Tracing Event-Driven Rust Systems: Channel-Based Context Propagation

This document provides a focused strategy for implementing distributed tracing in your non-async, event-driven Rust system. The goal is to perform all tracing operations at the **publish/dispatch layer** of your crossbeam_channel communication, keeping your core business logic components (Order Manager, Algorithm, Order Sender) clean and unaware of tracing concerns.

This approach leverages your existing NotificationHandlerOnce trait and introduces a generic channel message wrapper to carry OpenTelemetry trace context.

1. The Core Strategy: Intercepting at the Channel Boundary

The elegance of this solution lies in intercepting the notification flow precisely where it crosses channel boundaries. Your NotificationHandlerOnce<T> trait implementations for crossbeam_channel::Sender<T> are the ideal points to:

1. **Inject Context:** When a component calls handle_notification to publish an event, the current OpenTelemetry trace context is automatically captured and embedded into a wrapper message.
2. **Extract and Attach Context:** On the receiving end, before a component processes a message, the trace context is extracted from the wrapper and attached to the current thread. This ensures that any new spans created during processing are correctly linked as children of the originating span.

This strategy uses a **generic TracedChannelMessage<T> wrapper** and a custom TracedNotificationSender<T> that implements your traits.

2. src/tracing_channels.rs (Generic Traced Message & Helpers)

This file will define the generic TracedChannelMessage struct and the essential helper functions for injecting and extracting OpenTelemetry Context, including Baggage.

```
// src/tracing_channels.rs
use std::collections::HashMap;
use opentelemetry::Context;
use opentelemetry::propagation::{TextMapPropagator, Extractor, Injector};
use opentelemetry::sdk::propagation::TraceContextPropagator;
use tracing::Span;
use opentelemetry::KeyValue; // For Baggage
```

```rust
/// A generic wrapper for messages sent over internal crossbeam channels.
/// It encapsulates the original notification/event and the serialized OpenTelemetry trace
context.
#[derive(Debug, Clone)]
pub struct TracedChannelMessage<T> {
    pub notification: T,
    pub trace_context_carrier: HashMap<String, String>,
}

/// Helper struct that implements `Injector` and `Extractor` for a `HashMap`.
/// Used by OpenTelemetry's `TextMapPropagator` to serialize/deserialize context.
pub struct HashMapCarrier<'a>(pub &'a mut HashMap<String, String>);

impl<'a> Injector for HashMapCarrier<'a> {
    /// Sets a key-value pair in the underlying HashMap.
    fn set(&mut self, key: &str, value: String) {
        self.0.insert(key.to_string(), value);
    }
}

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

    /// Returns a vector of all keys in the HashMap.
    fn keys(&self) -> Vec<&str> {
        self.0.keys().map(|s| s.as_str()).collect()
    }
}

/// Injects the current OpenTelemetry Context (including Baggage) into a new HashMap.
/// This function should be called on the **sending/publishing** side of a channel.
/// It captures the context of the currently active `tracing` span.
pub fn inject_current_context() -> HashMap<String, String> {
    let propagator = TraceContextPropagator::new();
    let parent_context = Span::current().context(); // Captures the context of the active span
    let mut carrier = HashMap::new();
    propagator.inject_context(&parent_context, &mut HashMapCarrier(&mut carrier));
    carrier
}
```

```
/// Extracts an OpenTelemetry Context (including Baggage) from a HashMap.
/// This function should be called on the **receiving/consuming** side of a channel,
/// typically before processing the event, to re-activate the trace context.
pub fn extract_context_from_carrier(carrier: &HashMap<String, String>) -> Context {
    let propagator = TraceContextPropagator::new();
    propagator.extract_context(&HashMapCarrier(carrier))
}

/// Executes a closure with specific Baggage attached to the current OpenTelemetry Context.
/// Baggage items are key-value data that propagate along with the trace,
/// useful for carrying business correlation IDs (like UserID, ClientOrderID)
/// that might not be part of every span's direct attributes.
pub fn with_baggage<F, R>(baggage: Vec<KeyValue>, f: F) -> R
where
    F: FnOnce() -> R,
{
    let current_context = Context::current();
    // Fold new KeyValues into the current context to create a new context with baggage
    let new_context = baggage.into_iter().fold(current_context, |ctx, kv| ctx.with_value(kv));
    // Execute the closure with the new context as the current one, ensuring baggage
propagates
    new_context.with_current(f)
}
```

3. src/notification_system.rs (Your Traits and Traced Sender)

This file will contain your existing NotificationHandlerOnce and IntoNotificationHandlerOnceBox traits. Crucially, it will also define TracedNotificationSender, which implements these traits and performs the "automagical" context injection.

```
// src/notification_system.rs
use super::tracing_channels::{TracedChannelMessage, inject_current_context,
extract_context_from_carrier, with_baggage};
use crossbeam_channel::{Sender, Receiver}; // Import Receiver for consumers as well
use std::any::type_name;
use opentelemetry::{Context, KeyValue};
use tracing::{self, info, span, Level, Span}; // Import tracing crate for macros and Span

// Your existing traits (no changes needed here)
pub trait NotificationHandlerOnce<T>: Send + Sync {
    fn handle_notification(&self, notification: T);
}
```

```
pub trait IntoNotificationHandlerOnceBox<T> {
    fn into_notification_handler_once_box(self) -> Box<dyn NotificationHandlerOnce<T>>;
}

// --- New `TracedNotificationSender` Wrapper ---
/// A wrapper around a `crossbeam_channel::Sender` that automatically
/// injects the current OpenTelemetry trace context when a notification is sent.
/// This struct implements your `NotificationHandlerOnce` trait.
pub struct TracedNotificationSender<T> {
    sender: Sender<TracedChannelMessage<T>>,
}

impl<T> TracedNotificationSender<T> {
    /// Creates a new `TracedNotificationSender` from an underlying
`crossbeam_channel::Sender`.
    pub fn new(sender: Sender<TracedChannelMessage<T>>) -> Self {
        Self { sender }
    }
}

// Implementation of your `NotificationHandlerOnce` trait for the traced sender.
// This is where the "automagical" context injection happens.
impl<T> NotificationHandlerOnce<T> for TracedNotificationSender<T>
where
    T: Send + Sync, // Your notification type must be Send + Sync
{
    fn handle_notification(&self, notification: T) {
        // --- Automagical Context Injection ---
        // When `handle_notification` is called (i.e., a notification is emitted),
        // we automatically capture the OpenTelemetry context from the *current* active tracing
span.
        let trace_context_carrier = inject_current_context();

        // Wrap the original notification with the captured trace context.
        let traced_message = TracedChannelMessage {
            notification, // The actual data your components care about
            trace_context_carrier, // The tracing metadata that will propagate
        };

        // Send the wrapped message through the internal crossbeam channel.
        if let Err(err) = self.sender.send(traced_message) {
            tracing::warn!("Failed to send traced notification {}: {:?}", type_name::<T>(), err);
```

```
        } else {
            info!("Successfully sent traced notification {}", type_name::<T>());
        }
    }
}

// Implementation to convert TracedNotificationSender into a Boxed Trait Object.
// This allows you to pass the sender around as `Box<dyn NotificationHandlerOnce<T>>`.
impl<T> IntoNotificationHandlerOnceBox<T> for TracedNotificationSender<T>
where
    T: Send + Sync + 'static, // 'static bound required for Box<dyn Trait>
{
    fn into_notification_handler_once_box(self) -> Box<dyn NotificationHandlerOnce<T>> {
        Box::new(self)
    }
}

// --- Example Event Types for Your System ---
// These are your actual business-logic event structs.
// They remain clean and do not contain tracing-specific fields.
#[derive(Debug, Clone)]
pub struct ClientOrder {
    pub id: String, // Represents ClientOrderID
    pub user_id: String,
    pub price: f64,
    pub quantity: u64,
    pub order_type: String, // e.g., "Limit", "Market"
}

#[derive(Debug, Clone)]
pub struct ExchangeOrder {
    pub id: String, // Represents ExchangeOrderID
    pub parent_client_order_ids: Vec<String>, // Links back to original client orders
    pub quantity: u64,
    pub exchange: String,
}

#[derive(Debug, Clone)]
pub struct Fill {
    pub id: String, // Represents Fill ID
    pub exchange_order_id: String,
    pub filled_quantity: u64,
    pub filled_price: f64,
```

```
    pub timestamp: u64,
}

// --- Helper for Processing Traced Messages ---
/// Generic function to process a received `TracedChannelMessage`.
/// It extracts and attaches the trace context, creates a new span,
/// and then calls the provided processing logic.
pub fn process_traced_message<T, F>(traced_message: TracedChannelMessage<T>,
process_logic: F)
where
    T: Send + Sync + 'static, // Notification type must be Send + Sync and 'static
    F: FnOnce(&T), // The actual business logic to apply to the notification
{
    // 1. Extract the parent OpenTelemetry Context from the received message.
    let parent_otel_context =
extract_context_from_carrier(&traced_message.trace_context_carrier);

    // 2. Attach this context to the current thread.
    // This ensures any new `tracing` spans created within this block
    // will automatically become children of the emitter's span.
    let _guard = parent_otel_context.attach();

    // 3. Create a new tracing span for the processing of this specific event.
    // This span will automatically link to the parent context.
    let processing_span = span!(
        Level::INFO,
        "component_processing",
        event_type = type_name::<T>(), // Dynamic type name for span attribute
        // You can add more attributes here based on common fields in your notifications,
        // e.g., if T has `user_id` and `id` fields:
        // user_id = &traced_message.notification.user_id,
        // event_id = &traced_message.notification.id,
    );
    let _processing_guard = processing_span.enter();

    info!("Processing traced message: {:?}", type_name::<T>());

    // Execute the actual component logic with the unpacked notification
    process_logic(&traced_message.notification);

    // The `_processing_guard` drops here, ending the `component_processing` span.
    // The `_guard` drops here, detaching the parent context from the thread.
}
```

```
// Helper to add common correlation IDs from an event to the current span.
// This is useful for making spans searchable by business IDs in Kibana.
pub fn add_event_correlation_to_span(span: &Span, event_type: &str, user_id: &str, order_id:
&str) {
    span.record("event.type", event_type);
    span.record("user.id", user_id);
    span.record("order.id", order_id);
}
```

4. src/main.rs (Orchestration and Component Wiring)

This main.rs demonstrates how to set up the OpenTelemetry tracing, create your crossbeam channels with the TracedChannelMessage type, and then wire your components using the TracedNotificationSender and process_traced_message helpers.

```
// src/main.rs
use opentelemetry::{
    global,
    sdk::{
        trace::{self, Sampler},
        Resource,
    },
    KeyValue,
};
use opentelemetry_otlp::{self, WithExportConfig};
use tracing::{info, span, Level};
use tracing_subscriber::{prelude::*, registry::Registry, EnvFilter};
use crossbeam_channel::{unbounded, Sender, Receiver};
use std::thread;
use std::time::Duration;
use uuid::Uuid; // For generating unique IDs

mod notification_system; // Your traits, TracedNotificationSender, and event types
mod tracing_channels;   // TracedChannelMessage and context helpers

use notification_system::{
    NotificationHandlerOnce, IntoNotificationHandlerOnceBox, TracedNotificationSender,
    ClientOrder, ExchangeOrder, Fill, process_traced_message, add_event_correlation_to_span
};
use tracing_channels::TracedChannelMessage;

/// Sets up the global OpenTelemetry tracer and tracing subscriber.
```

```
fn setup_tracing() -> Result<(), Box<dyn std::error::Error>> {
    // Configure the OTLP exporter to send traces to an OpenTelemetry Collector.
    let otlp_exporter = opentelemetry_otlp::new_exporter()
        .http()
        .with_endpoint("http://localhost:4318/v1/traces") // Default OTLP HTTP endpoint for
Collector
        .with_timeout(Duration::from_secs(3)) // Timeout for sending traces
        .build()?;

    // Build the OpenTelemetry tracer pipeline.
    let tracer = opentelemetry_otlp::new_pipeline()
        .tracing()
        .with_exporter(otlp_exporter)
        .with_trace_config(
            trace::config()
                .with_sampler(Sampler::AlwaysOn) // Sample all traces
                .with_resource(Resource::new(vec![ // Define service-level attributes
                    KeyValue::new("service.name", "trading-system-core"),
                    KeyValue::new("environment", "development"),
                ])),
        )
        // Install the batch tracer, using the Tokio runtime for background span export.
        // Even if your core logic is non-async, the exporter runs in a background task.
        .install_batch(opentelemetry::runtime::Tokio)?;

    // Create the tracing-opentelemetry layer to bridge `tracing` to OpenTelemetry.
    let opentelemetry_layer = tracing_opentelemetry::OpenTelemetryLayer::new(tracer);

    // Create the tracing subscriber, combining multiple layers:
    // - `EnvFilter`: Allows controlling log levels via the RUST_LOG environment variable.
    // - `opentelemetry_layer`: Sends traces to OpenTelemetry.
    // - `fmt::layer().compact()`: Provides formatted console output for local debugging.
    let subscriber = Registry::default()
        .with(EnvFilter::from_default_env())
        .with(opentelemetry_layer)
        .with(tracing_subscriber::fmt::layer().compact());

    // Set the configured subscriber as the global default.
    global::set_subscriber(subscriber)?;
    Ok(())
}

fn main() -> Result<(), Box<dyn std::error::Error>> {
```

```
    setup_tracing()?; // Initialize tracing and OpenTelemetry
    info!("Trading system core starting...");

    // --- 1. Define Channel Types ---
    // All crossbeam channels will now carry `TracedChannelMessage<YourSpecificEvent>`.
    let (tx_client_order, rx_client_order): (Sender<TracedChannelMessage<ClientOrder>>,
Receiver<TracedChannelMessage<ClientOrder>>) = unbounded();
    let (tx_exchange_order, rx_exchange_order):
(Sender<TracedChannelMessage<ExchangeOrder>>,
Receiver<TracedChannelMessage<ExchangeOrder>>) = unbounded();
    let (tx_fills, rx_fills): (Sender<TracedChannelMessage<Fill>>,
Receiver<TracedChannelMessage<Fill>>) = unbounded();

    // --- 2. Instantiate Traced Senders for Components ---
    // Wrap the raw `crossbeam_channel::Sender`s with `TracedNotificationSender`.
    // These wrapped senders are then converted into `Box<dyn NotificationHandlerOnce<T>>`
    // for distribution to your components.

    // Order Manager's sender for Client Orders (to Algorithm)
    let order_manager_tx_algo = TracedNotificationSender::new(tx_client_order);
    let order_manager_handler: Box<dyn NotificationHandlerOnce<ClientOrder>> =
order_manager_tx_algo.into_notification_handler_once_box();

    // Algorithm's sender for Exchange Orders (to Order Sender)
    let algorithm_tx_order_sender = TracedNotificationSender::new(tx_exchange_order);
    let algorithm_handler: Box<dyn NotificationHandlerOnce<ExchangeOrder>> =
algorithm_tx_order_sender.into_notification_handler_once_box();

    // Order Sender's sender for Fills (to Algorithm)
    let order_sender_tx_algo = TracedNotificationSender::new(tx_fills);
    let order_sender_handler: Box<dyn NotificationHandlerOnce<Fill>> =
order_sender_tx_algo.into_notification_handler_once_box();

    // --- 3. Spawn Component Threads ---
    // Each component runs in its own thread, consuming from its input channel(s)
    // and publishing to its output channel(s) using the traced handlers.

    // Order Manager Thread (Simulates receiving client orders from WSS server and publishing)
    let om_handler_clone = order_manager_handler.clone(); // Clone the boxed handler for the
thread
    thread::spawn(move || {
        // Create a span for the entire Order Manager thread's execution.
        let om_span = span!(Level::INFO, "order_manager_thread");
```

```
        let _guard = om_span.enter(); // Enter the span for the thread's lifetime
        info!("Order Manager thread started.");

        // Simulate receiving client orders (e.g., from an external async WSS server)
        for i in 0..5 {
            let client_order = ClientOrder {
                id: Uuid::new_v4().to_string(), // Unique ClientOrderID
                user_id: format!("user_{}", i % 2), // Simulate different users
                price: 100.0 + (i as f64),
                quantity: 10 + (i as u64),
                order_type: if i % 2 == 0 { "Limit".to_string() } else { "Market".to_string() },
            };

            // Create a span representing the receipt of this specific client order by the OM.
            let client_order_receipt_span = span!(Level::INFO, "om_receive_client_order");
            let _co_guard = client_order_receipt_span.enter();
            // Add correlation IDs to the current span for easy searching in Kibana.
            add_event_correlation_to_span(&Span::current(), "ClientOrder", &client_order.user_id,
&client_order.id);
            info!("Order Manager: Simulating receiving client order: {}", client_order.id);

            // --- Inject Baggage for UserID and ClientOrderID ---
            // Baggage will propagate with the trace context to all downstream components.
            tracing_channels::with_baggage(
                vec![
                    KeyValue::new("user.id", client_order.user_id.clone()),
                    KeyValue::new("client.order_id", client_order.id.clone()),
                ],
                || {
                    // Call the traced handler, which will automatically inject the current span's context
(with baggage).
                    om_handler_clone.handle_notification(client_order);
                },
            );
            thread::sleep(Duration::from_millis(150)); // Simulate some work
        }
        info!("Order Manager thread finished publishing orders.");
    });

    // Algorithm Thread (Receives client orders, processes, sends exchange orders, receives
fills, matches)
    let algo_handler_clone = algorithm_handler.clone();
    let os_handler_for_algo_clone = order_sender_handler.clone(); // To simulate fill publishing
```

```
    thread::spawn(move || {
        let algo_span = span!(Level::INFO, "algorithm_thread");
        let _guard = algo_span.enter();
        info!("Algorithm thread started.");

        // --- Consume Client Orders from Order Manager ---
        info!("Algorithm: Listening for client orders.");
        for traced_client_order_msg in rx_client_order {
            // `process_traced_message` handles context extraction/attachment and creates a new
span.
            process_traced_message(traced_client_order_msg, |client_order| {
                // Add correlation IDs to the current span (created by process_traced_message).
                add_event_correlation_to_span(&Span::current(), "ClientOrder",
&client_order.user_id, &client_order.id);
                info!("Algorithm: Received client order: {}", client_order.id);

                // Simulate processing and generating exchange orders
                let exchange_order = ExchangeOrder {
                    id: Uuid::new_v4().to_string(), // Unique ExchangeOrderID
                    parent_client_order_ids: vec![client_order.id.clone()], // Link to parent client order
                    quantity: client_order.quantity,
                    exchange: "XCHANGE_A".to_string(),
                };

                let algo_generate_eo_span = span!(Level::INFO, "algo_generate_exchange_order");
                let _eo_guard = algo_generate_eo_span.enter();
                add_event_correlation_to_span(&Span::current(), "ExchangeOrder",
&client_order.user_id, &exchange_order.id);
                info!("Algorithm: Generating exchange order: {}", exchange_order.id);

                // --- Publish Exchange Order to Order Sender ---
                // The `handle_notification` call will automatically inject the current span's context.
                algo_handler_clone.handle_notification(exchange_order);
                thread::sleep(Duration::from_millis(50)); // Simulate work
            });
        }
        info!("Algorithm thread finished processing client orders.");

        // --- Consume Fills from Order Sender ---
        info!("Algorithm: Now listening for fills.");
        for traced_fill_msg in rx_fills {
            process_traced_message(traced_fill_msg, |fill| {
                add_event_correlation_to_span(&Span::current(), "Fill", &fill.exchange_order_id,
```

```
&fill.id); // UserID might not be on Fill directly
            info!("Algorithm: Received fill: {}", fill.id);

            // Simulate matching fills to client orders (the complex many-to-many part)
            let matching_span = span!(Level::INFO, "algo_match_fill");
            let _match_guard = matching_span.enter();
            info!("Algorithm: Matching fill {} to exchange order {}", fill.id, fill.exchange_order_id);

            // **IMPORTANT for Many-to-Many:**
            // This is where you would use
`Span::current().add_link(Link::new(span_context_of_client_order, vec![]));`
            // to link this `fill_matching` span back to the original `client_order_receipt_span` (or
multiple).
            // You would need a mechanism to retrieve those `SpanContext`s (e.g., a shared map,
as discussed in the previous response).
            // For this simplified channel-focused example, we omit the explicit linking here,
            // but the correlation IDs will still allow you to find related traces.

            thread::sleep(Duration::from_millis(20)); // Simulate work
         });
      }
      info!("Algorithm thread finished processing fills.");
   });


   // Order Sender Thread (Receives exchange orders, simulates external interaction, publishes
fills)
   let os_handler_clone = order_sender_handler.clone(); // Clone for publishing fills
   thread::spawn(move || {
      let os_span = span!(Level::INFO, "order_sender_thread");
      let _guard = os_span.enter();
      info!("Order Sender thread started.");

      // --- Consume Exchange Orders from Algorithm ---
      info!("Order Sender: Listening for exchange orders.");
      for traced_exchange_order_msg in rx_exchange_order {
         process_traced_message(traced_exchange_order_msg, |exchange_order| {
            add_event_correlation_to_span(&Span::current(), "ExchangeOrder",
&exchange_order.parent_client_order_ids[0], &exchange_order.id);
            info!("Order Sender: Sending exchange order to external exchange: {}",
exchange_order.id);

            // Simulate external exchange interaction (e.g., via an async adapter)
```

```
            thread::sleep(Duration::from_millis(100)); // Simulate network latency

            // Simulate receiving a fill for this exchange order
            let fill = Fill {
                id: Uuid::new_v4().to_string(), // Unique Fill ID
                exchange_order_id: exchange_order.id.clone(),
                filled_quantity: exchange_order.quantity / 2, // Partial fill for demo
                filled_price: exchange_order.price,
                timestamp: chrono::Utc::now().timestamp_millis() as u64,
            };

            let os_receive_fill_span = span!(Level::INFO, "os_receive_fill");
            let _fill_guard = os_receive_fill_span.enter();
            add_event_correlation_to_span(&Span::current(), "Fill",
&exchange_order.parent_client_order_ids[0], &fill.id);
            info!("Order Sender: Simulating receiving fill: {}", fill.id);

            // --- Publish Fill to Algorithm ---
            // The `handle_notification` call will automatically inject the current span's context.
            os_handler_clone.handle_notification(fill);
        });
    }
    info!("Order Sender thread finished.");
  });


  // Keep main alive for a bit to allow threads to run and traces to be exported
  thread::sleep(Duration::from_secs(10));

  info!("Application shutting down.");
  global::shutdown_tracer_provider(); // Ensure all traces are flushed before exit
  Ok(())
}
```

5. Verification and Key Benefits

To verify this setup:

1. **Run OpenTelemetry Collector:** Ensure your OpenTelemetry Collector is running with an OTLP receiver and an Elastic exporter (as configured in previous responses).
   ./opentelemetry-collector-contrib --config otel-collector-config.yaml

2. **Run Your Rust Application:**

```
cargo run
```

3. **Check Kibana APM:** Navigate to the APM section in Kibana. You should see:
   - A trading-system-core service.
   - Traces that span across the order_manager_thread, algorithm_thread, and order_sender_thread.
   - Spans like om_receive_client_order, component_processing (for client orders in Algorithm), algo_generate_exchange_order, component_processing (for exchange orders in Order Sender), os_receive_fill, and component_processing (for fills in Algorithm).
   - Crucially, these spans will be linked hierarchically, showing the flow.
   - The user.id and order.id attributes will be present on the spans, allowing you to filter for specific client orders or users.

Key Benefits of this Approach:

- **Clean Business Logic:** Your ClientOrder, ExchangeOrder, Fill structs, and the core logic within your components remain free of tracing boilerplate.
- **Decoupled Tracing:** Tracing concerns are encapsulated within tracing_channels.rs and TracedNotificationSender.
- **Automatic Context Propagation:** The TracedNotificationSender and process_traced_message helpers ensure trace context is automatically carried and activated across all crossbeam channel boundaries.
- **Searchable Traces:** Consistent use of add_event_correlation_to_span ensures that your business IDs are always available as span attributes for easy filtering and analysis in Kibana.
- **Scalable:** This pattern is robust and scales well across many channels and threads in a complex event-driven system.

This design provides an **intricate** and **meticulous** tracing solution for your non-async core, giving you deep visibility into the flow of orders and fills through your system.