Monitoring

Lecture 13
Srinivas Narayana

http://www.cs.rutgers.edu/~sn624/553-S23
Operations

• How to run and manage an Internet service?
• Monitoring, security
• Load management
• Release engineering, canarying
• Crafting and maintaining SLOs
• People and processes
• Incident response, postmortems
• Designing and managing configurations
• …
Autoscaling

- Sometimes, you just don’t have enough capacity
- **Vertical autoscaling**
- **Horizontal autoscaling**
- Don’t just rely on server utilization metrics. For example, error codes returned very quickly have low CPU utilization
- Creating new instances is never instant
- Doesn’t always work:
  - Failure to do useful work but consuming resources
  - Overloading downstream dependencies by autoscaling upstream tier
  - Shared quotas across tiers: reason with dependencies carefully
Load shedding

- Return errors upon high load; process what you can
- Combination of all techniques useful. But consider their interactions carefully
Monitoring

Distributed tracing
Why do we need monitoring?

• Validating functionality: failures, exceptions, latencies
• Understanding performance hotspots during development and after deployment
  • e.g. Components inducing long tail latencies
• Securing user data, intellectual property, infrastructure
  • e.g. system calls, data exfiltration, break-ins
  • e.g. validating conformance to security policies: access control
• Top-level view of large systems
  • e.g. inferring service dependencies
  • e.g. who is inflating the (wide-area) Internet bill?
Monitoring Interactive Applications

• Distributed application components (microservices)
• Monitoring at different levels: host, network, application
• Three “pillars” of application monitoring: logs, metrics, traces
• Logs: unstructured data, highly application and event specific
• Metrics: aggregated data over time or requests per component
  • E.g. system calls, file operations within a process, etc.
• Tracing: view of a single user-level request across distributed components
Taxonomy of tracing systems

- Closed box and open box monitoring
- Libraries and agents
- System events and application events
- Inter-process and intra-process events
Goals for Tracing systems

• Application transparency
• Low overheads
• Scalability to large applications
• Privacy of user data
• Interpreting and annotating traces with additional metadata
• Joining with other telemetry data

• Today: closed box tracing using libraries to monitor inter-process application-level events
Spans and traces

• **Span**: a process-level annotated event

• **Trace**: a series of spans linked to each other by being a part of the same high-level client request

• Q1. How to instrument applications to produce spans?

• Q2. What should spans contain?

• Q3. How are spans related to one another to produce a trace?

• Q4. How to extract the data of spans from the application?
Instrumentation: OpenTelemetry

Instrument widely used libraries rather than having each app instrument itself.
private static final Tracer tracer = GlobalOpenTelemetry.getTracer("demo-db-client", "0.1.0-beta1");

private Response selectWithTracing(Query query) {
    // check out conventions for guidance on span names and attributes
    Span span = tracer.spanBuilder(String.format("SELECT %s.%s", dbName, collectionName)).
        setSpanKind(SpanKind.CLIENT).
        setAttribute("db.name", dbName).
        ... .startSpan();

    // makes span active and allows correlating logs and nest spans
    try (Scope unused = span.makeCurrent()) {
        Response response = query.runWithRetries();
        if (response.isSuccessful()) {
            span.setStatus(StatusCode.OK);
        }

        if (span.isRecording()) {
            // populate response attributes for response codes and other information
        }
    } catch (Exception e) {
        span.recordException(e);
        span.setStatus(StatusCode.ERROR, e.getClass().getSimpleName());
        throw e;
    } finally {
        span.end();
    }
}
Example span: OpenTelemetry

```json
{
    "trace_id": "7ba9f33312b3dbb8b2c2c62bb7abe2d",
    "parent_id": "",
    "span_id": "086e837470e381e",
    "name": "/v1/sys/health",
    "start_time": "2021-10-22 16:04:01.209514132 +0000 UTC",
    "end_time": "2021-10-22 16:04:01.209514132 +0000 UTC",
    "status_code": "STATUS_CODE_OK",
    "status_message": "",
    "attributes": {
        "net.transport": "IP.TCP",
        "net.peer.ip": "172.17.0.1",
        "net.peer.port": "51820",
        "net.host.ip": "10.177.2.152",
        "net.host.port": "26040",
        "http.method": "GET",
        "http.target": "/v1/sys/health",
        "http.server_name": "mortar-gateway",
        "http.route": "/v1/sys/health",
        "http.user_agent": "Consul Health Check",
        "http.scheme": "http",
        "http.host": "10.177.2.152:26040",
        "http.flavor": "1.1"
    },
    "events": [
        {
            "name": "",
            "message": "OK!",
            "timestamp": "2021-10-22 16:04:01.209512872 +0000 UTC"
        }
    ]
}
```
Example span: Dapper
Combining spans into traces

• Carry all spans in headers between microservices?
  • Baggage: keep it small

• Carry parent-child ID relationships

• Trace IDs: probabilistically unique integers

• Actual mechanism of propagation: HTTP/RPC protocol headers (inter-process); function call arguments (intra-process)
Trace: putting spans together

```
{
  "name": "Hello-Greetings",
  "context": {
    "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "0x5fb397be34d26b51",
  },
  "parent_id": "0x051581bf3cb55c13",
  "start_time": "2022-04-29T18:52:58.114304Z",
  "end_time": "2022-04-29T22:52:58.114561Z",
  "attributes": {
    "http.route": "some_route1"
  },
  "events": [
    {
      "name": "hey there!",
    }
  ]
}
```

```
{
  "name": "Hello-Salutations",
  "context": {
    "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "0x935e4f51e1abe1c2",
  },
  "parent_id": "0x051581bf3cb55c13",
  "start_time": "2022-04-29T18:52:58.114492Z",
  "end_time": "2022-04-29T18:52:58.114631Z",
  "attributes": {
    "http.route": "some_route2"
  },
  "events": [
    {
      "name": "hey there!",
    }
  ]
}
```

```
{
  "name": "Hello",
  "context": {
    "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
    "span_id": "0x051581bf3cb55c13",
  },
  "parent_id": null,
  "start_time": "2022-04-29T18:52:58.114201Z",
  "end_time": "2022-04-29T18:52:58.114682Z",
  "attributes": {
    "http.route": "some_route3"
  },
  "events": [
    {
      "name": "Guten Tag!",
    }
  ]
}
```
Visualizing traces
Visualizing traces
Google Dapper system

![Diagram of the Google Dapper system]

<table>
<thead>
<tr>
<th>trace id</th>
<th>span 12</th>
<th>span 23</th>
<th>span 34</th>
<th>span 45</th>
<th>span 56</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>123456</td>
<td>nil</td>
<td>nil</td>
<td>&lt;data&gt;</td>
<td>&lt;data&gt;</td>
<td>nil</td>
<td>...</td>
</tr>
<tr>
<td>246802</td>
<td>&lt;data&gt;</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>&lt;data&gt;</td>
<td>...</td>
</tr>
<tr>
<td>357913</td>
<td>nil</td>
<td>&lt;data&gt;</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

(Central Bigtable repository for trace data)
Jaeger trace collector
Jaeger trace example
Jaeger trace example
Monitoring overheads

- **Instrumentation in the critical path**: latency and throughput issues
- Sample aggressively
  - Tradeoff with accuracy
- Head-based sampling vs. tail-based sampling
- Reduce baggage

<table>
<thead>
<tr>
<th>Sampling frequency</th>
<th>Avg. Latency (% change)</th>
<th>Avg. Throughput (% change)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1/1</td>
<td>16.3%</td>
<td>-1.48%</td>
</tr>
<tr>
<td>1/2</td>
<td>9.40%</td>
<td>-0.73%</td>
</tr>
<tr>
<td>1/4</td>
<td>6.38%</td>
<td>-0.30%</td>
</tr>
<tr>
<td>1/8</td>
<td>4.12%</td>
<td>-0.23%</td>
</tr>
<tr>
<td>1/16</td>
<td>2.12%</td>
<td>-0.08%</td>
</tr>
<tr>
<td>1/1024</td>
<td>-0.20%</td>
<td>-0.06%</td>
</tr>
</tbody>
</table>

Table 2: The effect of different [non-adaptive] Dapper sampling frequencies on the latency and throughput of a Web search cluster. The experimental errors
Collection overheads

- Collection agents can take up resources
- Sample separately at the collector as well
- Sample to target # traces per unit time

<table>
<thead>
<tr>
<th>Process Count (per host)</th>
<th>Data Rate (per process)</th>
<th>Daemon CPU Usage (single CPU core)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>10K/sec</td>
<td>0.125%</td>
</tr>
<tr>
<td>10</td>
<td>200K/sec</td>
<td>0.267%</td>
</tr>
<tr>
<td>50</td>
<td>2K/sec</td>
<td>0.130%</td>
</tr>
</tbody>
</table>
Monitoring concerns

• Teasing out interactions with shared systems
  • e.g., distributed storage

• Integration with public cloud systems

• Combining system and application visibility
  • Uncovering bottlenecks deeper in the stack, e.g. TCP

• Batch processing applications
Outro
Summary

- Internet services have many building blocks
- Content delivery at the user edge
- Application design patterns within the data center
- Infrastructure support within the system
- Networking design to achieve high performance and agility
- Operational considerations
Where to go from here?

• Carry a deeper appreciation for supporting technologies

• Learn how to evaluate system designs
  • Understand and diagnose problems lower down the stack

• Build your own better infrastructure

• Research or pursue careers developing (on) these technologies