Monitoring

Lecture 13

Srinivas Narayana

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

Operations

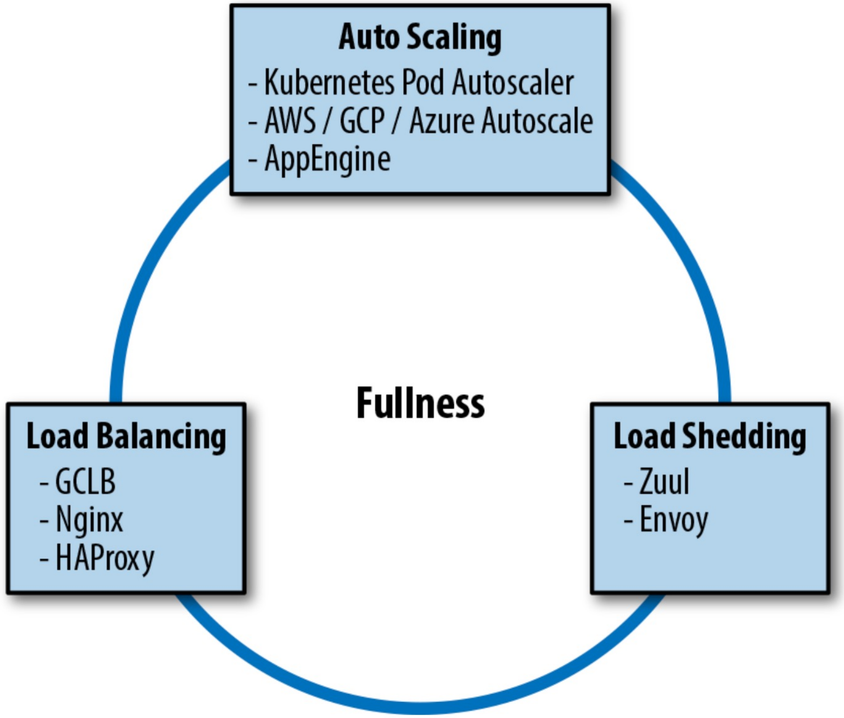
- How to run and manage an Internet service?
- Monitoring, security
- Load management
- Release engineering, canarying
- Crafting and maintaining SLOs
- People and processes
- Incident response, postmortems
- Designing and managing configurations
- …

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



Auto Scaling
- Kubernetes Pod Autoscaler
- AWS / GCP / Azure Autoscale
- AppEngine

Fullness

Load Balancing
- GCLB
- Nginx
- HAProxy

Load Shedding
- Zuul
- Envoy

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?

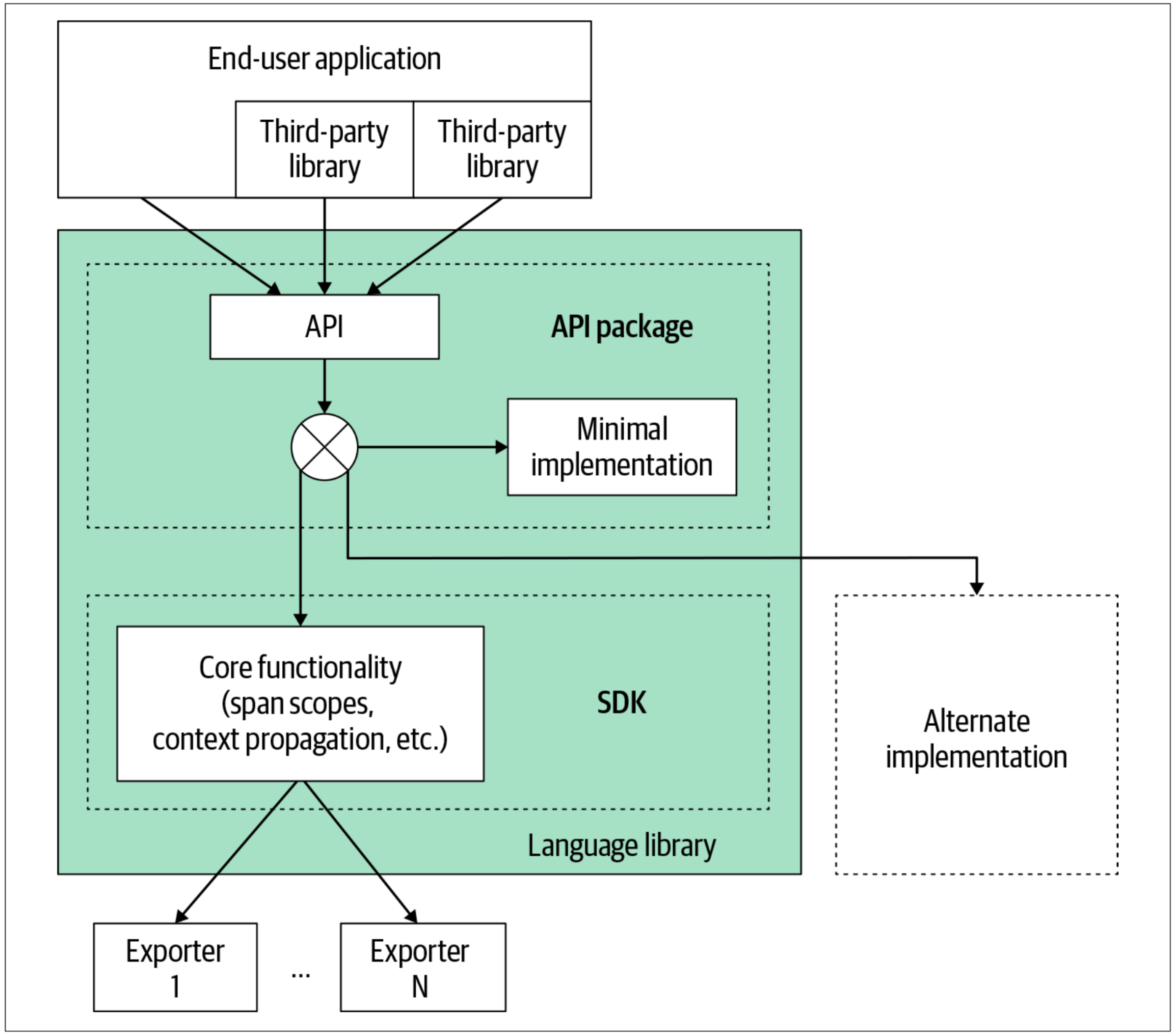
OpenTelemetry

Jaeger

Dapper

Instrumentation: OpenTelemetry

Instrument widely used libraries rather than having each app instrument itself.



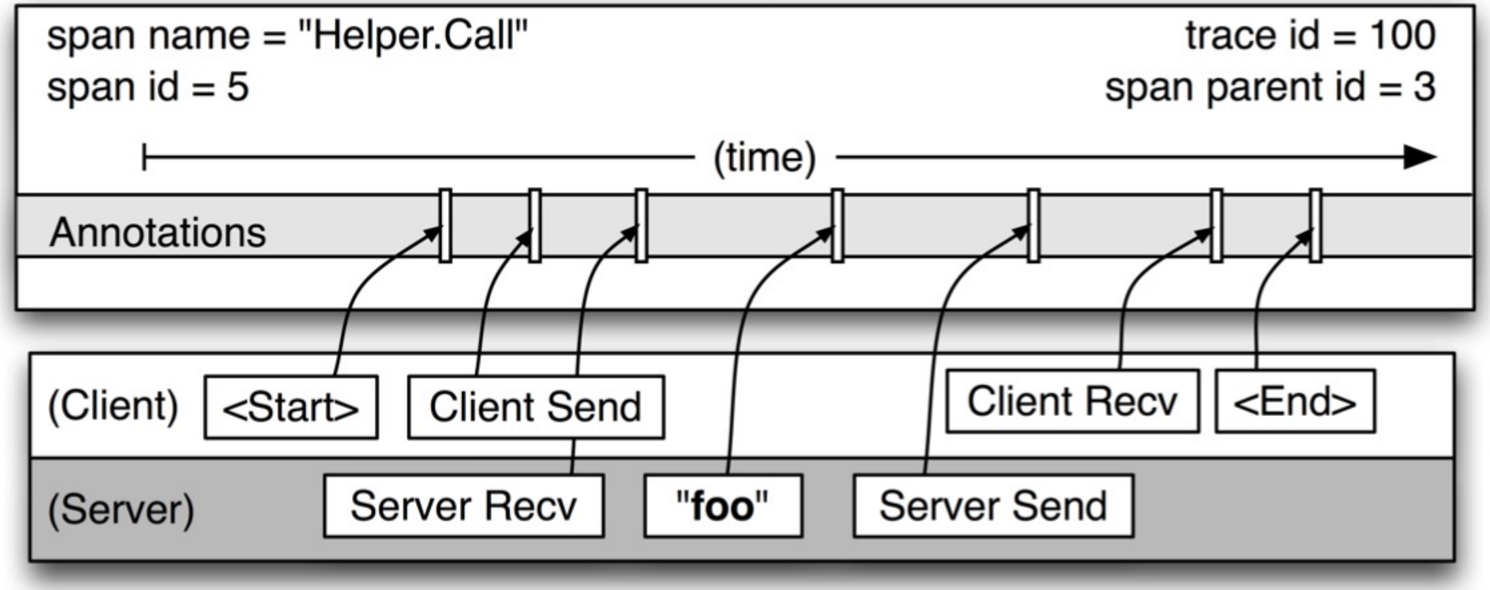
Instrumentation: OpenTelemetry

```java
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

```
{
  "trace_id": "7bba9f33312b3dbb8b2c2c62bb7abe2d",
  "parent_id": "",
  "span_id": "086e83747d0e381e",
  "name": "/v1/sys/health",
  "start_time": "2021-10-22 16:04:01.209458162 +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"
    }
  ]
}
```

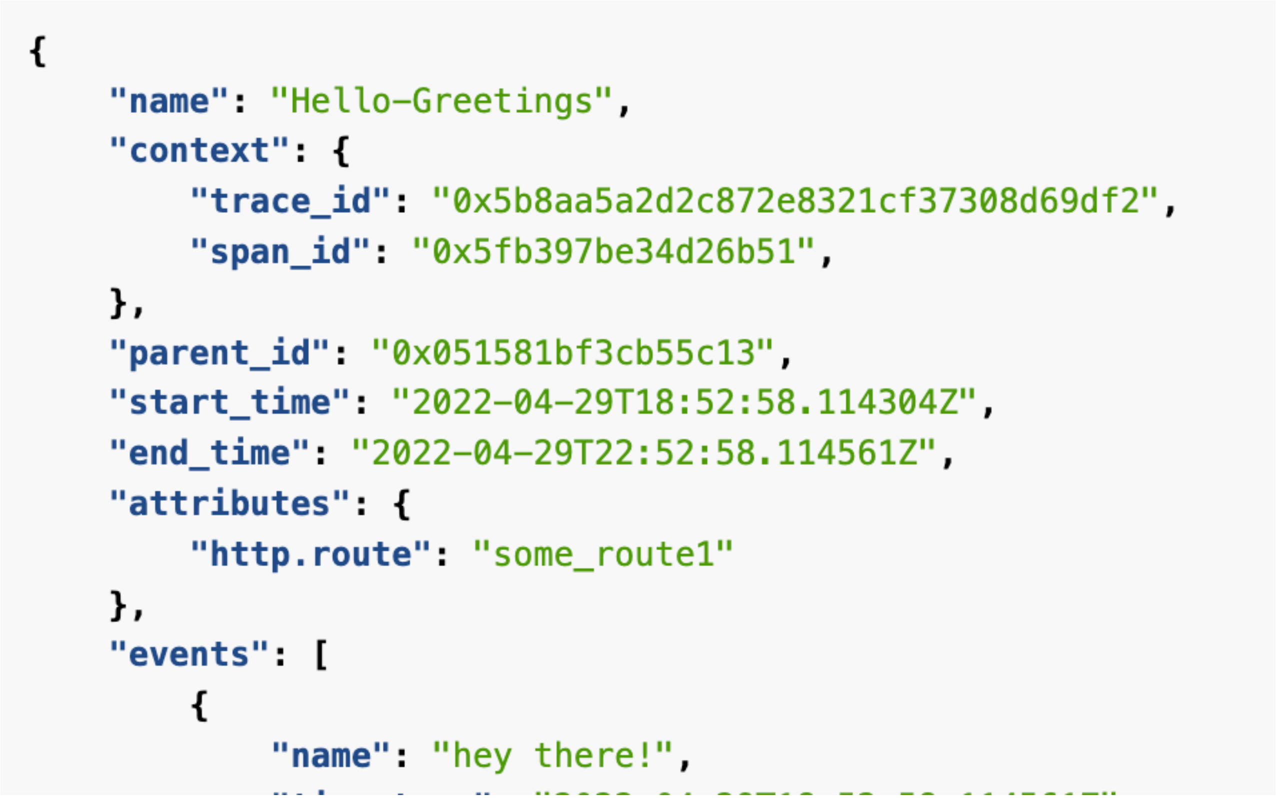
Example span: Dapper



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

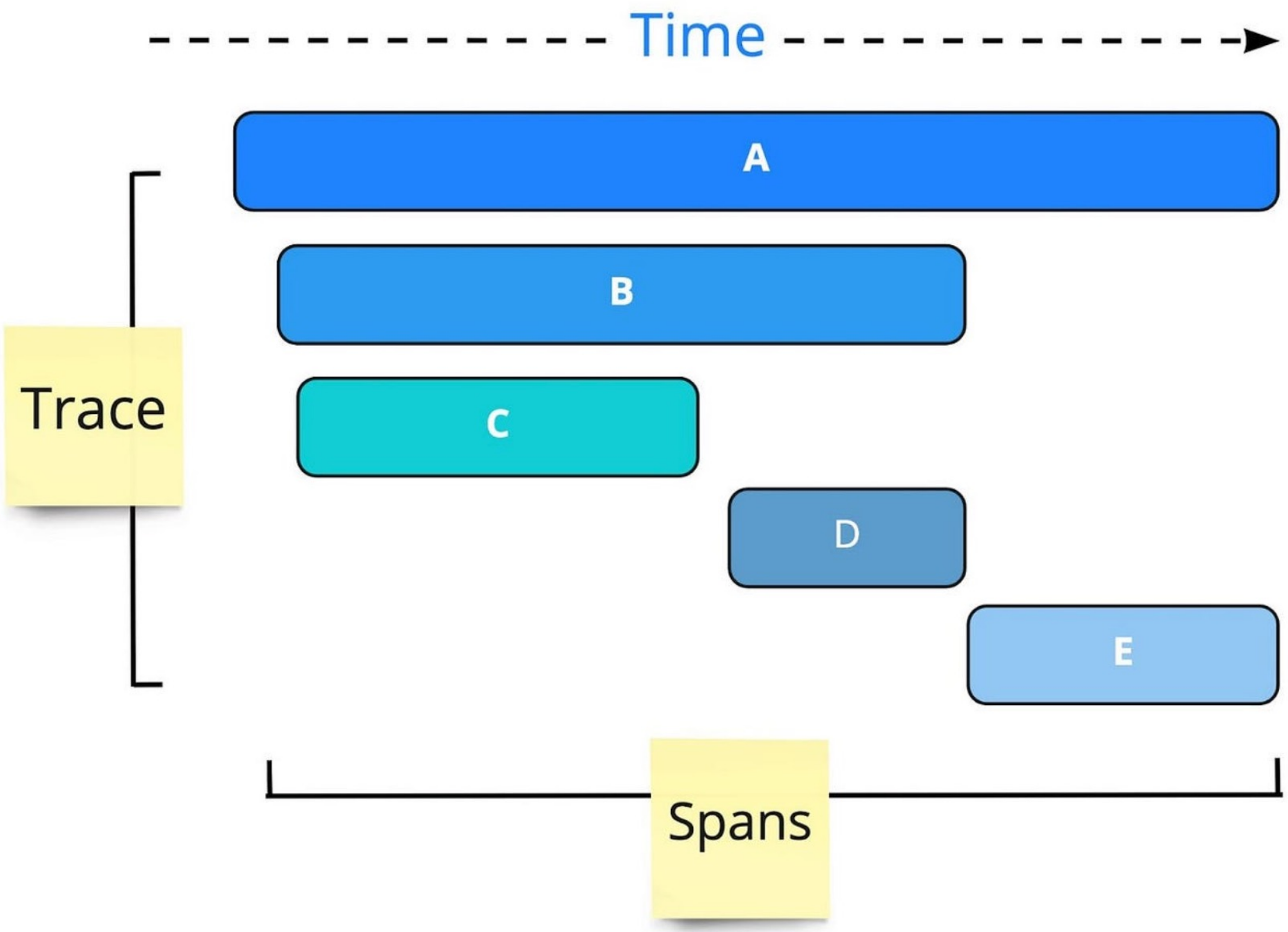


```
{
    "name": "Hello-Greetings",
    "context": {
        "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
        "span_id": "0x5fb397be34d26b51",
    },
    "parent_id": "0x051581bf3cb55c13",
    "start_time": "2022-04-29T18:52:58.114304Z",
    "end_time": "2022-04-29T22:52:58.114561Z",
    "attributes":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

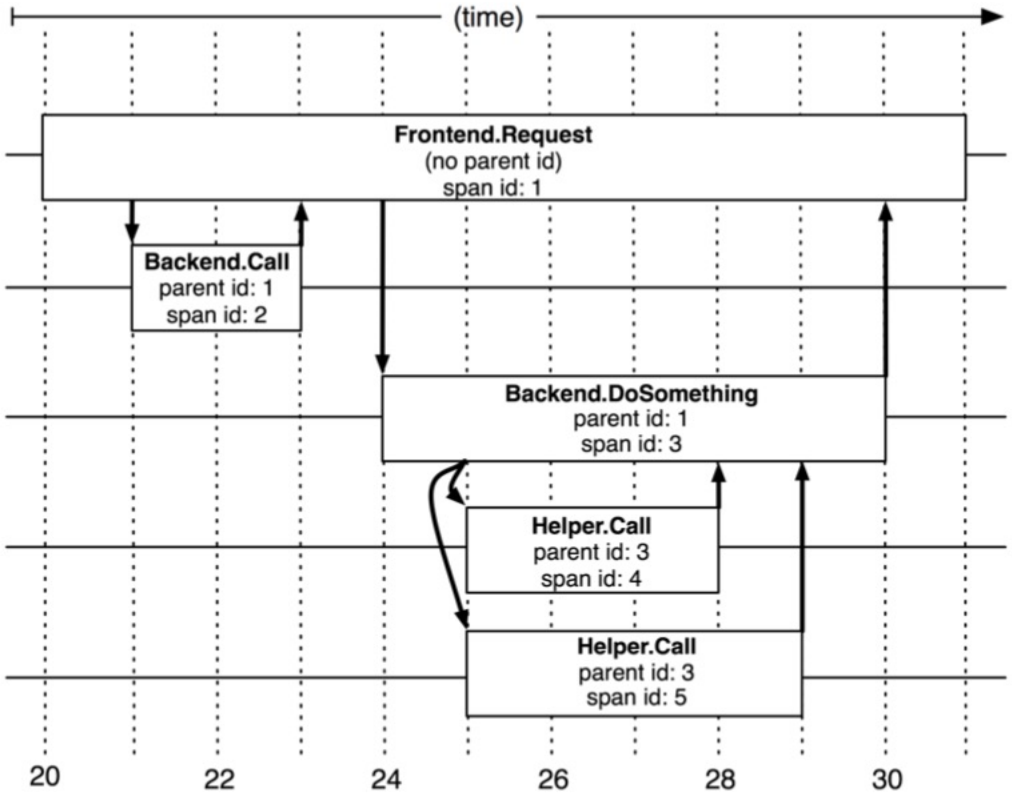
```
{
    "name": "Hello-Salutations",
    "context": {
        "trace_id": "0x5b8aa5a2d2c872e8321cf37308d69df2",
        "span_id": "0x93564f51e1abe1c2",
    },
    "parent_id": "0x051581bf3cb55c13",
    "start_time": "2022-04-29T18:52:58.114492Z",
    "end_time": "2022-04-29T18:52:58.114631Z",
    "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": "2022-04-29T18:52:58.114687Z",
    "attributes": {
        "http.route": "some_route3"
    },
    "events": [
        {
            "name": "Guten Tag!",
```

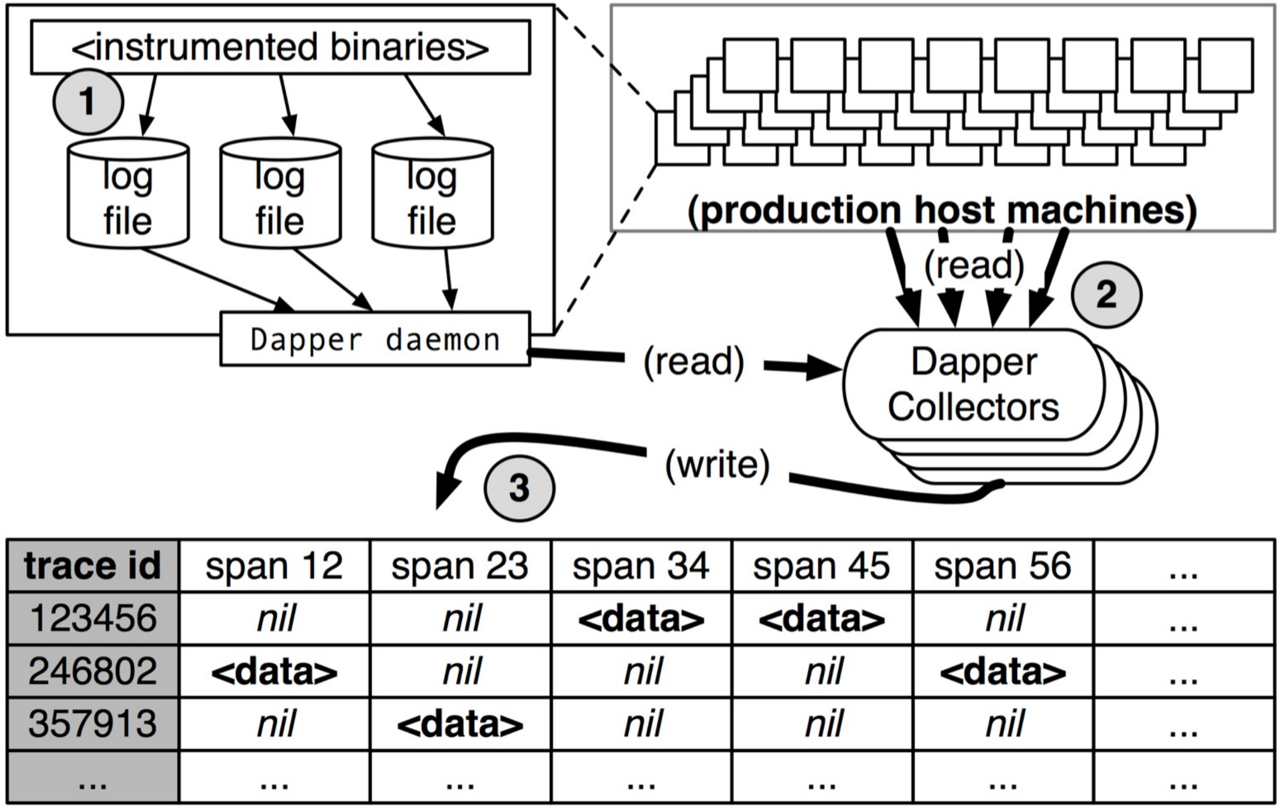
Visualizing traces



Visualizing traces



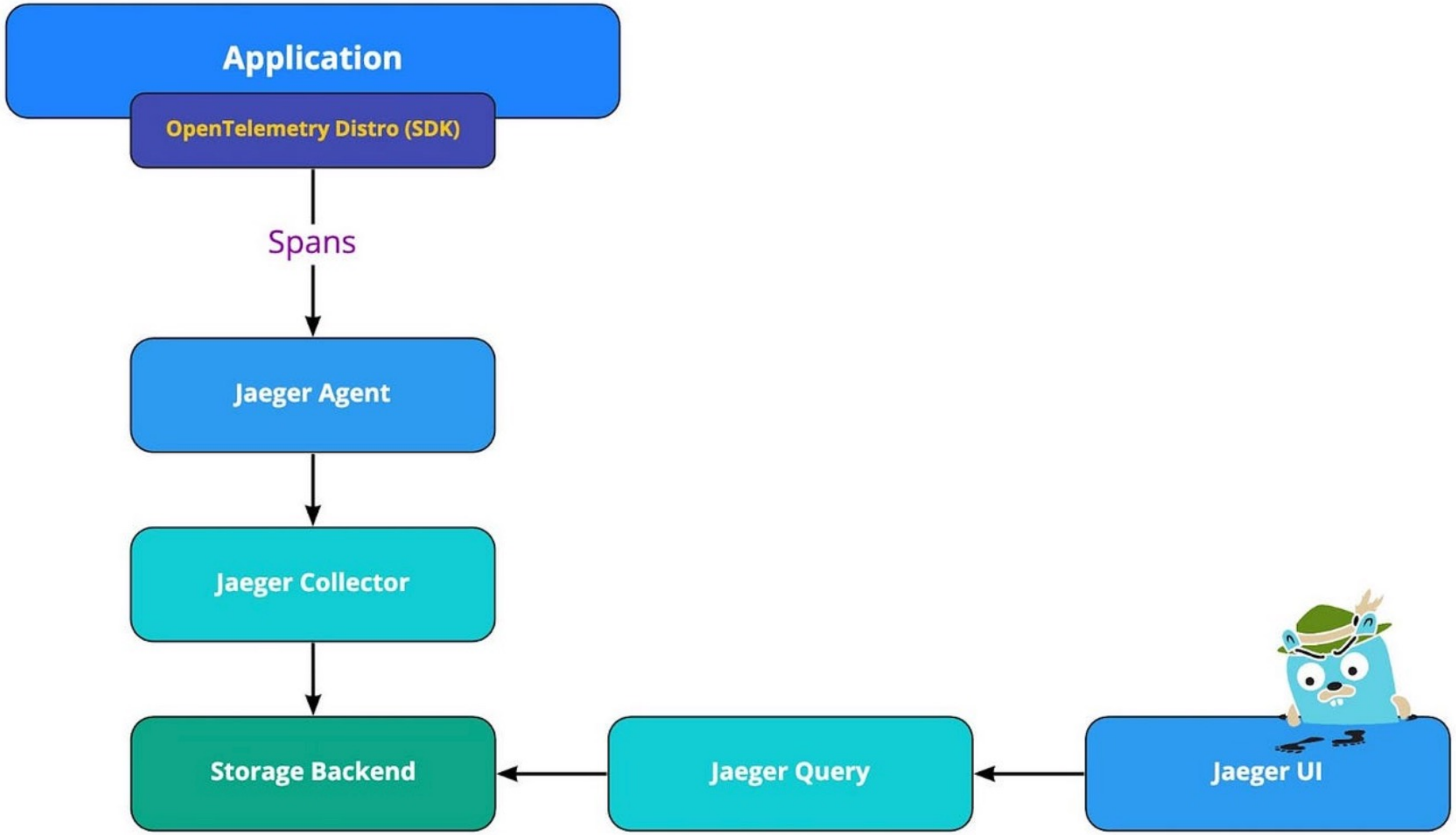
Google Dapper system



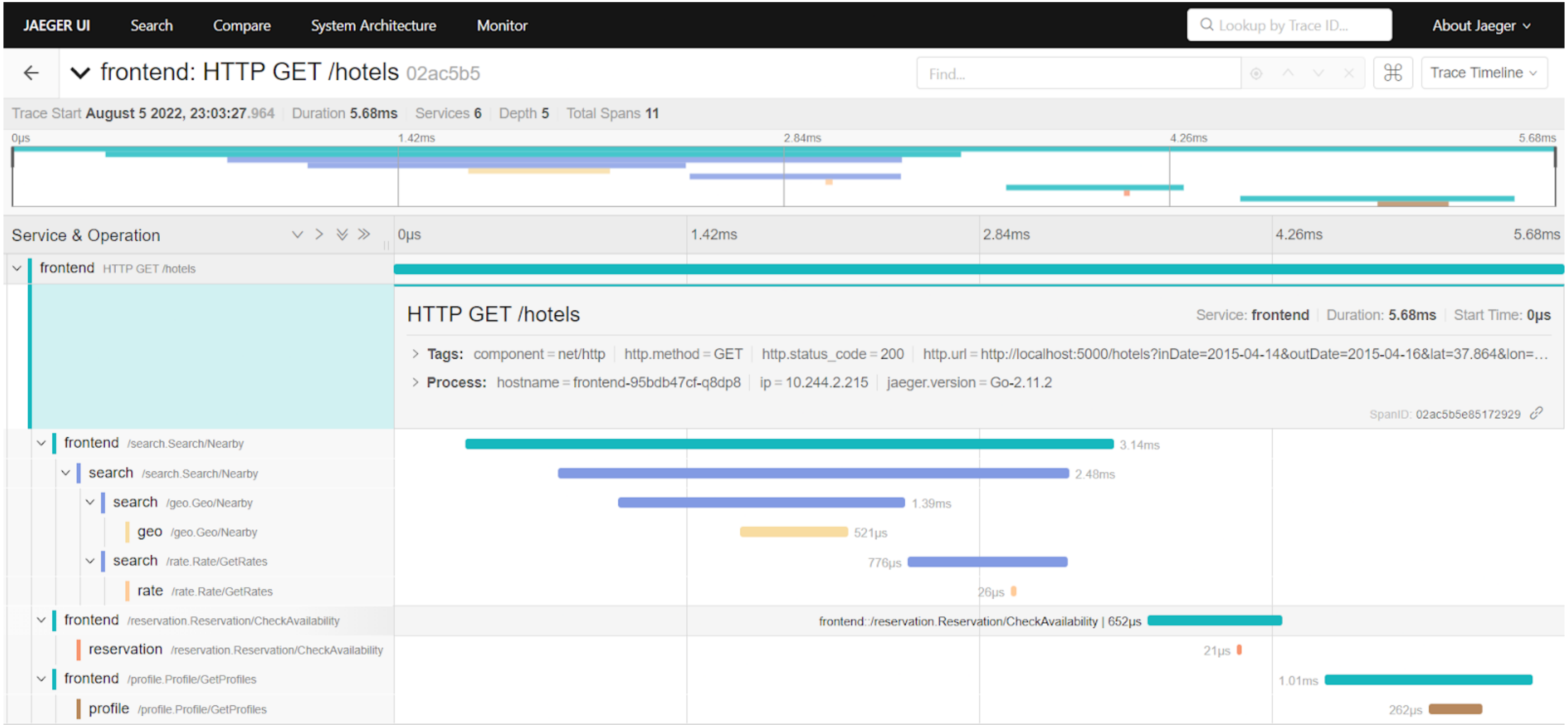
trace id	span 12	span 23	span 34	span 45	span 56	...
123456	*nil*	*nil*	**<data>**	**<data>**	*nil*	...
246802	**<data>**	*nil*	*nil*	*nil*	**<data>**	...
357913	*nil*	**<data>**	*nil*	*nil*	*nil*	...
...	...	...	...	...	...	...

(Central Bigtable repository for trace data)

Jaeger trace collector



Jaeger trace example



frontend: HTTP GET /hotels 02ac5b5

Trace Start **August 5 2022, 23:03:27**.964 | Duration **5.68ms** | Services **6** | Depth **5** | Total Spans **11**

frontend HTTP GET /hotels

HTTP GET /hotels

Service: **frontend** | Duration: **5.68ms** | Start Time: **0μs**

Tags: component = net/http | http.method = GET | http.status_code = 200 | http.url = http://localhost:5000/hotels?inDate=2015-04-14&outDate=2015-04-16&lat=37.864&lon=...

Process: hostname = frontend-95bdb47cf-q8dp8 | ip = 10.244.2.215 | jaeger.version = Go-2.11.2

SpanID: 02ac5b5e85172929

frontend /search.Search/Nearby 3.14ms

search /search.Search/Nearby 2.48ms

search /geo.Geo/Nearby 1.39ms

geo /geo.Geo/Nearby 521μs

search /rate.Rate/GetRates 776μs

rate /rate.Rate/GetRates 26μs

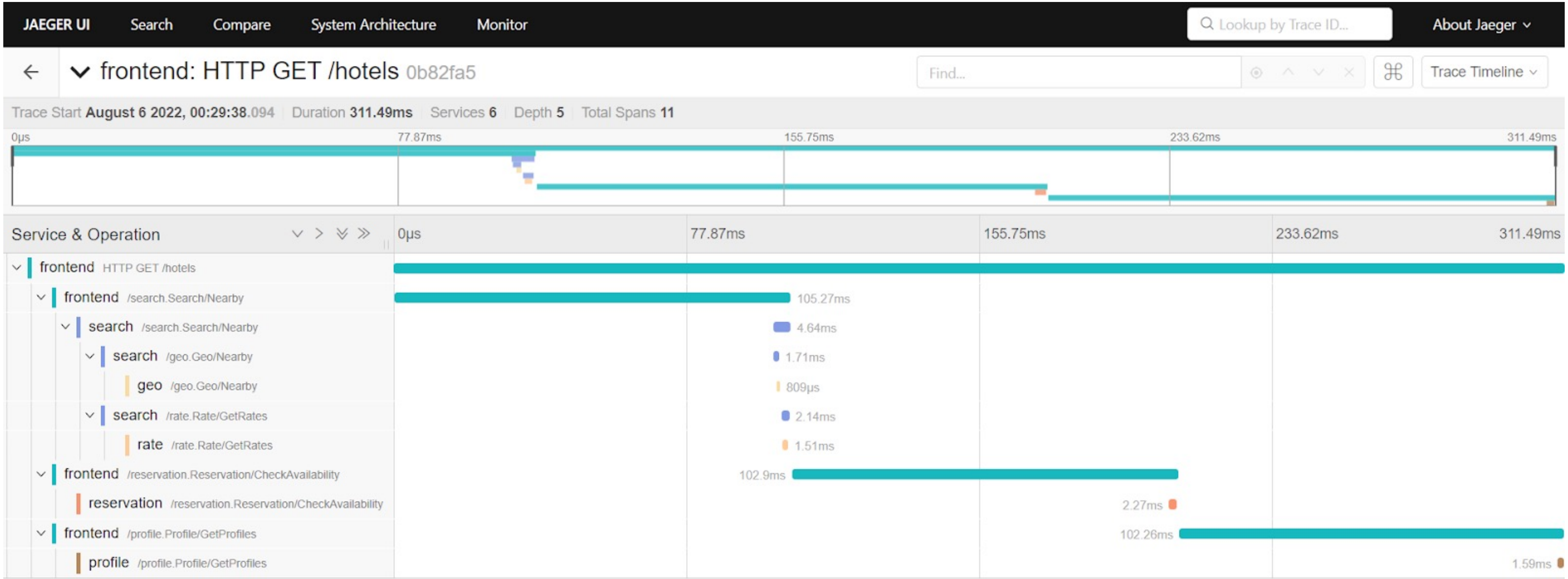
frontend /reservation.Reservation/CheckAvailability frontend::/reservation.Reservation/CheckAvailability | 652μs

reservation /reservation.Reservation/CheckAvailability 21μs

frontend /profile.Profile/GetProfiles 1.01ms

profile /profile.Profile/GetProfiles 262μs

Jaeger trace example



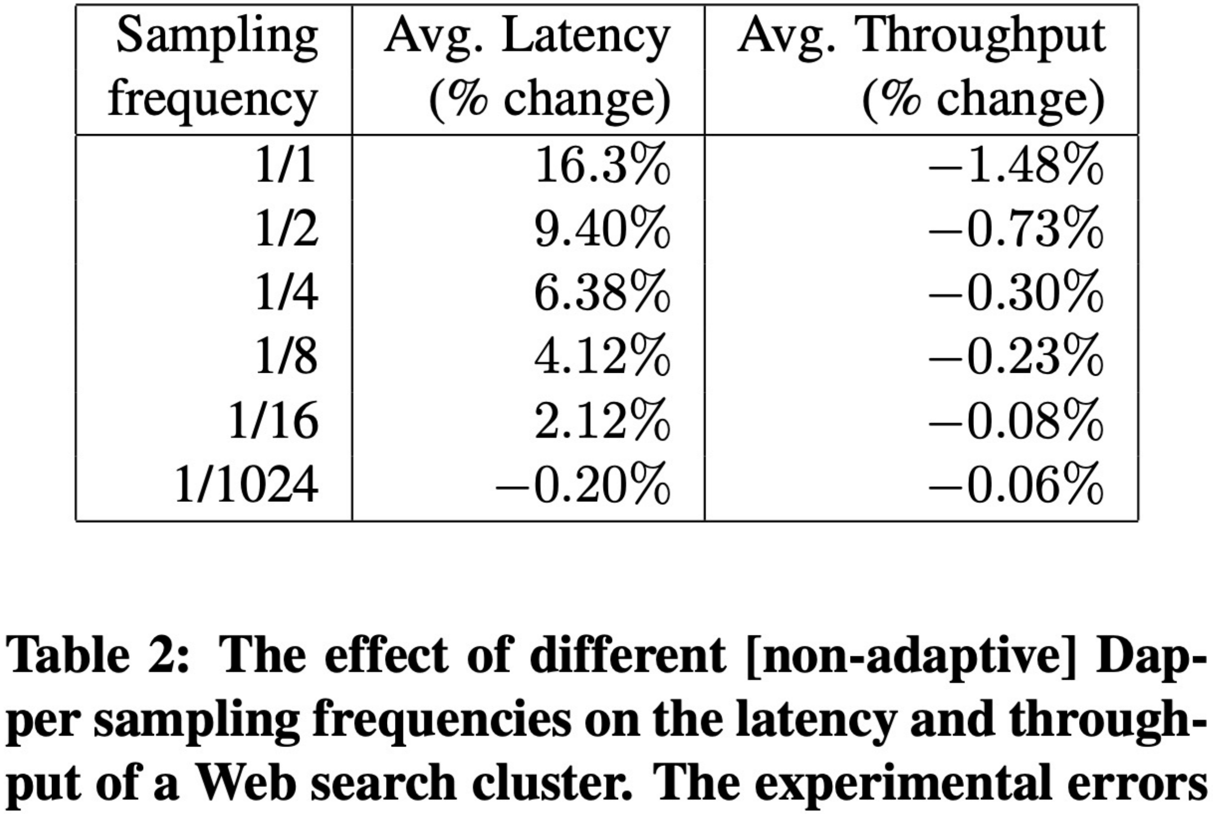
frontend: HTTP GET /hotels 0b82fa5

Trace Start **August 6 2022, 00:29:38**.094 Duration **311.49ms** Services **6** Depth **5** Total Spans **11**

frontend HTTP GET /hotels
frontend /search.Search/Nearby 105.27ms
search /search.Search/Nearby 4.64ms
search /geo.Geo/Nearby 1.71ms
geo /geo.Geo/Nearby 809μs
search /rate.Rate/GetRates 2.14ms
rate /rate.Rate/GetRates 1.51ms
frontend /reservation.Reservation/CheckAvailability 102.9ms
reservation /reservation.Reservation/CheckAvailability 2.27ms
frontend /profile.Profile/GetProfiles 102.26ms
profile /profile.Profile/GetProfiles 1.59ms

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

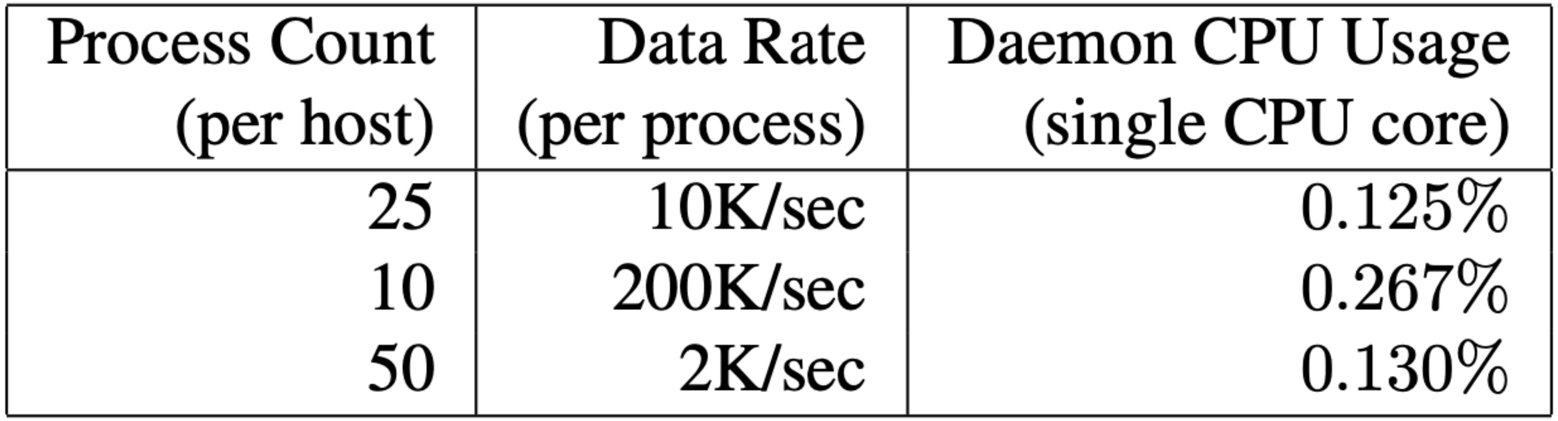


Sampling frequency	Avg. Latency (% change)	Avg. Throughput (% change)
1/1	16.3%	−1.48%
1/2	9.40%	−0.73%
1/4	6.38%	−0.30%
1/8	4.12%	−0.23%
1/16	2.12%	−0.08%
1/1024	−0.20%	−0.06%

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



Process Count (per host)	Data Rate (per process)	Daemon CPU Usage (single CPU core)
25	10K/sec	0.125%
10	200K/sec	0.267%
50	2K/sec	0.130%

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