Metrics API/SDK C++ Design Document Github Draft

This document outlines a proposed implementation of the OpenTelemetry Metrics API & SDK in C++. The design conforms to the current versions of the Metrics API Specification and the Metrics SDK Specification though both are currently under development and subject to change.

Use Cases

A *metric* is some raw measurement about a service, captured at runtime. Logically, the moment of capturing one of these measurements is known as a *metric event* which consists not only of the measurement itself, but the time that it was captured as well as contextual annotations which tie it to the event being measured. Users can inject instruments which facilitate the collection of these measurements into their services or systems which may be running locally, in containers, or on distributed platforms. The data collected are then used by monitoring and alerting systems to provide statistical performance data.

Monitoring and alerting systems commonly use the data provided through metric events, after applying various aggregations and converting into various exposition formats. However, we find that there are many other uses for metric events, such as to record aggregated or raw measurements in tracing and logging systems. For this reason, OpenTelemetry requires a separation of the API from the SDK, so that different SDKs can be configured at run time.

Various instruments also allow for more optimized capture of certain types of measurements. `Counter` instruments, for example, are monotonic and can therefore be used to capture rate information. Other potential uses for the `Counter` include tracking the number of bytes received, requests completed, accounts created, etc.

A `ValueRecorder` is commonly used to capture latency measurements. Latency measurements are not additive in the sense that there is little need to know the latency-sum of all processed requests. We use a `ValueRecorder` instrument to capture latency measurements typically because we are interested in knowing mean, median, and other summary statistics about individual events.

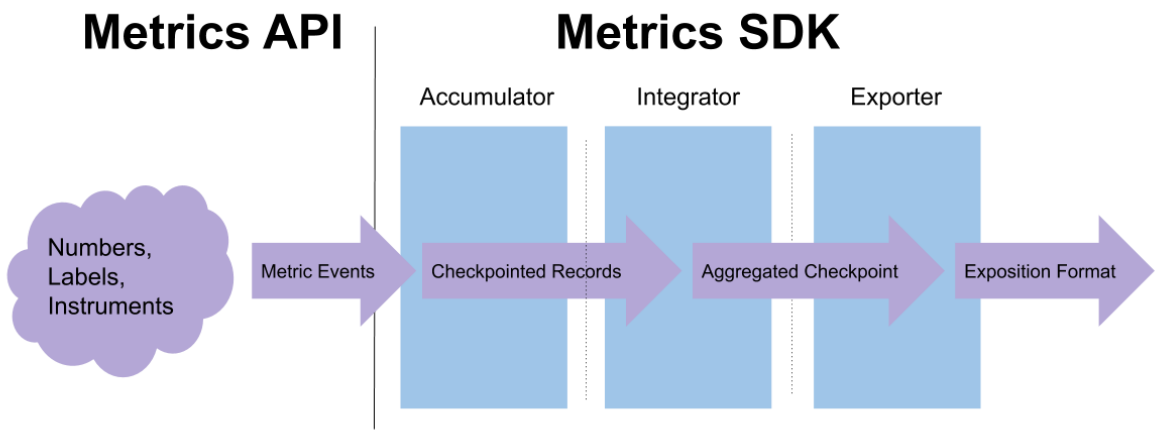
`Observers` are a good choice in situations where a measurement is expensive to compute, such that it would be wasteful to compute on every request. For example, a system call is needed to capture process CPU usage, therefore it should be done periodically, not on each request.

Design Tenets

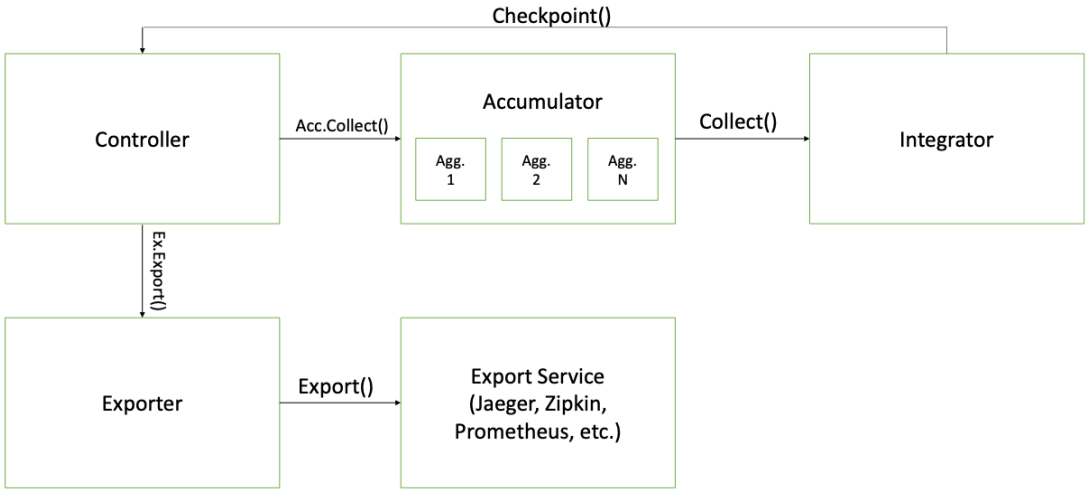
- Reliability
  - The Metrics API and SDK should be “reliable,” meaning that metrics data will always be accounted for. It will get back to the user or an error will be logged. Reliability also entails that the end-user application will never be blocked. Error handling will therefore not interfere with the execution of the instrumented program. The library may “fail fast” during the initialization or configuration path however.
  - Thread Safety
    - As with the Tracer API and SDK, thread safety is not guaranteed on all functions and will be explicitly mentioned in documentation for functions that support concurrent calling. Generally, the goal is to lock functions which change the state of library objects (incrementing the value of a Counter or adding a new Observer for example) or access global memory. As a performance consideration, the library strives to hold locks for as short a duration as possible to avoid lock contention concerns. Calls to create instrumentation may not be thread-safe as this is expected to occur during initialization of the program.

- Scalability
    - As OpenTelemetry is a distributed tracing system, it must be able to operate on sizeable systems with predictable overhead growth. A key requirement of this is that the library does not consume unbounded memory resource.
- Security
    - Currently security is not a key consideration but may be addressed at a later date.

The following diagrams illustrates the numerous components within the Metrics API and SDK as well as their relationships.



This diagram highlights the specific interactions between the disparate SDK classes.



Meter Interface (`MeterProvider` Class)

The singleton global `MeterProvider` can be used to obtain a global Meter by calling `global.GetMeter(name,version)` which calls `GetMeter()` on the initialized global `MeterProvider`

Global Meter Provider

The API should support a global `MeterProvider`. When a global instance is supported, the API must ensure that `Meter` instances derived from the global `MeterProvider` are initialized after the global SDK implementation is first initialized.

A `MeterProvider` interface must support a `global.SetMeterProvider(MeterProvider)` function which installs the SDK implementation of the `MeterProvider` into the API

Obtaining a Meter from MeterProvider

`GetMeter(name, version)` method must be supported

- Expects 2 string arguments:
  - name (required): identifies the instrumentation library.
  - version (optional): specifies the version of the instrumenting library (the library injecting OpenTelemetry calls into the code)

Implementation

The Provider class offers static functions to both get and set the global MeterProvider. Once a user sets the MeterProvider, it will replace the default No-op implementation stored as a private variable and persist for the remainder of the program's execution. This framework imitates the TracerProvider used in the Tracing SDK.

```
# meter_provider.h or an addition to Provider.h

/**
 * Stores the singleton global MeterProvider.
 */
class Provider
{
public:
  /**
   * Returns the singleton MeterProvider. By default, a no-op TracerProvider is returr
   * This will never return a nullptr TracerProvider.
   *
   */
  static nostd::shared_ptr<MeterProvider> GetMeterProvider()
  {
    // Call the GetProvider() function and return its value
  }

  /**
   * Changes the singleton TracerProvider.
   *
   * Arguments:
   * newMeterProvider, the MeterProvider instance to be set as the new global provider
   */
  static void SetMeterProvider(nostd::shared_ptr<MeterProvider> newMeterProvider)
  {
```

```
    GetProvider() = newMeterProvider;
  }

private:
  static nostd::shared_ptr<MeterProvider> &GetProvider() noexcept
  {
    //return a No-op MeterProvider
  }
};
```

Using this MeterProvider, users can obtain new Meters through the GetMeter function which first creates an InstrumentationInfo object to hold all the immutable information about the Meter then passes that to the Meter Constructor.

```
# meter_provider.h

/**
 * Creates new Meter instances.
 */
class MeterProvider
{
public:
  /**
   * Gets or creates a named meter instance.
   *
   * Optionally a version can be passed to create a named and versioned meter
   * instance.
   *
   * Arguments:
   * library_name, the name of the instrumenting library
   * library_version, the version of the instrumenting library
   */
  nostd::shared_ptr<Meter> GetMeter(nostd::string_view library_name,
                                    nostd::string_view library_version = ""){
      // Create an InstrumentationInfo object which holds the library name and versio
      // Call the Meter constructor with InstrumentationInfo
  }
};
```

Metric Instruments (`Meter` Class)

Metric Events

Metric instruments are primarily defined by their name. Names MUST conform to the following syntax:

- Non-empty string
- case-insensitive
- first character non-numeric, non-space, non-punctuation

- subsequent characters alphanumeric, ‘_’, ‘.’ , and ‘-’

`Meter` instances MUST return an error when multiple instruments with the same name are registered

The meter implementation will throw an illegal argument exception if the user-passed `name` for a metric instrument either conflicts with the name of another metric instrument created from the same `meter` or violates the `name` syntax outlined above.

Each distinctly named Meter (i.e. Meters derived from different instrumentation libraries) MUST create a new namespace for metric instruments descending from them. Thus, the same instrument name can be used in an application provided they come from different Meter instances.

In order to achieve this, each instance of the `Meter` class will have a container storing all metric instruments that were created using that meter. This way, metric instruments created from different instantiations of the `Meter` class will never be compared to one another and will never result in an error.

This interface consists of a set of **instrument constructors**, and a **facility for capturing batches of measurements** in a semantically atomic way.

```
# Meter.h

Class Meter {
public:
 /*
  * Constructor for Meter class
  *
  * Arguments:
  * MeterProvider, the MeterProvider object that spawned this Meter.
  * InstrumentationInfo, the name of the instrumentation library and, optionally,
  *                      the version.
  *
  */
  Meter(MeterProvider, InstrumentationInfo)

 /*
  * Record batch
  *
  * Allows the functionality of acting upon multiple metrics with the same set
  * of labels with a single API call. Implementations should find bound metric
  * instruments that match the key-value pairs in the labels.
  *
  * Arugments:
  * labels, labels associated with all measurements in the batch.
  * records, A sequence of pairs containing "Metrics" and the corresponding
  *          value to record for that metric.
  *
  */
  record_batch(labels, records)

//////////////////////////Metric Instrument Constructors/////////////////////////////

 /*
  * Double counter
  *
```

```
   * Function that creates and returns a Counter metric instrument with value
   * type double. Also adds this instrument to metrics container.
   *
   * Arguments:
   * name, the name of the metric instrument (must conform to the above syntax).
   * description, a brief, readable description of the metric instrument.
   * unit, the unit of metric values following the UCUM convention
   *       (https://unitsofmeasure.org/ucum.html).
   *
   */
   new_double_counter(name, description,  unit)

  /*
   * long counter
   *
   * Function that creates and returns a Counter metric instrument with value
   * type long. Also adds this instrument to metrics container.
   *
   * Arguments:
   * name, the name of the metric instrument (must conform to the above syntax).
   * description, a brief, readable description of the metric instrument.
   * unit, the unit of metric values following the UCUM convention
   *       (https://unitsofmeasure.org/ucum.html).
   *
   */
   new_long_counter(name, description,  unit)

////////////////////////////////////////////////////////////////////////////////
//                                                                            //
//                     Repeat above two functions for all                     //
//                     six (five other) metric instruments.                   //
//                                                                            //
////////////////////////////////////////////////////////////////////////////////

private:

  Vector<Metric> metrics_ //Needs to be nostd
  Batcher batcher_
}
```

Meter Class Design Considerations:

According to the specification, both signed integer and floating point value types must be supported. This implementation will use int64 and double types. Though int64 is more memory intensive than a standard int32, logs can easily measure quantities exceeding 2 billion. Different constructors are used for the different metric instruments and even for different value types due to C++ being a strongly typed language. This is identical to Java's implementation of the meter class. Python gets around this by passing the value type and metric type to a single function called `create_metric`. A C++ implementation in this way would require the use of templates which would greatly increase the complexity of the code while offering only small benefits to users.

Metric instruments created from this Meter class will be stored in a map (or another, similar container [needs to be nostd]) called "metrics." This is identical to the Python implementation and makes sense because the SDK implementation of the `Meter` class should have a function titled `collect_all()` that collects metrics for every instrument created from this meter. In contrast, Java's implementation has a `MeterSharedState` class that contains a

registry (hash map) of all metric instruments spawned from this meter. However, since each `Meter` has its own unique instruments it is easier to store the instruments in the meter itself.

The `Meter` class contains its own `Batcher` to assist with the `record_batch` function. Otherwise, each time the user wants to record values to a batch of instruments with the same label the meter will need to obtain a new `Batcher`.

The SDK implementation of the `Meter` class will contain a function called `collect_all()` that will collect the measurements from each metric stored in the `metrics` container. The implementation of this class acts as the accumulator in the SDK specification.

Pros of this implementation:

- Different constructors for the various metric instruments and types allows us to forego passing the metric type and value type along with the instrument through the data pipeline.
- Storing the metric instruments created from this meter directly in the meter object itself allows us to implement the collect_all method without creating a new class that contains the meter state and instrument registry.
- Storing a batcher in the Meter class allows us to call the method record_batch without constructing a new batcher every time.

Cons of this implementation:

- Different constructors for the different metric instruments means a lot of duplicated code. We could use templates but that will cause issues when exporting due to the requirement to adhere to the OT protocol.
- Storing the metric instruments in the Meter class means that if we have multiple meters, metric instruments are stored in various objects. Using an instrument registry that maps meters to metric instruments resolves this.
- Some users may never call record_batch. In this case, the batcher in the Meter class is simply taking up memory and serves no purpose.

The SDK implementation of the `Meter` class will act as the Accumulator mentioned in the SDK specification.

Instrument Types (`Metric` Class)

Metric instruments capture raw measurements of designated quantities in instrumented applications. All measurements captured by the Metrics API are associated with the instrument which collected that measurement.

Metric Instrument Data Model

Each instrument must have enough information to meaningfully attach its measured values with a process in the instrumented application. As such, metric instruments contain the following fields

- name (string) – Identifier for this metric instrument.
- description (string) – Short description of what this instrument is capturing.
- value_type (string or enum) – Determines whether the value tracked is an int64 or double.
- meter (Meter) – The Meter instance from which this instrument was derived.
- label_keys (wstring) – A string representing the labels associated with the Bound Instruments.
- enabled (boolean) – Determines whether the instrument is currently collecting data.
- bound_instruments (key value container) – Contains the bound instruments derived from this instrument.

Metric instruments are created through instances of the `Meter` class and each type of instrument can be described with the following properties:

- Synchronicity: A synchronous instrument is called by the user in a distributed Context (i.e., Span context, Correlation context) and is updated once per request. An asynchronous instrument is called by the SDK once per collection interval and only one value from the interval is kept.
- Additivity: An additive instrument is one that records additive measurements, meaning the final sum of updates is the only useful value. Non-additive instruments should be used when the intent is to capture information about the distribution of values.
- Monotonicity: A monotonic instrument is an additive instrument, where the progression of each sum is non-decreasing. Monotonic instruments are useful for monitoring rate information

The following instrument types will be supported:

Name	Instrument kind	Function(argument)	Default aggregation	Notes
Counter	Synchronous additive monotonic	Add(increment)	Sum	Per-request, part of a monotonic sum
UpDownCounter	Synchronous additive	Add(increment)	Sum	Per-request, part of a non-monotonic sum
ValueRecorder	Synchronous	Record(value)	MinMaxSumCount	Per-request, any non-additive measurement
SumObserver	Asynchronous additive monotonic	Observe(sum)	Sum	Per-interval, reporting a monotonic sum
UpDownSumObserver	Asynchronous additive	Observe(sum)	Sum	Per-interval, reporting a non-monotonic sum
ValueObserver	Asynchronous	Observe(value)	MinMaxSumCount	Per-interval, any non-additive measurement

Metric Event Data Model

Each measurement taken by a Metric instrument is a Metric event which must contain the following information:

- timestamp (implicit) – System time when measurement was captured.
- instrument definition(strings) – Name of instrument, kind, description, and unit of measure
- label set (key value pairs) – Labels associated with the capture, described further below.
- value (string) – Determines whether the class uses long or double values.
    - Metrics API spec: value (**signed integer** or **floating point number**)
- resources associated with the SDK at startup

Label Set

A key:value mapping of some kind MUST be supported as annotation each metric event. Labels must be represented the same way throughout the API (i.e. using the same idiomatic data structure) and duplicates are dealt with by taking the last value mapping. Languages may optionally choose to support ordered key labeling (e.g. OrderedLabels(“a”,“b”,“c”) → “a”:1,“a”:2,“a”:3)

Since label sets will often need to be compared, we have chosen to implement labels as a string data type.

Calling Conventions

Metric instruments must support bound instrument calling where the labels for each capture remain the same. After a call to `instrument.Bind(labels)`, all subsequent calls to `instrument.add()` will include the labels implicitly in their capture.

Direct calling must also be supported. The user can specify labels with the capture rather than binding beforehand by including the labels in the update call: `instrument.Add(x, labels).`

MUST support `RecordBatch` calling (where a single set of labels is applied to several metric instruments)
`meter.RecordBatch(labels, counter.measurement(1), updowncounter.measurement(10))`

Implementation

A base Metric class defines the constructor and binding functions which each metric instrument will need. Once an instrument is bound, it becomes a BoundInstrument which extends a BaseBoundInstrument class. The BaseBoundInstrument is what communicates with the aggregator and performs as actual updating of values. An enum helps to organize the numerous types of metric instruments that will be supported.

```
# Metric.h

/*
* Enum classes to hold the various types of Metric Instruments and their
* bound complements.
*/

enum class MetricKind
{
  Counter,
  UpDownCounter,
  ValueRecorder,
  SumObserver,
  UpDownSumObserver,
  ValueObserver,
};

enum class BoundMetricKind
{
  BoundCounter,
  BoundUpDownCounter,
  BoundValueRecorder,
  BoundSumObserver,
  BoundUpDownSumObserver,
  BoundValueObserver,
};

/*
 * Base class for all metric types.
 *
 * Also known as metric instrument. This is the class that is used to
 * represent a metric that is to be continuously recorded and tracked. Each
```

```
 * metric has a set of bound metrics that are created from the metric. See
 * `BaseBoundInstrument` for information on bound metric instruments.
 */
class Metric{

private:
    string name_;
    string description_;
    string value_type_;
    bool enabled_;
    wstring labels_;
    MetricKind kind_;
    KeyValueIterable bound_instruments_;

public:

    /*
    * Metric Class Constructor
    *
    * Arguments:
    * _name, the name of the new instrument
    * _description,  description of the instrument type
    * _value_type, type of the value being tracked (int64 or double)
    */
    Metric (string name, string description, string value_type, MetricKind kind,
                                                            bool enabled)

    /*
    * Bind function attaches a set of a labels to the instrument thereby
    * creating a boundInstrument type from the existing metric
    *
    * Arguments:
    * _labels, the labels being attached to the metric instrument
    */
    BoundMetricInstrument bind(string & labels){
        // if bound_instruments[labels] exists: return bound_instruments[labels]
        // else: create bound instrument, add to dictionary, then return
    }
}

   /*
   * Default no-op implementation of the Metric class used when no other
   * implementations are available
   */
class DefaultMetric: public Metric {

    /*
    * Default constructor returns a no-op BoundInstrument
    *
    * Arguments:
    * labels, the labels being attached to the metric instrument
    */
    DefaultBoundInstrument bind(string labels) {
        return DefaultBoundInstrument();
    }
```

```
    /*
    * No-op implementation of the add function intended to update metric value
    *
    * Arguments:
    * value, the quantity of change to be applied to the metric
    * labels, labels to be attached to the capture
    */
    void add(type value, string labels) {
        //No-op implementation
    }

    /*
    * No-op implementation of the record function intended to store observer value
    *
    * Arguments:
    * value, the quantity of change to be applied to the metric
    * labels, labels to be attached to the capture
    */
    void record(type value, string labels) {
        // No-op implementation
    }

}

class DefaultBoundInstrument {
    // No-op BoundInstrument implementation, used when no bound instrument implementat

    /*
    * No-op implementation of the add function intended to update metric value
    *
    * Arguments:
    * value, the quantity of change to be applied to the metric
    * labels, labels to be attached to the capture
    */
    void add(type value) {
        //No-op implementation of add
    }

    /*
    * No-op implementation of the record function intended to store observer value
    *
    * Arguments:
    * value, the quantity of change to be applied to the metric
    * labels, labels to be attached to the capture
    */
    void record(type value) {
        //No-op implementation of record
    }

    /*
    * No-op implementation of the release function intended to unbind an instrument
    * from its current labelSet
    */
    void release() {
```

```
        //No-op implementation of release
    }

}

class BaseBoundInstrument {

private:
    // How are we going to handle typing? Leaving it like this for now
    type value_type_;
    bool enabled_;
    int ref_count_;
    Aggregator aggregator_;

    /*
    * BaseBoundInstrument constructor
    *
    * Arguments:
    * _aggregator, the aggregator instance to use for tracking values
    * _value_type, kind of value the instrument will track
    * _enabled, whether or not the instrument should currently be tracking values
    */
    BaseBoundInstrument (Aggregator aggregator, type value_type, bool enabled) {
        // Set private vars, initialize ref_count to 0
    }

    /*
    * Used to refresh the metric with current values
    *
    * Arguments:
    * value, the latest state of the metric's target quantity
    */
    void update(type value) {
        // Call aggregator's update method
    }

    /*
    * Reduce the internal reference counter to signify unbinding of labels
    */
    void decrease_ref_count() {
        ref_count -= 1;
    }

    /*
    * Increase the internal reference counter to signify binding of labels
    */
    void increase_ref_count() {
        ref_count += 1;
    }

    /*
    * Return the current ref_count
    */
    int ref_count() {
        return ref_count;
```

```
    }

    /*
    * Decrement ref count with thread-safety measures
    */
    void release() {
        decrease_ref_count();
    }
}
```

The Counter below is an example of one Metric instrument. It is important to note that in the Counter's add function, it binds the labels to the instrument before calling add, then unbinds. Therefore all interactions with the aggregator take place through bound instruments and by extension, the BaseBoundInstrument Class.

```
class IntCounter: public Metric{
    """See `opentelemetry.metrics.Counter`.
    """

public:

    /*
    * Add a particular value to the counter's current state
    *
    * Arguments:
    * value, the latest state of the metric's target quantity
    * labels, the set of labels to include wih this capture
    */
    void add(type value, KeyValueIterable &labels) -> None{
        """See `opentelemetry.metrics.Counter.add`."""
        bound_intrument = self.bind(labels)
        bound_intrument.add(value)
        bound_intrument.release()
    }

private:
    BoundMetricKind kind_ = BoundCounter

}

class BoundIntCounter:public BaseBoundInstrument{

public:

    /*
    * Add a particular value to the counter's current state by sending
    * the update to the aggregator associated with the metric
    *
    * Arguments:
    * value, the latest state of the metric's target quantity
    */
```

```
    void add(__int64 value){
            //call to update() with value
        }
    }

}

class DoubleCounter: public Metric{
    """See `opentelemetry.metrics.Counter`.
    """

public:

    /*
    * Add a particular value to the counter's current state
    *
    * Arguments:
    * value, the latest state of the metric's target quantity
    * labels, the set of labels to include wih this capture
    */
    void add(double value, KeyValueIterable &labels) -> None{
        """See `opentelemetry.metrics.Counter.add`."""
        bound_intrument = self.bind(labels)
        bound_intrument.add(value)
        bound_intrument.release()
    }

private:
    BoundMetricKind kind = BoundCounter

}

class BoundDoubleCounter:public BaseBoundInstrument{

public:

    /*
    * Add a particular value to the counter's current state by sending
    * the update to the aggregator associated with the metric
    *
    * Arguments:
    * value, the latest state of the metric's target quantity
    */
    void add(double value){
            //call to update() with value
        }
    }

}


// The above Counter and BoundCounter are examples of 1 metric instrument.
// The remaining 5 will also be implemented in a similar fashion. With Int and
// Double for each metric instrument
```

```
class UpDownCounter: public Metric;
class ValueRecorder: public Metric;
class SumObserver: public Metric;
class UpDownSumObserver: public Metric;
class ValueObserver: public Metric;

class BoundUpDownCounter: public BaseBoundInstrument;
class BoundValueRecorder: public BaseBoundInstrument;
class BoundSumObserver: public BaseBoundInstrument;
class BoundUpDownSumObserver: public BaseBoundInstrument;
class BoundValueObserver: public BaseBoundInstrument;
```

In the meter class there will be functions such as the following

Metric Class Design Considerations:

OpenTelemetry requires several types of metric instruments with very similar core usage, but slightly different tracking schemes. As such, a base Metric class defines the necessary functions for each instrument leaving the implementation for the specific instrument type. Each instrument then inherits from this base class making the necessary modifications. In order to facilitate efficient aggregation of labeled data, a complementary BoundInstrument class is included which attaches the same set of labels to each capture. Knowing that all data in an instrument has the same labels enhances the efficiency of any post-collection calculations as there is no need for filtering or separation. In the above code examples, a Counter instrument is shown but all 6 mandated by the specification will be supported.

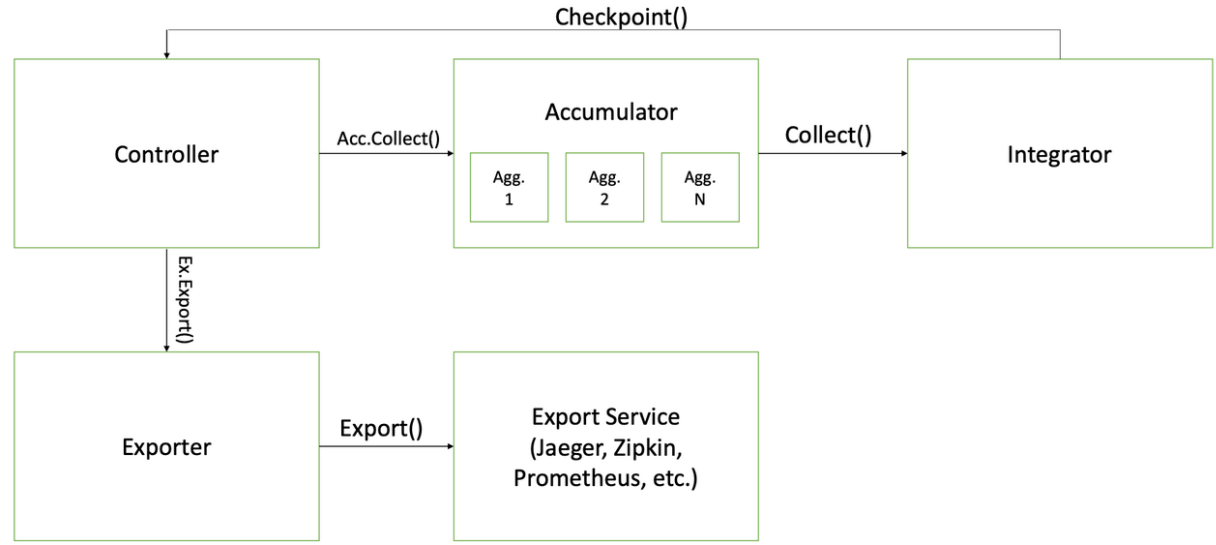
A base BoundInstrument class also serves as the foundation for more specific bound instruments and outlines the mandatory methods for updating values with calls to the Aggregator. It also includes reference counts which help determine when an instrument is inactive or finished with collection in other classes and potentially with memory optimization as inactive instruments can be scrubbed for performance.

The decision was also made to store labels as a string rather than in a key value container as they can then be used as a key for a mapping between labels and their respective bound instruments. A map is a less suitable choice as the key for a hash-based container. The user experience could also benefit as they would not have to create a container for a small number of keys each time they capture.

In other languages, reference counting was achieved through custom functions. In C++ however, the shared pointer class
is a better option as it disseminates the same information through the `use_count` function which returns the number of pointers sharing the object. The implementation benefits since it would be using a highly optimized reference counting scheme rather than repeated function calls. It also prevents accidental memory leaks and improves readibility by reducing the number of repeated function calls. That said, shared pointers are more performance intensive than regular unique_ptrs though this would likely be offset by the reduced number of calls to custom ref_count functions and their supporting data members.

Metrics SDK Requirements

Note: these requirements come from a specification currently under development. Changes and feedback are in PR #347 and the current document is linked here.



Accumulator

The Accumulator is responsible for computing aggregation over a fixed unit of time. It essentially takes a set of captures and turns them into a quantity that can be collected and used for meaningful analysis by maintaining aggregators for each active instrument and each distinct label set instruments. For example, the aggregator for a counter must combine multiple calls to Add(increment) into a single sum.

Accumulators MUST support a `Checkpoint()` operation which saves a snapshot of the current state for collection and a `Merge()` operation which combines the state from multiple aggregators into one.

Calls to the Accumulator's `Collect()` sweep through metric instruments with un-exported updates, checkpoints their aggregators, and submits them to the integrator/exporter. This and all other accumulator operations should be extremely efficient and follow the shortest code path possible.

```
    /*
    * The accumulator tracks records from the aggregators of each bound instrument
    * associated with a particular metric.  It provides facilities for checkpointing,
    * clearing old records, calling the instrument, and exporting
    *
    */
class Accumulator {


public:

    /*
    * Accumulator Constructor
    *
    * Arguments:
    * _m, the metric instrument this accumulator tracks
    */
    Accumulator(Metric & m);
```

```
    /*
    * Add a particular value to the counter's current state
    *
    * Arguments:
    * value, the latest state of the metric's target quantity
    * labels, the set of labels to include wih this capture
    */
    void RecordDirect(type value, string labels){
        // Bind m to labels then call add with the passed value
    }

    /*
    * Add a particular value to the counter's current state
    *
    * Arguments:
    * value, the latest state of the metric's target quantity
    */
    void RecordBound(type value){
        // Call the metric's add function with the passed value
    }

    /*
    * Store the current value of each bound instrument associated with this metric
    */
    KeyValueIterable<string, Record> Collect(){
        // For each boundInstrument in m, call the aggregator.current() method
        // and store results in records with the labels corresponding to the instrumen
        // as the key.
        // return get_records()
    }

    /*
    * Take a snapshot of the current set of records for future use
    */
    void take_checkpoint(){
        // set checkpoint to records
        // clear current records
    }

    void RecordBatch(vector<type> values, string labels){
        // Bind m to labels
        // Loop over values vector, calling RecordBound() on each
    }

    /*
    * Remove inactive records from the records set.  How the records are classified
    * as inactive depends on user parameters in the controller
    *
    * Arguments:
    * labels, the set of labels with records deemed as stale
    */
    void RemoveStaleRecords(const vector<string> & labels){
        // Loop over labels removing stale records from the records map
    }
```

```
    /*
    * Return the current set of records
    */
    KeyValueIterable<string, Record> get_records(){
        //return and clear records
    }

    /*
    * Return the lates checkpoint of records
    */
    KeyValueIterable<string, Record> get_checkpoint(){
        //return checkpoint of records
    }

private:
    Metric m_;
    KeyValueIterable<string, Record> records_;
    KeyValueIterable<string, Record> checkpoint_;
}
```

Notes on Aggregator Design:

Removing stale records is done in this implementation by passing a set of labels preselected for removal. The specification mandates that the ability to remove stale metrics be present with few implementation details. Using labels is a logical extension of indexing format used in the Metric class.

Reference the Recommended Implementation

Aggregator

The term *aggregator* refers to an implementation that can combine multiple metric updates into a single, combined state for a specific function. Aggregators MUST support `Update()`, `Checkpoint()`, and `Merge()` operations. `Update()` is called directly from the Metric instrument in response to a metric event, and may be called concurrently. The `Checkpoint()` operation is called to atomically save a snapshot of the Aggregator. The `Merge()` operation supports dimensionality reduction by combining state from multiple aggregators into a single Aggregator state.

The SDK must include the Counter aggregator which maintains a sum and the gauge aggregator which maintains last value and timestamp. In addition, the SDK should include MinMaxSumCount, Sketch, Histogram, and Exact aggregators
All operations should be atomic in languages that support them.

```
class Aggregator {
    """Base class for aggregators.
    Aggregators are responsible for holding aggregated values and taking a
    snapshot of these values upon export (checkpoint).
    """

public:
    Aggregator(){
        self.current = nullptr
        self.checkpoint = nullptr
```

```
    }

    @abc.abstractmethod
    virtual void update(<T>t value);
        """Updates the current with the new value."""

    @abc.abstractmethod
    virtual void take_checkpoint();
        """Stores a snapshot of the current value."""

    @abc.abstractmethod
    virtual void merge(Aggregator other);
        """Combines two aggregator values."""

private:
    Type current;
    Type checkpoint;
    timestamp last_update_timestamp

}
class CounterAggregator(Aggregator){
    """Aggregator for Counter metrics."""

public:
    CounterAggregator(): current(0), checkpoint(0), last_update_timestamp(nullptr){}

    void update(<T> value){
        // thread lock
        // current += value
        this->last_update_timestamp = time_ns()
    }

    void take_checkpoint(){
        // thread lock
            this->checkpoint = this->current
            this->current = 0
    }

    void merge(CounterAggregator * other){
        // thread lock
            this->checkpoint += other->checkpoint
            this->last_update_timestamp = get_latest_timestamp(
                this-> last_update_timestamp, other-> last_update_timestamp
            )
     }
```

This CounterAggregator() is an example Aggregator. We plan on implementing all the Aggregators in the specification: Counter, Gauge, MinMaxSumCount, Sketch, Histogram, and Exact aggregators

Integrator

The Integrator SHOULD act as the primary source of configuration for exporting metrics from the SDK. The two kinds of configuration are:

1. Given a metric instrument, choose which concrete aggregator type to apply for in-process aggregation.
2. Given a metric instrument, choose which dimensions to export by (i.e., the "grouping" function).

During the collection pass, the Integrator receives a full set of check-pointed aggregators corresponding to each (Instrument, LabelSet) pair with an active record managed by the Accumulator. According to its own configuration, the Integrator at this point determines which dimensions to aggregate for export; it computes a checkpoint of (possibly) reduced-dimension export records ready for export. It can be thought of as the business logic or processing phase in the pipeline.

Change of dimensions: The user-facing metric API allows users to supply LabelSets containing an unlimited number of labels for any metric update. Some metric exporters will restrict the set of labels when exporting metric data, either to reduce cost or because of system-imposed requirements. A *change of dimensions* maps input LabelSets with potentially many labels into a LabelSet with a fixed set of label keys. A change of dimensions eliminates labels with keys not in the output LabelSet and fills in empty values for label keys that are not in the input LabelSet. This can be used for different filtering options, rate limiting, and alternate aggregation schemes. Additionally, it will be used to prevent unbounded memory growth through capping collected data. The community is still deciding exactly how metrics data will be pruned and this document will be updated when a decision is made.

The following is a pseudo code implementation of a 'simple' Integrator. Josh MacDonald is working on implementing a 'basic' Integrator which allows for further Configuration that lines up with the specification in Go. He will be finishing the implementation and updating the specification within the next few weeks. We recommend that we implement the 'simple' Integrator first as apart of the MVP and then will also implement the 'basic' Integrator later on. Josh recommended having both for doing different processes.

```
class Integrator {
    AggregationSelector aggSelector
    Bool stateful
    Batch batch

    Integrator (AggregationSelector _aggSelector, Bool _stateful) {
        aggSelector = _aggSelector
        stateful = _stateful
    }

    void Process(Record record) {

        key = batchKey(record)

        if (this.batch.values.ok) {
            /* The call to Merge here only combines identical records. Required even
            for a stateless Integrator because such identical records may arise in the
            Meter implementation due to race conditions */
            record.aggreagtor.Merge(record.Aggregator(), record.Descriptor())
        }

        if(this.stateful) {
            /* If this integrator is stateful, create a copy of the Aggregator for
            long-term storage. Otherwise the Meter implementation will checkpoint the
            aggregator again, overwriting the long-lived state.*/

            temp = record.Aggregator()

            // Essentially cloning data
```

```
            agg = this.AggregatorFor(record.Descriptor())
            agg.Merge(temp,record.Descriptor())
        }

        this.batch.values[key] = batchValue(agg,record)

    }

    Checkpointset CheckpointSet() {
        return this.batch
    }

    void FinishedCollection() {
        if !b.stateful {
            b.batch.values = map[batchKey]batchValue{}
        }
    }

    Error ForEach( f(export.Record)) {
        for key, value := range this.batch.values {
            if err := f(export.NewRecord(
                key.descriptor,
                value.labels,
                value.resource,
                value.aggregator,
            )); err != nil && !errors.Is(err, aggregation.ErrNoData) {
                return err
            }
        }
        return nullptr
    }

}
```

Controller

Controllers generally are responsible for binding the Accumulator, the Integrator, and the Exporter. The controller initiates the collection and export pipeline and manages all the moving parts within it. It also governs the flow of data through the SDK components. Users interface with the controller to begin collection process.

Once the decision has been made to export, the controller must call `Collect()` on the Accumulator, then read the checkpoint from the Integrator, then invoke the Exporter.

Java's IntervalMetricReader class acts as a parallel to the controller. The user gets an instance of this class, sets the configuration options (like the tick rate) and then the controller takes care of the collection and exporting of metric instruments the user defines.

There are two different controllers: Push and Pull. The "Push" Controller will establish a periodic timer to regularly collect and export metrics. A "Pull" Controller will await a pull request before initiating metric collection.

We recommend implementing the PushController as the initial implementation of the Controller. This Controller is the base controller in the specification. We may also implement the PullController if we have the time to do it.

```
class PushController {

    private:
        Mutex lock
        Accumulator accumulator
        MeterProvider provider
        Integrator integrator
        Exporter exporter
        Int period
        Int timeout
        Clock clock
        chan struct{} ch
        Ticker ticker
        WaitGroup wg


    PullController(Aggregator aggregator, Exporter exporter, Int _period, Int _timeout
        integrator = Integrator(aggregator);
        accumulator = Accumulator(integrator);
        exporter =    exporter;
        period = _period;
        timeout = _timeout;
        clock = Clock.clock();
        provider = NewMeterProvider(accumulator);
        ch = make(chan struct{})
    }

    // Controller.h
    // SetClock supports setting a mock clock for testing.  This must be
    // called before Start().
    void SetClock(Clock clock) {
        this.lock.Lock()
        this.clock = clock
        this.lock.Unlock()
    }

    // Provider returns a metric.Provider instance for this controller.
    metric.Provider Provider {
        return this.provider
    }

    // Start begins a ticker that periodically collects and exports
    // metrics with the configured interval.
    void Start() {
        this.lock.Lock()

        if this.ticker != nil {
            c.lock.Unlock()
            return
        }

        this.ticker = this.clock.Ticker(this.period)
        this.run(this.ch)
        this.lock.Unlock()
    }
```

```
    // Stop waits for the background to return and then collects
    // and exports metrics one last time before returning.
    void Stop() {
        this.lock.Lock()
        if this.ch == nil {
            this.lock.Unlock()
            return
        }

        close(this.ch)
        this.ch = nullptr
        this.wg.Wait()
        this.ticker.Stop()
        this.tick()
        this.lock.Unlock()
    }

    void run(ch chan struct{}) {
        for {
            select {
                case <-ch:
                    c.wg.Done()
                    return
                case <-this.ticker.C():
                    this.tick()
            }
        }
    }

   void tick() {
        ctx, cancel := context.WithTimeout(context.Background(), this.timeout)

        this.integrator.Lock()

        this.accumulator.Collect(ctx)

        err := this.exporter.Export(ctx, this.integrator.CheckpointSet())
        this.integrator.FinishedCollection()

        if err != nil {
            cancel()
            this.integrator.Unlock()
            global.Handle(err)
        }
        this.integrator.Unlock()
    }
}
```

Exporter

The exporter SHOULD be called with a checkpoint of finished (possibly dimensionally reduced) export records. Most configuration decisions have been made before the exporter is invoked, including which instruments are enabled,

which concrete aggregator types to use, and which dimensions to aggregate by.

There is very little left for the exporter to do other than format the metric updates into the desired format and send them on their way.

Our idea is to take the simple trace example StdoutExporter and add Metric functionality to it. This will allow us to verify that what we are implementing in the API and SDK works as intended. The exporter will go through the different aggregators and print out their accumulator(s) to stdout, **for simplicity only Sum is shown here, but all aggregators will be implemented**.

```
class StdoutExporter
{

    ExportResult Export(export.CheckpointSet checkpointSet) noexcept
    {

       if(checkpointSet != NULL) {

            ForEach(checkpoint in checkpointSet) {
                Aggreagtor agg = checkpoint.record.Aggregator();

                // Check what type of Aggregator it is
                if(agg.type == Sum) {
                    cout << agg.Sum() << endl;
                }
            }
       }


        return ExportResult::kSuccess;
    }

    void shutdown() {
    try {

      managedChannel.shutdown().awaitTermination(5, TimeUnit.SECONDS);

    } catch (InterruptedException e) {

      logger.log(Level.WARNING, "Failed to shutdown the gRPC channel", e);

    }
  }

};
```

Test Strategy / Plan

Since there is a specification we will be following, we will not have to write out user stories for testing. We will generally only be writing functional unit tests for this project. The C++ Open Telemetry repository uses Googletest because it provides test coverage reports, also allows us to easily integrate code coverage tools such as codecov.io with the project. A required coverage target of 90% will help to ensure that our code is fully tested.

An open-source header-only testing framework called Catch2 is an alternate option which would satisfy our testing needs. It is easy to use, supports behavior driven development, and does not need to be embedded in the project as source files to operate (unlike Googletest). Code coverage would still be possible using this testing framework but would require us to integrate additional tools such as gcov or lcov. This framework may be preferred as an agnostic replacement for Googletest and is widely used in open source projects.

Design Questions

- As the SDK specification is an evolving document, there is no consensus on how the Accumulator class should be implemented. Some languages create a separate class while others fold its functionality into the Meter class. Which option better suits C++ and the library's end users?
- As instruments are injected into user code and often export to remote locations, they create an attack vector which may undermine user applications. What security considerations should we take when developing this library?
- What about design considerations for handling concurrency issues?

Contributors

- Alolita Sharma
- Ankit Bhargava
- Brandon Kimberly
- Hudson Humphries