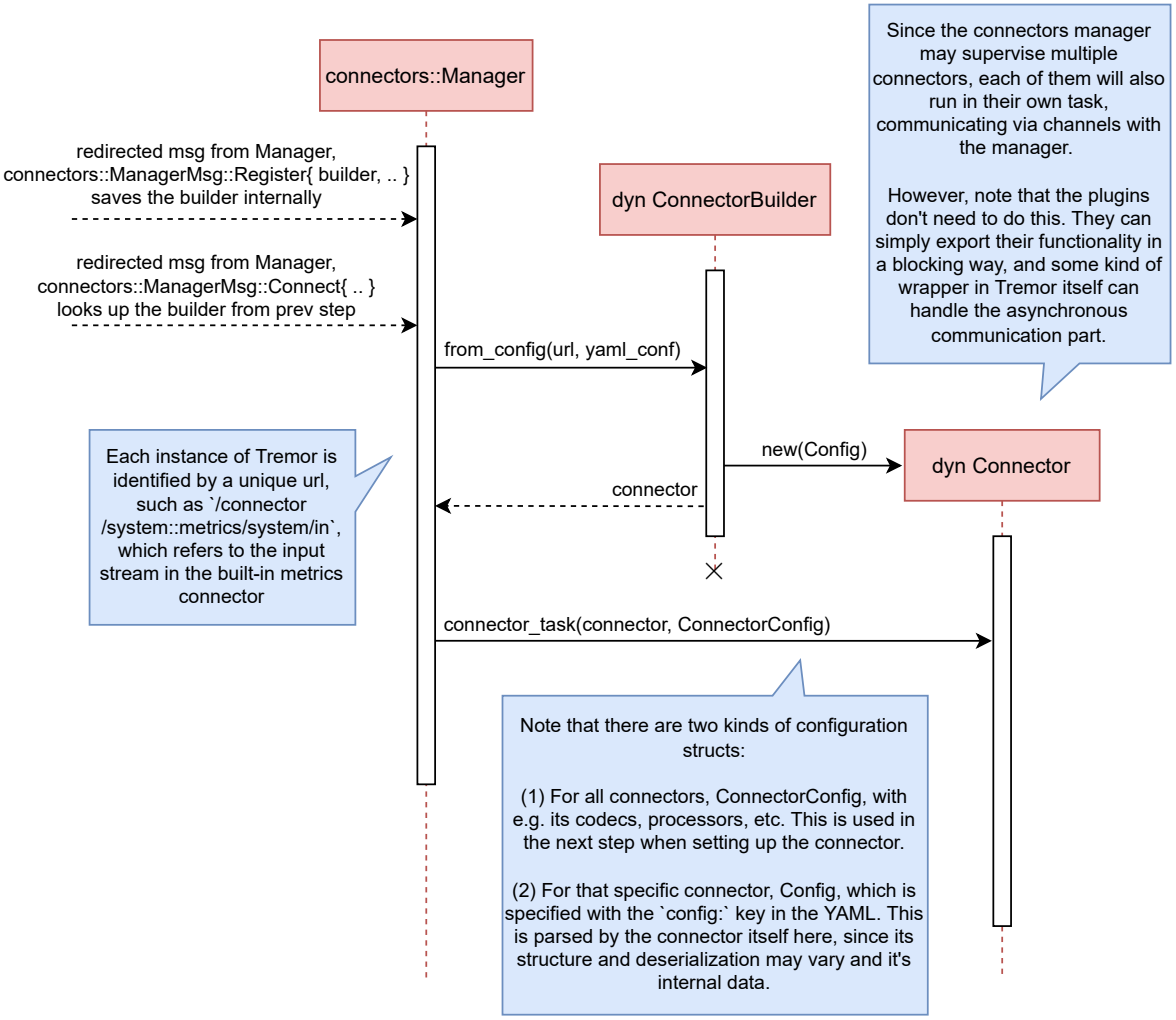

Figura 2.5: Inicialización de un conector en el programa

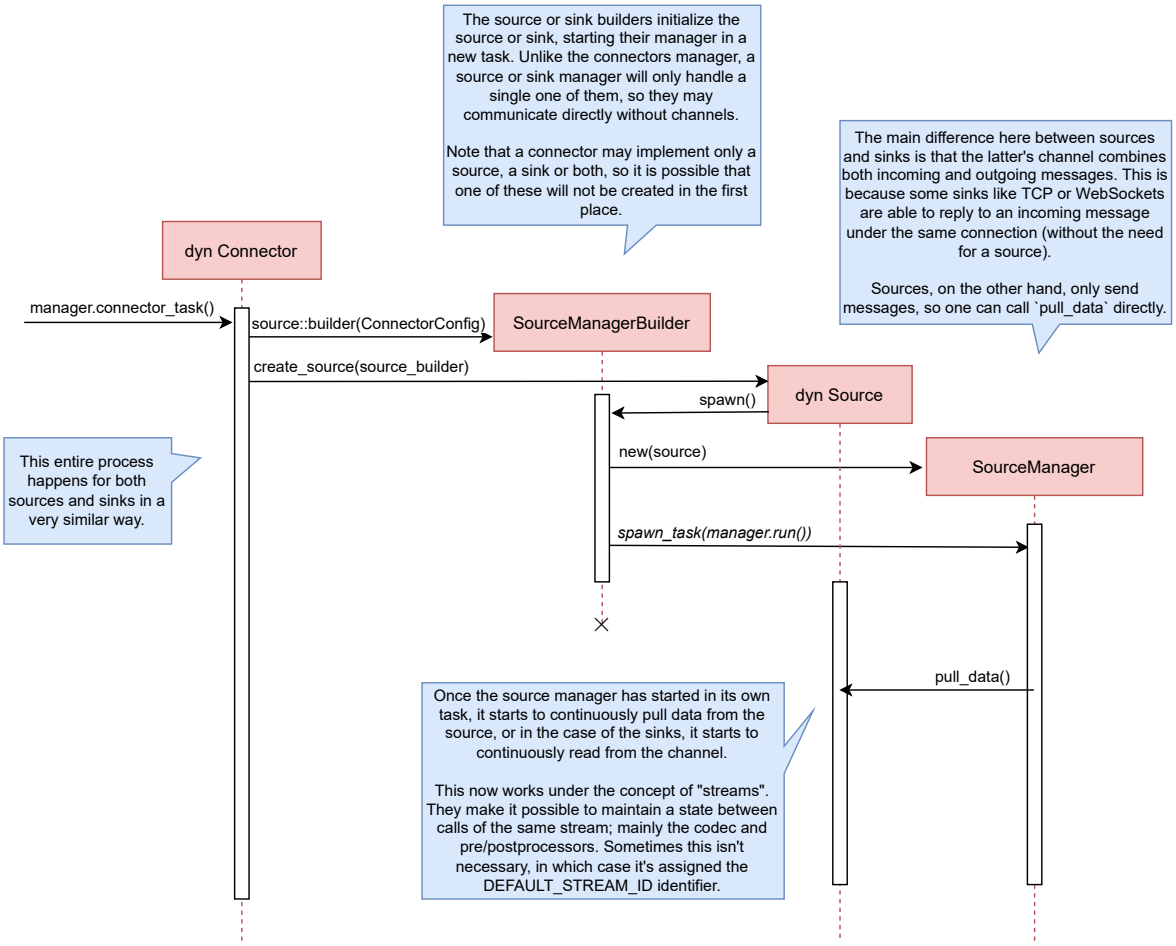

Figura 2.6: Configuración de un conector en el programa

Capítulo 3

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 3.1.
- Debería ser retro-compatible con el código ya existente, como indica la sección 3.2.

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

3.1. Seguridad

Muchas de las tecnologías que se pueden aplicar para un sistema de plugins usan código `unsafe`. Técnicamente, esto no es necesariamente un problema si la implementación está autocontenida y auditada exhaustivamente, pero se pierden algunas garantías que proporcionadas por Rust, incrementando el coste de mantenimiento de la librería.

Asegurarse de que la implementación es segura implica una cantidad considerablemente mayor de trabajo, aun cuando existen herramientas como MIRI[1] — que integraría en Tremor en caso de tener que recurrir a `unsafe`.

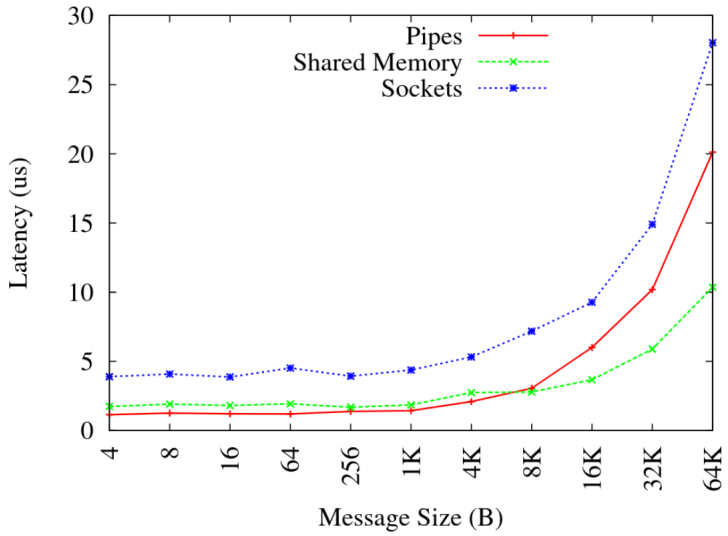

Figura 3.1: Latencia vs. Tamaño de Mensaje [21]

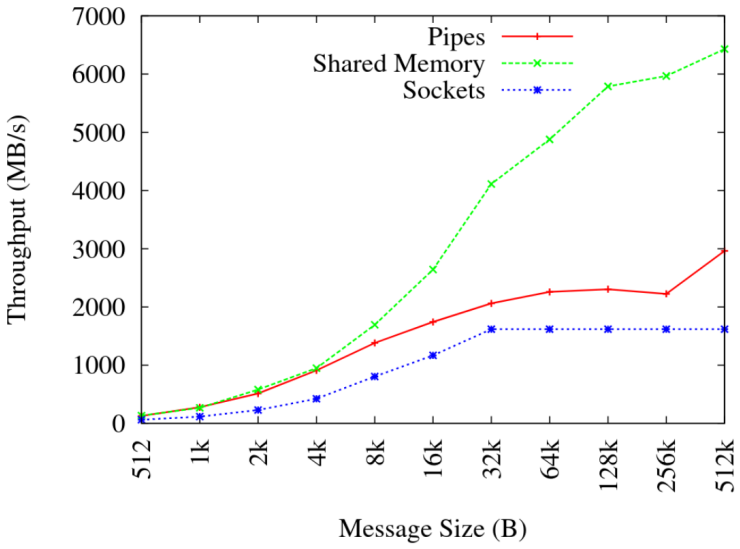

Figura 3.2: Rendimiento vs. Tamaño de Mensaje [21]

3.2. Retro-compatibilidad

3.3. Tecnologías a considerar

3.3.1. Lenguajes de *scripting*

3.3.2. Comunicación Inter-Proceso

3.3.3. Cargado dinámico

3.3.4. WebAssembly

3.3.5. eBPF

3.4. Conclusión

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

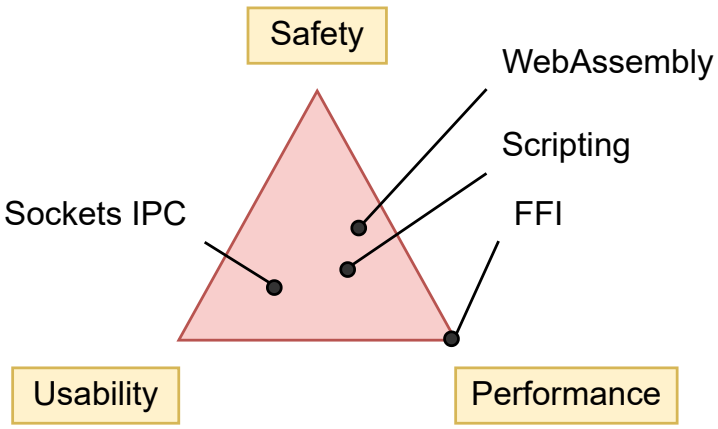

Figura 3.3: Ejemplo de uso de Tremor

Capítulo 4

Implementación

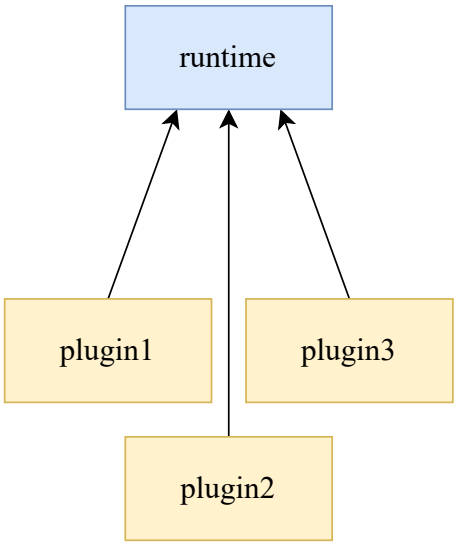

Figura 4.1: Ejemplo de uso de Tremor

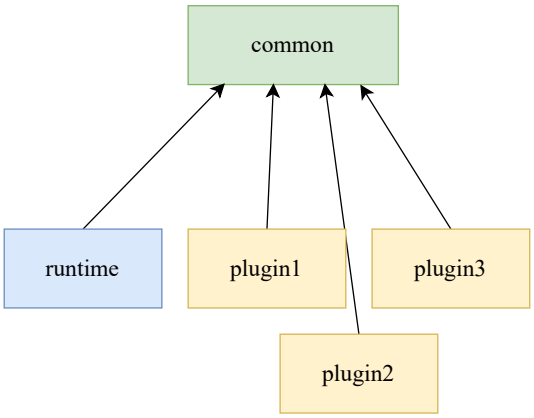

Figura 4.2: Ejemplo de uso de Tremor

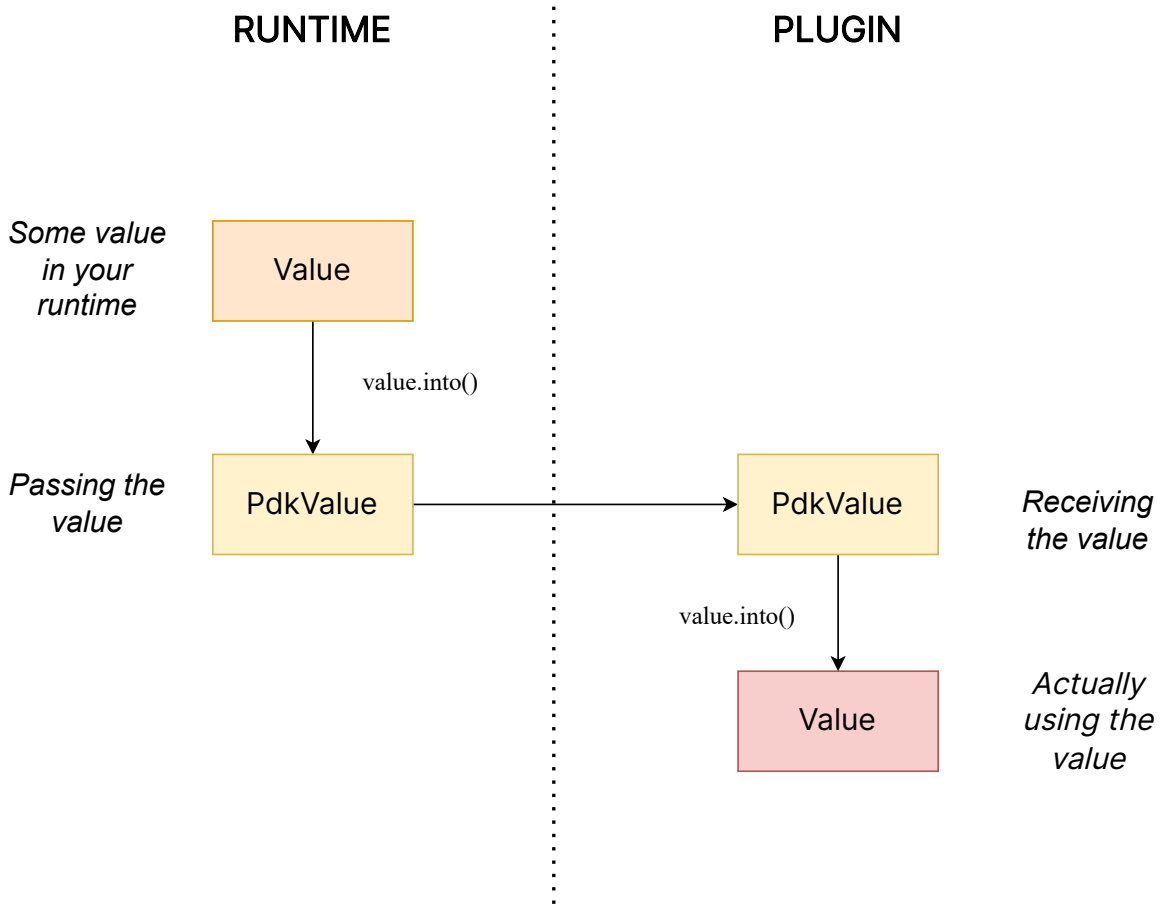

Figura 4.3: Ejemplo de uso de Tremor

Capítulo 5

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

[9] *Logstash*. 2011. URL: https://www.elastic.co/logstash/.

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

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

[12] 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.

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

[14] 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.

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

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

[17] *Tremor Pipelines*. 2021. URL: https://www.tremor.rs/docs/next/language/#pipelines.

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

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

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

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

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

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

Lista de Figuras

Lista de Tablas

Anexos

Anexos A

Contribuciones de código abierto

TODO: copiar de blog