JAEGER – AN OPENSOURCE DISTRIBUTED TRACING SYSTEM

- Jegan Balakrishnan

Overview

Jaeger is an open-source distributed tracing system developed by Uber Technologies. It is designed to monitor and troubleshoot the performance of microservices-based architectures. Jaeger helps in tracking request flows across services, measuring latency, and identifying performance bottlenecks.

Key Features

- **Distributed Context Propagation**: Tracks the complete request lifecycle across services.
- **Performance Optimization**: Identifies slow services or endpoints.
- **Root Cause Analysis**: Helps in debugging errors and latency issues.
- **Service Dependency Analysis**: Visualizes dependencies between services.
- **Monitoring and Alerting**: Supports integration with monitoring tools for real-time alerts.

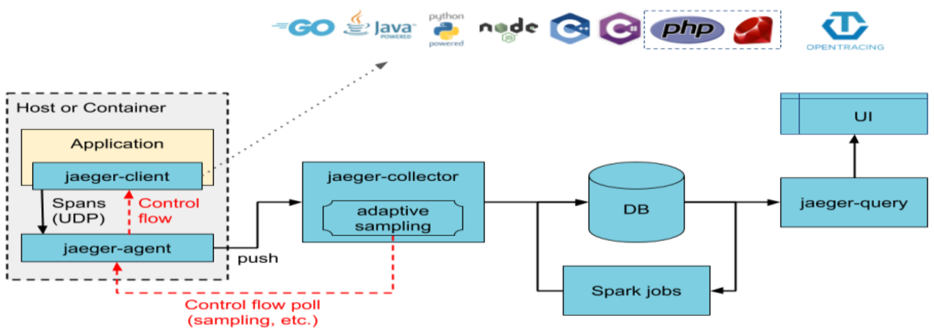
Use Cases

- Monitoring and troubleshooting microservices-based applications.
- Analyzing service dependencies.
- Optimizing system performance by identifying bottlenecks.
- Understanding request flows and pinpointing failure points.

Jaeger Architecture

Jaeger comprises the following core components:

1. **Agent**: A lightweight daemon that collects trace data from applications and forwards it to the Collector.
2. **Collector**: Aggregates trace data received from agents and stores it in a database.
3. **Query Service**: Provides a UI or API to query and visualize traces.
4. **Storage Backend**: Stores trace data, supports options like Elasticsearch, Cassandra, Kafka, etc.
5. **UI**: A web-based interface for visualizing trace data.



Architecture of Jaeger

Spans and Traces:

1. Spans

A **span** represents a single unit of work in a distributed system. It is the building block of a trace and typically corresponds to an operation, such as:

- A database query
- An HTTP request
- A function execution

Key Components of a Span:

- **Operation Name**: Describes the task performed (e.g., `GET /api/users`).
- **Start Time and Duration**: Indicates when the span started and how long it took to complete.
- **Span Context**: Contains metadata, including a unique `span_id`.
- **Tags**: Key-value pairs providing additional metadata (e.g., `http.status_code=200`).
- **Logs**: Time-stamped events or errors associated with the span (e.g., `error=true`).
- **Parent Span ID**: Identifies the parent span if the span is part of a larger trace.

2. Traces

A **trace** is a collection of spans that represents the entire journey of a request across the distributed system. It provides a holistic view of how a request propagates through various services and components.

Key Characteristics of a Trace:

- **Trace ID**: A unique identifier for the entire trace.
- **Structure**: A trace is a directed acyclic graph (DAG) of spans, often visualized as a tree. The root span represents the entry point of the request, and child spans represent subsequent operations.
- **End-to-End Visibility**: Captures the flow of the request across services, allowing for latency analysis and troubleshooting.

Real World Applications:

1. E-commerce Platforms

- **Problem**: Identifying latency issues in high-traffic applications.
- **Use Case**: Track user requests across multiple services such as product search, cart management, and checkout.
- **Benefit**: Quickly identify bottlenecks like slow database queries or overloaded APIs, ensuring a seamless shopping experience.

---

2. Financial Services

- **Problem**: Ensuring low-latency transactions and debugging complex workflows.
- **Use Case**: Trace transactions across services like fraud detection, payment processing, and ledger updates.
- **Benefit**: Improve reliability and maintain compliance with SLAs (Service Level Agreements).

---

3. Media Streaming Services

- **Problem**: Delivering content with minimal buffering or delays.
- **Use Case**: Monitor request flows through services responsible for content discovery, transcoding, and streaming delivery.
- **Benefit**: Optimize performance and minimize service interruptions for users.

---

4. Healthcare Applications

- **Problem**: Monitoring critical health data pipelines.
- **Use Case**: Trace requests in systems that handle appointment scheduling, patient data retrieval, and medical record storage.
- **Benefit**: Ensure accuracy, efficiency, and reliability of critical operations in compliance with regulations like HIPAA.

---

5. IoT Applications

- **Problem**: Debugging and monitoring high-volume data pipelines from IoT devices.
- **Use Case**: Trace telemetry data flows from edge devices to processing services and storage systems.
- **Benefit**: Detect and resolve performance bottlenecks in near-real-time.

Installation and Setup

Jaeger can be deployed using multiple methods, including Docker, Kubernetes, or binary installation. Below are the steps for setting up Jaeger using Docker Compose.

Prerequisites

- Docker and Docker Compose installed on your system.

Steps

1. **Create a `docker-compose.yml` File**
2. `version: '3.7'`
3. `services:`
4. `  jaeger:`
5. `    image: jaegertracing/all-in-one:latest`
6. `    ports:`
7. `      - "5775:5775/udp"`
8. `      - "6831:6831/udp"`
9. `      - "6832:6832/udp"`
10. `      - "5778:5778"`
11. `      - "16686:16686"`
12. `      - "14268:14268"`
13. `      - "14250:14250"`
14. `      - "9411:9411"`
15. **Start the Jaeger All-in-One Container**
16. `docker-compose up -d`
17. **Verify the Installation**
    - Open your browser and navigate to `http://localhost:16686`.
    - You should see the Jaeger UI for querying and visualizing traces.

Integrating Jaeger with Applications

Instrumenting Code

Jaeger supports multiple programming languages. Below is an example for Python using the `opentelemetry` library.

Steps:

1. **Install OpenTelemetry Libraries**
2. `pip install opentelemetry-api opentelemetry-sdk opentelemetry-exporter-jaeger`
3. **Configure Jaeger Exporter**
4. `from opentelemetry import trace`
5. `from opentelemetry.exporter.jaeger.thrift import JaegerExporter`
6. `from opentelemetry.sdk.trace import TracerProvider`
7. `from opentelemetry.sdk.trace.export import BatchSpanProcessor`
8.
9. `# Set up a Tracer Provider`
10. `trace.set_tracer_provider(TracerProvider())`

```

# Configure Jaeger Exporter
jaeger_exporter = JaegerExporter(
    agent_host_name='localhost',
    agent_port=6831,
)

# Add the Jaeger exporter to the tracer provider
trace.get_tracer_provider().add_span_processor(
    BatchSpanProcessor(jaeger_exporter)
)

tracer = trace.get_tracer(__name__)

# Create spans
with tracer.start_as_current_span("example-span"):
    print("Tracing with Jaeger")
```

28. **Run the Application**

    Execute the script and verify the trace appears in the Jaeger UI.

Advanced Configuration

Storage Options

Jaeger supports various storage backends:

- **Elasticsearch**: Ideal for large-scale deployments.
- **Cassandra**: Suitable for high-write workloads.
- **Kafka**: Used for buffering and streaming trace data.

Update your deployment configuration to use these storage backends.

Sampling Strategies

Sampling determines which traces are captured and stored. Jaeger supports:

- **Probabilistic Sampling**: Captures a percentage of requests.
- **Rate Limiting**: Limits the number of traces per second.
- **Adaptive Sampling**: Dynamically adjusts sampling rates.

Monitoring and Maintenance

- Use Prometheus or Grafana to monitor Jaeger metrics.
- Regularly clean up old trace data based on storage capacity.
- Optimize sampling strategies to balance performance and resource usage.

Troubleshooting

Common Issues

1. **No Traces in UI**: Verify the application is correctly instrumented and the agent is reachable.
2. **High Latency**: Optimize the storage backend and sampling rates.
3. **Collector Errors**: Check logs for configuration issues or storage backend errors.

Conclusion

Jaeger is a powerful tool for distributed tracing and performance monitoring in microservices-based architectures. With its robust features and extensible ecosystem, it is an essential component for modern application observability.