NOTE: this document is now outdated, the persistent buffering has been implemented in OpenTelemetry Collector

OpenTelemetry Collector WAL Design Doc [Draft]

Issue: https://github.com/open-telemetry/opentelemetry-collector/issues/2285

Related work: https://github.com/open-telemetry/opentelemetry-collector/pull/3017/files (Prometheus exporter WAL)

Requirements:

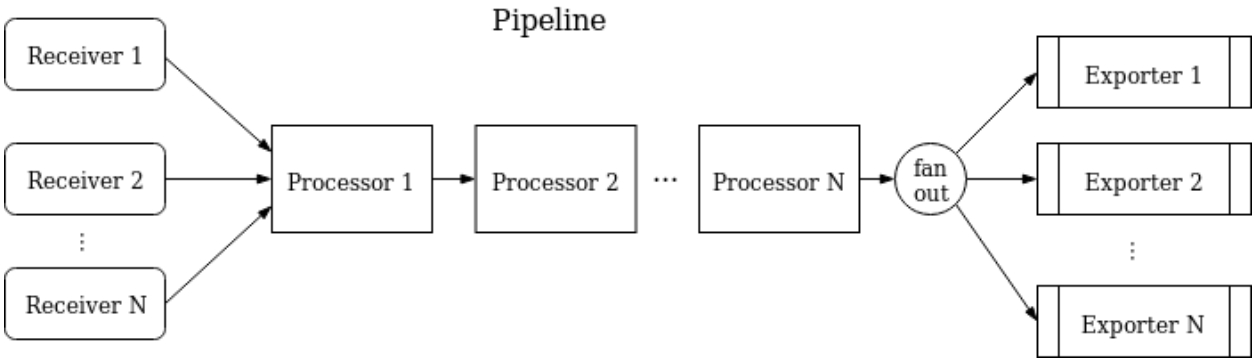
- the records are persisted to disk so in case the export fails and memory queue hits the limit or collector is killed with a non-empty queue, data is not being lost
- when a record was successfully exported, it is removed from the WAL (or marked as such via a tombstone)
- when OTC restarts, the existing records are loaded and added to the respectful send queues (replayed)
- there is a maximum allowed disk usage set; which is never exceeded (also see below)
- when multiple exporters are configured, each record should be successfully sent only once for each of them
- all signal types supported

Nice to haves:

- when maximum size is exceeded, the old records are overwritten with new ones or no new records are accepted, depending on the configuration variable (similar to FluentD's overflow_action)
- the same like above, but also when less than X% of disk space is available (and/or less than X MB)
- compress WAL data on disk

WAL in the context of pipeline

OpenTelemetry Collector architecture is built around the concept of pipelines, where a number of receivers writes to a sequence of processors, which ultimately fans out data to a number of exporters. Multiple pipelines can be defined for each of the signal (trace, metrics, logs). They can share components (the same configuration will be used, though via a separate instance).



The processors always operate on the internal data model, while Receiver and Exporters convert from/to accordingly.
It is worth observing that there are several processors which are recommended to be used in all pipelines (especially when collector is run in gateway mode). This includes the batch processor (which groups records together into larger batches) and memorylimiter (which stops accepting data after too much memory is being used).
Additionally, exporters commonly use exporterhelper, which provides queued retry capability among others. This allows to retry sending a given batch, if it failed due to server error or timeout. The current implementation stores the batches in memory.
More recently, a filestorage extension was added into OpenTelemetry Collector Contrib. It is a generic extension built on top of *bbolt*, which allows to store/load blobs using the local storage.

Alternatives for which component to include the WAL capability

Let's consider components in which WAL capability could be located:

1. **Receiver** - this could be achieved via e.g. some sort of helper, perhaps after the conversion to native format, though it's unclear when buffering capability would be preferred in this component, as it's most disconnected from batching and receiving response on the status of the export.
2. **Processor** - a separate "WAL" processor could provide the buffering for each signal, accordingly. That would provide a clear separation, however it requires retrieving back information from exporters if the operation was successful (currently, the processors are following a "fire and forget" approach). Changing that would require more significant effort.
   Additionally, while it would limit the number of copies in each WAL when multiple exporters are configured (as given record would be kept only once), the associated information on which exporters failed and which succeeded would also need to be kept, which may add to the complexity
3. **Exporter** - each exporter could handle WAL separately. This could be provided by extending `queued_retry` helper. It simplifies the design considerably, as queued_retry already knows the outcome of sending a given batch and has a memory-backed queue. Conceptually, the queue would be backed by WAL rather than memory in such case. Also, WAL can be configured more fine-granularly.
   Additionally, the exporter internal API could be refactored or extended in a way that

provides a method that deals with marshaled data already, so no additional serialization/deserialization would be required (however ToOtlpProtoBytes() has very low latency).

Alternatives for disk storage implementations

1. filestorage extension could be leveraged for WAL purposes. Each batch would need to be serialized and stored using a unique key, which might be based on timestamp (or UUID).
2. An existing library for WAL purposes could be leveraged. This is e.g. done for prometheusremotewrite WAL proposal, where tidwall/wal library is being leveraged.
3. Leverage diskqueue (as noted by David Ashpole)
4. Something else (e.g. SQLite?)

Other queues

Some processors, such as batchprocessor or groupbytraceprocessor are using currently memory-backed queues. In the event of the collector process getting killed, the data present in those queues would be lost. Hence, if the solution is going to provide WAL in exporter, there should be a possibility to persist those queues as well. A common internal library could be leveraged for persisting queue here and for the exporters.

Various concerns

Handling Prometheus data

Discussion at #3017 and wg-prometheus#9
Prometheus data have specific requirements on the order of data which limit the options on solving WAL problem heer (hence the separate PR). However, it should be possible to use a generic solution based on queued_retry, with a remark that number of consumers must be set to 1.

Identifying exporters (when multiple ones used)

It is anticipated that several exporters in each pipeline will be used. This might be challenging when there are existing WAL's for no-longer existing exporters (and the other way around). Old entries should be removed regardless if the exporter is configured or not. Each exporter needs to be identified accordingly too. This can be achieved via its name in the pipeline. For example, following pipelines:

```
service:
  pipelines:
    traces:
```

```
    receivers: [otlp]
    processors: [memory_limiter, batch]
    exporters: [otlp, zipkin]
  metrics/foo:
    receivers: [prometheus]
    processors: [batch]
    exporters: [otlp]
```

would yield following exporter ids used in WALs:

- `traces#otlp`
- `traces#zipkin`
- `metrics/foo#otlp`