tf2_rs: Bringing tf2 to Rust

Théo Engels[1], Antonio Paolillo[2] and Ken Hasselmann[1]

[1]Royal Military Academy, Avenue de la Renaissance 30, 1000 Brussels, Belgium
[2]Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium

Abstract

tf2 is a core Robot Operating System 2 (ROS 2) subsystem used to manage time-aware coordinate transforms throughout robotic software stacks. While interest in Rust for robotics has grown due to its memory-safety guarantees, performance, and data-race-free concurrency model, the ROS 2 Rust ecosystem still lacks several core components, including an ergonomic and high-performance `tf2` interface. This paper presents `tf2_rs`, a Rust crate providing bindings to ROS 2 `tf2` together with an idiomatic Rust API for transform buffering, time-aware lookup, and integration with core ROS 2 workflows. We evaluate `tf2_rs` against the C++ and Python implementations on lookup-latency benchmarks covering static and dynamic transform workloads. Results show that `tf2_rs` closely tracks C++ performance and substantially outperforms Python, indicating that the Rust-side abstraction and FFI layers introduce only limited overhead.

Keywords

ROS 2, tf2, Rust

1. Introduction

Robotic systems usually integrate data from multiple sensors, actuators, and estimates that are expressed in different coordinate frames and at different times. In ROS 2, this problem is addressed by `tf2` [1, 2], which maintains a transform tree and provides lookup and interpolation utilities. As such, `tf2` is a fundamental dependency for many ROS 2 stacks, including navigation, perception, and manipulation stacks.

In recent years, Rust for robotics software has seen growing interest due to its performance comparable to C or C++, its strong memory-safety guarantees, and compile-time enforcement of ownership and borrowing rules. These properties are attractive in robotic software, where low-level resource control, concurrency, and robustness must coexist with latency and throughput constraints. Existing efforts in Rust for ROS 2 and robotics include Zenoh [3], `rclrs` [4], and ROS-Z [5], a Zenoh-native ROS 2 Rust client library, alongside broader Rust-based robotics infrastructure such as Rerun [6] and Copper [7]. However, despite this momentum, the ROS 2 Rust ecosystem still lacks several core infrastructure components. One notable gap is an ergonomic and efficient `tf2` solution: without it, Rust-based ROS 2 nodes must either avoid transforms in their workflows, rely on ad hoc foreign-function interfaces (FFI), or reimplement critical functionality, increasing engineering overhead and slowing adoption.

This article presents `tf2_rs`[1], a Rust crate that provides ROS 2 `tf2` bindings to the core `tf2` C++ functionalities together with an idiomatic Rust API for transform buffering, time-aware lookup, and integration with standard ROS 2 `tf2` workflows. The current release focuses on the core transform path needed for practical use in Rust-based ROS 2 nodes, while broader feature coverage remains future work. We describe the crate's binding-oriented architecture, which is designed to preserve upstream `tf2` semantics while keeping foreign-function and unsafe boundaries narrow. To assess the cost of this approach, we report benchmark results comparing `tf2_rs` against the reference C++ and Python `tf2` implementations on static and dynamic transform lookup workloads.

8th International Workshop on Robotics Software Engineering (RoSE 2026), co-located with the IEEE International Conference on Robotics and Automation (ICRA 2026), June 1, 2026, Vienna, Austria
theo.engels@mil.be (T. Engels); antonio.paolillo@vub.be (A. Paolillo); ken.hasselmann@mil.be (K. Hasselmann)
0009-0009-5405-186X (T. Engels); 0000-0001-6608-6562 (A. Paolillo); 0000-0002-8196-9889 (K. Hasselmann)

[1]https://github.com/olingo99/tf2_rs

The contributions of this article are: (1) an open-source, MIT-licensed Rust `tf2` binding and API design intended for practical ROS 2 usage; (2) a preliminary benchmark of its performance and comparison against `tf2_cpp` and its Python bindings, `tf2_py`; and (3) an analysis of current limitations and future directions toward broader `tf2` feature coverage in Rust.

2. Background and Motivation

In ROS 2, a transform is not just a spatial relation between two frames, but a relation that can vary over time. For that reason, `tf2` [1, 2] does not store only the latest transform for each frame relation. Instead, it maintains a tree of frame relationships together with buffered timestamped transform samples, allowing a node to query the relation between frames either at the latest available time or at the timestamp associated with the data being processed. When sufficient temporal information is present, `tf2` can interpolate between buffered samples. This time-buffered design is essential in robotics because sensor messages, odometry, and state estimates are produced asynchronously and are often consumed later than when they were recorded.

In practice, each process maintains its own local transform buffer, typically populated from the `/tf` and `/tf_static` topics through a transform listener. A lookup therefore consists of resolving a path through the frame tree and, for each edge on that path, retrieving the transform value corresponding to the requested time. For static edges this is straightforward, whereas dynamic edges may require selecting buffered samples and interpolating between them. As a result, lookup cost depends not only on tree depth but also on whether the path traverses static or dynamic transforms and on how much history is buffered for each dynamic edge. Beyond direct lookup, `tf2` also provides helpers for applying transforms to stamped geometric data such as points and poses, which is why it commonly appears in latency-sensitive perception, localization, and control pipelines.

Providing this functionality in Rust is not simply a matter of exposing existing C++ symbols. The first challenge is error handling: `tf2` reports many lookup failures through exceptions, whereas idiomatic Rust requires explicit `Result`-based APIs and structured error types. The second challenge is concurrency: transform buffers are updated asynchronously while being queried from other parts of the system. In the C++ ROS 2 usage pattern, the user-facing transform buffer is commonly shared between the node and its `TransformListener` via `std::shared_ptr`, while transform insertion and lookup are internally synchronized through the same `frame_mutex_` protecting shared frame data [8]. A Rust interface must therefore account for the same shared-access and thread-safety concerns while exposing them through an idiomatic API. Finally, the design must balance safety, ergonomics, and performance: a thin FFI layer may preserve upstream behavior, but exposing it directly would yield an unusual API, while an overly abstract wrapper risks unnecessary copying or synchronization overhead.

The use of memory-safe and statically verified languages in robotics has precedent beyond Rust. Work on SPARK/Ada has demonstrated formal verification of robot navigation algorithms [9], and prior work has addressed security hardening of the ROS ecosystem [10]. In the Rust ecosystem specifically, prior efforts such as `rustros_tf` [11] attempted a pure-Rust `tf` reimplementation for ROS 1, and crates like `ros_pointcloud2` [12] demonstrate Rust FFI patterns for ROS 2 message types.

The reference `tf2` implementation in ROS 2 is provided in C++ [8], and official Python bindings preserve the same core concepts and workflows [13]. This makes a binding-oriented approach a natural starting point for Rust, rather than a full native reimplementation of the transform system. Our objective is to preserve upstream `tf2` semantics, reduce the risk of semantic drift, and evolve toward broader feature coverage while still presenting an idiomatic Rust interface.

3. tf2_rs Design and Implementation

The implementation of `tf2_rs` is guided by five requirements: (i) preserving the semantics of the upstream `tf2` implementation; (ii) supporting the core ROS 2 workflow of local buffer maintenance and time-aware lookup; (iii) exposing an idiomatic Rust API with explicit error handling; (iv) keeping unsafe

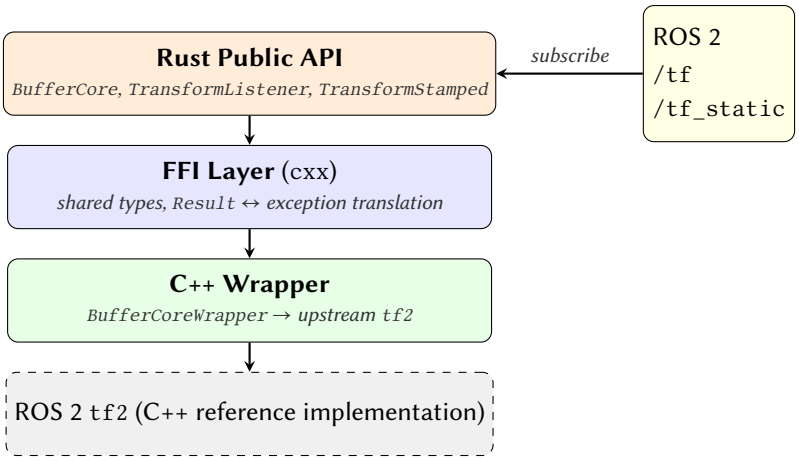

Figure 1: Layered architecture of `tf2_rs`. The Rust public API exposes idiomatic abstractions backed by a narrow `cxx`-based FFI layer and a thin C++ wrapper that delegates to the upstream ROS 2 `tf2` implementation. The `TransformListener` subscribes to standard ROS 2 transform topics.

and foreign-function boundaries narrow; and (v) maintaining performance for frequent transform lookup operations.

To satisfy these requirements, `tf2_rs` adopts a layered architecture composed of three components (Figure 1). The first is a public Rust API exposing the main application-facing abstractions. The second is a narrow FFI layer, implemented with the `cxx` [14] crate, that defines the shared types and function calls exchanged between Rust and C++. The third is a thin C++ wrapper around the reference `tf2` implementation, responsible for invoking the upstream library and adapting its exception-based interface to a form that can be handled idiomatically in Rust. The public API is centered on a small set of concepts: a `BufferCore` for transform storage and queries, a `TransformListener` for ROS 2 integration, a Rust-owned `TransformStamped` representation, and time and transformation traits that support message-oriented usage.

The central abstraction in the Rust layer is `BufferCore`. It wraps a shared pointer to a C++ `BufferCoreWrapper` and exposes the core `tf2` workflow through methods for transform insertion, availability checks, and lookup. This keeps the Rust interface close to the usual `tf2` programming model while expressing failures through explicit `Result`-based APIs rather than exceptions. The same principle applies to time and transform data. Time-aware lookup is represented explicitly in Rust and translated into an FFI-safe `Tf2Time` structure, while transforms are carried as Rust-owned `TransformStamped` values and converted explicitly across the language boundary. This avoids exposing raw C++ `tf2` objects directly in Rust while preserving standard `tf2` semantics, including the convention that a zero timestamp denotes the latest available transform.

The implementation follows the standard ROS 2 `tf2` usage described in Section 2. Each process maintains a local transform buffer. Transforms are inserted into that buffer as they are received, and queries are resolved against the local state rather than through the communication layer. In `tf2_rs`, this pattern is realized by the combination of `BufferCore` and `TransformListener`, which subscribes to `/tf` and `/tf_static` and forwards incoming transforms into the local buffer. The Rust side does not maintain a separate copy of that buffer; it holds shared ownership of the same underlying C++ `BufferCore` through a shared-pointer handle to a `BufferCoreWrapper`. By keeping the wrapper thin and relying on upstream `tf2` for the core transform logic, the crate stays close to established `tf2` behavior while localizing foreign-function and unsafe concerns to a small part of the codebase.

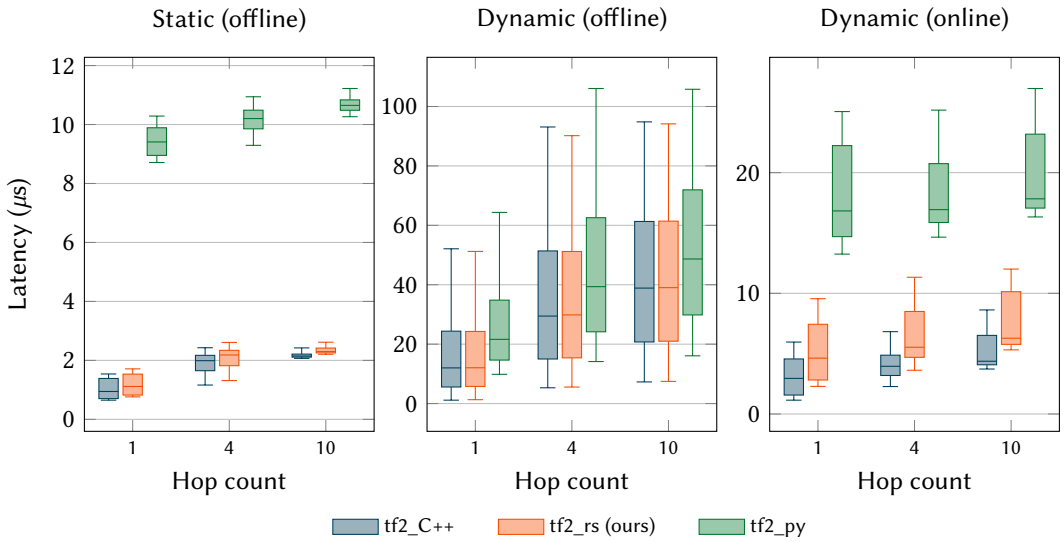

Figure 2: Lookup latency distributions for `tf2_cpp`, `tf2_rs`, and `tf2_py` across static offline, dynamic offline, and dynamic online workloads at hop counts 1, 4, and 10. Latency statistics are computed from successful lookups only. Boxes show the 25th, 50th, and 75th percentiles, and whiskers show the 5th and 95th percentiles.

4. Evaluation

We evaluate `tf2_rs` against the reference C++ and Python implementations using lookup-latency benchmarks[2] [15]. Our objective is to assess how closely the current implementation tracks the performance of upstream `tf2`. Three scenarios are considered:

- **Static offline**: only static transforms are present, and all transforms are loaded before the benchmark starts.
- **Dynamic offline**: both static and dynamic transforms are present, and all transforms are loaded before the benchmark starts.
- **Dynamic online**: both static and dynamic transforms are present, with dynamic transforms published continuously while lookups are performed at scheduled times during execution.

For each scenario, lookup latency is measured for source and target pairs separated by 1, 4, or 10 hops in the transform tree. Since all three implementations rely on the same C++ core for transform lookup and interpolation, the measured latency can be interpreted as the combination of a language-specific overhead and the underlying `tf2` lookup cost.

The offline benchmark uses six datasets (`h1`, `h4`, and `h10`, each in static and dynamic variants) with 500 frames per dataset. The online benchmark uses three dynamic datasets (`h1`, `h4`, `h10`) with 300 frames over a 60-second run. Queries are generated as random frame pairs at an exact hop count. All implementations use a buffer cache time of 60 seconds, and each timed section is preceded by 5000 warmup queries. Experiments were run on an Intel Core Ultra 7 155H with 30 GiB RAM, Ubuntu 24.04.4, Linux 6.17.0, and ROS 2 Jazzy. Rust was compiled with rustc 1.93.0 in release mode, C++ with g++ 13.3.0 using `-O3`, and Python was run with Python 3.12.3 using `PYTHONOPTIMIZE=1`.

In addition to latency, we performed a functional consistency check on the static transforms by comparing the transform outputs returned by `tf2_py` and `tf2_rs` against the C++ reference on the same queries. For these static cases, all queried lookups succeeded, and the measured output delta was zero in every case, indicating exact agreement with the reference `tf2_cpp` results on these workloads.

Across all three scenarios, `tf2_rs` remains close to `tf2_cpp` while substantially outperforming `tf2_py`. Measured as the ratio of median successful lookup latency relative to `tf2_cpp`, `tf2_rs`

[2]Benchmark and dataset generation code available at https://github.com/olingo99/tf2_benchmark.git and https://github.com/olingo99/tf2_tree_generator.git

Table 1
`tf2` feature parity snapshot

Feature	tf2 (C++)	tf2_py	tf2_rs
Core buffer API	✓	✓	✓
Listener + broadcasters	✓	✓	✓
Advanced lookup API (`target_time` != `source_time`)	✓	✓	✗
Typed geometry transforms and transform application	✓	✓	△
Async/wait API	✓	✓	✗
MessageFilter	✓	✗	✗

Legend: ✓implemented, △partial, ✗missing.

stays within 1.07–1.18× in the static offline scenario, is nearly identical in the dynamic offline scenario (1.00–1.01×), and remains within 1.40–1.57× in the dynamic online scenario. By contrast, `tf2_py` is about 5–10× slower in the static offline case, 1.25–1.80× slower in the dynamic offline case, and 4–6× slower in the dynamic online case. These results suggest that the additional Rust-side representation, conversion, and FFI layers introduce limited overhead relative to the reference C++ implementation, while remaining substantially faster than the Python interface. Additional offline throughput experiments showed a similar trend, with `tf2_rs` remaining close to `tf2_cpp` and outperforming `tf2_py`.

A plausible explanation for the larger gap in the dynamic online scenario is that the Rust and C++ implementations do not follow identical callback execution paths. In the C++ benchmark, `/tf` callbacks are processed directly by the same single-threaded executor as the benchmark loop, whereas in `tf2_rs` they pass through `rclrs` [4] worker-subscription machinery, which may introduce additional scheduling and thread-management overhead. In addition, incoming transforms in `tf2_rs` undergo Rust-to-FFI-to-C++ conversion before insertion into the underlying `tf2::BufferCore`, which may contribute further overhead during online updates.

The scenarios explain the main trends observed in the latency distributions. Static offline lookups exhibit the lowest latency because they involve only static transforms, which makes the fixed language-level overhead more apparent. Dynamic offline lookups are substantially slower because execution is dominated by transform chain traversal, per-edge timestamp lookup, and potential interpolation in the dynamic caches. In our benchmark, dynamic online latency is lower than dynamic offline latency because the history buffer is filled progressively during execution, so lookups traverse shorter dynamic-cache histories than in the fully preloaded offline case. The dynamic scenarios also show wider latency distributions because lookups may traverse different combinations of static and dynamic edges, some requiring more cache processing than others. Online results should be interpreted with care since they report successful lookups only and exclude queries that fail due to extrapolation.

Since all three implementations rely on the same `tf2` core, the comparison evaluates the overhead of the Rust-side representation, conversion, and FFI layers rather than whether Rust can outperform C++. The results indicate that this overhead remains small and preserves most of the reference implementation's performance.

5. Conclusion and Future Work

This paper presented `tf2_rs`, a Rust crate that brings ROS 2 `tf2` functionality to Rust. It uses a binding-oriented architecture that preserves upstream `tf2` semantics while exposing an idiomatic Rust interface. The crate addresses an important gap in the ROS 2 Rust ecosystem by providing the core capabilities required for transform buffering, time-aware lookup, and integration with standard ROS 2 `tf2` workflows.

The evaluation shows that `tf2_rs` achieves performance close to the reference C++ implementation and substantially outperforms the Python interface across all tested scenarios. This indicates that

the additional Rust-side representation, conversion, and FFI layers introduce only limited overhead, preserving most of the performance of upstream `tf2` while providing an idiomatic API for Rust users. Although the current release is still under active development, it already provides a practical foundation for transform-heavy ROS 2 applications and a basis for the continued growth of the ROS 2 Rust and Rust robotics ecosystems.

`tf2_rs` already supports the core transform workflow needed for practical ROS 2 use, but several features available in the C++ and Python `tf2` implementations are not yet implemented. Table 1 summarizes the current status of selected features. Currently, the crate focuses on buffered transform insertion, time-aware lookup, ROS 2 listener integration, and the core Rust abstractions required to use `tf2` from Rust nodes.

The most immediate priority is the expansion of feature coverage. In particular, important next steps include broader support for transform application helpers and integration with message filter workflows where callbacks are delayed until the required transforms are available. Beyond feature parity, future work also includes a thorough safety analysis and an expanded test suite with cross-checks against C++ behavior on multiple ROS 2 distributions.

Declaration on Generative AI

During the preparation of this work, the authors used GPT-5 and Claude Opus 4.7 in order to: grammar and spelling check, improve writing style, and formatting assistance. After using these tools, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] Open Robotics, tf2: ROS 2 Documentation, 2026. URL: https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Tf2.html.

[2] T. Foote, tf: The transform library, in: 2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA), 2013, pp. 1–6. doi:10.1109/TePRA.2013.6556373.

[3] Eclipse Zenoh Project, zenoh, 2026. URL: https://github.com/eclipse-zenoh/zenoh.

[4] ros2-rust contributors, ros2_rust, 2026. URL: https://github.com/ros2-rust/ros2_rust.

[5] ZettaScale Labs, ros-z, 2026. URL: https://github.com/ZettaScaleLabs/ros-z.

[6] Rerun Development Team, Rerun: A Visualization SDK for Multimodal Data, 2026. URL: https://www.rerun.io.

[7] G. Binet, Copper Project contributors, Copper-rs: A deterministic runtime and SDK for robotics, 2026. URL: https://github.com/copper-project/copper-rs.

[8] ROS 2, geometry2, 2026. URL: https://github.com/ros2/geometry2.

[9] P. Trojanek, K. Eder, Verification and Testing of Mobile Robot Navigation Algorithms: A Case Study in SPARK, in: IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2014, pp. 1489–1494. doi:10.1109/IROS.2014.6942753.

[10] R. White, G. Caiazza, H. Christensen, A. Cortesi, SROS1: Using and Developing Secure ROS1 Systems, in: Robot Operating System (ROS): The Complete Reference (Volume 3), Springer, 2019, pp. 373–405. doi:10.1007/978-3-319-91590-6_11.

[11] A. Chakravarty, rustros_tf: A port of ROS's TF library to Rust, 2022. URL: https://github.com/arjo129/rustros_tf.

[12] C. Sieh, ros_pointcloud2: A PointCloud2 message conversion library for ROS1 and ROS2, 2025. URL: https://github.com/stelzo/ros_pointcloud2.

[13] Open Robotics, Tf2_ros_py 0.30.0 documentation: tf2_ros_buffer, 2026. URL: https://docs.ros.org/en/jazzy/p/tf2_ros_py/tf2_ros.buffer.html.

[14] D. Tolnay, CXX: Safe Interop between Rust and C++, 2026. URL: https://cxx.rs/.

[15] V. Mayoral-Vilches, I. Lütkebohle, C. Bédard, R. Araújo, REP 2014 – Benchmarking Performance in ROS 2 (ROS.Org), 2014. URL: https://ros.org/reps/rep-2014.html.