Maintaining Trace Context in Hybrid Async/Non-Async Rust Applications

This document addresses the challenge of maintaining a single, continuous trace across an event-driven Rust application that mixes asynchronous (Axum, WebSockets) and multi-threaded non-asynchronous processing, linked by unbounded channels.

1. The Challenge: Bridging Context Gaps

Your architecture involves several transitions where trace context can be lost if not explicitly handled:

1. **Async to Non-Async (WebSocket Server -> Processing Queue):** The initial event comes from an async Axum handler, but its processing starts on a non-async worker thread.
2. **Non-Async Thread Hops (Processing Thread A -> Processing Thread B/C):** Events are dispatched between different non-async worker threads.
3. **Non-Async to Async (Processing -> External WebSocket Client):** Processed data is sent from a non-async worker to an async WebSocket client for external communication.

The core solution lies in **explicitly propagating the OpenTelemetry Context** through your channel messages.

2. Key Concepts for Context Propagation

- **opentelemetry::Context**: This is the central object that carries trace information (like trace ID, span ID of the parent, baggage) across different parts of your application.
- **Span::current().context()**: When a tracing span is active, you can retrieve the underlying OpenTelemetry Context associated with it. This context contains the information about the currently active span.
- **Context::attach() / Context::with_current()**: These methods allow you to make a specific opentelemetry::Context the "current" context for the executing thread. Any new tracing spans created while this context is active will automatically become children of the span represented by that context.
- **TextMapPropagator**: While opentelemetry::Context is an in-memory object, you often need to serialize it to pass it across boundaries (like network requests or, in your case, channels). The TextMapPropagator (specifically TraceContextPropagator for W3C Trace Context) helps inject context into and extract it from a HashMap<String, String> (or similar text-based carrier).

3. Implementation Strategy

The strategy involves modifying your event structures to carry the trace context and then explicitly injecting and extracting this context at each channel boundary.

Step 3.1: Modify Your Event Message Structure

Your event messages (the ones passed through the unbounded channels) need a new field to carry the trace context. A HashMap<String, String> is a good choice as it's easily serializable and compatible with TextMapPropagator.

```
// In your common types or event definitions file (e.g., `src/types.rs`)

use std::collections::HashMap;
use opentelemetry::Context;
use opentelemetry::propagation::{TextMapPropagator, Extractor, Injector};
use opentelemetry::sdk::propagation::TraceContextPropagator; // W3C Trace Context

/// A custom struct to hold your event data and the trace context.
#[derive(Debug, Clone)] // Clone is often useful for channel messages
pub struct MyEvent {
    pub id: String,
    pub payload: String,
    /// This HashMap will carry the serialized OpenTelemetry trace context.
    pub trace_context_carrier: HashMap<String, String>,
}

/// Helper struct for injecting/extracting context into/from a HashMap.
/// Implements `Injector` and `Extractor` traits from OpenTelemetry.
pub struct HashMapCarrier<'a>(pub &'a mut HashMap<String, String>);

impl<'a> Injector for HashMapCarrier<'a> {
    fn set(&mut self, key: &str, value: String) {
        self.0.insert(key.to_string(), value);
    }
}

impl<'a> Extractor for HashMapCarrier<'a> {
    fn get(&self, key: &str) -> Option<&str> {
        self.0.get(key).map(|s| s.as_str())
    }

    fn keys(&self) -> Vec<&str> {
        self.0.keys().map(|s| s.as_str()).collect()
    }
}

/// Injects the current OpenTelemetry Context into a new HashMap.
```

```rust
pub fn inject_current_context() -> HashMap<String, String> {
    let propagator = TraceContextPropagator::new();
    let parent_context = Context::current(); // Get the current active context
    let mut carrier = HashMap::new();
    propagator.inject_context(&parent_context, &mut HashMapCarrier(&mut carrier));
    carrier
}

/// Extracts an OpenTelemetry Context from a HashMap.
pub fn extract_context_from_carrier(carrier: &HashMap<String, String>) -> Context {
    let propagator = TraceContextPropagator::new();
    propagator.extract_context(&HashMapCarrier(carrier))
}
```

Step 3.2: Initialize OpenTelemetry and tracing Subscriber (Main)

Ensure your main.rs (or application entry point) has the OpenTelemetry and tracing setup as discussed previously. This provides the global tracer and subscriber.

Cargo.toml dependencies (reiterated for clarity):

```toml
[dependencies]
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter", "fmt"] }
tracing-opentelemetry = "0.23"
opentelemetry = { version = "0.22", features = ["trace", "metrics"] }
opentelemetry-sdk = { version = "0.22", features = ["trace", "rt-tokio"] }
opentelemetry-otlp = { version = "0.15", features = ["trace", "http-proto", "reqwest-client",
"tokio"] }
tokio = { version = "1", features = ["full"] } # For async runtime
uuid = { version = "1.0", features = ["v4"] } # For unique event IDs
futures-util = "0.3" # For WebSocket stream/sink
tokio-tungstenite = "0.21" # For WebSocket client
axum = "0.7" # For WebSocket server
tower-http = { version = "0.5", features = ["trace"] } # For Axum tracing middleware
```

src/main.rs (Illustrative, showing core setup and channel usage):

```rust
// src/main.rs
mod types; // Assuming your MyEvent and context helpers are in src/types.rs

use axum::{
    extract::{
        ws::{Message, WebSocket, WebSocketUpgrade},
        State,
    },
```

```
    response::Response,
    routing::get,
    Router,
};
use opentelemetry::{
    global,
    sdk::{
        trace::{self, Sampler},
        Resource,
    },
    KeyValue,
};
use opentelemetry_otlp::{self, WithExportConfig};
use tokio::sync::mpsc;
use tracing::{info, instrument, Span};
use tracing_subscriber::{prelude::*, registry::Registry, EnvFilter};
use types::{inject_current_context, MyEvent, extract_context_from_carrier}; // Import your
types

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    // 1. Configure OpenTelemetry Tracer Provider
    let otlp_exporter = opentelemetry_otlp::new_exporter()
        .http()
        .with_endpoint("http://localhost:4318/v1/traces") // OTLP Collector endpoint
        .with_timeout(std::time::Duration::from_secs(3))
        .build()?;

    let tracer = opentelemetry_otlp::new_pipeline()
        .tracing()
        .with_exporter(otlp_exporter)
        .with_trace_config(
            trace::config()
                .with_sampler(Sampler::AlwaysOn)
                .with_resource(Resource::new(vec![
                    KeyValue::new("service.name", "hybrid-rust-app"),
                    KeyValue::new("service.version", "0.1.0"),
                ])),
        )
        .install_batch(opentelemetry::runtime::Tokio)?;

    // 2. Create tracing-opentelemetry layer
    let opentelemetry_layer = tracing_opentelemetry::OpenTelemetryLayer::new(tracer);
```

```
    // 3. Create tracing subscriber for console output and OpenTelemetry export
    let subscriber = Registry::default()
        .with(EnvFilter::from_default_env())
        .with(opentelemetry_layer)
        .with(tracing_subscriber::fmt::layer().compact());

    global::set_subscriber(subscriber)?;

    info!("Application starting...");

    // Setup channels for communication between async and non-async parts
    let (ws_to_processor_tx, ws_to_processor_rx) = mpsc::unbounded_channel::<MyEvent>();
    let (processor_to_ws_client_tx, processor_to_ws_client_rx) =
mpsc::unbounded_channel::<MyEvent>();

    // Spawn non-async worker threads
    for i in 0..2 { // Example: 2 worker threads
        let rx_clone = ws_to_processor_rx.clone(); // Clone receiver for each thread
        let next_tx_clone = processor_to_ws_client_tx.clone(); // Clone sender for each thread
        std::thread::spawn(move || {
            non_async_event_processor(rx_clone, next_tx_clone, i);
        });
    }
    // Drop the original receiver to ensure channels close when all senders are dropped
    drop(ws_to_processor_rx);
    drop(processor_to_ws_client_tx);


    // Spawn async WebSocket client task
    tokio::spawn(async move {
        // In a real app, you'd connect to an external WebSocket service here
        let (ws_stream, _) = tokio_tungstenite::connect_async("ws://echo.websocket.org")
            .await
            .expect("Failed to connect to external WebSocket");
        info!("Connected to external WebSocket service.");
        external_websocket_sender(processor_to_ws_client_rx, ws_stream).await;
    });

    // Setup Axum server
    let app = Router::new()
        .route("/ws", get(websocket_handler))
        .with_state(ws_to_processor_tx) // Pass sender to Axum handlers
```

```
        .layer(tower_http::trace::TraceLayer::new_for_http()); // Basic HTTP tracing for Axum

    let listener = tokio::net::TcpListener::bind("127.0.0.1:3000").await?;
    info!("Axum server listening on http://127.0.0.1:3000");
    axum::serve(listener, app).await?;

    info!("Application finished.");
    global::shutdown_tracer_provider(); // Ensure all traces are flushed
    Ok(())
}

// --- Async Axum WebSocket Server ---
#[instrument(skip(ws, tx))]
async fn websocket_handler(
    ws: WebSocketUpgrade,
    State(tx): State<mpsc::UnboundedSender<MyEvent>>,
) -> Response {
    ws.on_upgrade(move |socket| handle_socket(socket, tx))
}

#[instrument(skip(socket, tx))]
async fn handle_socket(mut socket: WebSocket, tx: mpsc::UnboundedSender<MyEvent>) {
    info!("WebSocket connection established.");

    while let Some(msg) = socket.recv().await {
        if let Ok(msg) = msg {
            if let Message::Text(text) = msg {
                info!("Received message: {}", text);

                // --- INJECT CONTEXT: Async to Non-Async ---
                // Get the current OpenTelemetry context from the active tracing span (created by
#[instrument] or TraceLayer)
                let trace_context_carrier = inject_current_context();

                let event = MyEvent {
                    id: uuid::Uuid::new_v4().to_string(),
                    payload: text,
                    trace_context_carrier,
                };

                if let Err(e) = tx.send(event) {
                    tracing::error!("Failed to send event to processing queue: {:?}", e);
                    break;
```

```rust
            }
        }
    } else {
        tracing::warn!("WebSocket receive error or close.");
        break;
    }
  }
  info!("WebSocket connection closed.");
}

// --- Non-Async Event Processor (Runs on dedicated threads) ---
fn non_async_event_processor(
  mut rx: mpsc::UnboundedReceiver<MyEvent>,
  next_tx: mpsc::UnboundedSender<MyEvent>,
  thread_id: usize,
) {
  info!("Non-async processor thread {} started.", thread_id);
  // Use `blocking_recv` for non-async context
  while let Some(event) = rx.blocking_recv() {
    // --- EXTRACT & ATTACH CONTEXT: Channel Receive ---
    let parent_otel_context = extract_context_from_carrier(&event.trace_context_carrier);

    // Make this context the current active context for the current thread.
    // All new tracing spans created within this block will be children of this context.
    let _guard = parent_otel_context.attach();

    // Create a new tracing span for the processing logic.
    // This span will automatically link to the parent context.
    let processing_span = tracing::span!(
      tracing::Level::INFO,
      "event_processing",
      event.id = &event.id,
      thread_id = thread_id,
      payload_len = event.payload.len()
    );
    let _processing_guard = processing_span.enter();

    info!("Thread {}: Processing event ID: {}", thread_id, event.id);
    // Simulate some CPU-bound work
    std::thread::sleep(std::time::Duration::from_millis(50));

    // Example: If this processing emits another event to another queue/thread:
    let new_payload = format!("Processed by Thread {}: {}", thread_id, event.payload);
```

```
        // --- INJECT CONTEXT: Non-Async to Async (or another Non-Async) ---
        // Get the current OpenTelemetry context from the active span (processing_span)
        let trace_context_carrier_for_next_event = inject_current_context();

        let next_event = MyEvent {
            id: uuid::Uuid::new_v4().to_string(),
            payload: new_payload,
            trace_context_carrier: trace_context_carrier_for_next_event,
        };

        if let Err(e) = next_tx.send(next_event) {
            tracing::error!("Thread {}: Failed to send next event: {:?}", thread_id, e);
        }

        info!("Thread {}: Finished processing event ID: {}", thread_id, event.id);
        // `_processing_guard` and `_guard` are dropped here, ending spans and detaching
context
    }
    info!("Non-async processor thread {} stopped.", thread_id);
}

// --- Async WebSocket Client (Sending to External Service) ---
#[instrument(skip(rx, ws_stream))]
async fn external_websocket_sender(
    mut rx: mpsc::UnboundedReceiver<MyEvent>,
    mut ws_stream:
tokio_tungstenite::WebSocketStream<tokio_tungstenite::MaybeTlsStream<tokio::net::TcpStrea
m>>,
) {
    info!("External WebSocket sender task started.");
    while let Some(event_to_send) = rx.recv().await {
        // --- EXTRACT & ATTACH CONTEXT: Channel Receive ---
        let parent_otel_context =
extract_context_from_carrier(&event_to_send.trace_context_carrier);
        let _guard = parent_otel_context.attach();

        // Create a span for the actual sending over WebSocket
        let send_ws_span = tracing::span!(
            tracing::Level::INFO,
            "external_websocket_send",
            event.id = &event_to_send.id,
            final_payload_len = event_to_send.payload.len()
```

```
        );
        let _send_ws_guard = send_ws_span.enter();

        info!("Sending via external WebSocket: {}", event_to_send.payload);
        let message = tokio_tungstenite::tungstenite::Message::Text(event_to_send.payload);
        if let Err(e) = ws_stream.send(message).await {
            tracing::error!("Failed to send message over external WebSocket: {:?}", e);
        }
        info!("Message sent over external WebSocket.");
        // `_send_ws_guard` and `_guard` are dropped here
    }
    info!("External WebSocket sender task stopped.");
}
```

Step 3.3: Configure and Run OpenTelemetry Collector

This remains the same as in the previous explanation. The collector will receive OTLP traces and forward them to your Elastic APM Server.

otel-collector-config.yaml:

```yaml
# otel-collector-config.yaml
receivers:
  otlp:
    protocols:
      grpc:
      http:
        endpoint: 0.0.0.0:4318 # Default OTLP HTTP port

processors:
  batch:
    send_batch_size: 100
    timeout: 1s
  resource:
    attributes:
      - key: host.name
        value: my-hybrid-app-host
        action: upsert

exporters:
  otlphttp/elastic:
    endpoint: "http://<your-elastic-apm-server-host>:8200" # **IMPORTANT: Replace with your Elastic APM Server URL**
    compression: gzip
    # headers:
```

```yaml
  #   authorization: "Bearer <your_elastic_apm_api_key>"

service:
  pipelines:
    traces:
      receivers: [otlp]
      processors: [batch, resource]
      exporters: [otlphttp/elastic]
```

Run the collector:

```
./opentelemetry-collector-contrib --config otel-collector-config.yaml
```

Step 3.4: Run Your Rust Application

```
cargo run
```

You can test this by connecting to ws://127.0.0.1:3000/ws using a WebSocket client (e.g., a browser's developer console or a tool like Postman/Insomnia) and sending text messages. You should see the traces appear in Kibana.

4. Explanation of Context Flow

1. **Axum WebSocket Handler (handle_socket):**
   - An initial tracing span is created by #[instrument] or Axum's TraceLayer.
   - When a message is received and an MyEvent is created: inject_current_context() is called. This function gets the opentelemetry::Context from the *current active span* (the Axum handler span) and serializes it into the trace_context_carrier HashMap.
   - The MyEvent with the embedded context is sent to the ws_to_processor_tx channel.
2. **Non-Async Event Processor (non_async_event_processor):**
   - When rx.blocking_recv() receives an MyEvent:
     - extract_context_from_carrier() deserializes the HashMap back into an opentelemetry::Context. This is the *parent context* from the Axum handler.
     - parent_otel_context.attach() makes this parent context the *current active context* for the worker thread.
     - A new tracing::span! (event_processing) is created. Because a parent context is active, this new span automatically becomes a *child* of the Axum handler span, linking them together in the trace.
   - If this processor then emits a next_event:
     - inject_current_context() is called again. This time, it captures the context of the event_processing span (which is a child of the original Axum span).
     - This new context is embedded in the next_event and sent to the

processor_to_ws_client_tx channel.

3. **Async WebSocket Client (external_websocket_sender):**
   - When rx.recv().await receives an MyEvent:
     - Similar to the non-async processor, extract_context_from_carrier() extracts the context (which now represents the event_processing span).
     - parent_otel_context.attach() makes it active.
     - A new tracing::span! (external_websocket_send) is created, automatically becoming a child of the event_processing span.
   - The actual WebSocket send operation happens within this span, completing the trace path from the initial incoming message to the final outgoing message.

By explicitly passing and activating the opentelemetry::Context at each boundary, you ensure that all segments of your event-driven flow are correctly linked together into a single, comprehensive distributed trace in Elastic.