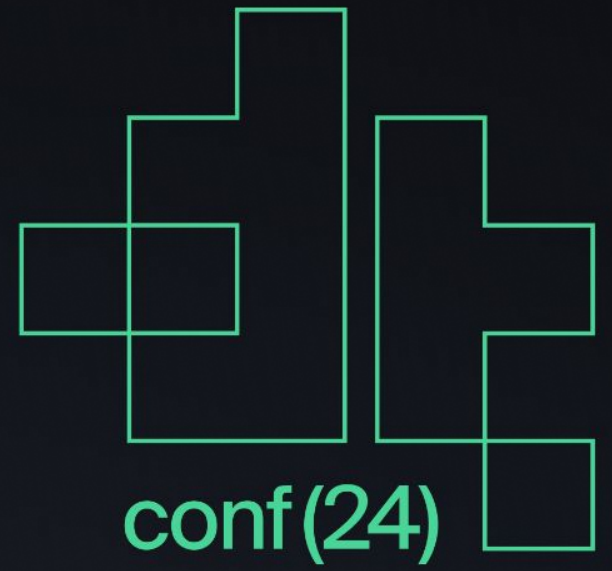
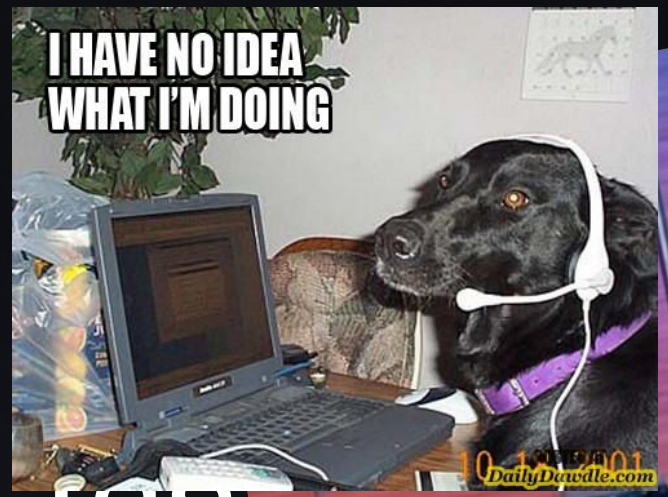
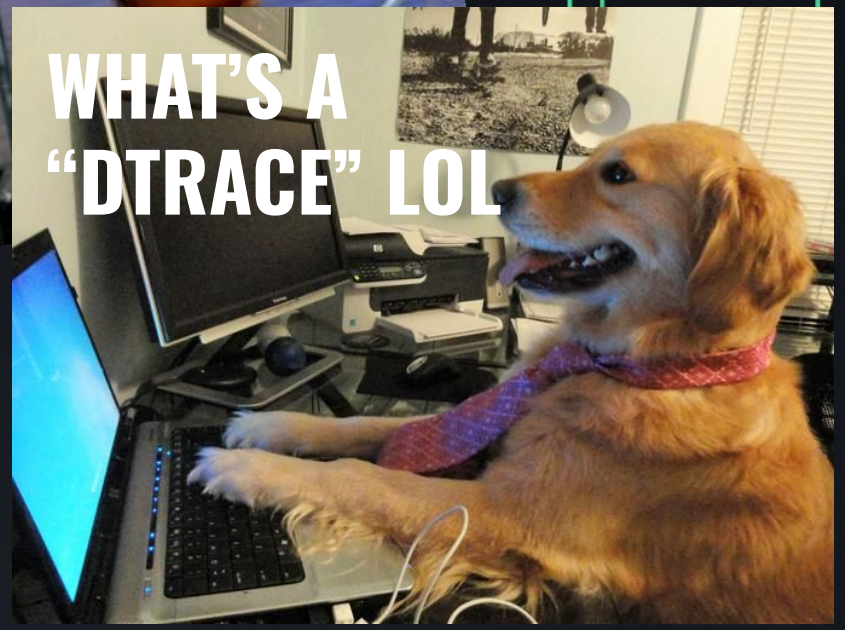
DTracing: Two great tastes…better together?



Eliza Weisman





tas
toge

Eliza Weisman

WHY I MADE ANOTHER LOGGING LIBRARY

A little backstory

- In 2019, I worked at Buoyant, where we were writing a high-performance application-layer proxy in Rust
- We were very **early adopters of the Rust async ecosystem**
- The debugging experience using `log` was **really bad**…

Logs in a **synchronous** system make sense…

```
DEBUG server: accepted connection from 106.42.126.8:56975
DEBUG server::http: received request
 WARN server::http: invalid request headers
TRACE server: closing connection
```

…but in an **asynchronous system**, they don't

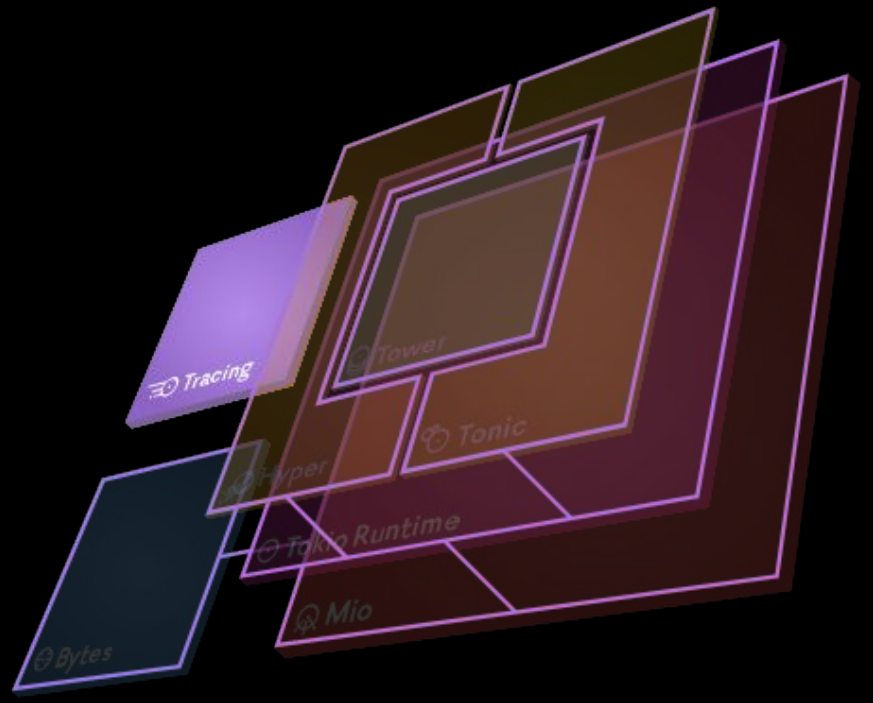
```
DEBUG server: accepted connection from 106.42.126.8:56975
TRACE server: closing connec
DEBUG server::http: receive
DEBUG server: accepted co                                  49123
DEBUG server::http: receiv
DEBUG server: accepted conne                            .37.105:51342
 WARN server::http: invalid request headers
TRACE server: closing connection
```

Solution: add **context**

```
DEBUG server{client.addr=106.42.126.8:56975}: accepted connection
TRACE server{client.addr=82.5.70.2:53121}: closing connection
DEBUG server{client.addr=89.56.1.12:55601}:request{path="/posts/tracing"}: received
request
DEBUG server{client.addr=111.103.8.9:49123}: accepted connection
DEBUG server{client.addr=106.42.126.8:56975}:request{path="/"}: received request
DEBUG server{client.addr=113.12.37.105:51342}: accepted connection
 WARN server{client.addr=106.42.126.8:56975}:request{path="/"}: invalid request headers
TRACE server{client.addr=106.42.126.8:56975}:request{path="/"}: closing connection
```

Tracing

Structured, application-level diagnostics



- Part of the **Tokio project**
- Data model **inspired by distributed tracing**
- **First-class contexts** (spans)
- A **facade** for modular **subscribers**

Core concepts: **events**

```
tracing::event!(level: Level::INFO, “hello world”);

let question = “life, the universe, and everything”;
tracing::debug!(question, answer = 42);
```

Core concepts: **spans**

```
{
    let _span = tracing::info_span!(“my_span”).entered();

    tracing::info!(inside_span = true);
}

tracing::info!(inside_span = false);
```

Core concepts: **spans**

```
#[tracing::instrument]
async fn connect(addr: SocketAddr) -> Result<TcpStream> {
    tracing::debug!(“connecting...”);
    // ...
}
```

So why are we talking about this at **dtrace.conf**?

tracing

Unfollow

Application-level tracing for Rust.

#async #logging #metrics #tracing

Readme | 43 Versions | Dependencies | Dependents | Settings

Displaying **1-10** of **8984** reverse dependencies of tracing

tokio DEPENDS ON ^0.1.29 — 250,038,346

An event-driven, non-blocking I/O platform for writing asynchronous I/O backed applications.

hyper DEPENDS ON ^0.1 — 226,683,754

A protective and efficient HTTP library for all.

tokio-util DEPENDS ON ^0.1.29 — 217,146,240

Additional utilities for working with Tokio.

h2 DEPENDS ON ^0.1.35 — 204,333,217

An HTTP/2 client and server

async-trait DEPENDS ON ^0.1.40 — 180,550,110

Type erasure for async trait methods

tracing-attributes DEPENDS ON ^0.1.35 — 176,125,310

Procedural macro attributes for automatically instrumenting functions.

Two great tastes…**better together?**

Imagine **one Rust macro** that can:

- Generate a **USDT probe**
- Emit a **text log**
- Located within a **distributed trace**

…all configured at runtime!

tracing and DTrace have some **compatible values**…

1. Minimize **disabled** probe effects
2. Don’t lie
3. Minimize **enabled** probe effects

Required Methods

fn enabled(&self, metadata: &Metadata<'_>) -> bool Source

Returns true if a span or event with the specified metadata would be recorded.

By default, it is assumed that this filter needs only be evaluated once for each callsite, so it is called by `register_callsite` when each callsite is registered. The result is used to determine if the subscriber is always interested or never interested in that callsite. This is intended primarily as an optimization, so that expensive filters (such as those involving string search, et cetera) need not be re-evaluated.

However, if the subscriber's interest in a particular span or event may change, or depends on contexts only determined dynamically at runtime, then the `register_callsite` method should be overridden to return `Interest::sometimes`. In that case, this function will be called every time that span or event occurs.

fn new_span(&self, span: &Attributes<'_>) -> Id Source

fn record(&self, span: &Id, values: &Record<'_>) Source

fn record_follows_from(&self, span: &Id, follows: &Id) Source

fn event(&self, event: &Event<'_>) Source

fn enter(&self, span: &Id) Source

fn exit(&self, span: &Id) Source

```rust
use ::tracing::__macro_support::Callsite as _;
static __CALLSITE: ::tracing::callsite::DefaultCallsite = {
    static META: ::tracing::Metadata<'static> = {
        ::tracing_core::metadata::Metadata::new(
            "event src/main.rs:2",
            "tracing_demo",
            ::tracing::Level::INFO,
            ::tracing_core::__macro_support::Option::Some("src/main.rs"),
            ::tracing_core::__macro_support::Option::Some(2u32),
            ::tracing_core::__macro_support::Option::Some("tracing_demo"),
            ::tracing_core::field::FieldSet::new(
                &["message"],
                ::tracing_core::callsite::Identifier(&__CALLSITE),
            ),
            ::tracing::metadata::Kind::EVENT,
        )
    };
    ::tracing::callsite::DefaultCallsite::new(&META)
};
let enabled = ::tracing::Level::INFO
    <= ::tracing::level_filters::STATIC_MAX_LEVEL
    && ::tracing::Level::INFO <= ::tracing::level_filters::LevelFilter::current()
    && {
        let interest = __CALLSITE.interest();
        !interest.is_never()
            && ::tracing::__macro_support::__is_enabled(
                __CALLSITE.metadata(),
                interest,
            )
    };
if enabled {
    (|value_set: ::tracing::field::ValueSet| {
        let meta = __CALLSITE.metadata();
        ::tracing::Event::dispatch(meta, &value_set);
    })({
        #[allow(unused_imports)]
        use ::tracing::field::{debug, display, Value};
        let mut iter = __CALLSITE.metadata().fields().iter();
        __CALLSITE
            .metadata()
            .fields()
            .value_set(
                &[
                    (
                        &::tracing::__macro_support::Iterator::next(&mut iter)
                            .expect("FieldSet corrupted (this is a bug)"),
                        ::tracing::__macro_support::Option::Some(
                            &format_args!("hello world") as &dyn Value,
                        ),
                    ),
                ],
            )
    });
} else {
}
```

…but we have some pretty **different constraints**

- **No help from the kernel**
- No dynamic binary patching, just **normal Rust code**
- Subscribers and filtering **are in-process**
- Integration with **distributed tracing** (like OpenTelemetry)

Tracing's model for integration **won't work nicely** for USDT…

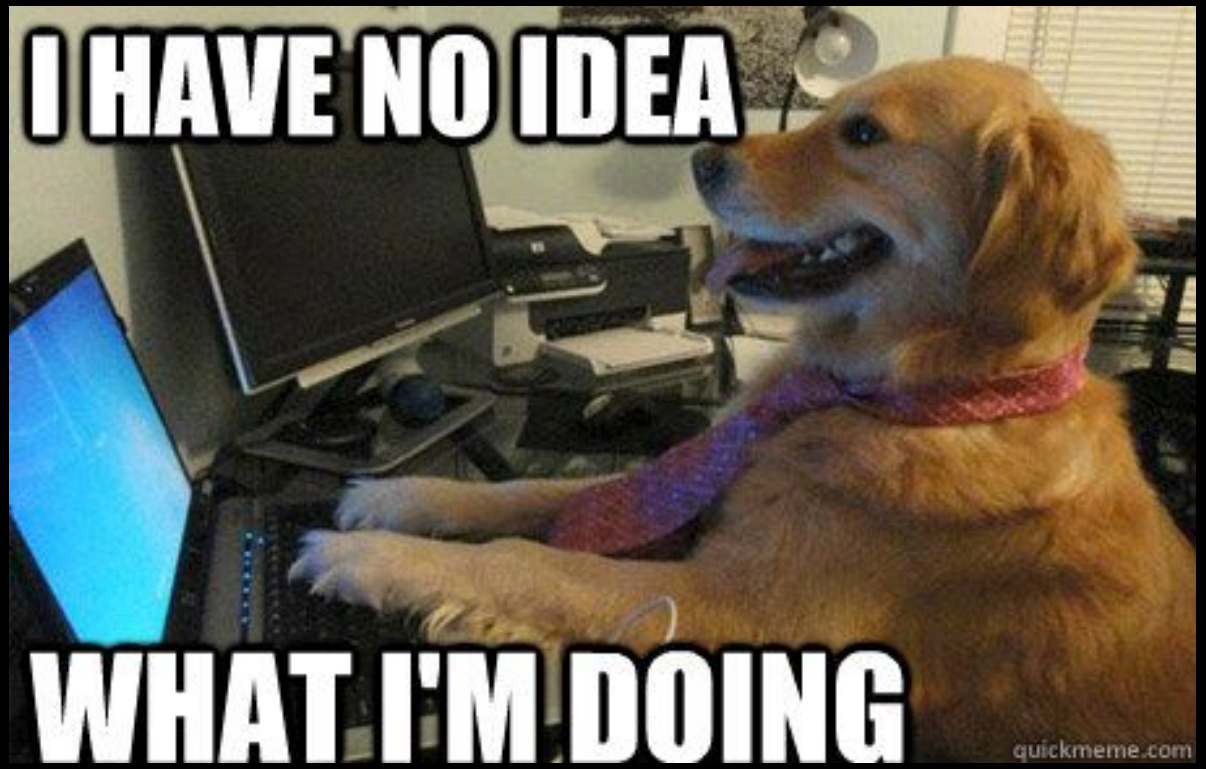
- A subscriber could turn tracing events into USDT probes…
- …but this kind of misses the point!
- Both `tracing` and DTrace rely on **locating instrumentation at the tracepoint**

Solution: **first-party** integration?

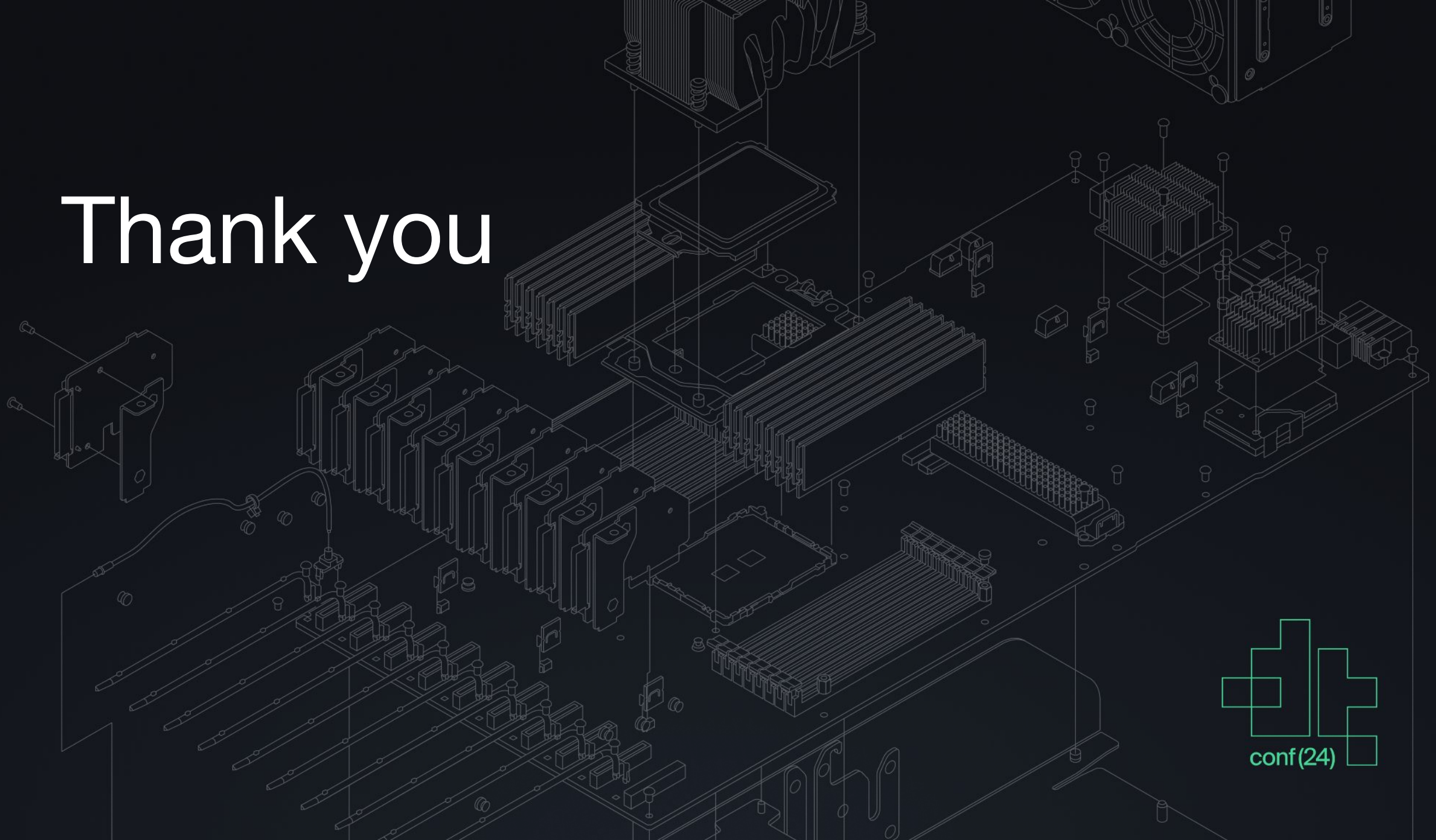
- `tracing`'s macros could (optionally) **emit USDT probes** in addition to all the other stuff
- Fortunately, **I'm the first party** and I can just do that (sort of)

Modeling **spans** in DTrace

- A span IDs is just a **u64**
- Generate **a set of probes per span**:
  - **new_span**
  - **enter**
  - **exit**
  - **close**



Thank you



dt conf(24)