Advanced Tracing for Event-Driven Trading Systems with Batching and Many-to-Many Matching

This document outlines a strategy for implementing comprehensive distributed tracing in a complex, event-driven trading system. The system features:

1. **Fan-in:** Multiple client orders merging into a single batch.
2. **Fan-out:** A single batch generating multiple orders to an external exchange.
3. **Many-to-Many Matching:** Exchange fills matching back to multiple client orders.
4. **Hybrid Async/Non-Async Processing:** Communication via unbounded channels.

The goal is to create a single, cohesive trace that spans the entire lifecycle of a client order, even across these complex relationships.

1. Core OpenTelemetry Concepts for Complex Tracing

To tackle this scenario, we'll heavily rely on:

- **Spans:** Represent individual operations (e.g., "client order received," "batching," "send to exchange," "fill matching").
- **Trace ID:** The unique identifier for an entire end-to-end transaction. All spans belonging to the same logical flow share this ID.
- **Span ID:** Unique identifier for a single span.
- **Parent Span ID:** Establishes a hierarchical relationship (child-of) between spans.
- **Span Links: Crucial for many-to-many relationships.** A Span Link connects a span to one or more *other* spans that are causally related but not directly parent-child. This is perfect for showing that multiple client orders contributed to a batch, or that a single fill affected multiple client orders.
- **Baggage:** A set of key-value pairs propagated along with the trace context. This is ideal for carrying business-level identifiers (like UserID, ClientOrderID) that are useful for filtering and searching, without modifying your core event payloads.
- **Context Propagation:** As previously discussed, explicitly passing and activating the opentelemetry::Context across channel boundaries and thread hops is fundamental.

2. Refined TracedChannelMessage and Context Helpers

Your MyEvent (representing client orders, exchange orders, fills, etc.) will remain clean. We'll enhance TracedChannelMessage and the context helpers to include Baggage.

```
// src/tracing_channels.rs
use std::collections::HashMap;
use opentelemetry::{Context, KeyValue};
```

```
use opentelemetry::propagation::{TextMapPropagator, Extractor, Injector};
use opentelemetry::sdk::propagation::TraceContextPropagator;
use tracing::{Span, span::With}; // `span::With` is for baggage in tracing 0.1

// --- Your original MyEvent struct (unchanged) ---
// This struct will represent different types of events (client order, exchange order, fill)
// based on context. It always carries UserID and ClientOrderID for correlation.
#[derive(Debug, Clone)]
pub struct MyEvent {
    pub user_id: String,
    pub client_order_id: String, // This could be ClientOrderID for client orders, or BatchID for
batches, or ExchangeOrderID for exchange orders
    pub event_type: String, // e.g., "ClientOrder", "ExchangeOrder", "Fill"
    pub payload: String,    // Specific data for this event
    // ... other fields relevant to the specific event type
}

/// A generic wrapper struct for messages sent over channels that need tracing.
/// It carries the original `MyEvent` and the serialized OpenTelemetry trace context.
#[derive(Debug, Clone)]
pub struct TracedChannelMessage {
    pub event: MyEvent,
    pub trace_context_carrier: HashMap<String, String>,
}

/// Helper struct for injecting/extracting context into/from a HashMap.
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()
    }
}
```

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

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

/// Helper to execute a closure with specific Baggage attached to the current context.
/// Baggage items will propagate with the trace context.
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
    new_context.with_current(f)
}
```

3. Tracing Strategy by Component

Let's break down the tracing implementation for each stage of your system.

3.1. WSS Server: Client Order Ingestion (Entry Point)

- **Action:** Start a new trace for each incoming client order. Add UserID and ClientOrderID to both span attributes and Baggage.
- **Span:** client_order_received (root span for this order's journey).
- **Context Propagation:** Inject the context (including Baggage) into TracedChannelMessage for the Order Manager.

```rust
// In your Axum WebSocket handler (simplified)
use crate::tracing_channels::{TracedChannelMessage, inject_current_context, with_baggage};
use opentelemetry::KeyValue;
```

```
use tracing::{info, instrument, Span};

#[instrument(skip(ws, tx))]
async fn handle_socket(mut socket: WebSocket, tx:
mpsc::UnboundedSender<TracedChannelMessage>) {
    // ... (receive message, parse into MyEvent) ...
    if let Message::Text(text) = msg {
        let incoming_order_event = MyEvent {
            user_id: "user_abc".to_string(),
            client_order_id: "client_order_123".to_string(),
            event_type: "ClientOrder".to_string(),
            payload: text,
        };

        // Start a new span for this specific client order's journey.
        // This is the root span for this client order's trace.
        let client_order_span = tracing::span!(
            tracing::Level::INFO,
            "client_order_received",
            user.id = &incoming_order_event.user_id,
            client.order_id = &incoming_order_event.client_order_id
        );
        let _guard = client_order_span.enter();

        info!("Received client order: (UserID: {}, OrderID: {})",
              incoming_order_event.user_id, incoming_order_event.client_order_id);

        // --- Propagate UserID and ClientOrderID as Baggage ---
        // Baggage will travel with the trace context to all downstream components.
        let traced_message = with_baggage(
            vec![
                KeyValue::new("user.id", incoming_order_event.user_id.clone()),
                KeyValue::new("client.order_id", incoming_order_event.client_order_id.clone()),
            ],
            || TracedChannelMessage {
                event: incoming_order_event,
                trace_context_carrier: inject_current_context(), // Context now includes baggage
            },
        );

        if let Err(e) = tx.send(traced_message) {
            tracing::error!("Failed to send traced client order to Order Manager: {:?}", e);
        }
```

```
    }
}
```

3.2. Order Manager & Batching (Fan-In)

- **Action:** Collect multiple client orders. When a batch is formed, create a new batch-specific span. Crucially, **link** all contributing client order spans to this batch span.
- **Span:** order_manager_processing (child of client_order_received), order_batching (new span, linked to multiple client_order_received spans).
- **Context Propagation:** Inject the order_batching span's context into TracedChannelMessage for the Algorithm.
- **State Management:** You'll need a way to temporarily store the SpanContext of active client_order_received spans to create links later. A HashMap keyed by (UserID, ClientOrderID) is suitable.

```
// In your Order Manager component (non-async worker thread)
use crate::tracing_channels::{TracedChannelMessage, extract_context_from_carrier,
inject_current_context};
use opentelemetry::trace::{SpanContext, Link};
use std::collections::HashMap;
use std::sync::{Arc, Mutex};
use tracing::{info, span, Level, Span};

// Shared state to keep track of active client order spans for linking
type ActiveClientOrderSpans = Arc<Mutex<HashMap<(String, String), SpanContext>>>;

fn order_manager_processor(
    mut rx: crossbeam_channel::Receiver<TracedChannelMessage>, // From WSS Server
    batch_tx: crossbeam_channel::Sender<TracedChannelMessage>, // To Algorithm
    active_client_order_spans: ActiveClientOrderSpans,
) {
    info!("Order Manager processor started.");
    for traced_message in rx {
        // Extract and attach context from the incoming client_order_received span
        let parent_otel_context =
extract_context_from_carrier(&traced_message.trace_context_carrier);
        let _guard = parent_otel_context.attach(); // This makes `client_order_received` the
parent context

        // Create a span for the Order Manager's processing of this individual order
        let order_manager_span = span!(
            Level::INFO,
            "order_manager_processing",
            user.id = &traced_message.event.user_id,
```

```
    client.order_id = &traced_message.event.client_order_id
);
let _manager_guard = order_manager_span.enter();

info!("Order Manager: Processing client order ({}, {})",
      traced_message.event.user_id, traced_message.event.client_order_id);

// Store the SpanContext of the current `client_order_received` span.
// This SC will be used to create links to the batch span later.
let client_order_sc = Span::current().context();
active_client_order_spans.lock().unwrap().insert(
    (traced_message.event.user_id.clone(), traced_message.event.client_order_id.clone()),
    client_order_sc,
);

// --- Simulate Batching Logic ---
// In a real system, you'd collect orders here until a batch is ready (e.g., N orders or
timeout)
// For demonstration, let's assume a batch is ready after every 2 orders.
static mut BATCH_COUNT: usize = 0;
static mut BATCHED_ORDERS: Vec<MyEvent> = Vec::new();
unsafe {
    BATCH_COUNT += 1;
    BATCHED_ORDERS.push(traced_message.event);

    if BATCH_COUNT % 2 == 0 { // Batch every 2 orders for demo
        let batch_id = format!("batch_{}", BATCH_COUNT / 2);
        let mut span_links = Vec::new();
        let mut batch_users = Vec::new();
        let mut batch_orders = Vec::new();

        // Create Span Links from the new batch span to each client order's root span
        for client_order_in_batch in &BATCHED_ORDERS {
            if let Some(sc) =
active_client_order_spans.lock().unwrap().get(&(client_order_in_batch.user_id.clone(),
client_order_in_batch.client_order_id.clone())) {
                span_links.push(Link::new(*sc, vec![])); // Link to the client_order_received span
                batch_users.push(client_order_in_batch.user_id.clone());
                batch_orders.push(client_order_in_batch.client_order_id.clone());
            }
        }

        // Create a new span for the batching operation.
```

```
            // This span will be the parent for the subsequent exchange orders.
            let order_batching_span = span!(
                Level::INFO,
                "order_batching",
                batch.id = &batch_id,
                batch.size = BATCHED_ORDERS.len(),
                batch.users = tracing::field::debug(&batch_users),
                batch.orders = tracing::field::debug(&batch_orders),
            );
            let _batch_guard = order_batching_span.enter();

            // Add the collected Span Links to the current batching span.
            // This explicitly shows the fan-in relationship.
            for link in span_links {
                Span::current().add_link(link);
            }

            info!("Batch {} ready for algorithm processing. Contains {} orders.", batch_id,
BATCHED_ORDERS.len());

            // Create the message representing the batch to send to the Algorithm
            let batch_event = MyEvent {
                user_id: "SYSTEM_BATCHER".to_string(), // A system ID for the batch
                client_order_id: batch_id.clone(), // Use batch_id as the correlation ID for this event
                event_type: "OrderBatch".to_string(),
                payload: format!("Batch of {} orders", BATCHED_ORDERS.len()),
            };

            // Inject the context of the `order_batching_span` for the next hop (Algorithm)
            let traced_batch_message = TracedChannelMessage {
                event: batch_event,
                trace_context_carrier: inject_current_context(),
            };
            batch_tx.send(traced_batch_message).unwrap(); // Send to algorithm

            BATCHED_ORDERS.clear(); // Clear the batch
        }
    }
  }
  info!("Order Manager processor stopped.");
}
```

3.3. Algorithm: Sending to Exchange & Receiving Fills (Fan-Out & Many-to-Many)

- **Action:** From a batch, generate multiple exchange orders. For each, create a new span that is a child of the order_batching span. When fills are received, create spans for them and **link** them back to the original client order spans.
- **Span:** algorithm_processing (child of order_batching), exchange_order_send (child of algorithm_processing), exchange_fill_received (child of exchange_order_send), fill_matching (new span, linked to exchange_fill_received and potentially multiple client_order_received spans).
- **Context Propagation:** Inject context when sending to the external WebSocket client.

```
// In your Algorithm component (non-async worker thread)
use crate::tracing_channels::{TracedChannelMessage, extract_context_from_carrier,
inject_current_context, with_baggage};
use opentelemetry::trace::{SpanContext, Link};
use opentelemetry::KeyValue;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};
use tracing::{info, span, Level, Span};

// Shared state for active client order spans (same as in Order Manager)
type ActiveClientOrderSpans = Arc<Mutex<HashMap<(String, String), SpanContext>>>;

fn algorithm_processor(
   mut rx: crossbeam_channel::Receiver<TracedChannelMessage>, // From Order Manager
   exchange_ws_tx: crossbeam_channel::Sender<TracedChannelMessage>, // To external
Exchange WS client
   active_client_order_spans: ActiveClientOrderSpans, // Shared state
) {
   info!("Algorithm processor started.");
   for traced_batch_message in rx {
      // Extract and attach context from the incoming order_batching span
      let parent_otel_context =
extract_context_from_carrier(&traced_batch_message.trace_context_carrier);
      let _guard = parent_otel_context.attach();

      let algorithm_span = span!(
         Level::INFO,
         "algorithm_processing",
         batch.id = &traced_batch_message.event.client_order_id,
      );
      let _algo_guard = algorithm_span.enter();
```

```
    info!("Algorithm processing batch: {}", traced_batch_message.event.client_order_id);

    // --- Simulate generating multiple exchange orders from the batch ---
    let num_exchange_orders = 2; // Example: 2 exchange orders per batch
    for i in 0..num_exchange_orders {
        let exchange_order_id = format!("exchange_order_{}_{}",
traced_batch_message.event.client_order_id, i);
        let exchange_order_type = if i % 2 == 0 { "LimitOrder" } else { "MarketOrder" };

        let exchange_order_event = MyEvent {
            user_id: "EXCHANGE_ALGO".to_string(), // System ID for exchange orders
            client_order_id: exchange_order_id.clone(), // Use exchange_order_id as correlation
ID
            event_type: exchange_order_type.to_string(),
            payload: format!("Order for {}. Price: 100, Qty: 10", exchange_order_type),
        };

        // Create a span for sending this specific exchange order
        let exchange_send_span = span!(
            Level::INFO,
            "exchange_order_send",
            exchange.order_id = &exchange_order_id,
            exchange.type = exchange_order_type,
        );
        let _send_guard = exchange_send_span.enter();

        info!("Sending exchange order: {}", exchange_order_id);

        // Inject context of `exchange_send_span` for the WS client
        let traced_exchange_order = with_baggage(
            vec![
                KeyValue::new("exchange.order_id", exchange_order_id.clone()),
                KeyValue::new("exchange.type", exchange_order_type.to_string()),
            ],
            || TracedChannelMessage {
                event: exchange_order_event,
                trace_context_carrier: inject_current_context(),
            },
        );
        exchange_ws_tx.send(traced_exchange_order).unwrap(); // Send to external WS client
    }
```

```
    // --- Simulate receiving fills (this would come from the external WS client's receiver) ---
    // For demonstration, let's assume we get some fills back related to the batch
    let fills_for_batch: Vec<MyEvent> = vec![
        MyEvent {
            user_id: "EXCHANGE_FILL".to_string(),
            client_order_id: format!("fill_{}_1", traced_batch_message.event.client_order_id),
            event_type: "Fill".to_string(),
            payload: "Filled 5 units of client_order_123".to_string(),
        },
        MyEvent {
            user_id: "EXCHANGE_FILL".to_string(),
            client_order_id: format!("fill_{}_2", traced_batch_message.event.client_order_id),
            event_type: "Fill".to_string(),
            payload: "Filled 3 units of client_order_456".to_string(),
        },
    ];

    for fill_event in fills_for_batch {
        let fill_id = fill_event.client_order_id.clone(); // Using client_order_id for fill ID here

        // Assume fill came from a traced external WS client, so it has context.
        // If not, you'd start a new trace/span here or link to the original exchange_order_send
span.
        let fill_received_span = span!(
            Level::INFO,
            "exchange_fill_received",
            fill.id = &fill_id,
            fill.data = &fill_event.payload,
        );
        let _fill_guard = fill_received_span.enter();

        info!("Received fill: {}", fill_id);

        // --- Matching Fills to Client Orders (Many-to-Many Relationship) ---
        let fill_matching_span = span!(
            Level::INFO,
            "fill_matching",
            fill.id = &fill_id,
        );
        let _matching_guard = fill_matching_span.enter();

        // Link the `fill_matching_span` to the `exchange_fill_received` span
        Span::current().add_link(Link::new(fill_received_span.context(), vec![]));
```

```
        // --- Logic to determine which client orders this fill matches ---
        // This is where you'd use your business logic to find the original client orders.
        // For demo, let's assume it matches two specific client orders from the batch.
        let matched_client_orders: Vec<(String, String)> = vec![
            ("user_abc".to_string(), "client_order_123".to_string()),
            ("user_xyz".to_string(), "client_order_456".to_string()),
        ];

        for (user_id, client_order_id) in matched_client_orders {
            // Link the `fill_matching_span` to the *original `client_order_received` span*
            // This shows the reverse causality: this fill affected that client order.
            if let Some(client_order_sc) =
active_client_order_spans.lock().unwrap().get(&(user_id.clone(), client_order_id.clone())) {
                Span::current().add_link(Link::new(*client_order_sc, vec![]));
                info!("Linked fill {} to client order ({}, {})", fill_id, user_id, client_order_id);
            }

            // Create a span for updating the client order status
            let client_order_update_span = span!(
                Level::INFO,
                "client_order_update",
                user.id = &user_id,
                client.order_id = &client_order_id,
                status = "partially_filled", // or "fully_realized"
            );
            let _update_guard = client_order_update_span.enter();
            info!("Updating client order ({}, {}) status.", user_id, client_order_id);

            // If a client order is fully realized, you might want to mark its root span as complete.
            // This is generally handled by the component that "owns" the lifecycle of that root
span.
            // For example, the Order Manager could have a final "order_completed" span.
        }
    }
  }
  info!("Algorithm processor stopped.");
}
```

3.4. External WebSocket Client (Sending/Receiving)

- **Action:** This component acts as a proxy. It receives TracedChannelMessages, extracts context, sends to the external exchange, and when fills are received, it creates

TracedChannelMessages with context and sends them back to the Algorithm.

- **Span:** external_exchange_ws_send, external_exchange_ws_recv.

```
// In your Async WebSocket Client (sending/receiving from external exchange)
use crate::tracing_channels::{TracedChannelMessage, extract_context_from_carrier,
inject_current_context, with_baggage};
use tokio_tungstenite::{WebSocketStream, MaybeTlsStream};
use tokio::net::TcpStream;
use futures_util::{StreamExt, SinkExt};
use opentelemetry::KeyValue;
use tracing::{info, instrument, Span, Level};

#[instrument(skip(rx, ws_stream))]
async fn external_websocket_client(
    mut rx: crossbeam_channel::Receiver<TracedChannelMessage>, // From Algorithm
(exchange orders)
    fill_tx: crossbeam_channel::Sender<TracedChannelMessage>, // To Algorithm (fills)
    mut ws_stream: WebSocketStream<MaybeTlsStream<TcpStream>>,
) {
    info!("External WebSocket client started.");

    // Separate tasks for sending and receiving to avoid blocking
    let (mut write, mut read) = ws_stream.split();

    // Task for sending orders to the exchange
    let send_task = tokio::spawn(async move {
        for traced_message in rx {
            // Extract and attach context from the incoming `exchange_order_send` span
            let parent_otel_context =
extract_context_from_carrier(&traced_message.trace_context_carrier);
            let _guard = parent_otel_context.attach();

            let ws_send_span = span!(
                Level::INFO,
                "external_exchange_ws_send",
                exchange.order_id = &traced_message.event.client_order_id,
                exchange.type = &traced_message.event.event_type,
            );
            let _send_guard = ws_send_span.enter();

            info!("Sending to external exchange: {:?}", traced_message.event);
            let message =
tokio_tungstenite::tungstenite::Message::Text(traced_message.event.payload);
            if let Err(e) = write.send(message).await {
```

```
                tracing::error!("Failed to send message over external WebSocket: {:?}", e);
                break;
            }
            info!("Message sent to external exchange.");
        }
        info!("External WebSocket send task stopped.");
    });

    // Task for receiving fills from the exchange
    let recv_task = tokio::spawn(async move {
        while let Some(msg) = read.next().await {
            if let Ok(msg) = msg {
                if let tokio_tungstenite::tungstenite::Message::Text(text) = msg {
                    // This is a new incoming message (a fill) from an external service.
                    // Start a new span for receiving this fill.
                    let ws_recv_span = span!(
                        Level::INFO,
                        "external_exchange_ws_recv",
                        fill.raw_data = &text,
                    );
                    let _recv_guard = ws_recv_span.enter();

                    info!("Received from external exchange: {}", text);

                    // Parse the raw text into a MyEvent representing a fill
                    let fill_event = MyEvent {
                        user_id: "EXCHANGE_FILL_SOURCE".to_string(),
                        client_order_id: format!("fill_{}", uuid::Uuid::new_v4()), // Generate a unique ID for
the fill
                        event_type: "Fill".to_string(),
                        payload: text,
                    };

                    // Inject the context of the `ws_recv_span` for the next hop (Algorithm)
                    let traced_fill_message = with_baggage(
                        vec![
                            KeyValue::new("fill.id", fill_event.client_order_id.clone()),
                        ],
                        || TracedChannelMessage {
                            event: fill_event,
                            trace_context_carrier: inject_current_context(),
                        },
                    );
```

```
                if let Err(e) = fill_tx.send(traced_fill_message) {
                    tracing::error!("Failed to send traced fill to Algorithm: {:?}", e);
                    break;
                }
            }
        } else {
            tracing::warn!("External WebSocket receive error or close.");
            break;
        }
    }
    info!("External WebSocket receive task stopped.");
  });

  // Wait for both send and receive tasks to complete
  let _ = tokio::try_join!(send_task, recv_task);
  info!("External WebSocket client closed.");
}
```

4. Key Takeaways for this Intricate Scenario

- **Span Links are Indispensable:** For any many-to-many relationship (multiple inputs to one output, or one input affecting multiple outputs), Span Links are the correct OpenTelemetry primitive. They show causal relationships without forcing a strict parent-child hierarchy that wouldn't make sense.
- **Baggage for Business Context Propagation:** Use OpenTelemetry Baggage to carry UserID, ClientOrderID, BatchID, ExchangeOrderID, FillID etc., across your trace. This allows any component to access these business identifiers for logging, metrics, or conditional logic without modifying your core MyEvent payload.
- **Centralized SpanContext Storage (Carefully):** For linking back from fills to original client orders, you need a mechanism to store the SpanContext of the initial client_order_received spans. An Arc<Mutex<HashMap<CorrelationKey, SpanContext>>> is a common pattern, but be mindful of memory usage and contention in high-throughput scenarios. Consider using a bounded queue or a cache with eviction for this map.
- **Span Granularity:** Each significant logical step in your system (receiving order, batching, sending to exchange, receiving fill, matching, updating client order) should be its own span. This provides the detailed visibility you need.
- **Correlation IDs as Span Attributes:** Always add relevant correlation IDs (like user.id, client.order_id, batch.id, exchange.order_id, fill.id) as attributes to *every relevant span*. This makes filtering and searching in Kibana incredibly powerful and allows you to quickly find all telemetry related to a specific client order or batch.
- **Context::attach() at Every Boundary:** The Context::attach() call is critical *every time*

you receive a TracedChannelMessage and are about to do work related to that message. It ensures that subsequent spans created on that thread are children of the span that emitted the message.

- **#[instrument] vs. Manual Spans:** Use #[instrument] for functions where the entire function's execution is a logical unit. Use span! and _guard.enter() for finer-grained control over specific blocks of code within a function, or for dynamic span creation (like iterating over a batch).

This comprehensive approach will provide an **intricate** and **meticulous** view of your trading system's operations, allowing you to trace the lifecycle of an order end-to-end, understand bottlenecks, and debug complex interactions across your hybrid architecture.