Collecting telemetry data from low latency microservices

Eya-Tom Augustin SANGAM

Dorsal Lab, Polytechnique Montréal
Agenda

- About me and DORSAL
- Context, goals and considerations
- Related work
- Proposed solution
- Benchmarks
- Future work
- Conclusion
About me and
DORSAL
About me

- Masters student in DORSAL lab under Prof. Michel Dagenais supervision
  - DORSAL stands for Distributed Open Reliable Systems Analysis Lab
- Located in the Computer and Software Engineering Department at Polytechnique Montreal in Canada
About DORSAL

- Research in collaboration with Ericsson, Ciena, AMD, EfficiOS and others about:
  - Monitoring and Debugging of High Performance Distributed Heterogeneous Systems
  - Dynamic instrumentation (uftrace, LTTng, libpatch), hardware tracing
  - GPU tracing, profiling and debugging (ROCm, ROCgdb and Theia TraceCompass)
  - Runtime verification (lower overhead alternatives to ASan and TSan)
  - Scalable trace analysis and visualisation (parallel Theia Trace compass extension)
  - Trace analysis with Machine Learning (Trace Compass)
Context, goals and considerations
Context and goals

- We have C microservices communicating with each other using ZeroMQ
- We want to **collect telemetry data (TD)**:
  - Host metrics (CPU usage, RAM usage, ...)
  - Application logs
  - Application metrics (queue size, request duration, ...)
  - Distributed requests (aka tracing spans)
Considerations

- We want to do cross-hosts TD analysis
  - We need to bring some TD together at some point
- Some hosts have limited hard drive storage
  - A filtering mechanism is required to minimize the amount of data saved on the disk
  - e.g., we should be able to decide at runtime whether we want to save heartbeat traces or not
Considerations

- Some applications run on hosts with limited resources
  - Installing any agent or observability backend may highly affect the application behaviour
- Live monitoring is desired but not required.
Related work
Related work: LTTng/LTTng-UST

- Open source tracing framework for Linux
- LTTng is well suited for tracing low latency programs
Related work: LTTng and host metrics

- LTTng can help collect host metrics (CPU, RAM, Network usage, ...)
- We capture only necessary events
  - e.g. sched_switch to be able to compute CPU usage
- Big advantage of LTTng and LTTng-UST: We can modify recording rules at runtime
Related work: LTTng-UST and spans

- To collect spans we need to log a message at beginning and end of an operation

```c
1 void operation() {
2    lttng_ust_tracepoint(provider, "span_start", ... )
3    // do the operation
4    lttng_ust_tracepoint(provider, "end_start", ... )
5 }
```

- Problem: We need to agree on how trace ids are generated, how the trace contexts are propagated to other microservices, ....
To collect metrics we can log all metrics variations to LTTng.

During analysis phase, we can aggregate all those variations across all the hosts.

Problem: We need to add more logic to support:

- Synchronous counters: counters invoked inline with application/business processing logic
- Asynchronous counters: counters modified on demand (e.g. every 30s)
- Histograms
- Standardize schemas for the data collected
Related work: LTTng/LTTng-UST verdict

- LTTng and LTTng-UST are a good start point, but they do not solve all our problems
- We need to define a protocol over the standard LTTng-UST, to help us collect, aggregate and structure the data we collect
  - Here comes the OpenTelemetry specification
Related work: OpenTelemetry

- OpenTelemetry (OTel) is becoming the industry standard for creating and collecting TD.
- OTel specification describes cross-language requirements and expectations for all OTel implementations.
  - It defines how and what TD should be collected, processed and sent.
  - Standardizes TD schemas.
  - Gives a reference implementation in most common languages (C++, Java, C#, Python ...).
Related work: OpenTelemetry

- Many telemetry backend/visualisation tools like Jaeger or Prometheus support OTel data schemas out of the box
- OTel created the OTel Collector which is a vendor-agnostic way to receive, process and export TD
Related work: OpenTelemetry

- Protobuf definition of a Span:

```protobuf
define Span {
  bytes trace_id = 1;
  bytes span_id = 2;
  string trace_state = 3;
  bytes parent_span_id = 4;
  string name = 5;
}
define SpanKind {
  SPAN_KIND_UNSPECIFIED = 0;
  SPAN_KIND_INTERNAL = 1;
  SPAN_KIND_SERVER = 2;
  SPAN_KIND_CLIENT = 3;
  SPAN_KIND_PRODUCER = 4;
  SPAN_KIND_CONSUMER = 5;
} define Span {
  SpanKind kind = 6;
  fixed64 start_time_unix_nano = 7;
  fixed64 end_time_unix_nano = 8;
  repeated opentelemetry.proto.common.v1.KeyValue attributes = 9;
  uint32 dropped_attributes_count = 10;
}
```

Span protobuf definition from
https://github.com/open-telemetry/opentelemetry-proto/blob/d1468b7700309cec0a3fdfffbfba4e84acfb4072/opentelemetry/proto/trace/v1/trace.proto
Combining LTTng and OpenTelemetry
Different ways of collecting TD and moving them around
Using proprietary instrumentation
Using Opentelemetry instrumentation

Collecting telemetry data from low latency microservices - Eya-Tom Augustin SANGAM, DORSAL Lab
Combining LTTng and OpenTelemetry

- OTLP = OpenTelemetry Protocol (OTLP)
- OTLP describes the encoding, transport, and delivery mechanism of telemetry data between telemetry sources, intermediate nodes such as collectors and telemetry backends.
- Data are protobufs
Combining LTTng and OpenTelemetry

- OTLP = OpenTelemetry Protocol (OTLP)
- OTLP describes the encoding, transport, and delivery mechanism of telemetry data between telemetry sources, intermediate nodes such as collectors and telemetry backends.
- Data are protobufs
Proposed solution
Proposed solution

- Combine both OpenTelemetry and LTTng
- Two phases
  - Online phase: TD collection, when application runs
  - Offline phase: Analysis, later on
**Proposed solution**

**Online phase**

(When application runs)

- LTTng is used to collect host metrics
- We use OTel to instrument the application
- TD generated (Protobufs binary data) is logged to LTTng-UST and saved in CTF (Common Trace Format) files
  - We can control what runtime data we save this way
Proposed solution

Offline phase

(Only when we want to do analysis)

- CTF files are copied from the host
- Host metrics could be viewed in Trace Compass directly
- The OTel Replay System reads TD and sends them to the OTel collector which will send them later to observability backends (Jaeger, Prometheus, etc.)
Source code

- Otel C wrapper
  - Wrapper around the official C++ OpenTelemetry client
  - Code: [https://github.com/dorsal-lab/opentelemetry-c](https://github.com/dorsal-lab/opentelemetry-c)
Source code

- Simple ZeroMQ client, proxy and server application traced using opentelemetry-c

Code: https://github.com/dorsal-lab/opentelemetry-c-demo
Source code

- OTel Replay System which reads the telemetry data and sends them to the OTel collector which will send them later to observability backends (Jaeger, Prometheus, etc.)
  - Code: [https://github.com/dorsal-lab/opentelemetry-c-replayer](https://github.com/dorsal-lab/opentelemetry-c-replayer)
Source code

- Benchmarks
  - Deep dive doc
Benchmarks
Trace benchmarks

- Scenario: Start a span and end it right away. Measure the time to do the operation.
- Multiple configurations tested:
  - LTTng configuration: No LTTng session running, LTTng session without recording, LTTng session recording UST telemetry data, LTTng remote session recording UST telemetry data
  - Type of instrumentation: No instrumentation, OpenTelemetry
  - Type of exporter: LTTng Exporter, Local OTel collector, Remote OTel collector
  - OTel Traces Processor (applies only for traces benchmarks): Simple, Batching processor
Trace benchmarks: Simple Span Processor

Exporting spans one by one as they are created using remote OTel collector VS using Local Lttng exporter VS Exporting one by one to remote LTTng

<table>
<thead>
<tr>
<th></th>
<th>Remote OTel collector</th>
<th>Local LTTng session</th>
<th>Remote LTTng session</th>
</tr>
</thead>
<tbody>
<tr>
<td>n spans</td>
<td>500</td>
<td>20,000</td>
<td>20,000</td>
</tr>
<tr>
<td>min (ns)</td>
<td>1,931,562</td>
<td>94,947</td>
<td>61,689</td>
</tr>
<tr>
<td>mean (ns)</td>
<td>2,945,936</td>
<td>288,689</td>
<td>287,596</td>
</tr>
<tr>
<td>max (ns)</td>
<td>15,251,23</td>
<td>957,472</td>
<td>1,512,586</td>
</tr>
<tr>
<td>median (ns)</td>
<td>2,796,951</td>
<td>305,975</td>
<td>283,274</td>
</tr>
<tr>
<td>std (ns)</td>
<td>478,621</td>
<td>22,681</td>
<td>23,003</td>
</tr>
<tr>
<td>real (ms)</td>
<td>65,391</td>
<td>208,483</td>
<td>208,473</td>
</tr>
<tr>
<td>user (ms)</td>
<td>8,079</td>
<td>6,029</td>
<td>5,969</td>
</tr>
<tr>
<td>sys (ms)</td>
<td>369</td>
<td>407</td>
<td>461</td>
</tr>
</tbody>
</table>

- When using simple processor, spans are processed synchronously after their creation.
- In this situation, using LTTng to log spans should be preferred over sending traces over the network.
Trace benchmarks: Batching Processor

Same comparison but we export traces every 5s in batches of a maximum of 512 spans in a background thread.

<table>
<thead>
<tr>
<th></th>
<th>Remote OTel collector</th>
<th>Local LTTng session</th>
<th>Remote LTTng session</th>
</tr>
</thead>
<tbody>
<tr>
<td>n spans</td>
<td>20,000</td>
<td>20,000</td>
<td>20,000</td>
</tr>
<tr>
<td>min (ns)</td>
<td>21,101</td>
<td>23,063</td>
<td>43,641</td>
</tr>
<tr>
<td>mean (ns)</td>
<td>116,657</td>
<td>117,143</td>
<td>116,836</td>
</tr>
<tr>
<td>max (ns)</td>
<td>455,129</td>
<td>536,921</td>
<td>396,297</td>
</tr>
<tr>
<td>median (ns)</td>
<td>117,134</td>
<td>113,691</td>
<td>131,189</td>
</tr>
<tr>
<td>std (ns)</td>
<td>9,668</td>
<td>9,394</td>
<td>9,482</td>
</tr>
<tr>
<td>real (ms)</td>
<td>204,911</td>
<td>205,077</td>
<td>205,048</td>
</tr>
<tr>
<td>user (ms)</td>
<td>3,663</td>
<td>3,259</td>
<td>3,268</td>
</tr>
<tr>
<td>sys (ms)</td>
<td>330</td>
<td>405</td>
<td>379</td>
</tr>
</tbody>
</table>

- Using LTTng reduce the overall CPU time used
- In production, the remote collector could be in a different network, which could make these results vary
- The preferred solution should be logging all traces locally to LTTng. This avoids running an OTel collector and dealing with all the network communications troubles it could add
Trace benchmarks: Flooding OTel ring buffers

- Pattern: export spans back to back for 1 minute without sleeping
- We use the Batching Span processor and export at most 512 spans per batch
- OTel ring buffers accepts up to 2048 spans. Pass that limit, old spans are overwritten
- Table format: Number of spans successfully exported / number of spans created

<table>
<thead>
<tr>
<th></th>
<th>Exporting to remote OTel collector</th>
<th>Exporting to local LTTng session</th>
<th>Exporting to remote LTTng session</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple Span Processor</td>
<td>29,275 / 29,275</td>
<td>1,569,145 / 1,569,145</td>
<td>1,546,970 / 1,546,970</td>
</tr>
<tr>
<td>Batching Span Processor</td>
<td>222,219 / 7,418,929 (97% loss rate)</td>
<td>2,863,534 / 7,829,110 (64% loss rate)</td>
<td>2,757,642 / 7,551,265 (64% loss rate)</td>
</tr>
</tbody>
</table>
Metrics benchmarks

- Pattern: We measure the time to do an operation without collecting any kind of metrics. And we repeat the same operation while exporting metrics every 500/1000 ms

- Comparison: No instrumentation VS exporting metrics to a remote Otel collector VS exporting metrics to a local LTTng session VS exporting metrics to a remote LTTng session
### Metrics benchmarks

<table>
<thead>
<tr>
<th></th>
<th>No instrumentation</th>
<th>Exporting to remote OTel collector</th>
<th>Exporting to local LTTng session</th>
<th>Exporting to remote LTTng session</th>
</tr>
</thead>
<tbody>
<tr>
<td>Export delay (ms)</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>duration (ms)</td>
<td>114,541</td>
<td>115,290</td>
<td>114,712</td>
<td>114,649</td>
</tr>
<tr>
<td>overhead (%)</td>
<td>-</td>
<td>0.654</td>
<td>0.149</td>
<td>0.094</td>
</tr>
<tr>
<td>cpu time (ms)</td>
<td>114,537</td>
<td>115,816</td>
<td>114,836</td>
<td>114,776</td>
</tr>
<tr>
<td>cpu time overhead (%)</td>
<td>-</td>
<td>1.116</td>
<td>0.261</td>
<td>0.208</td>
</tr>
</tbody>
</table>

- For all configurations, the execution time overhead is less than 1.2% and the larger the export interval, the lower the overhead.
- LTTng Metrics exporter is approximatively 50% faster than the remote exporter but the CPU time spent in user space is similar for the two configurations.
Future work
Future work

- Analyse OTel userspace traces directly in Tracecompass without having to use any telemetry backend
  - Add new Spans Life Analysis: Support OTel schemas, trace synchronisation and add filtering capabilities
  - Add a Metrics View: Add counters view and support basic query language (e.g. metric1 + metric2)
Conclusion
Conclusion

- We proposed a strategy of collecting telemetry data from low latency microservices
- We benefit both from OpenTelemetry specification standards and LTTng speed and filtering capabilities
- Total overhead of our solution is lower than another one using OpenTelemetry for both collecting and exporting telemetry data
Thanks !

Questions ?