Rust in produzione

Cristiano Chieppa

Smart Security

Cosa facciamo

Nasciamo nel 2015.

I primi progetti sono in ambito sicurezza (video sorveglianza) da cui il nome.

Da allora sono stati messi in produzione tanti prodotti e servizi in ambiti diversi: medicale, manufacturing, banking, automotive, IoT.

Rust è già tra noi

AZURE — RUST — MICROSOFT — CLOUD — CYBERSECURITY — NEWS

Azure rewrites it in Rust

"One of the most significant security improvements in Azure history"

EDWARD TARGETT

September 19, 2024 . 1:19 PM — 2 min read

AWS Open Source Blog

Why AWS loves Rust, and how we'd like to help

by Matt Asay | on 24 NOV 2020 | in Announcements, Open Source, Programing Language | Permalink | Comments | Share

One of the most exciting things about the Rust programming language is that it makes infrastructure incredibly boring. That's not a bad thing, in this case. No one wants their electrical wiring to be exciting; most of us prefer the safety that comes with being able to flip a switch and have light to see by. For similar reasons, at AWS we increasingly build critical infrastructure like the Firecracker VMM using Rust because its out-of-the-box features reduce the time and effort needed [illegible]formance similar to C and C++.

Microsoft continues push to switch code over to Rust

FLORIS HULSHOFF POL
5 feb 2024, 15:57 CET

Microsoft is continuing the push to move over all of its C code to Rust in all of its product portfolios. The company is doing this as part of an acceleration of the previously established "Rewrite Everything in Rust" initiative.

Microsoft has been working for some time to replace the currently C/C++-based kernel code in all its products, from Windows and Office to the public cloud Azure, with Rust-written code. Reasons for this replacement include the fact that the old programming languages still contain many memory issues. It costs developers a lot of time to fix these errors.

30 year old code killed! Microsoft rewrites Windows kernel with 180,000 lines of Rust

Aaron 0928 · Follow
5 min read · May 15, 2023

3K 27

Rust's disruption of C has already begun.

Microsoft is rewriting its core Windows libraries using the Rust programming language.May 11 — The latest Windows 11 Insider P[illegible] release is the first to include the memory-safe programming langu[illegible] according to Azure Chief Technology Officer Mark Russinovich.

"If you're on the Win11 Insider ring, you'll get your first taste of Rust in the Windows kernel," Russinovich tweeted last night.

Mark Russinovich
@markrussinovich

If you're on the Win11 Insider ring, you're getting the first taste of Rust in the Windows kernel!

```
C:\Windows\System32>dir win32k*
 Volume in drive C has no label.
 Volume Serial Number is E60B-9A9E

 Directory of C:\Windows\System32

04/15/2023  09:50 PM           708,608 win32k.sys
04/15/2023  09:49 PM         3,424,256 win32kbase.sys
04/15/2023  09:49 PM           110,592 win32kbase_rs.sys
04/15/2023  09:50 PM         4,194,304 win32kfull.sys
04/15/2023  09:49 PM            40,960 win32kfull_rs.sys
04/15/2023  09:49 PM            69,632 win32kns.sys
04/15/2023  09:49 PM            98,304 win32ksgd.sys
               7 File(s)      8,646,656 bytes
               0 Dir(s)  116,366,049,280 bytes free
```

_rs = Rust!

Cost Efficiency

By optimizing performance and resource usage, Rust can significantly contribute to cost savings on AWS. Efficient code execution can reduce the need for high-end, costly instances, allowing businesses to achieve their performance goals at a lower cost. Rust's ability to handle high concurrency and throughput without extensive resource consumption enables developers to optimize AWS infrastructure usage effectively. Furthermore, Rust's focus on safety and performance can reduce the frequency of bugs and runtime errors, which in turn minimizes downtime and the need for costly debugging and maintenance. For cloud environments where every second of runtime and every byte of memory counts, Rust's efficiency translates into tangible cost benefits.

Integration with AWS Services

Integration with AWS services is streamlined by the AWS SDK for Rust, which provides APIs to interact with a wide range of AWS services like Amazon S3, DynamoDB, AWS Lambda, and more. This SDK is designed to be idiomatic to Rust, offering type-safe interactions with AWS services, which reduces runtime errors and increases developer productivity. Using Rust, developers can build robust cloud-native applications that integrate deeply with the AWS ecosystem. For example, they can leverage AWS's managed services for databases, machine learning, and analytics directly within Rust applications, creating highly integrated and efficient cloud solutions. This makes Rust an appealing choice for developers looking to harness the full power of AWS services with minimal friction.

TechPort

Search by keyword or select Advanced Search

Advanced Search

Center Independent Research & Development: GSFC IRAD

Rust in cFS: Prevent Bugs with Memory-Safe Programming

Completed Technology Project | 119 views

Download PDF | Download Excel

Project Description

This project will provide Rust language support for NASA's core Flight System (cFS). The Rust language is designed to be memory-safe: it detects a wide range of programmer errors at compile-time while allowing low-level access to hardware and high performance. Rust's safety features make it ideal for writing new cFS applications.

Rust cFS Application

spacexfsw OP · 14h
Official SpaceX

2 Awards

We are definitely excited about Rust! Its emphasis on safety, performance, and modern tooling all stand out. We're also excited that we could use one language across embedded systems, simulators, tooling, and web apps. We are starting to prototype some new projects in Rust, but we are certainly just at the beginning of this journey.- Asher

Perché Rust

Microsoft | MSRC Report an issue Customer guidance Engage Who we are Blogs Acknowledgments

Blog / 2019 / 07 / a proactive approach to more secure code

A proactive approach to more secure code

Security Research & Defense / By MSRC Team / July 16, 2019

What if we could eliminate an entire class of vulnerabilities before they ever happened?

Since 2004, the Microsoft Security Response Centre (MSRC) has triaged every reported Microsoft security vulnerability. From all that tria
astonishing fact sticks out: as Matt Miller discussed in his 2019 presentation at BlueHat IL, the majority of vulnerabilities fixed and with
assigned are caused by developers inadvertently inserting memory corruption bugs into their C and C++ code. As Microsoft increases i
base and uses more Open Source Software in its code, this problem isn't getting better, it's getting worse. And Microsoft isn't the only
exposed to memory corruption bugs—those are just the ones that come to MSRC.

Figure 1

Google's Rust belts bugs out of Android, helps kill off unsafe code substantially

Memory safety flaws used to represent 76% of 'droid security holes. Now they account for 24%

Thomas Claburn Wed 25 Sep 2024 // 17:00 UTC

Google says its effort to prioritize memory-safe software development over the past six years has substantially reduced the number of memory safety vulnerabilities in its

Feb 25, 2019

Rust Case Study:

Community makes Rust an easy choice for npm

The npm Registry uses Rust for its CPU-bound bottlenecks

PRODUCT CUSTOMERS PRICING SOLUTIONS DOCS DATADOG ABOUT BLOG LOGIN GET STARTED FREE

ENGINEERING

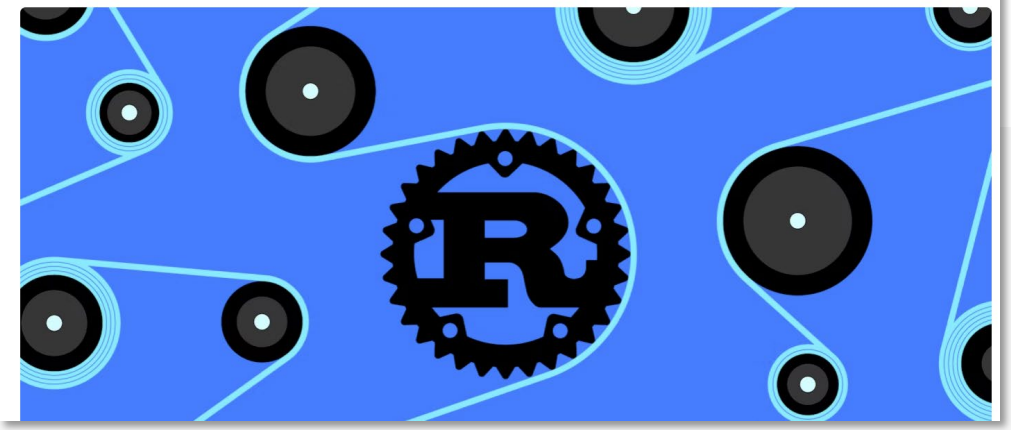
9 MINUTE READ

Published MAY 23, 2024

Share

How we migrated our static analyzer from Java to Rust

JAVA



Caratteristiche di Rust 1/3

Sicurezza della memoria (ownersh
borrowing)

Sicurezza nei thread

Type safe

Prestazioni elevate

Efficace sistema di async-await

```
1: let mut v = vec![1, 2, 3];
2: let num = &v[2];
3: Vec::push(&mut v, 4);
4: println!("{}", *num);
```

Immutable / shared reference

Mutable / unique reference

```
$ rustc vec.rs
error: cannot borrow
`v` as mutable because
it is also borrowed as
immutable
```

Caratteristiche di Rust 2/3

Sicurezza della memoria (ownership e borrowing)

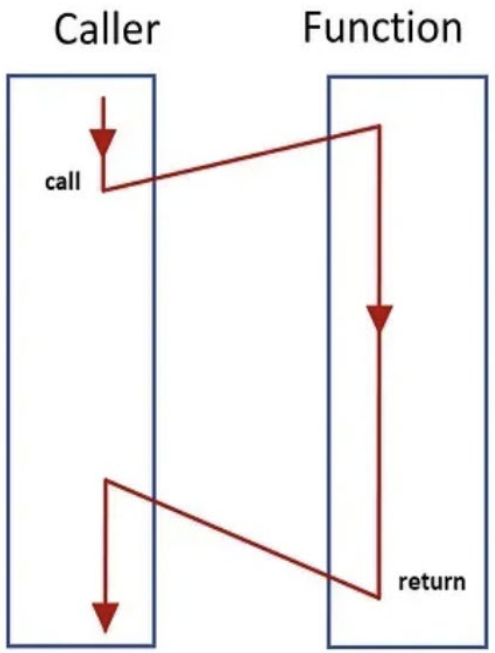
Sicurezza nei thread

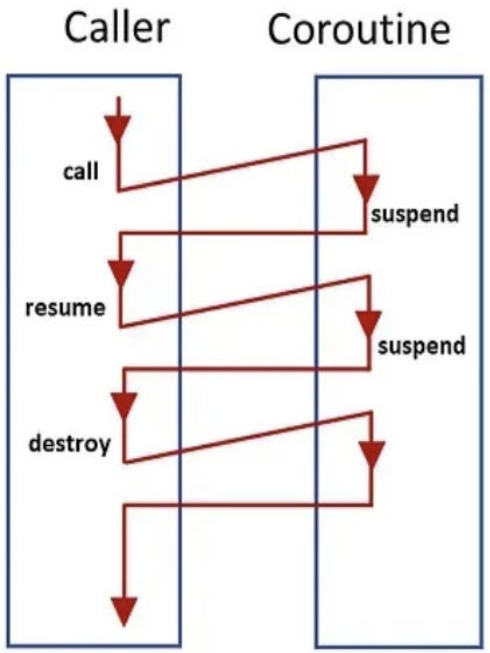
Type safe

Prestazioni elevate

Efficace sistema di async-await

Caratteristiche di Rust 3/3






```rust
use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};

/// La stessa Future che conta i poll.
struct CounterFuture {
    count: usize,
    target: usize,
}

impl Future for CounterFuture {
    type Output = String;

    fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output> {
        let this = self.get_mut();

        if this.count >= this.target {
            println!("✅ CounterFuture: Pronto! (Conteggio: {})", this.count);
            Poll::Ready(format!("Future completata dopo {} poll.", this.count))
        } else {
            this.count += 1;
            println!("⏳ CounterFuture: In attesa... (Conteggio: {})", this.count

            // Cruciale: notifica all'esecutore di riprovare a chiamare 'poll'
            // al prossimo ciclo. Questo evita che l'esecutore ignori la future
            // finché non c'è un evento esterno.
            cx.waker().wake_by_ref();

            Poll::Pending
        }
    }
}
```

Borrow Checker salva da errori subdoli

```cpp
// ========== PARTE 1: Le strutture dati ==========
struct Image {
    std::vector<uint8_t> rawData;  // Contiene i pixel dell'immagine
    size_t width;
    size_t height;
};

// ========== PARTE 2: Library wrapper per C (semplificata) ==========
class VisionLibrary {
private:
    const uint8_t* savedImagePointer;  // ⚠️ Salva un PUNTATORE ai dati!

public:
    void start(const std::vector<uint8_t>& imageData, ImageInfo info) {
        // ⚠️ PROBLEMA: Salva il puntatore per processamento asincrono
        savedImagePointer = imageData.data();

        // Avvia un thread worker che processerà l'immagine più tardi
        std::thread([this]() ->void {
            std::this_thread::sleep_for(100ms);  // Simula lavoro asincrono

            // ❌ USA IL PUNTATORE SALVATO
            processImageData(savedImagePointer);  // BOOM! Potrebbe essere invalido!
        }).detach();
    }
};

// ========== PARTE 3: Il nostro codice (VERSIONE BUGGY) ==========
class ImageProcessor {
public:
    void processImage(const Image& image) {
        // ❌ CATTURA PER COPIA
        boost::asio::post(executor,
            [self = shared_from_this(), image, context]() mutable ->void {
            //                  ^^^^^
            //                  Copia di Image!
                // Passa image.rawData alla vision library
                context->getVlInstance().start(
                    image.rawData,  // std::vector<uint8_t>
                    {.imageWidth = image.width, .imageHeight = image.height}
                );
            }
        );
        // La lambda esce da questo scope e potrebbe essere mossa
    }
};
// Cosa succede:
// 1. Lambda viene creata con una COPIA di image
// 2. boost::asio::post MUOVE la lambda internamente
// 3. Il move della lambda causa il move di image.rawData
// 4. Il vector originale dopo move ha data() == nullptr
// 5. vision_library ha un puntatore DANGLING → CRASH!
```

```rust
// ========== Rust equivalente ==========
struct Image {
    raw_data: Vec<u8>,
    width: usize,
    height: usize,
}

struct VisionLibrary {
    // Rust NON ti permette di salvare un puntatore nudo senza lifetime!
    // saved_image_pointer: *const u8,  X Unsafe! Devi dichiarare il lifetime
    // Meglio:
    image_data: Vec<u8>,
    ...
}

impl ImageProcessor {
    // X QUESTO NON COMPILA
    fn process_image_broken(&self, image: &Image) {
        let executor = self.executor.clone();

        executor.spawn(async move {
            // ERROR: `image` does not live long enough
            //        cannot move a borrowed reference into async block
            self.vision_lib.start(&image.raw_data);
            //                    ^^^^^^^^^^^^^^^^
            //                    borrowed value does not live long enough
        });
        // `image` dropped here while still borrowed
    }

    // ✅ RUST TI FORZA A SCEGLIERE UNA SOLUZIONE SICURA
    // Reference counting (shared ownership)
    fn process_image_arc(self, image: Arc<Image>) {
        executor.spawn(async move {
            // OK: Arc garantisce che Image viva finché qualcuno lo usa
            self.vision_lib.start(&image.raw_data).await;
        });
    }
```

Scenario: box per automotive

Sistema ARM Linux progettato per bikers e city car, con un'applicazione a servizi (*etc/systemd/system*) sviluppata in C per:

- Gestione percorsi tramite GPS e storico.
- Fences (definizione di confini da non oltrepassare)
- Rilevamento furto mediante sensori MEMS integrati sulla scheda
- Telemetria mezzo
- Integrazione di tutti i servizi con Azure, supportata da una console dedicata agli amministratori

Problemi:

- Codice C con scarsa documentazione.
- Nessuna memoria storica del sistema
- Memory leak vari
- No threads con CPU bloccata su alcuni task
- Core non sfruttati
- Massiccio uso di messaggistica D-Bus per interprocessing
- Difficile reverse engineering in poco tempo.
- Mancanza del log dai dispositivi.

Servizio di log remoto 1/2

Invio del journal come tar.gz a Azure per analisi da remoto allo spegnimento del mezzo.

Poche righe per inviare il file rispetto a *libcurl* e invio in async senza bloccare il main thread del servizio.

CPU con i task ripartiti equamente sui vari core.

Maggiore facilità della gestione degli errori (insito nel linguaggio)

```rust
/// Carica un file su ECP invocano l'opportuno endpoint
/// # Argomenti
/// * `sas_token` - Il token SAS per l'autenticazione
/// * `filename` - il file con il path da caricare, es: /tmp/journal.tar.gz
///
/// # Ritorno
/// Restituisce un risultato `std::io::Result<()>` che indica se l'operazione è andata a buon fine o meno.
async fn upload_log_file(sas_token: &str, filename: &str) -> std::io::Result<()> {
    task::block_in_place(|| {
        // crea il client reqwest
        let client = reqwest::blocking::Client::new();

        // prepariamo la form per il  multipart
        let form = multipart::Form::new()
            .file("File", filename)
            .map_err(|e| {
                Error::new(
                    ErrorKind::Other,
                    format!("Failed to add file to form: {}", e),
                )
            })?
            .text("Type", "journal")
            .text("Identifier", ".tar.gz");

        // Costruisci la richiesta HTTP
        let request = client
            .post(ECP_END_POINT)
            .header("Authorization", sas_token)
            .multipart(form)
            .timeout(HTTP_SEND_TIMEOUT)
            .build()
            .map_err(|e| Error::new(ErrorKind::Other, format!("Failed to build request: {}", e)))?;

        // Invia la richiesta
        let res = client
            .execute(request)
            .map_err(|e| Error::new(ErrorKind::Other, format!("Failed to send request: {}", e)))?;

        let response = res
            .error_for_status()
            .map_err(|e| Error::new(ErrorKind::Other, format!("Request failed: {}", e)))?;

        tracing::info!("response: {:?}", response);

        if response.status() != 204 {
            // fa proseguire l'app manager nello shutdown
            if let Err(e) = write_upload_status(0) {
                tracing::error!("Failed to write upload status: {}", e);
            }
```

Servizio di log remoto 2/2

Adozione di Axum per creare un REST server multi-thread e async con avanzato logging (Tower::Tracing), pulizia e leggibilità. L'end point viene invocato allo spegnimento del mezzo, triggerando il servizio.

Serializzazione/Deserializzazione naturale in Rust (non si scrive quasi nulla con Serde).

```rust
let app = Router::new()
    .route("/debug", post(handle_upload_journal))
    .with_state(state)
    .layer(
        ServiceBuilder::new()
            .layer(
                TraceLayer::new_for_http()
                    .on_request(|request: &axum::http::Request<_>, _span: &tracing::Span| {
                        tracing::info!(
                            "Received request: {} {}",
                            request.method(),
                            request.uri()
                        );
                    })
                    .on_response(
                        |response: &axum::http::Response<_>,
                         latency: std::time::Duration,
                         _span: &tracing::Span| {
                            tracing::info!(
                                "Response: {} (latency: {:?})",
                                response.status(),
                                latency
                            );
                        },
                    )
                    .on_failure(
                        |error: ServerErrorsFailureClass,
                         _latency: std::time::Duration,
                         _span: &tracing::Span| {
                            tracing::error!("Request failed: {}", error);
                        },
                    ),
            )
            .layer(TimeoutLayer::new(Duration::from_secs(60))),
    );

// creiamo il listener.
// Attenzione: se si cambia la porta, bisogna farlo anche in es-connection-manager.
let listener = match tokio::net::TcpListener::bind("127.0.0.1:56000").await {
    Ok(listener) => listener,
    Err(e) => {
        tracing::error!("Failed to bind to address: {}", e);
        return;
    }
```

Servizio di tracciamento GPS 1/4

Il servizio deve costantemente leggere le sentenze NMEA via UART da chip (Ublox) e trasferire al cloud le informazioni.

La vecchia versione C non gestiva il multi-thread, ma aveva una macchina a stati complessa.

La macchina a stati usa D-Bus con messaggi da/a altri moduli per avanzare.

Ci sono problemi con la logica della macchina a stati per cui non si riesce a domare la transizione da alcuni stati ad altri. Questo ha comportato *patch* e *workaround* che rendono il codice molto complesso da comprendere e da sistemare.

Risultato: il servizio non funziona come ci si aspetta e modificarlo è complesso.

Servizio di tracciamento GPS 2/4

La riscrittura in Rust ha permesso di usare il paradigma async-await e multi-thread per migliorare il carico dei task.

OOP e principio open/closed per modellare la parte Ublox e D-Bus (con zbus, crate nativo Rust)

Impiego di design pattern per sistemare e rendere minima la macchina a stati.

La parte complessa è stata quella con zbus per la mancanza di documentazione esaustiva.

```rust
/// # send_gps_data
/// Invia il signal dbus con i dati gps
///
/// # Arguments
/// * `gps_data` - Dati gps da inviare
///
/// # Returns
/// Restituisce un'istanza di Result<(), Box<dyn std::error::Error>>
pub async fn send_gps_data(
    &self,
    gps_data: &GpsData,
) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let last_emit_time = *self.last_emit_time.lock().unwrap();

    let now : Instant = Instant::now();
    if now.duration_since(last_emit_time) >= Duration::from_secs( secs: 1) {
        debug!("Sending to dbus {:?}", &gps_data);

        self.signal_emitter_gps : SignalEmitter
            .emit(
                interface: "it._____.GpsFixData.signal.Type",
                signal_name: "gps_fix_data",
                &(gps_data,),
            ) : impl Future<Output=Result<…>>
            .await?;

        // aggiorna il tempo dell'ultimo invio
        *self.last_emit_time.lock().unwrap() = now;
    }
    Ok(())
}
```

Servizio di tracciamento GPS 3/4

Tokio + zbus per la sottoscrizione async dei messaggi D-Bus senza appesantire la CPU.

```rust
/// # UpdateConfig
/// Struttura dati per la gestione della configurazione del modulo gps.
/// Questa struttura viene inviata da es-configuration-manager via signal dbus.
/// La struttura viene riempita esternamente.
20 usages
#[derive(Debug, Clone, Default, Serialize, Deserialize, Value, OwnedValue, Type)]
pub struct UpdateConfig {
    enable: u8,
    rate: u8,
    mode: u8,
}
```

```rust
/// # start_subscription
/// Fa partire il thread di sottoscrizione ai soli segnali dbus a cui siamo interessati.
///
/// # Returns
/// Restituisce un'istanza di Result<(), Box<dyn std::error::Error>>
async fn start_subscription(&mut self) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let mut dbus_cfg_update_stream : MessageStream = self.dbus_cfg_update_stream.clone();

    let tx_clone : Arc<Mutex<Sender<…>>> = Arc::clone(&self.tx);

    self.handle_thread_listener = TOKIO_RUNTIME.get().unwrap().spawn( future: async move {
        while let Some(msg : Message) = (&mut dbus_cfg_update_stream).try_next().await.unwrap() {
            let tx_clone : Arc<Mutex<Sender<…>>> = Arc::clone(&tx_clone);

            info!("Received signal: {:?}", &msg);

            let body : Body = msg.body();
            let body: zbus::zvariant::Structure = match body.deserialize() {
                Ok(b : Structure) => b,
                Err(e : Error) => {
                    warn!("Error: {:?}", e);
                    continue;
                }
            };

            let mut config : UpdateConfig = UpdateConfig::default();
            let fields : &[Value] = body.fields();
            let u_8 : &Value = fields.first().unwrap_or(&zvariant::Value::U8(0));

            if let Ok(v : u8) = u_8.downcast_ref::<u8>() {
                (&mut config).set_enable(v);
            } else {
                warn!("Failed to downcast Enable value to u8");
            }
```

PS 4/4

```rust
/// # parse_nmea_message
/// Processa il messaggio NMEA e lo pulisce da caratteri non pertinenti.
/// Passa poi il vettore di messaggi filtrati al parser NMEA.
///
/// # Arguments
/// * `buffer` - buffer contenente il messaggio NMEA
/// * `tx` - channel per la comunicazione tra threads
///
/// # Returns
/// Result<(), Box<dyn Error>>
1 usage  Cristiano Chieppa
async fn parse_nmea_message(
    buffer: &[u8],
    tx: Arc<Mutex<mpsc::Sender<UpdateConfig>>>,
) -> Result<(), Box<dyn Error + Send + Sync>> {
    let message = String::from_utf8_lossy(buffer);
    let filtered_message: String = message
        .chars()
        .filter(|c| c.is_ascii_graphic() || c.is_ascii_whitespace())
        .collect();
    let str = filtered_message.trim_start_matches(&['\r', '\n'][..]);

    // Dividi la stringa in singoli messaggi NMEA
    let messages: Vec<&str> = str.split('\r').collect();

    // Filtra solo i messaggi NMEA validi
    let valid_messages: Vec<&str> = messages
        .into_iter()
        .filter(|msg| msg.contains("$GPGGA") || msg.contains("$GNGGA") || msg.contains("$GNRMC"))
        .collect();

    for msg in valid_messages {
        if let Some(start) = msg.find('$') {
            let nmea_message : &? = &msg[start..];
            analyze_sentence(nmea_message, Arc::clone(&tx)).await?;
        }
    }
    Ok(())
```

```rust
let handle = tokio::spawn(async move {
    info!("Lettura nmea...");
    let mut buffer: [u8; 512] = [0u8; 512];

    loop {
        if WANT_TO_EXIT.load(Ordering::SeqCst) {
            info!("### exit nmea thread ###");
            break;
        }

        buffer.fill(0);

        // Se uart_nmea_clone.read() è bloccante, usa spawn_blocking
        let result = tokio::task::spawn_blocking({
            let uart = uart_nmea_clone.clone();
            let mut buf = buffer.clone();
            move || uart.lock().unwrap().read(&mut buf).map(|n| (n, buf))
        }).await.unwrap();

        if let Ok((_n, buf)) = result {
            if let Err(e) = parse_nmea_message(&buf.to_vec(), Arc::clone(&tx_clone))
                error!("Failed to parse NMEA message: {:?}", e);
            }
        }
    }
});
```

Benefici e problemi riscontrati

Benefici:

- Pochi mesi di sviluppo (3).
- Servizi sempre attivi, mai un crash nei test.
- L'async si sposa bene con l'ottimizzazione delle poche ri
  embedded con un carico ripartito dei core.
- Comunità open source estremamente vivace.
- Maggior rigore nella CI: ad es. la PR non passa se *clippy*
  hanno warning e si automatizza *fmt* e *rustdoc*.
- Un unico tool, cargo, per fare tutto: dalla compilazione a
  package manager, al test, all'analisi del codice, alla
  documentazione e tanto altro.

```rust
use tokio::time::Duration;

async fn sleep_then_print(timer: i32) {
    println!("Start timer {}.", timer);

    tokio::time::sleep(Duration::from_secs(1)).await;
//                                             ^ execution can be paused here

    println!("Timer {} done.", timer);
}

#[tokio::main]
async fn main() {
    // The join! macro lets you run multiple things concurrently.
    tokio::join!(
        sleep_then_print(1),
        sleep_then_print(2),
        sleep_then_print(3),
    );
}
```

```
Start timer 1.
Start timer 2.
Start timer 3.
Timer 1 done.
Timer 2 done.
Timer 3 done.
```

Altri progetti in sviluppo

Sistema di automazione industriale.

Front end disegna la UI e la logica da applicare in modo visuale (no/low code). Il backend in Rust con Axum e gRPC trasforma la UI in una rete di nodi. Ogni nodo ha una logica (es: timer, selector...). La rete dei nodi viene poi distribuita su pannelli ARM dove gira un secondo backend Rust in grado di eseguire i task e parlare con i sensori, inclusi i PLC. Beta a fine 2025.

Domotica - KNX su embedded.

PoC per azienda di ingegneria su Raspberry PI per sniffing del protocollo KNX da dispositivi in campo, invio dei dati a AWS IoT con broker MQTT e app desktop scritta in Slint per mostrare/impostare dati elementari (es: temperatura).

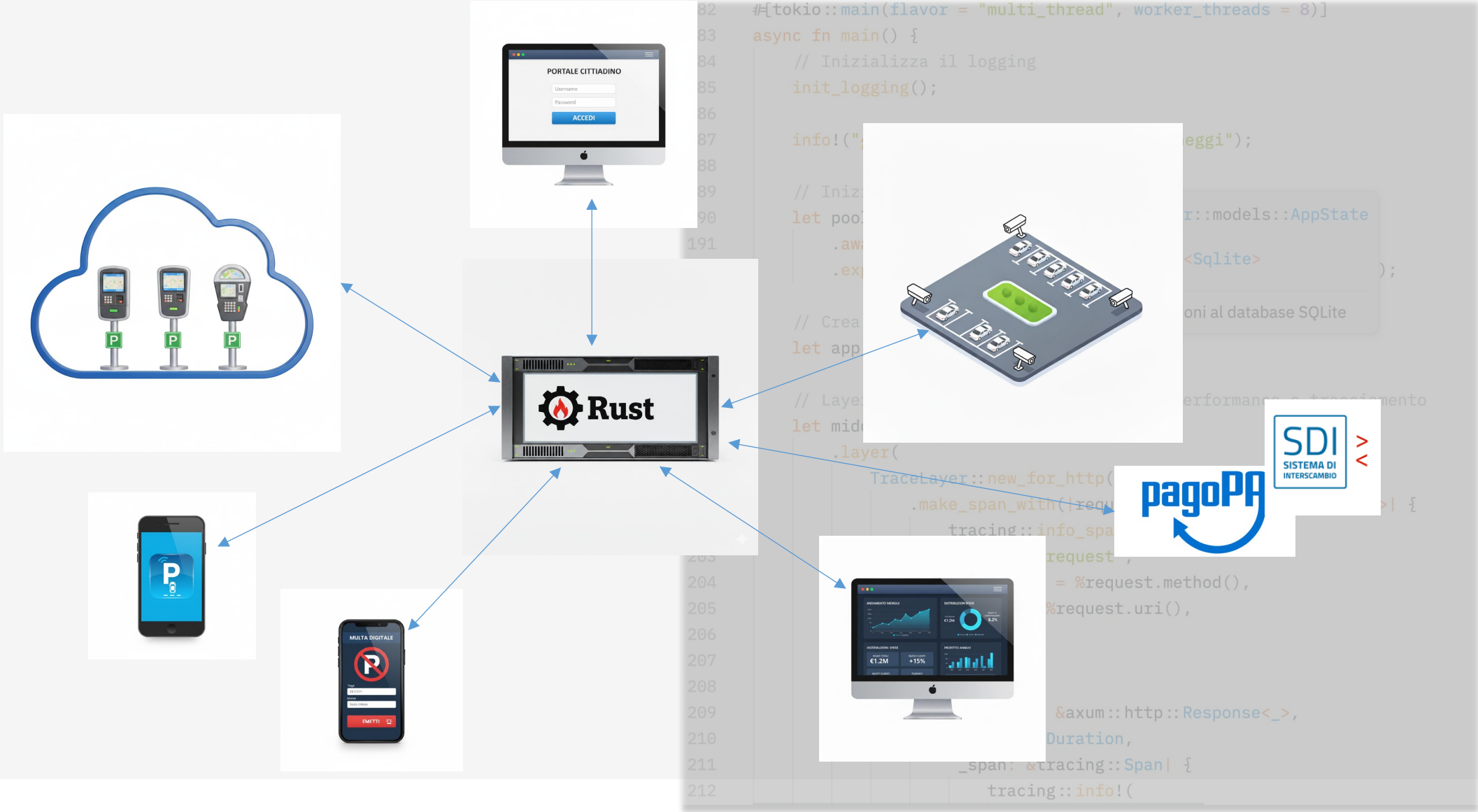
Sistema di gestione di parcheggio pubblico.

Pubblica Amministrazione

Sistema complesso in sviluppo composto da:

- Sistema di parchimetri collegati in cloud che parlano con il backend (Rust).
- Telecamere di sicurezza per il riconoscimento targa per parcheggi chiusi.
- Wallet prepagati digitali (integrazione con PagoPA).
- App per utenti e ausiliari.
- Telemetria apparati in campo
- Front end per accreditamento utenti + Cruscotto per forza di polizia locale.
- Integrazione con SDI per fatturazione elettronica
- Tariffazione dinamica.

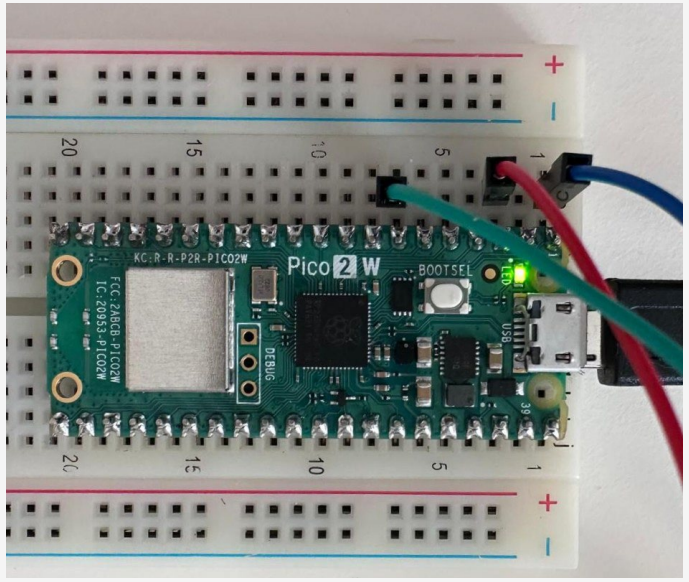
Schema sintetico del sistema di sosta



Rust su Microcontrollori 1/2

Rust ha pieno supporto su microcontrollori come:

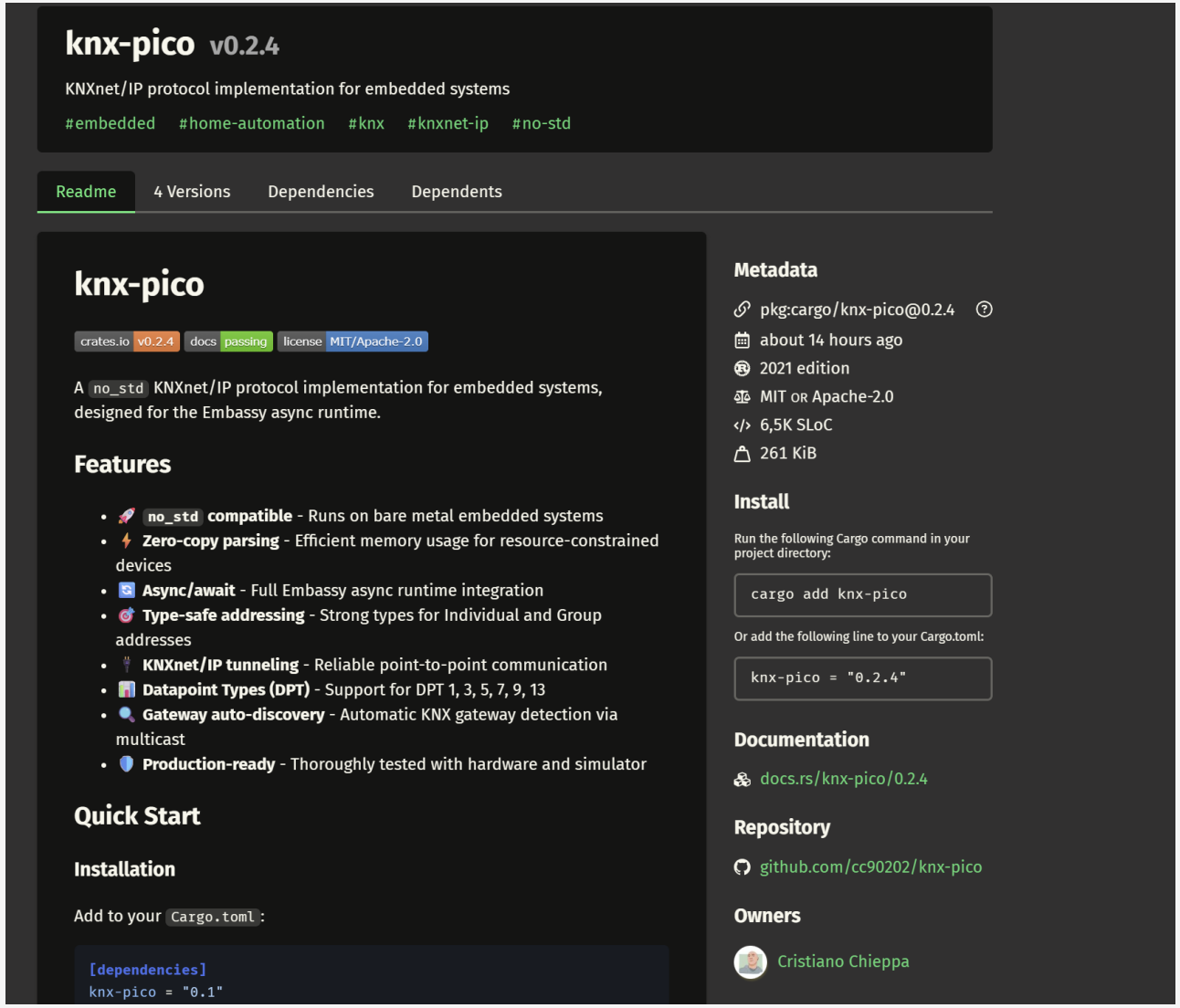
- ESP32
- Raspberry Pico/2
- ST (Stm32)
- Nordic
- NXP
- Microchip



https://github.com/rust-embedded/awesome-embedded-rust?tab=readme-ov-file

```rust
/// Entry point principale secondo Embassy
#[embassy_executor::main]
async fn main(spawner: Spawner) {
    let p = embassy_rp::init(Default::default());

    // Parte il logger su USB
    let driver = Driver::new(p.USB, UsbIrqs);
    spawner.must_spawn(logger_task(driver)); //<---- 1
    if let Some(panic_message) = panic_persist::get_panic_message_utf8() {
        log::error!("{panic_message}");
        loop {
            embassy_time::Timer::after_secs(5).await;
        }
    }

    // Firmware files for the CYW43xxx WiFi chip.
    let fw = include_bytes!("../cyw43-firmware/43439A0.bin");
    let clm = include_bytes!("../cyw43-firmware/43439A0_clm.bin");

    // To make flashing faster for development, you may want to flash the firmwares independently
    // at hardcoded addresses, instead of baking them into the program with `include_bytes!`:
    //     probe-rs download ../../cyw43-firmware/43439A0.bin --binary-format bin --chip RP235x --base-address 0x10100000
    //     probe-rs download ../../cyw43-firmware/43439A0_clm.bin --binary-format bin --chip RP235x --base-address 0x10140000
    //let fw = unsafe { core::slice::from_raw_parts(0x10100000 as *const u8, 230321) };
    //let clm = unsafe { core::slice::from_raw_parts(0x10140000 as *const u8, 4752) };

    let pwr = Output::new(p.PIN_23, Level::Low);
    let cs = Output::new(p.PIN_25, Level::High);
    let mut pio = Pio::new(p.PIO0, Irqs);
    let spi = PioSpi::new(
        &mut pio.common,
        pio.sm0,
        // SPI communication won't work if the speed is too high, so we use a divider larger than `DEFAULT_CLOCK_DIVIDER`.
        // See: https://github.com/embassy-rs/embassy/issues/3960.
        RM2_CLOCK_DIVIDER,
        pio.irq0,
        cs,
        p.PIN_24,
        p.PIN_29,
        p.DMA_CH0,
    );

    static STATE: StaticCell<cyw43::State> = StaticCell::new();
    let state = STATE.init(cyw43::State::new());
    let (net_device, mut control, runner) = cyw43::new(state, pwr, spi, fw).await;

    // parte il task di gestione del chip WiFi
    spawner.must_spawn(cyw43_task(runner)); //<---- 2
```

Rust su Microcontrollori 2/2



knx-pico v0.2.4

KNXnet/IP protocol implementation for embedded systems

#embedded #home-automation #knx #knxnet-ip #no-std

Readme | 4 Versions | Dependencies | Dependents

knx-pico

crates.io v0.2.4 | docs passing | license MIT/Apache-2.0

A `no_std` KNXnet/IP protocol implementation for embedded systems, designed for the Embassy async runtime.

Features

- 🚀 **`no_std` compatible** - Runs on bare metal embedded systems
- ⚡ **Zero-copy parsing** - Efficient memory usage for resource-constrained devices
- 🔄 **Async/await** - Full Embassy async runtime integration
- 🎯 **Type-safe addressing** - Strong types for Individual and Group addresses
- 🔌 **KNXnet/IP tunneling** - Reliable point-to-point communication
- 📊 **Datapoint Types (DPT)** - Support for DPT 1, 3, 5, 7, 9, 13
- 🔍 **Gateway auto-discovery** - Automatic KNX gateway detection via multicast
- 🛡️ **Production-ready** - Thoroughly tested with hardware and simulator

Quick Start

Installation

Add to your `Cargo.toml`:

```
[dependencies]
knx-pico = "0.1"
```

Metadata

pkg:cargo/knx-pico@0.2.4
about 14 hours ago
2021 edition
MIT OR Apache-2.0
6,5K SLoC
261 KiB

Install

Run the following Cargo command in your project directory:

```
cargo add knx-pico
```

Or add the following line to your Cargo.toml:

```
knx-pico = "0.2.4"
```

Documentation

docs.rs/knx-pico/0.2.4

Repository

github.com/cc90202/knx-pico

Owners

Cristiano Chieppa

```rust
253         // ============================================================================
254         // Test: READ Commands
255         // ============================================================================
256
257         pico_log!(info, "3. Testing READ commands...");
258
259         pico_log!(info, "Sending READ to 1/2/3...");
260         // Note: Read operations are fire-and-forget in this implementation
261         // The response would come as a GroupResponse event
262         // Using knx_read! macro for concise syntax
263         match knx_read!(client, 1/2/3).await {
264             Ok(_) => pico_log!(info, "✓ READ command sent (response would be received as event)"),
265             Err(_) => pico_log!(error, "✗ Failed to send READ command"),
266         }
267
268         Timer::after(Duration::from_secs(1)).await;
269
270         // ============================================================================
271         // Test: WRITE Commands
272         // ============================================================================
273
274         pico_log!(info, "4. Testing WRITE commands...");
275
276         pico_log!(info, "Sending WRITE: bool=true to 1/2/3");
277         // Using knx_write! macro for concise syntax
278         match knx_write!(client, 1/2/3, KnxValue::Bool(true)).await {
279             Ok(_) => {
280                 pico_log!(info, "✓ WRITE command sent successfully (fire-and-forget)");
281             }
282             Err(_) => {
283                 pico_log!(error, "✗ Failed to send WRITE command");
284             }
285         }
```

Piccoli suggerimenti per Rust

- Usare se possibile RwLock anziché mutex, parking-lot.
- Evitare il clone di dati grandi, provare Arc/Rc/Copy-on-Wri (CoW).
- Lock brevissimi, solo per il tempo necessario.
- Mai mischiare async e sync (l'executor si blocc
- Non fare in async operazioni molto lunghe (alt spawn_blocking).
- Usare type system guidance ove possibile.
- Codice ripetuto -> valutare macro se è chiaro quello che fa.

```
3 usages
struct Grounded;
3 usages
struct Launched;
8 usages
let slow_data : Arc<Mutex<SharedData>> = Arc::clone(&data);
let slow_handle : JoinHandle<()> = thread::spawn(move || {
    println!("[LENTO] Attendo di acquisire il lock...");

    // --- 🔒 ACQUISIZIONE DEL LOCK ---                                    arker::PhantomData} }
    let mut guard : MutexGuard<SharedData> = slow_data.lock().unwrap();
    println!("Lock acquisito. Eseguo operazione I/O lenta...");

    // Questa simula una richiesta di rete, accesso a file, o calcolo intensivo.
    thread::sleep(Duration::from_secs( secs: 3));

    guard.value += 1;
    // andrebbe scritto cosi: slow_data.lock().unwrap().value += 1;

    println!("Operazione lenta completata. Rilascio il lock.");
    // --- 🔓 RILASCIO DEL LOCK (implictamente quando 'guard' esce dallo scope) ---
});
        rocket.decelerate();
    }
```

Rust in produzione
Q&A

cristiano.chieppa@smartsecuritysw.it