

ECE 473/573
Cloud Computing and Cloud Native Systems
Lecture 25 Observability

Professor Jia Wang
Department of Electrical and Computer Engineering
Illinois Institute of Technology

November 12, 2025

Outline

Observability

Tracing

Metrics

Logging

Reading Assignment

- ▶ This lecture: 11
- ▶ Next two lectures: batch and stream processing
 - ▶ MapReduce: Simplified Data Processing on Large Clusters
<https://research.google/pubs/pub62/>
 - ▶ Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing http://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf

Outline

Observability

Tracing

Metrics

Logging

Observability

- ▶ The need to understand our systems better.
 - ▶ Complexity of software comes from complex requirements.
 - ▶ Good software design needs good visibility into systems.
 - ▶ No amount of fancy frameworks or protocols can solve the problem of bad software.
- ▶ Observability: the ability to infer system's internal states from knowledge of its external outputs. E.g.
 - ▶ What does that error message mean and what triggers it?
 - ▶ Why the performance is not as expected?
 - ▶ While logging may be available for general troubleshooting purposes, is it possible to answer specific questions that the developers haven't thought of yet?

Evolution of Traditional Monitoring

- ▶ Traditional monitoring focuses on the “known unknowns”
 - ▶ Identify/predict expected or previously observed failure modes.
 - ▶ Work well for simple systems through trial and error.
 - ▶ Require code updates that is not flexible.
- ▶ However, understanding and monitoring all possible failure (or non-failure) states in a complex system is impossible.
 - ▶ Scale of data is beyond human brain power and attention span.
 - ▶ Non-deterministic behaviors are difficult to reason with.
 - ▶ Interactions between component faults and system failures are very complicated.
- ▶ Monitoring shows that system is not working and observability answers why it is not working.

Three Pillars of Observability

- ▶ Tracing: details from one request to its response consisting of all functions called and messages communicated.
 - ▶ E.g. arguments and return values and time spent.
- ▶ Metrics: numerical data points representing system states at specific points in time.
 - ▶ E.g. CPU/memory/disk/network usage.
- ▶ Logging: appending records of noteworthy events to the log for later review or analysis.
 - ▶ But how can we manage and search many log files for specific information?
- ▶ A synergy of the three leads to better observability.
 - ▶ When? What? Where? Why?

Outline

Observability

Tracing

Metrics

Logging

Tracing

- ▶ Track requests as they propagate through the system.
 - ▶ Not limited to function calls within a specific process or thread.
 - ▶ Need to consider queues and communications across process, network, and even security boundaries.
 - ▶ Help to pinpoint component failures, identify performance bottlenecks, and analyze service dependencies.
- ▶ Model of requests: spans and traces
 - ▶ A request may consist of many works and addition requests, that are running recursively and parallelly.
- ▶ Span: a unit of work from beginning to end.
 - ▶ Identified by a name with start/end times.
 - ▶ Model heirarchy of works and requests as nested spans.
 - ▶ Model causal relationships as ordered spans.
- ▶ Trace: collection of spans and their relationships.

Tracing with OpenTelemetry

```
const serviceName = "foo"

func main() {
    setupTracerProvider()

    tr := otel.GetTracerProvider().Tracer(serviceName)
    ctx, sp := tr.Start(context.Background(), "main") // Start the root span
    defer sp.End() // End completes the span
    SomeFunction(ctx)
}

func SomeFunction(ctx context.Context) {
    tr := otel.GetTracerProvider().Tracer(serviceName)
    _, sp := tr.Start(ctx, "SomeFunction")
    defer sp.End()
    ... // Do something MAGICAL here!
}
```

- ▶ Record begin of span at the beginning of a function.
 - ▶ Usually the function name is used for the span.
- ▶ Make use of `defer` to record end of span.

Tracing with OpenTelemetry (cont.)

```
func setupTracerProvider() {  
    stdExporter, err := stdout.NewExporter(  
        stdout.WithPrettyPrint(),  
    )  
    jaegerEndpoint := "http://localhost:14268/api/traces"  
    serviceName := "fibonacci"  
    jaegerExporter, err := jaeger.NewRawExporter(  
        jaeger.WithCollectorEndpoint(jaegerEndpoint),  
        jaeger.WithProcess(jaeger.Process{  
            ServiceName: serviceName,  
        }),  
    )  
    tp := sdktrace.NewTracerProvider(  
        sdktrace.WithSyncer(stdExporter),  
        sdktrace.WithSyncer(jaegerExporter))  
    otel.SetTracerProvider(tp)  
}
```

- ▶ Use a remote exporter to collect spans for a single request across multiple servers.
- ▶ Use multiple exporters so the information can be found in convenient locations like local logs.

Additional Tracing Features

- ▶ Attributes and events can be added to spans.

```
span.AddEvent("Canceled by external signal",  
    label.Int("pid", 1234),  
    label.String("signal", "SIGHUP"))
```

- ▶ Attributes are key-value pairs.
- ▶ Events are points in time.

- ▶ Autoinstrumentation is available as wrappers for many popular libraries.

```
func main() {  
    // http.HandleFunc("/", helloGoHandler)  
    http.Handle("/", otelhttp.NewHandler(  
        http.HandlerFunc(helloGoHandler), "root"))  
    log.Fatal(http.ListenAndServe(":3000", nil))  
}
```

Outline

Observability

Tracing

Metrics

Logging

Metrics

- ▶ Collection of numerical data about a component, process, or activity over time. E.g.
 - ▶ Computing resources: CPU, memory used, disk/network I/O
 - ▶ Infrastructure: instance replica count, autoscaling events
 - ▶ Applications: request count, error count
 - ▶ Business metrics: revenue, customer sign-ups
- ▶ Metrics consist of data points as samples.
 - ▶ Sample should have a name, a value, and a timestamp, possibly annotated with labels as key-value pairs.
 - ▶ A set of samples form a time series that can be visualized and analyzed, e.g. for anomaly detection.
- ▶ Push vs. Pull metric collection.
 - ▶ Applications push metrics to collector: simple but needs scaling mechanisms like message queues.
 - ▶ Collector contact applications to pull metrics back: more flexible and allow ad-hoc inspections, but less friendly for service discovery, load balancer, and ephemeral services.

Metrics with OpenTelemetry

```
func main() {  
    ...  
    prometheusExporter, err := prometheus.NewExportPipeline(prometheus.Config{})  
    mp := prometheusExporter.MeterProvider()  
    otel.SetMeterProvider(mp)  
    http.Handle("/metrics", prometheusExporter)  
  
    log.Fatal(http.ListenAndServe(":3000", nil))  
}
```

- ▶ Prometheus is an open source monitoring and alerting toolkit
 - ▶ Use a pull model over HTTP to scrape metric data
 - ▶ Manage them in its time series database.

Synchronous Instruments

```
var requests metric.Int64Counter
func buildRequestsCounter() error {
    meter := otel.GetMeterProvider().Meter(serviceName)
    requests, err := meter.NewInt64Counter("fibonacci_requests_total",
        metric.WithDescription("Total number of Fibonacci requests."),
    )
    return err
}

var labels = []label.KeyValue{
    label.Key("application").String(serviceName),
    label.Key("container_id").String(os.Getenv("HOSTNAME")),
}
func Fibonacci(ctx context.Context, n int) chan int {
    requests.Add(ctx, 1, labels...)
    // The rest of the function...
}
```

- Call `buildRequestsCounter` in `main` to initialize the counter `requests` that is used later.

Synchronous Instruments (cont.)

```
func updateMetrics(ctx context.Context) {  
    meter := otel.GetMeterProvider().Meter(serviceName)  
    mem, _ := meter.NewInt64UpDownCounter("memory_usage_bytes",  
        metric.WithDescription("Amount of memory used."),  
    )  
    goroutines, _ := meter.NewInt64UpDownCounter("num_goroutines",  
        metric.WithDescription("Number of running goroutines."),  
    )  
    var m runtime.MemStats  
    for {  
        runtime.ReadMemStats(&m)  
        mMem := mem.Measurement(int64(m.Sys))  
        mGoroutines := goroutines.Measurement(int64(runtime.NumGoroutine()))  
        meter.RecordBatch(ctx, labels, mMem, mGoroutines)  
        time.Sleep(5 * time.Second)  
    }  
}
```

- ▶ Metrics can be measured and recorded in a periodic manner.
- ▶ `Int64UpDownCounter` allows to record metrics that can increase or decrease.

Outline

Observability

Tracing

Metrics

Logging

Logging

- ▶ Why can't we just use `fmt.Printf` (and so on)?
 - ▶ Easy to provide lots of context-rich data for a component.
- ▶ It is difficult to extract information from verbose and unstructured logs.
 - ▶ In particular at scale, when you are interested in logs from many, but not one, components.
- ▶ To generate and store logs consumes CPU and I/O resources.
 - ▶ Without careful planning, could easily consume significant amount of resources.
- ▶ How to store logs for access at scale?
 - ▶ Many services we have discussed and will discuss are created for processing logs!

Structured Logging

- ▶ Treat logs as streams of events.
 - ▶ Instead of lines in files that should be read by humans.
 - ▶ Applications generate events for logs.
 - ▶ Underlying infrastructure takes care of routing, storage, indexing, and analysis.
 - ▶ Developers could still read logs as lines in files for development but will access them differently, e.g. through database queries, in production.
- ▶ Structured logging: describe events as key-value pairs.
 - ▶ There is no need to generate human friendly lines that need to be parsed later.
 - ▶ Instead of,

2020/11/09 02:15:10AM User 12345: GET /help in 23ms

2020/11/09 02:15:11AM Database error: connection reset by peer

Store logs in JSON,

```
{"time":1604888110, "level":"info", "method":"GET", "path":"/help", "peer":}
```

```
{"time":1604888111, "level":"error", "error":"connection reset by peer", "peer":}
```

Logging with Zap

```
logger, err := zap.NewProduction()
if err != nil {
    log.Fatalf("can't initialize zap logger: %v", err)
}
logger.Info("failed to fetch URL",
    zap.String("url", url),
    zap.Int("attempt", 3),
    zap.Duration("backoff", time.Second),
)
```

- ▶ Zap is a popular open source logging library.
 - ▶ Known for its speed and low memory usage.
 - ▶ In particular with strong typing, through a bit awkward to use.
 - ▶ The Sugar method provides a easier but slower interface.

```
logger, _ := zap.NewProduction()
sugar := logger.Sugar()
sugar.Infof("failed to fetch URL: %s", url)
```

Summary

- ▶ Make your application observable by integrating OpenTelemetry solutions.