HADOOP-6728-MetricsV2

This page keeps the design notes for HADOOP-6728 only. Current dev/user documentation for metrics system should be kept elsewhere (say, package.html and/or package-info.java in respective packages).

Scope

- Allow multiple metrics output plugins to be used correctly in parallel (HADOOP-6508).
- Allow dynamic (without server restart) reconfiguration of metrics plugins
    - Including filtering (glob patterns preferred by ops. cf. HADOOP-6787) metrics by names.
- Allow all metrics to be exported via JMX.

While we realize that there are many optimization opportunities, the main goal is to fix existing issues and lay down the foundation for future improvements.

Design Overview

In the v2 framework, metrics sources are where the metrics are generated/updated, and metrics sinks consume the records generated by the metrics sources. A metrics system would poll the metrics sources periodically and pass the metrics records to metrics sinks (Figure 1). The source getMetrics interface allows lockless implementations of metrics instrumentation (with volatile metrics values). The sink interface is simple, where the putMetrics method would be called with an immutable metrics record (Figure 2), so that plugin implementers don't have to worry about thread safety.

{{ https://issues.apache.org/jira/secure/attachment/12452869/metrics2-uml-r2.png	Metrics system overview}}
Figure 1: Metrics system overview	

Figure 1 is a UML sketch class diagram illustrating the involved passive objects (different colors indicating different kinds of driving threads in discussion): MetricsSource (in cyan) is driven by a timer thread (for getMetrics()) and MetricSink (in green) is driven by a thread for each Sink. The MetricsFilter objects (in orange) can be used either to filter the metrics from sources in the timer thread or to filter metrics per sink in its respective thread. The metrics system expects that the getMetrics call would return fairly quickly (i.e., latency smaller than the polling period). The MetricsSinkQueue is a nonblocking queue with preconfigured size (tolerance of sink latency: n * period). New metrics records would be lost if the queue is full. The JMX MBean interface would be implemented to allow existing JMX clients (JConsole, jManage etc.) to stop and start the metrics system at run time.

{{ https://issues.apache.org/jira/secure/attachment/12452693/metrics2-record-r2.png	Immutable metrics objects}}
Figure 2: Immutable metrics objects	

As an example, a JobTracker metrics instrumentation would implement the MetricsSource interface and override the getMetrics method to return a snapshot of metrics when the timer thread polls it with a MetricsBuilder. The timer thread would then enqueue the resulting metrics records from the MetricsBuilder to each MetricsSinkQueue. The thread for each sink would blocked/wait on each MetricsSinkQueue until it's notified by the timer thread after new metrics records are enqueued and then proceeds to dequeue and call the putMetrics method of the corresponding sink object. Figure 3 and 4 illustrate the new data flow vs the old data flow.

{{ https://issues.apache.org/jira/secure/attachment/12445685/metrics1-flow.png	Before}}	{{ https://issues.apache.org/jira/secure/attachment/12445686/metrics2-flow.png	After}}
Figure 3: V1 data flow diagram	Figure 4: V2 data flow diagram		

The following new packages are proposed for a smooth transition of the metrics framework:

package	Contents	Notes
`org.apache.hadoop.metrics2`	Public interface/abstract classes of the metrics system	
`org.apache.hadoop.metrics2.annotation`	Public annotations for implementing simple metrics sources	
`org.apache.hadoop.metrics2.lib`	Public reusable components for using the metrics system	
`org.apache.hadoop.metrics2.filter`	Public (class names) builtin metrics filter (Glob/Regex) classes	
`org.apache.hadoop.metrics2.source`	Public (class names) builtin metrics source (JVM etc.) classes	
`org.apache.hadoop.metrics2.sink`	Public (class names) builtin metrics sink (file, ganglia etc.) classes	

`org.apache.hadoop.metrics2.util`	Public utility classes for developing metrics system (including plugins)	
`org.apache.hadoop.metrics2.impl`	Metrics system internal implementation classes	

Metrics Filtering

The framework supports 3 levels of filters: source, record and metrics names, thus 6 ways to filter metrics with increasing cost (in terms of memory/CPU):

1. Global source name filtering: any sources with matching names are skipped for getMetrics calls.
2. Per sink source name filtering: any sources with matching names are skipped for putMetrics calls.
3. Per source record filtering: any records with matching names or tag values are skipped in the MetricsBuilder.add* calls in the getMetrics calls.
4. Per sink record filtering: any records with matching names or tag values are skipped for the putMetrics calls.
5. Per source metrics filtering: any metrics with matching names are skipped in the Metric.sample* calls in the getMetrics calls.
6. Per sink metrics filtering: any metrics with matching names are skipped in the iteration of the MetricsRecord in putMetrics calls.

These can be mixed and matched to optimize for lower total filtering cost if necessary. See below for configuration examples.

Configuration

The new framework uses the PropertiesConfiguration from the apache commons configuration library for backward compatibility (java properties) and more features (include, variable substitution, subset etc.)

Proposed configuration examples:

```
# Basic syntax: <prefix>.(source|sink).<instance>.<option>
*.sink.file.class=org.apache.hadoop.metrics2.sink.FileSink
*.source.filter.class=org.apache.hadoop.metrics2.filter.GlobFilter
*.record.filter.class=${*.source.filter.class}
*.metric.filter.class=${*.source.filter.class}
*.period=10

# Filter out any sources with names end with Details
jobtracker.*.source.filter.exclude=*Details

# Filter out records with names that matches foo* in the source named "rpc"
jobtracker.source.rpc.record.filter.exclude=foo*

# Filter out metrics with names that matches foo* for sink instance "file" only
jobtracker.sink.file.metric.filter.exclude=foo*
jobtracker.sink.file.filename=jt-metrics.out

# Custom sink plugin
jobtracker.sink.my.class=com.example.hadoop.metrics.my.MyPlugin
# MyPlugin only handles metrics in "foo" context
jobtracker.sink.my.context=foo
```

Metrics Source (Instrumentation) Development

A minimal metrics source:

```


// default record name is the class name
// default context name is "default"
@Metrics(context="bar")
public class MyPojo {
  // Default name of metric is method name sans get
  // Default type of metric is gauge
  @Metric("An integer gauge named MyMetric")
  public int getMyMetric() { return 42; }

  // Recommended helper method
  public MyPojo registerWith(MetricsSystem ms) {
    return ms.register("MyPojo", "MyPojo metrics", this);
  }
}
```

By using annotations, one can add simple metrics to any methods returning supported types (int, long, float and double and their object counter parts) in any java classes.

An example using the mutable metric library objects:

```
@Metrics{context="bar")
public class MyMetrics {
  // Default metric name is the variable name
  @Metric("An integer gauge") MutableGaugeInt g1;
  // Default type is inferred from the mutable metric type
  @Metric("An long integer counter") MutableCounterLong c1;

  // Recommended helper method
  public MyMetrics registerWith(MetricsSystem ms) {
    return ms.register("MyMetrics", "MyMetrics description", this);
  }
}
```

{{ https://issues.apache.org/jira/secure/attachment/12452694/metrics2-mutable-r2.png	Mutable metrics for instrumentation}}
Figure 5: Mutable metrics helper objects for instrumentation	

Initialize the metrics system:

```
// Somewhere in your app's startup code, initialize the metric system.
DefaultMetricsSystem.initialize("JobTracker");

// Create the metrics source object
MyMetrics myMetrics = new MyMetrics().registerWith(DefaultMetricsSystem.instance());

// Update the metrics
myMetrics.foo.set(someValue);
myMetrics.bar.incr();
```

Note, for simple metrics sources, using annotations make the code declarative and concise. For more advanced metrics source implementations, one might need to explicitly implement the MetricsSource interface and override the getMetrics method and use the metrics builder API (Figure 6.):

```
class MyMetricsSource implements MetricsSource {

  @Override
  public void getMetrics(MetricsBuilder builder) {
    builder.addRecord("foo")
      .addGauge("g0", "an integer gauge", 42)
      .addCounter("c0", "a long counter", 42L);

    // Typical metrics sources generate one record per snapshot.
    // We can add more records, which is not supported by annotations.
    builder.addRecord("bar")
      .addGauge("g1", "a float gauge", 42.0)
      .addCounter("c1", "a integer counter", 42);
  }

  public MyMetricSource registerWith(MetricsSystem ms) {
    return ms.register("MyMetrics", "MyMetrics description", this);
  }
}
```

{{ https://issues.apache.org/jira/secure/attachment/12452695/metrics2-builder-r2.png	Metrics builders}}
Figure 6: Metrics builders	

Metrics Sink (Plugin) Developement

Implementing a sink plugin with schema conversion (without a forest of if/switches):

```
public class EchoPlugin implements MetricsSink, MetricsVisitor {

  @Override // MetricsPlugin
  public void init(SubsetConfiguration conf) {
    // do plugin specific initialization here
  }

  @Override // MetricsSink
  public void putMetrics(MetricsRecord rec) {
    echoHeader(rec.name(), rec.context());

    for (MetricTag tag : rec.tags())
      echoTag(tag.getName(), tag.getValue());

    for (AbstractMetric metric : rec.metrics())
      metric.visit(this);
  }

  @Override // MetricsSink
  public void flush() {
    // do sink specific buffer management here
  }

  @Override // MetricsVisitor
  public void counter(MetricInfo info, int value) {
    echoCounterInt32(info.name(), value);
  }

  @Override // MetricsVisitor
  public void counter(MetricInfo info, long value) {
    echoCounterInt64(info.name(), value);
  }

  @Override // MetricsVisitor
  public void gauge(MetricInfo info, int value) {
    echoGaugeInt32(info.name(), value);
  }

  @Override // MetricsVisitor
  public void gauge(MetricInfo info, long value) {
    echoGaugeInt64(info.name(), value);
  }

  @Override // MetricsVisitor
  public void gauge(MetricInfo info, float value) {
    echoGaugeFp32(info.name(), value);
  }

  @Override // MetricsVisitor
  public void gauge(MetricInfo info, double value) {
    echoGaugeFp64(info.name(), value);
  }
}
```