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

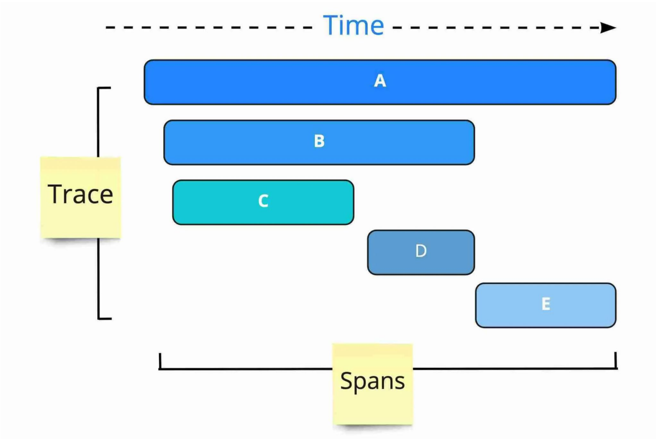
About DORSAL

- 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)



Example of span

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

```
operation() {
    lttng_ust_tracepoint(provider, "span_start", ... )
    // do the operation
    lttng_ust_tracepoint(provider, "end_start", ... )
}
```

- Problem : We need to agree on how trace ids are generated, how the trace contexts are propagated to other microservices, ….

Related work: LTTng-UST and app metrics

- 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:

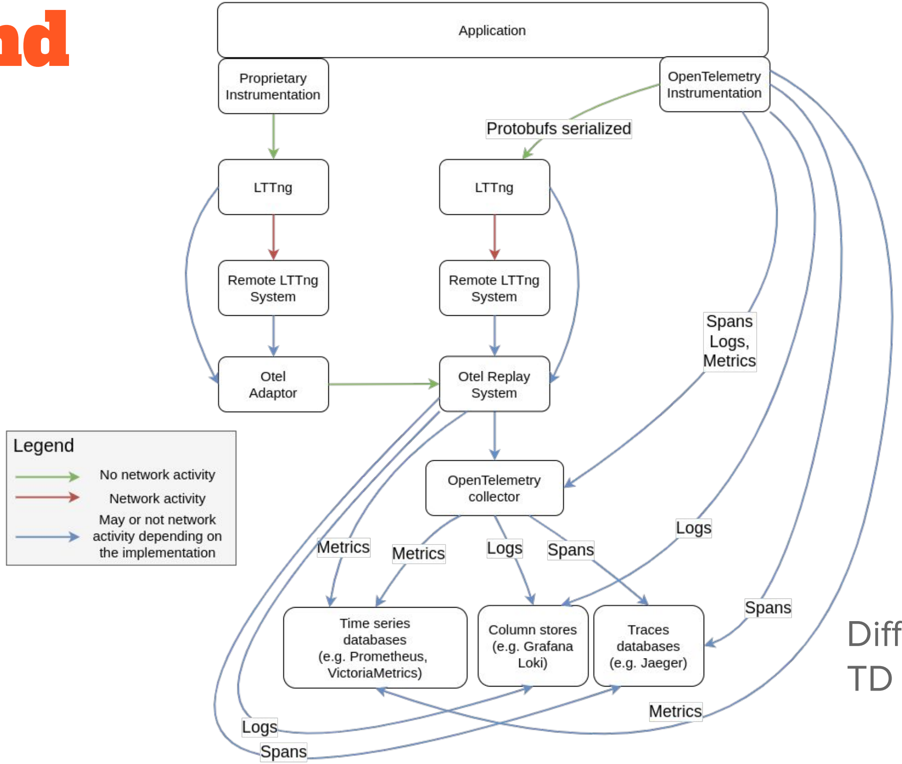
```
message Span {
  bytes trace_id = 1;
  bytes span_id = 2;
  string trace_state = 3;
  bytes parent_span_id = 4;
  string name = 5;
  enum SpanKind {
    SPAN_KIND_UNSPECIFIED = 0;
    SPAN_KIND_INTERNAL = 1;
    SPAN_KIND_SERVER = 2;
    SPAN_KIND_CLIENT = 3;
    SPAN_KIND_PRODUCER = 4;
    SPAN_KIND_CONSUMER = 5;
  }
  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;
```

```
  message Link {
    bytes trace_id = 1;
    bytes span_id = 2;
    string trace_state = 3;
    repeated opentelemetry.proto.common.v1.KeyValue attributes = 4;
    uint32 dropped_attributes_count = 5;
  }
  repeated Link links = 13;
  uint32 dropped_links_count = 14;
  Status status = 15;
}
```

Span protobuf definition from
https://github.com/open-telemetry/opentelemetry-proto/blob/d1468b7700309cec0a3fdfffbfba4e84acf94072/opentelemetry/proto/trace/v1/trace.proto

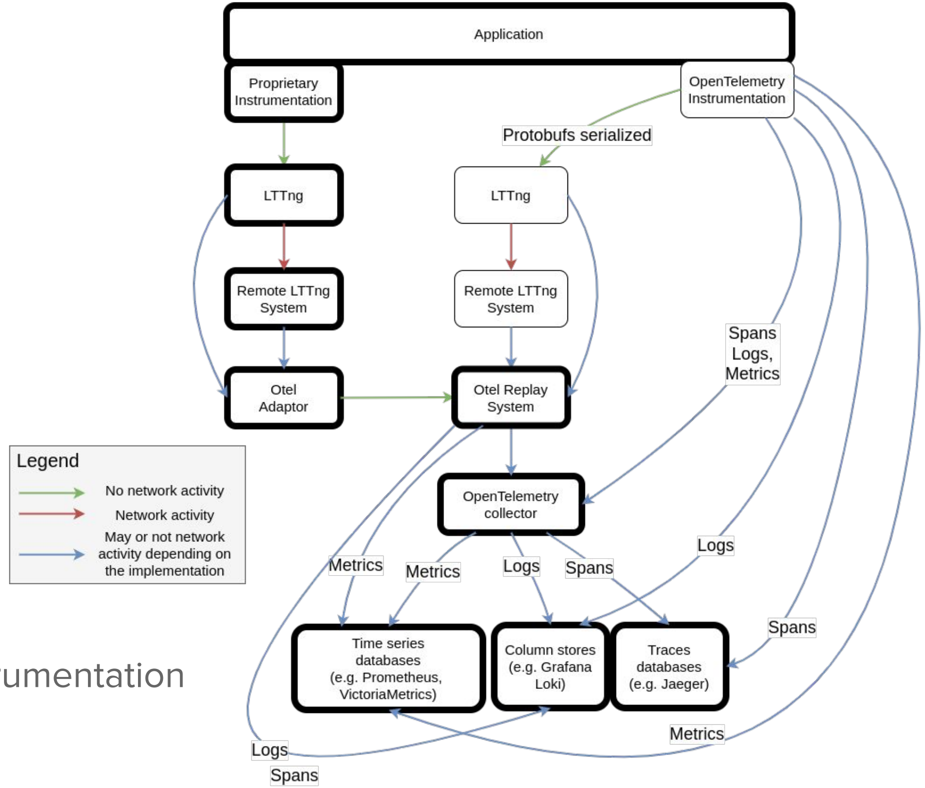
Combining LTTng and OpenTelemetry

Different ways of collecting TD and moving them around



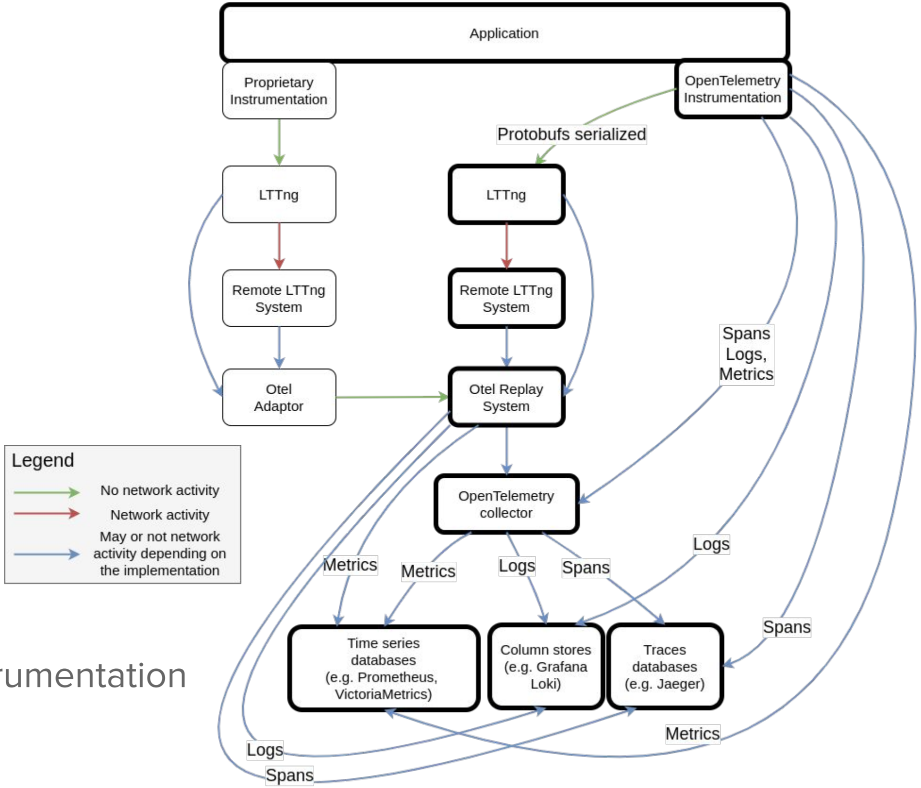
Different ways of collecting TD and moving them around

Using proprietary instrumentation



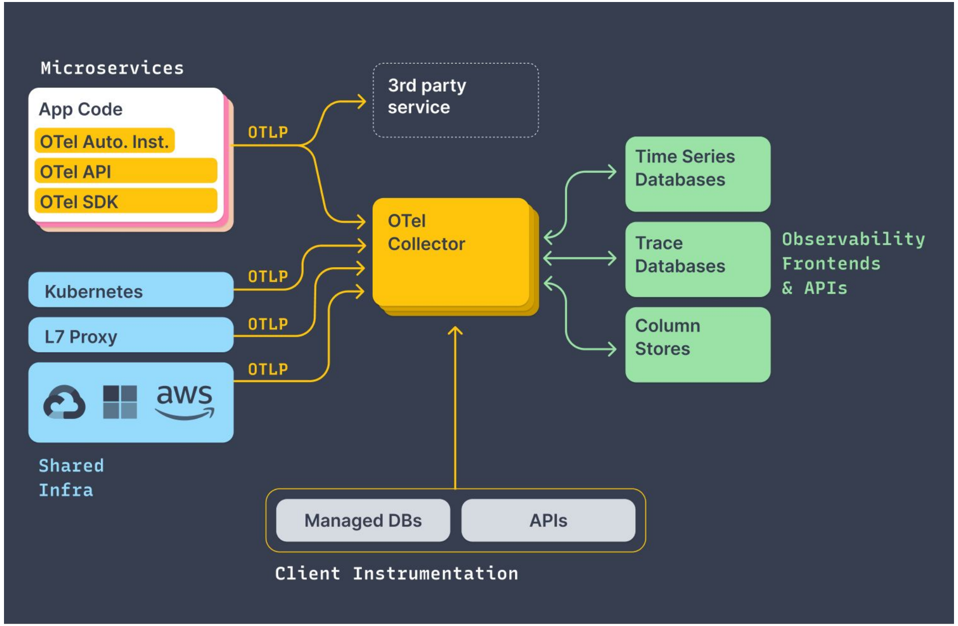
Collecting TD using proprietary instrumentation

Using Opentelemetry instrumentation



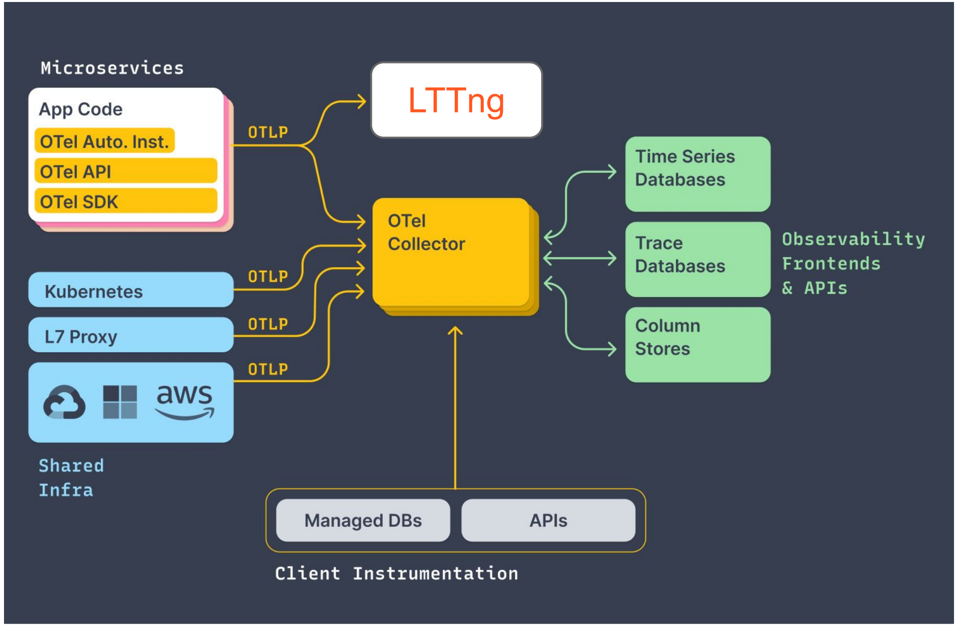
Collecting TD using Opentelemetry instrumentation

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

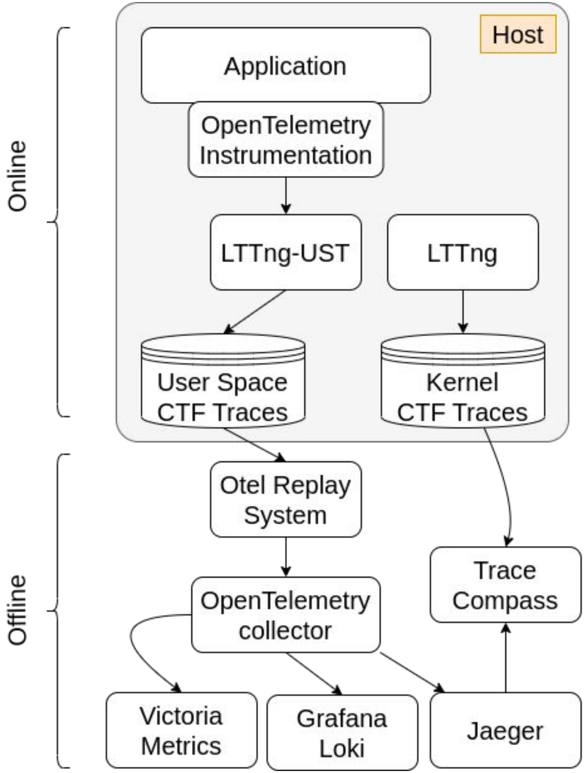


- OTLP = OpenTelemetry Protocol (OTLP)
- OTLP describes the encoding, transport, and delivery mechanism of telemetry data between telemetry sources, intermediate nodes such as collectors and telemetry backends.
- Data are protobufs

Proposed solution

Proposed solution

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

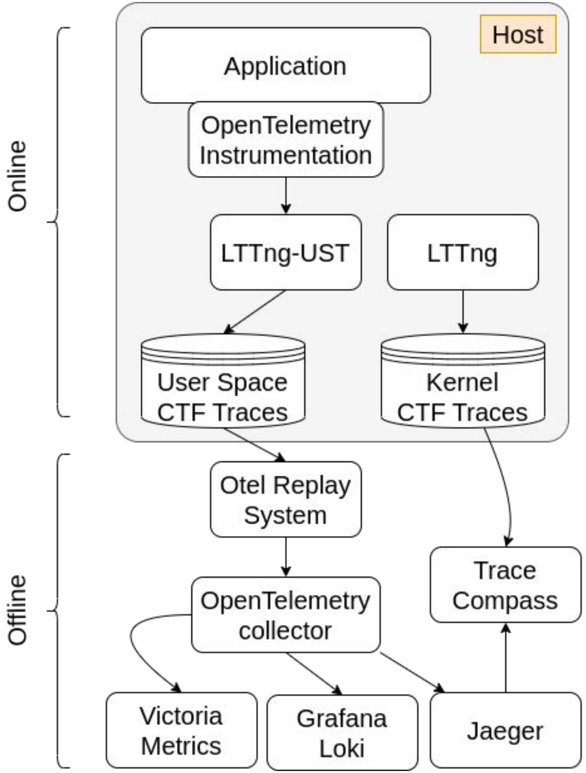


Proposed solution

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

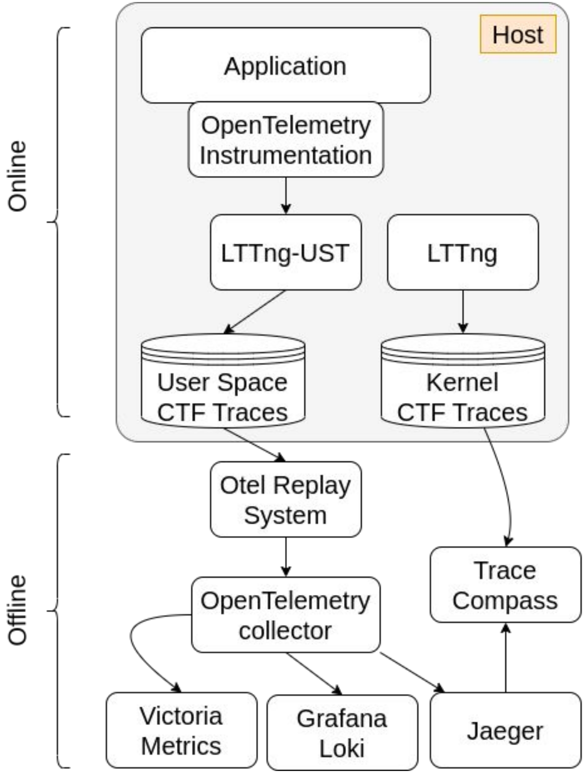


Proposed solution

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.)



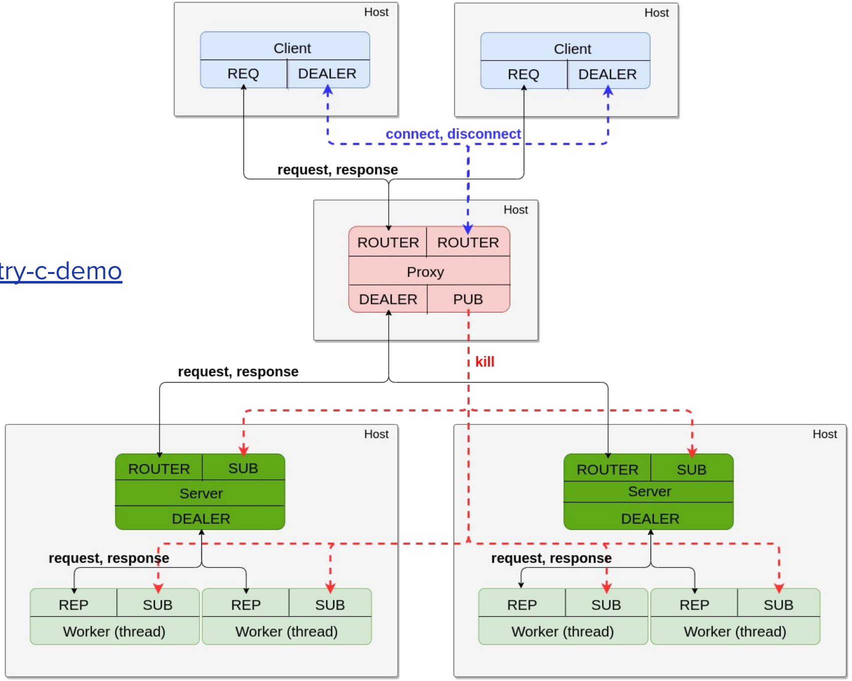
Source code

- Otel C wrapper
  - Wrapper around the official C++ OpenTelemetry client
  - Code: https://github.com/dorsal-lab/opentelemetry-c

Source code

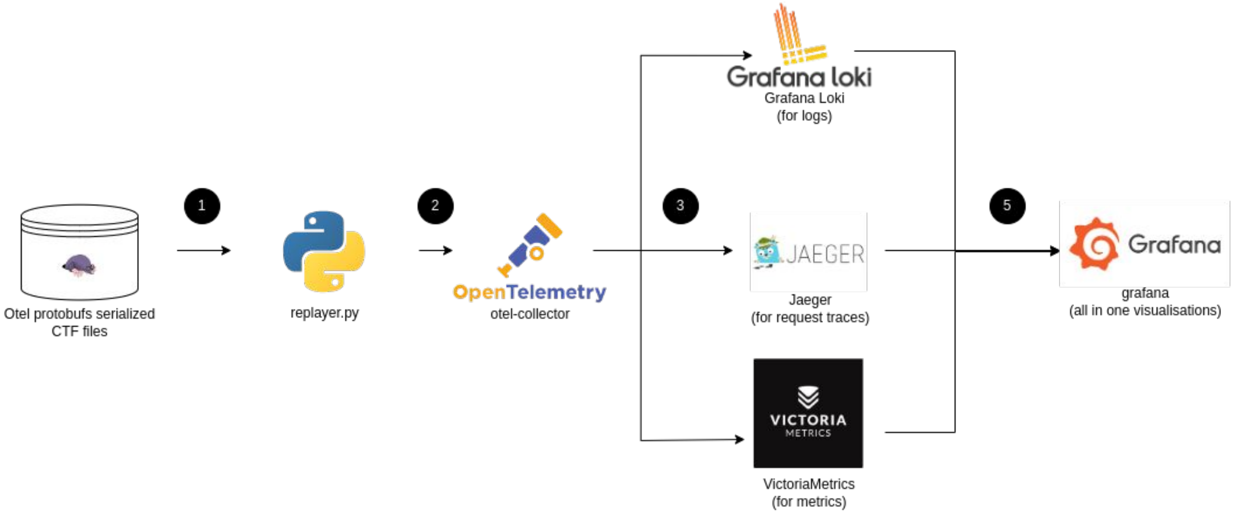
- Simple ZeroMQ client, proxy and server application traced using opentelemetry-c

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



Source code

- Benchmarks
  - https://github.com/dorsal-lab/opentelemetry-c-performance
  - 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

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

- 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.

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

- 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

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

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

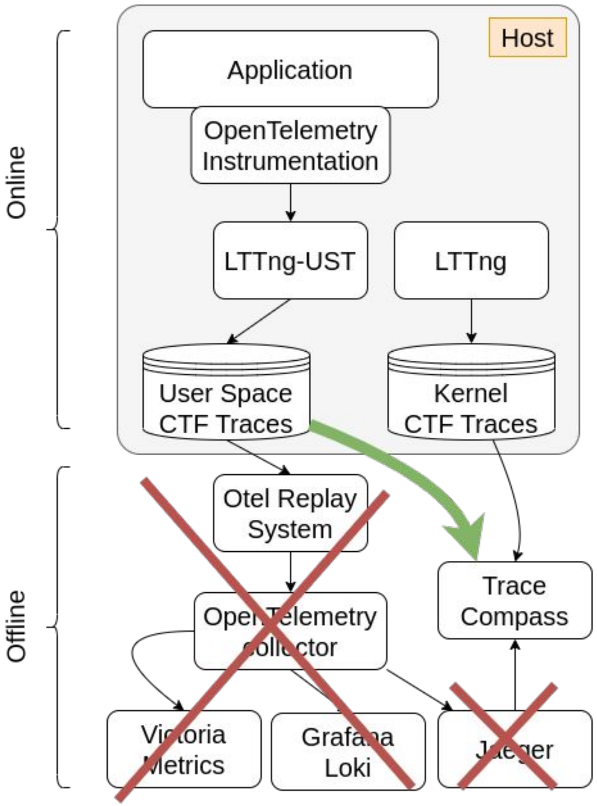
	No instrumentation		Exporting to remote OTel collector		Exporting to local LTTng session		Exporting to remote LTTng session	
Export delay (ms)	500	1000	500	1000	500	1000	500	1000
duration (ms)	114,541	114,539	115,290	115,030	114,712	114,681	114,649	114,572
overhead (%)	-	-	0.654	0.656	0.149	0.151	0.094	0.096
cpu time (ms)	114,537	114,535	115,816	115,348	114,836	114,749	114,776	114,650
cpu time overhead (%)	-	-	1.116	0.71	0.261	0.187	0.208	0.1

- 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 ?