Monitoring Custom Metrics

How I Learned to Instrument First and Ask Questions Later

Maxime Petazzoni – @mpetazzoni – SignalFx, Inc

Intro

What's going on? Who am I? Why are we here!?

Agenda

- About Metrics
- Understanding Your Applications From Within
- Instrument First, Ask Questions Later
- Tales of Timeseries

Metrics

The Most Important Pillar of Observability

What Are Metrics?

- Measurements; datapoints
- Usually taken at regular intervals
- Reported with the identity of what they measure
- Over time, form a *timeseries*

Metrics Identity

- Metric name
- Dimensions
  - Key/value pairs
  - Cardinality
- Minimum set to uniquely identity the timeseries

Metrics Identity

```
{
  "metric": "requests",
  "dimensions": {
    "host": "h-1",
    "endpoint": "/endpoint",
    "customerId": "c-1"
  },
  "timestamp": 1540808955484,
  "value": 42
}
```

Building a Metadata Model

- Distinct timeseries for each set of dimensions
- Population of timeseries; you can look at
  - Individuals
  - Aggregates, across any subset of dimensions

Building a Metadata Model

- Common dimension names and values
  - Makes it easy to work with different metrics & datasets
  - Aggregations or computations across common pivot values
- Metadata properties on dimensions
  - Reduces dimensions to the strict subset for uniqueness
  - Can be manipulated out of band and w/o code changes

Benefits of a metadata model

- Reduces burden on developers
- Adds value
  - by making comparisons and computations possible
  - by giving more ways to slice and dice timeseries data

Understanding Your Applications From Within

The Need for Data

- ➔ RED metrics aren't everything
- ➔ Need data about internal state and behavior
- ➔ Need history and trends, not just point in time
- ➔ Custom metrics are the best path to this visibility

How To Instrument

- ➔ Metrics libraries exist for every language
- ➔ Instrumentation is just a line of code away
- ➔ Let's go through the basics...

How To Instrument: Counters

```
public void doAction(Action action) {
  try {
    // Count it.
    metrics.counter("actions").inc();
    action.execute();
    // Do it.
  } catch (ActionExecutionException e) {
    // Count errors.
    metrics.counter("action.errors").inc();
  }
}
```

How To Instrument: Counters

```
public void doAction(Action action) {
  try {
    // Count actions by type.
    metrics.counter("actions", "type", action.getType()).inc();
    action.execute();
  } catch (ActionExecutionException e) {
    // Count errors by action type and error code.
    metrics.counter("action.errors",
      "type", action.getType(),
      "error", e.getErrorCode()
    ).inc();
  }
}
```

How To Instrument: Counters

```
executions = data('actions').sum(by='type')
errors = data('actions.errors').sum(by='type')
(100 * errors / executions).publish('error rate')
```

How To Instrument: Gauges

```
Queue<Action> actions = new ArrayBlockQueue<>(1024);
metrics.registerGauge(
  "queue.size",      // metric name
  "name", "actions", // a dimension
  actions::size);    // provider for the value of the gauge

// Don't forget to unregister in teardown; this holds a strong
// reference to the queue.
```

How To Instrument: Histograms

```
public Collection<Result> search(Query query) {
  Collection<Result> results = query.execute();
  metrics.histogram("num.results").update(results.size());
  return results;
}
```

How To Instrument: Timers

```
public void doAction(Action action) {
  Timer t = metrics.timer("actions", "type", action.getType());
  try (Timer.Context c = t.time()) {
    // Do it; it's getting counted and timed.
    action.execute();
  }
}
```

How To Instrument: Timers

```
public void doAction(Action action) {
  Timer t = metrics.timer("actions", "type", action.getType());
  long start = System.nanoTime();
  try {
    // Do it.
  } finally {
    t.update(System.nanoTime() - start);
  }
}
```

Instrument First, Ask Questions Later

Culture of Instrumentation

- Today's systems are complex
  - Difficult to predict failure modes
  - Need a lot of information and history to troubleshoot
  - Don't know what metrics you'll really need
- Better to instrument as code is written
- Identitfy patterns and structures, make it a habit

Culture of Instrumentation

- ➔ Yes, it generates a lot of data, but that's ok
  - ◆ Ingest is a solved problem now
  - ◆ Need scalable and real-time analytics
- ➔ Make it your culture:
  - ◆ Instrument as you go
  - ◆ Be consistent and follow your metadata model
  - ◆ Know that you'll get the answers you seek

Tales of Timeseries

Practical Custom Metrics Examples

Cache Hit Ratio

```
metrics.register("cache.size", cache::size);

public synchronized V get(K key) {
  V value = cache.get(key);
  if (value != null) {
    metrics.counter("cache.hits").inc();
    return value;
  }
  value = backend.load(key);
  cache.put(key, value);
  metrics.counter("cache.misses").inc();
  return value;
}
```

Cache Hit Ratio

```
hits = data('cache.hits')
misses = data('cache.misses')
(100 * hits / misses).publish('hit ratio')



hits = data('cache.hits').sum(by='customer')
misses = data('cache.misses').sum(by='customer')
(100 * hits / misses).mean(over='5m').bottom(5).publish('hit ratio by customer')
```

Logging Insights with Metrics

```
public FilterReply decide(Marker marker, Logger logger, Level level,
                          String format, Object[] params, Throwable t) {
  // Count logging messages by level (memoizing the counter)
  counters.computeIfAbsent(level, (level) -> metrics.counter(
    "logging.messages",
    "level", level.name().toLowerCase())).inc();

  // If an exception was also passed to the log statement, count those by class name.
  if (t != null) {
    metrics.counter("logging.exceptions", "class", t.getClass().getSimpleName()).inc();
  }
  return FilterReply.NEUTRAL;
}
```

Logging Insights with Metrics

```
data('logging.messages', rollup='sum') # To get sum of increments instead of rate
  .sum(by='service')
  .sum(over='1w')
  .top(1).publish()
```

➔ Not as flexible, but still a high-value signal

➔ Helps reduce time to clue / resolution

Commit SHAs In Production

```
metrics.registerGauge(
  "build_info.commit",
  "sha", buildInfo.getCommitSHA(),
  "canary", buildInfo.isCanary(),
  () -> 1);
```

➔ Started as just "let's report timeseries for this"

➔ Ended up powering an important CI/CD check

Extra: Threadpool Monitoring

If time allows...

- Gauges:
  - Thread pool size
  - Task queue depth
- Counters:
  - Tasks submitted, executed, failed
- Timers:
  - Task start delay
  - Task execution time

Conclusion

Takeaways & Questions