haproxy-spoe-rs: A Rust SPOA Agent Library for HAProxy

2026-04-12

Building a HAProxy Stream Processing Offload Agent (SPOA) library in Rust — zero-dependency async design, mpsc write batching, 95.9% test coverage, and 2.8–4.9× higher throughput than the Go reference implementation.

---

haproxy-spoe-rs is a Rust library for writing HAProxy SPOE agents. This post covers what SPOE is, the design decisions behind the library, and how its throughput compares to the Go reference implementation.

Why this library?

This library is not a replacement for the Go implementation — it is a **Rust-native alternative** for teams that are already using Rust and want to keep their SPOA agent in the same language and toolchain.

Both implementations are solid choices. The Go version is mature, well-tested, and integrates naturally into Go-based infrastructures. This library exists for the Rust ecosystem:

- **Type safety end-to-end.** The SPOE wire types (`TypedData`, `Scope`, `Action`) are Rust enums — the compiler rejects mismatched types at build time, not at runtime.
- **No separate process to operate.** Embed the agent directly in an existing Rust binary via `Agent::new(handler).serve(listener)`, or run it standalone.
- **Minimal dependencies.** Three crates: tokio, mimalloc, log — no codec library, no serialization framework.
- **Performance.** The mpsc write-batching architecture shows 2.8–4.9× higher throughput than the Go reference under pipelining load (see Throughput below).

What is SPOE?

HAProxy's **Stream Processing Offload Engine** (SPOE) lets HAProxy offload work to an external process — a **SPOA (Stream Processing Offload Agent)** — over a persistent TCP connection. For every HTTP request (or TCP session), HAProxy sends a `NOTIFY` frame with key-value arguments to the agent and waits for an `ACK` containing variables to set. Those variables are then available in HAProxy ACLs and rules for the rest of the request lifecycle.

A typical use case is **IP reputation**: HAProxy forwards `src` (client IP) to the agent on every frontend request; the agent looks it up in a database and returns a score; HAProxy blocks or rate-limits based on that score — all without a Lua script or module recompile.

A second common use case is **GCRA rate limiting**: the agent maintains per-client token buckets in shared memory and returns a `blocked` boolean; HAProxy denies requests that exceed the configured rate. The haproxy-spoe-rs: Deployment post shows a complete GCRA implementation as a worked example.

The protocol is fully documented in `doc/SPOE.txt` in the HAProxy source tree.

{{ bounded_image(src="/img/haproxy-spoe-request-flow.drawio.png", alt="HAProxy SPOE request flow: Client ⟶ HAProxy (stream suspended) ⟶ SPOA Rust Agent ⟶ HAProxy (stream resumed) ⟶ Backend", max_width=800) }}

The diagram shows the six steps of a single request: the client sends a request (①), HAProxy suspends the stream and forwards a NOTIFY to the agent (②), the agent runs the handler and returns an ACK with variables (③), HAProxy resumes the stream and applies ACL rules (④), forwards to the backend (⑤), and returns the response (⑥). The annotated bands highlight where stream suspension, handler execution, write batching, and stream resumption occur.

HAProxy version compatibility. The library implements SPOP 2.0 and has been tested with HAProxy 2.9 and 3.x. The `async` capability advertised in older SPOE configs is ignored since HAProxy 3.x — only `option pipelining` is relevant.

HAProxy configuration

A minimal SPOE setup looks like this:

```
# spoe filter in frontend
frontend http-in
    bind *:80
    filter spoe engine ip-reputation config /etc/haproxy/ip-
reputation.conf
    http-request deny if { var(sess.ip.ip_score) -m int ge 80 }

# spoe config file
[ip-reputation]
spoe-agent ip-reputation
    messages         check-client-ip
    option           var-prefix ip
    timeout hello    100ms
    timeout idle     30s
    timeout processing 15ms
    use-backend      spoe-backend

spoe-message check-client-ip
    args ip=src
    event on-frontend-http-request

# backend pointing to the agent
backend spoe-backend
```

```
    mode tcp
    server spoa 127.0.0.1:9000
```

spoe-group: multiple messages per NOTIFY

A `spoe-group` bundles several `spoe-message` entries and sends them all in **one NOTIFY frame** instead of one frame per message. Groups are not bound to a fixed event — they are triggered explicitly via TCP or HTTP rules, giving fine-grained control over when the SPOA is consulted (e.g. only for specific paths or content types).

From the agent's perspective nothing changes: the NOTIFY payload is already a `Vec<Message>`, so a group simply means more entries in that list. The handler calls `get_message()` for each name it cares about:

```
Agent::new(|req| {
    if let Some(msg) = req.get_message("check-client-ip") { /* … */ }
    if let Some(msg) = req.get_message("check-user-agent") { /* … */ }
    // both arrive in the same req — one NOTIFY, two messages
})
```

HAProxy configuration using a group:

```
[my-engine]
spoe-agent my-agent
    groups   check-all          # reference the group, not individual
messages
    option   var-prefix app
    timeout processing 15ms
    use-backend spoe-backend

spoe-group check-all
    messages check-client-ip check-user-agent

spoe-message check-client-ip
    args ip=src
    # no event — triggered by the group rule below

spoe-message check-user-agent
    args ua=req.hdr(user-agent)
```

```
# in the frontend — trigger the group only where needed
http-request send-spoe-group my-engine check-all
http-request deny if { var(txn.app.ip_score) -m int ge 80 }
```

This is more efficient than two separate `event`-based messages when both checks are only needed for a subset of requests, because HAProxy sends a single NOTIFY carrying all arguments at once instead of two round-trips to the agent.

Processing model: serializing by design

SPOE is fundamentally **serializing per stream**: when HAProxy encounters a `filter spoe` on a request, it suspends that stream — the HTTP request is held — sends a `NOTIFY` frame to the agent, and waits for the `ACK` before resuming. This is not a fire-and-forget side channel; it is a blocking call on the critical path of every request.

```
HAProxy stream          SPOA
     |                    |
     |---- NOTIFY ------->|
     |     (suspended)    |  handler runs
     |<--- ACK -----------|
     |     (resumed)      |
```

Two consequences fall out of this immediately:

1. **The handler must be fast.** `timeout processing` in the SPOE config is a hard deadline (15 ms in the example above). If the agent does not respond in time, HAProxy either aborts the request or lets it through, depending on `on-error` settings. This means handlers should be CPU-bound lookups against in-memory data, not synchronous database calls. For slow I/O, pre-populate a cache and consult it from the handler.
2. **Pipelining removes the per-stream bottleneck.** Without it, a single SPOE TCP connection can only carry one NOTIFY/ACK exchange at a time. With `option pipelining`, HAProxy sends NOTIFYs from many streams on the same connection without waiting for each ACK — the agent must correlate responses via `stream_id` / `frame_id`. The agent advertises `capabilities="pipelining"` in its AGENT-HELLO to signal this support. (The `async` capability that existed in older SPOE versions is deprecated and ignored by HAProxy since at least v3.x — only `pipelining` is relevant today.)

This is also why the write-batching in the architecture section matters: under pipelining, many NOTIFYs arrive in a burst and the agent must return ACKs quickly. The mpsc channel + `try_recv` loop ensures ACKs are batched into one syscall per burst rather than one syscall per frame.

Quick Start

Add the library to `Cargo.toml`:

```toml
[dependencies]
haproxy-spoe = "1.0"
tokio = { version = "1", features = ["macros", "net"] }
```

The public API is a single closure passed to `Agent::new`:

```rust
use haproxy_spoe::{Agent, Scope, TypedData};
use tokio::net::TcpListener;

#[tokio::main]
async fn main() {
```

```
    let listener = TcpListener::bind("0.0.0.0:9000").await.unwrap();

    Agent::new(|req| {
        let Some(msg) = req.get_message("check-client-ip") else
{ return };
        let Some(TypedData::IPv4(ip)) = msg.get("ip") else { return };

        let score = if ip.octets()[0] == 10 { 100i32 } else { 0 };
        req.set_var(Scope::Session, "ip_score", TypedData::Int32(score));
    })
    .serve(listener)
    .await
    .unwrap();
}
```

The handler is a plain `Fn(&mut Request)` — synchronous, no async, no trait object boilerplate. `req.set_var` / `req.unset_var` build the ACK actions; the library handles framing, pipelining, healthchecks, and graceful disconnect.

For graceful shutdown (SIGTERM + Ctrl-C), `tokio::select!` on signal futures works naturally because `Agent::serve` is just an async function:

```
tokio::select! {
    res = agent.serve(listener) => res.expect("agent error"),
    _ = tokio::signal::ctrl_c() => {}
    _ = async {
        tokio::signal::unix::signal(SignalKind::terminate())
            .expect("signal handler")
            .recv().await
    } => {}
}
```

Real-world handler pattern

The quick-start handler is intentionally minimal. In production the handler typically consults a shared in-memory data structure — an `Arc<RwLock<…>>` populated by a background refresh task:

```
use std::collections::HashSet;
use std::net::IpAddr;
use std::sync::{Arc, RwLock};

let blocklist: Arc<RwLock<HashSet<IpAddr>>> =
Arc::new(RwLock::new(HashSet::new()));

// Background task: refresh blocklist from Redis / DB every 60 s.
let bl = Arc::clone(&blocklist);
tokio::spawn(async move {
```

```
    loop {
        let fresh = fetch_blocklist().await;
        *bl.write().unwrap() = fresh;
        tokio::time::sleep(std::time::Duration::from_secs(60)).await;
    }
});

// Handler: read-lock is cheap and contention-free under pipelining load.
let bl = Arc::clone(&blocklist);
Agent::new(move |req| {
    let Some(msg) = req.get_message("check-client-ip") else { return };
    let Some(TypedData::IPv4(ip)) = msg.get("ip") else { return };
    let blocked = blocklist.read().unwrap().contains(&IpAddr::V4(*ip));
    req.set_var(Scope::Session, "blocked", TypedData::Boolean(blocked));
})
.serve(listener)
.await?;
```

The `RwLock` read-lock is uncontended during normal operation because the background writer only holds it briefly during the swap. The handler never does I/O — it only reads from memory, which keeps it well inside the `timeout processing` budget.

Handler panics. If the handler panics, the tokio task for that NOTIFY is aborted and the ACK is never sent — HAProxy will hit `timeout processing` for that stream. The connection stays alive for all other streams. Panics are logged as worker errors via `log::warn!`. Prefer `unwrap_or` / `?` patterns over `unwrap()` in handlers.

Environment variables

The `ip_reputation` example is configured entirely through environment variables:

Variable	Default	Description
`SPOE_ADDR`	`0.0.0.0:9000`	TCP address the agent listens on
`TOKIO_WORKER_THREADS`	number of CPU cores	Tokio async worker threads; read automatically by the runtime — no code change needed
`RUST_LOG`	*(logging off)*	Log filter passed to `env_logger`, e.g. `warn` or `haproxy_spoe=debug`

The agent prints its effective configuration at startup:

```
spoe agent listening on 0.0.0.0:9000
  TOKIO_WORKER_THREADS : 8
  RUST_LOG             : warn
```

What lives on the HAProxy side instead. Most operational tuning is done in the SPOE config, not in the agent:

HAProxy directive	Effect
`timeout processing`	Hard deadline per NOTIFY/ACK round-trip
`timeout idle`	Idle connection keepalive
`maxconn` on `server` entry	Max simultaneous connections to the agent
`maxconnrate`	Max new connections per second
Multiple `server` entries	Horizontal scaling — run N agent instances

This clean separation means you can tune throughput and reliability entirely from HAProxy config without touching or redeploying the agent.

Dependencies

The library has exactly **three runtime dependencies**:

Crate	Role
`tokio` (rt-multi-thread, net, io-util, sync, macros, signal, time)	Async I/O and task runtime
`mimalloc` (default-features = false)	Global allocator for better multi-threaded allocation
`log = "0.4"`	Logging facade — zero overhead when no subscriber is registered

No serialization framework, no codec crate, no trait-heavy abstractions — the SPOE wire format is simple enough to implement directly.

Architecture

Concurrency model

```
TcpListener
    │
    ├─ tokio::spawn  (one task per connection)
    │       │
    │       ├─ reader loop  ──NOTIFY──►  JoinSet::spawn  (one task per
frame)
    │       │                                   │
    │       │                              handler(&mut req)
    │       │                              tx.send(ack_bytes)   ◄── non-
blocking
    │       │
    │       └─ writer task  (BufWriter, drains channel, one flush per
batch)
    │               ▲
    │               └─ mpsc::unbounded_channel
```

The write path is the key design decision. An early version used `Arc<Mutex<OwnedWriteHalf>>` — every handler task locked the mutex, wrote its ACK, and unlocked. Under pipelining (HAProxy sends many NOTIFYs without waiting for ACKs), this caused **per-ACK lock contention and one `write` syscall per frame**.

The replacement is an `mpsc::unbounded_channel<Vec<u8>>` plus a dedicated writer task:

```
let (tx, mut rx) = mpsc::unbounded_channel::<Vec<u8>>();
let writer_task = tokio::spawn(async move {
    let mut writer = BufWriter::new(write_half);
    while let Some(bytes) = rx.recv().await {
        writer.write_all(&bytes).await?;
        // Drain everything queued before flushing — batch multiple ACKs
per syscall.
        loop {
            match rx.try_recv() {
                Ok(more) => writer.write_all(&more).await?,
                Err(_) => break,
            }
        }
        writer.flush().await?;
    }
});
```

Handler tasks do `tx.send(ack)` — a non-blocking channel push, no await, no mutex. The writer task wakes once, drains with `try_recv` to collect all ACKs that arrived while it was writing, and issues a single `flush`. Multiple ACKs ship in one syscall.

InFrame enum

Parsed inbound frames are typed enum variants — each carries exactly its own fields:

```
pub enum InFrame {
    HaproxyHello {
        stream_id: u64, frame_id: u64,
        engine_id: String,
        /// Comma-separated SPOP version list, e.g. `"2.0"`.
        supported_versions: String,
        max_frame_size: u32, healthcheck: bool,
    },
    HaproxyDisconnect {
        stream_id: u64, frame_id: u64,
        /// SPOE status code (0 = normal, 1 = I/O error, 2 = timeout, …).
        status_code: u32,
        message: String,
    },
    Notify { stream_id: u64, frame_id: u64, messages: Vec<Message> },
}
```

The worker uses an exhaustive `match` — the compiler enforces that all variants are handled, and there is no catch-all arm needed.

begin_frame / finish_frame (zero extra allocation)

Encoding a response frame allocates **one** `Vec<u8>`:

```
// Reserve 4 zero bytes as length placeholder.
fn begin_frame(frame_type: u8, flags: u32, stream_id: u64, frame_id: u64)
-> Vec<u8> { … }

// Patch the placeholder in-place once the full length is known.
fn finish_frame(mut out: Vec<u8>) -> Vec<u8> {
    let len = (out.len() - 4) as u32;
    out[0..4].copy_from_slice(&len.to_be_bytes());
    out
}
```

The previous design computed the header separately and then copied payload into a second `Vec`. `finish_frame` patches the 4-byte length field in-place — one allocation, no copy.

Protocol robustness

A protocol implementation that only handles the happy path is incomplete. Four correctness points are worth calling out:

Version validation. The SPOE spec (§3.2.4) requires the agent to reject HAPROXY-HELLO if the `supported-versions` field is missing or contains no 2.x token. The library sends an AGENT-DISCONNECT with status 5 ("version value not found") or 8 ("unsupported version") and returns `Error::Protocol` — the connection never enters the ready state.

```
// worker.rs — rejects "1.0", accepts "2.0" and "2.0,1.0"
fn spop_version_supported(supported: &str) -> bool {
    !supported.is_empty()
        && supported.split(',').any(|v| v.trim().starts_with("2."))
}
```

HAPROXY-DISCONNECT payload. HAProxy can close a connection with a non-zero status code and human-readable message (`status-code` + `message` KV-LIST). These are now parsed into `InFrame::HaproxyDisconnect` and logged at `warn` level when non-zero, so connection problems are visible without needing a packet capture.

Unknown frame types → skip, not error. A future HAProxy version could introduce new frame types. Instead of disconnecting on an unrecognised type byte, the agent reads the frame body (the 4-byte length field is always present) and discards it, then continues. This forward compatibility costs nothing.

`max_frame_size` enforcement. The `max-frame-size` negotiated during HELLO is now enforced on every inbound frame before any buffer allocation. A peer claiming a 4 GB frame would previously cause an OOM; now it returns `Error::Protocol` immediately.

Logging

The library uses the `log` facade. Two events are emitted at `warn` level:

- Worker errors (protocol violation, I/O error on a connection) — in `Agent::serve`
- HAProxy sending DISCONNECT with a non-zero status code

When no subscriber is registered the macros compile to nothing — zero runtime overhead. To enable logging in your application add a subscriber of your choice:

```
// Cargo.toml: env_logger = "0.11"
env_logger::init(); // RUST_LOG=warn ./your-agent
```

or with `tracing`:

```
// Cargo.toml: tracing-subscriber = { version = "0.3", features = ["env-filter"] }
tracing_subscriber::fmt::init();
```

SPOE wire format essentials

The protocol is compact and straightforward:

Field	Size	Notes
Frame length	4 bytes BE	counts everything after itself
Frame type	1 byte	0x01 HAPROXY-HELLO, 0x03 NOTIFY, 0x65 AGENT-HELLO, 0x67 ACK …
Flags	4 bytes BE	bit 0 = FIN
Stream ID	varint	
Frame ID	varint	
Payload	variable	type-specific KV or message list

Varint uses the SPOE Peers encoding: values < 240 fit in 1 byte; larger values use continuation bytes with the formula `(n-128) >> 7` (not the more common `n >> 7`).

TypedData encodes the type in the lower nibble and flags in the upper nibble of the first byte. Integers are encoded as varints, not fixed-width. `bool true` = `0x11`, `bool false` = `0x01`.

Module layout

After splitting the three largest files (each was 250–464 lines) using the Rust 2018 module system (`src/foo.rs` + `src/foo/submod.rs`, no `mod.rs` needed):

```
src/
  lib.rs               # global_allocator + public re-exports
  error.rs             # Error enum
  varint.rs            # SPOE Peers varint encode/decode
  typeddata.rs         # TypedData enum + encode(); declares mod decode
  typeddata/
    decode.rs          # impl TypedData { pub fn decode() } + all decode
```

```
tests
  action.rs            # Action enum (SetVar/UnsetVar), Scope enum
  message.rs           # Message struct
  frame.rs             # FrameType, InFrame enum, encode_agent_*;
declares mod parse
  frame/
    parse.rs           # parse() + parse helpers + parse tests
  request.rs           # Request with set_var/unset_var helpers
  worker.rs            # async TCP handler; declares mod util, writer,
tests
  worker/
    writer.rs          # spawn_writer(): BufWriter batching task
    util.rs            # spop_version_supported, broken_pipe, drain,
                       #   is_connection_close + unit tests
    tests.rs           # 12 tokio TCP integration tests + wire helpers
  agent.rs             # Agent::new(handler).serve(TcpListener)
examples/
  ip_reputation.rs     # complete working example (SIGTERM + Ctrl-C)
  bench.rs             # throughput benchmark (fake HAProxy, pipelined
NOTIFYs)
tests/
  spoe_agent.vtc       # VTest end-to-end test (real HAProxy + Rust
agent)
```

`impl TypedData` blocks may live in any module within the same crate, so `decode.rs` adds the `decode()` method from the submodule without any re-export gymnastics. The worker submodules use Rust 2018 path conventions (`src/worker/writer.rs` etc.) — no `mod.rs` needed, and `pub(super)` visibility keeps each helper scoped to the worker module.

Testing and coverage

65 tests in total, exercising the library at multiple levels:

Test location	Count	What it tests
`typeddata::decode::tests`	9	roundtrip encode/decode for all types + error paths
`frame::parse::tests`	16	parse happy paths + malformed input + disconnect payload + supported-versions
`frame::tests`	4	encode correctness
`worker::tests` (in `worker/tests.rs`)	12	full TCP integration: hello/notify/disconnect, healthcheck, protocol errors, version rejection, unknown frame skip, oversized frame, EOF mid-frame, agent-sent frame type skip, non-zero DISCONNECT status
`worker::util::tests`	2	`is_connection_close` logic
`agent::tests`	2	`is_transient` logic
other inline `mod tests`	~20	varint, action, message, request, error

The integration tests in `worker/tests.rs` spin up a real `TcpListener` and exchange raw SPOE frames over TCP — no mocks, no stubs.

Combined line coverage via `cargo llvm-cov` + VTest (real HAProxy against the `ip_reputation` example):

Module	Lines	Coverage
action.rs	63	100%
error.rs	50	100%
frame.rs + frame/parse.rs	309	97.7%
message.rs	31	100%
request.rs	69	100%
typeddata.rs + typeddata/decode.rs	138	100%
varint.rs	78	99.1%
worker.rs + worker_tests.rs	304	93.1%
agent.rs	43	79.1%
Total	**1140**	**95.9%**

The `agent.rs` gap is the accept-error branch (lines 36–41) — exercising it requires a real failing listener and is not worth the complexity.

VTest end-to-end test

`tests/spoe_agent.vtc` runs a real HAProxy instance against the compiled `ip_reputation` binary. A few non-obvious constraints when writing VTest scripts for SPOE:

- **Process names must start with `p`**: `process foo` must be `process p_foo`.
- **VTest does not propagate environment variables** to child processes — set them inside the command string.
- **`option pipelining`** must be present in the SPOE agent config for HTTP/1.1 keepalive reuse.

- **`process -stop` uses SIGKILL** (`kill(-pid, 9)`) — `atexit` never runs, the profraw file stays at 0 bytes. Manage the agent via `shell` blocks and `kill -TERM` instead.

Throughput: Rust vs Go

How `bench.rs` works

`examples/bench.rs` is **self-contained** — it starts the `ip_reputation` agent in-process on a random port, then immediately acts as a fake HAProxy client on the same machine:

```
let listener = TcpListener::bind("127.0.0.1:0").await.unwrap();
let addr = listener.local_addr().unwrap();
tokio::spawn(async move { agent.serve(listener).await.unwrap(); });
```

After the HAPROXY-HELLO / AGENT-HELLO handshake, each connection does exactly what a pipelining HAProxy does: concatenate all NOTIFY frames into **one buffer** and send them in a single `write_all`, then read ACKs one by one until all are received. The clock runs from the first send to the last ACK.

```
// Pipeline: send all NOTIFYs in one write, then drain ACKs
let mut all_notifies = Vec::with_capacity(frames as usize * 32);
for i in 1..=frames {
    all_notifies.extend_from_slice(&wire_frame(0x03, 1, i, i,
notify_payload));
}
stream.write_all(&all_notifies).await.unwrap();

for _ in 0..frames {
    stream.read_exact(&mut ack_buf).await.unwrap();
    // …read body…
}
```

Run it with:

```
cargo run --release --example bench -- [FRAMES] [CONNECTIONS]
# defaults: 100_000 frames, 1 connection
```

Example output:

```
Benchmarking 200000 NOTIFY frames over 1 connection(s) (200000 each)…
Done in 0.291s → 687241 frames/s  (1.46 µs/frame)
```

Results

Config	Rust	Go	Speedup
1 connection, 200k frames	~687k fps	~244k fps	**~2.8×**
4 connections, 1M frames	~1.74M fps	~354k fps	**~4.9×**

Why Rust wins — and why the gap grows with connections

The Go bottleneck is the write path: each handler goroutine calls `conn.Write()` directly, so every ACK is a separate syscall. Under pipelining, with many NOTIFYs queued at once, this means N goroutines each waking, locking the connection, and issuing one write — no batching.

The Rust writer task drains the mpsc channel with `try_recv` and issues a single `flush()` per burst. Under heavy pipelining dozens of ACKs ship in one syscall. With more connections the batching benefit compounds: the tokio scheduler keeps all threads busy and the combined throughput grows super-linearly, while the Go agent's per-connection write contention stays constant.

The Go benchmark lives in `cmd/bench/main.go` of haproxy-spoe-go and uses the same payload (IPv4 `ip` argument), the same pipelining strategy, and the same measurement window.

Comparison with `spop`

There is one other Rust SPOE crate: `spop` (GitHub: `nbari/spop`, v0.11.0, March 2026).

	`spop`	haproxy-spoe-rs
Level	Protocol codec (`SpopCodec`, `SpopFrame` trait)	High-level agent framework
API	Frames manually handled	`Agent::new(handler).serve(listener)`
Extra deps	tokio-util	none beyond tokio

`spop` is a frame-level codec — connection state (hello handshake, pipelining, ACK routing) must be managed by the caller. `haproxy-spoe-rs` abstracts all of that away so you write only the handler logic.

Source

github.com/git001/haproxy-spoe-rs