Performance Debugging: Finding Bottlenecks in Distributed Systems

Christian Grabowski - Backend Engineer
Github: cpg1111 Twitter: @christiang0817
• About Me

• Backend Engineer @ NS1
• Total Linux Nerd
• <3 open source
About NS1

- We run a global network of authoritative DNS servers
- This network is made up of many different services that each provide a unique function in order to deliver our product
- These services are distributed globally across 25 POPs
- We’re always receiving traffic, and spikes in traffic will of course happen, so performance and reliability are important to us
In a nutshell...

- Distributed systems are complicated, and as the system scales to handle the growth of a company, the complexity grows along with it.
- As a system scales, bottlenecks are bound to occur, but with this complexity, it becomes more and more difficult to identify these bottlenecks.
- This can be frustrating, because logically speaking, your system is working, but it’s just not cutting it.
In A Little More Detail

- Where do you start?
  - There are so many metrics to look at
  - There are so many tools out there for performance debugging
- How do I identify the bottleneck in a multiple services of a distributed system?
- How do I identify the bottleneck in a single process?
  - Is it my code? Is it a Library? Is it the OS?

There are many ways to approach the identification of said issues. Based on my experience, here are a couple cases that will aim to answer these questions.
Case 1: Metrics Aggregation Problems
Case 1

Initial Issue
Case 1

Initial Issue

- A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster
Case 1

Initial Issue

- A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster
- This required a unique set of tags for each metric, so we add a tag per process
Case 1

Initial Issue

- A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster
- This required a unique set of tags for each metric, so we add a tag per process
- Throughput wasn’t ideal
Case 1

Initial Issue

• A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster
• This required a unique set of tags for each metric, so we add a tag per process
• Throughput wasn’t ideal
• It consisted of too many Python processes
Case 1

Initial Issue

• A service aggregating query metrics from our POPs and placing it into a OpenTSDB cluster
• This required a unique set of tags for each metric, so we add a tag per process
• Throughput wasn’t ideal
• It consisted of too many Python processes
• More timeseries, more problems
Case 1
Case 1

Hypothesised Solution (came up with a few, but ultimately)
Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
Case 1

Hypothesised Solution (came up with a few, but ultimately)

• Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
• From Python to Go
Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines
Case 1

Hypothesised Solution (came up with a few, but ultimately)

• Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process

• From Python to Go
  • From Twisted to Goroutines
  • Easier in-process horizontal scaling
Case 1

Hypothesised Solution (came up with a few, but ultimately)

• Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process

• From Python to Go
  • From Twisted to Goroutines
  • Easier in-process horizontal scaling
  • Better supported drivers to our databases and message queue
Case 1

Hypothesised Solution (came up with a few, but ultimately)

• Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process

• From Python to Go
  • From Twisted to Goroutines
  • Easier in-process horizontal scaling
  • Better supported drivers to our databases and message queue

• Aggregate in-memory
Case 1

Hypothesised Solution (came up with a few, but ultimately)

• Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
• From Python to Go
  • From Twisted to Goroutines
  • Easier in-process horizontal scaling
  • Better supported drivers to our databases and message queue
• Aggregate in-memory
  • First approach was a COW btree, but that turned out to cause a lot of memory issues
Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines
  - Easier in-process horizontal scaling
  - Better supported drivers to our databases and message queue
- Aggregate in-memory
  - First approach was a COW btree, but that turned out to cause a lot of memory issues
  - Second approach was a sorted array of metrics sorted by last update that coincided with a map of metric name = aggregated value
Case 1

Hypothesised Solution (came up with a few, but ultimately)

- Have the message queue consistently hash the query metrics by customer, such that any unique set of tags was ensured to be delivered to the same aggregation process
- From Python to Go
  - From Twisted to Goroutines
  - Easier in-process horizontal scaling
  - Better supported drivers to our databases and message queue
- Aggregate in-memory
  - First approach was a COW btree, but that turned out to cause a lot of memory issues
  - Second approach was a sorted array of metrics sorted by last update that coincided with a map of metric name = aggregated value
- Sounds great right?
Case 1

What was wrong?
Case 1

What was wrong?

- There was a memory leak directly related to each metric received in the aggregation process
Case 1

What was wrong?

• There was a memory leak directly related to each metric received in the aggregation process
• The initial data structure used for aggregation was using copy-on-write, so that needed to be replaced
• **Case 1**

What was wrong?

• There was a memory leak directly related to each metric received in the aggregation process
• The initial data structure used for aggregation was using copy-on-write, so that needed to be replaced
• At this level of throughput, this becomes a problem quickly, with rapidly increasing memory utilization.
Case 1

What was wrong?

• There was a memory leak directly related to each metric received in the aggregation process
• The initial data structure used for aggregation was using copy-on-write, so that needed to be replaced
• At this level of throughput, this becomes a problem quickly, with rapidly increasing memory utilization.
• So what caused this?
Case 1

Steps taken
Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   • In our case, we created in-memory counters that would be sampled off of an http endpoint
GET /metrics?snapshot=true

[
  {
    "name": "msgs.recv",
    "time": 1257894000,
    "stags": {
      "host": "a.host"
    },
    "utags": [
      "bulk_size"
    ],
    "T": [20],
    "V": 18500,
    "N": 8
  },
  {
    "name": "metrics.recv",
    "time": 1257894000,
    "stags": {
      "host": "a.host"
    },
    "utags": [
      "type"
    ],
    "T": [
      "by_customer",
      "by_zone",
      "by_record"
    ],
    "V": 370000,
    "N": 160
  }
]
Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   • In our case, we created in-memory counters that would be sampled off of an http endpoint
   • We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)
Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   - In our case, we created in-memory counters that would be sampled off of an http endpoint
   - We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)

2. Reproduced the symptoms
Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   • In our case, we created in-memory counters that would be sampled off of an http endpoint
   • We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)
2. Reproduced the symptoms
   • Recreated production traffic in a controlled testing environment
Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   • In our case, we created in-memory counters that would be sampled off of an http endpoint
   • We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)

2. Reproduced the symptoms
   • Recreated production traffic in a controlled testing environment

3. Profile the heap allocations of a single of the aggregation process
   • First with Go’s built-in profiler and pprof visualizations
Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   - In our case, we created in-memory counters that would be sampled off of an http endpoint
   - We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)

2. Reproduced the symptoms
   - Recreated production traffic in a controlled testing environment

3. Profile the heap allocations of a single of the aggregation process
   - First with Go’s built-in profiler and pprof visualizations
   - Switched to Go-Torch (https://github.com/uber/go-torch)
Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   • In our case, we created in-memory counters that would be sampled off of an http endpoint
   • We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)

2. Reproduced the symptoms
   • Recreated production traffic in a controlled testing environment

3. Profile the heap allocations of a single of the aggregation process
   • First with Go’s built-in profiler and pprof visualizations
   • Switched to Go-Torch ([https://github.com/uber/go-torch](https://github.com/uber/go-torch))
   • Used eBPF with bcc’s memleak.py (slightly modified) tool for greater detail ([https://github.com/iovisor/bcc](https://github.com/iovisor/bcc))
Case 1

Steps taken

1. Add operational metrics for greater visibility into functionality and events of the system
   • In our case, we created in-memory counters that would be sampled off of an http endpoint
   • We then sampled these metrics on an interval and placed them into a separate OpenTSDB cluster (now ELK)

2. Reproduced the symptoms
   • Recreated production traffic in a controlled testing environment

3. Profile the heap allocations of a single of the aggregation process
   • First with Go’s built-in profiler and pprof visualizations
   • Switched to Go-Torch (https://github.com/uber/go-torch)
   • Used eBPF with bcc’s memleak.py (slightly modified) tool for greater detail (https://github.com/iovisor/bcc)

4. Used Go’s Benchmark testing to be able to compare changes
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BenchmarkBTree_Insert1-8</td>
<td>2000000000</td>
<td>753</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Insert10-8</td>
<td>3000000</td>
<td>5460</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Insert100-8</td>
<td>2000000</td>
<td>67481</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Insert1000-8</td>
<td>2000000</td>
<td>831932</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Insert10000-8</td>
<td>2000000</td>
<td>9863654</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Insert100000-8</td>
<td>2000000</td>
<td>127574333</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Insert1000000-8</td>
<td>1</td>
<td>1890309427</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Remove1-8</td>
<td>3000000</td>
<td>5.05</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Remove10-8</td>
<td>3000000</td>
<td>53.9</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Remove100-8</td>
<td>3000000</td>
<td>543</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Remove1000-8</td>
<td>3000000</td>
<td>5841</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Remove10000-8</td>
<td>2000000</td>
<td>691415</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkBTree_Remove100000-8</td>
<td>2000000</td>
<td>4412963369</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Insert1-8</td>
<td>3000000</td>
<td>483</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Insert10-8</td>
<td>3000000</td>
<td>4486</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Insert100-8</td>
<td>3000000</td>
<td>52976</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Insert1000-8</td>
<td>2000000</td>
<td>672284</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Insert10000-8</td>
<td>2000000</td>
<td>7843585</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Insert100000-8</td>
<td>2000000</td>
<td>104467790</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Remove1-8</td>
<td>1</td>
<td>1597383476</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Remove10-8</td>
<td>3000000</td>
<td>5.39</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Remove100-8</td>
<td>3000000</td>
<td>51.6</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Remove1000-8</td>
<td>3000000</td>
<td>519</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Remove10000-8</td>
<td>3000000</td>
<td>5163</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Remove100000-8</td>
<td>3000000</td>
<td>54560</td>
<td>ns/op</td>
</tr>
<tr>
<td>BenchmarkLLRB_Remove1000000-8</td>
<td>2000000</td>
<td>673192</td>
<td>ns/op</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>BenchmarkMapAggPut1-4</td>
<td>1000000</td>
<td>52489</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>1658</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggPut10-4</td>
<td>50000</td>
<td>1013205</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>8763</td>
<td>140</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggPut100-4</td>
<td>10000</td>
<td>522884</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>78651</td>
<td>1228</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggPut1000-4</td>
<td>1000</td>
<td>1343539</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>777050</td>
<td>12020</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggPut10000-4</td>
<td>100</td>
<td>13487897</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>776138</td>
<td>120920</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggPut100000-4</td>
<td>10</td>
<td>133272840</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>7766172</td>
<td>1280921</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggFlush1-4</td>
<td>5000000</td>
<td>14246</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>2272</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggFlush10-4</td>
<td>2000000</td>
<td>96255</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>11789</td>
<td>175</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggFlush100-4</td>
<td>190000</td>
<td>947855</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>83108</td>
<td>1264</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggFlush1000-4</td>
<td>1800000</td>
<td>3805475</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>788817</td>
<td>12803</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggFlush10000-4</td>
<td>1000000</td>
<td>14738426</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>7846638</td>
<td>120964</td>
<td></td>
</tr>
<tr>
<td>BenchmarkMapAggFlush100000-4</td>
<td>10</td>
<td>133809511</td>
<td>ns/op</td>
</tr>
<tr>
<td>allocs/op</td>
<td>78407281</td>
<td>1280651</td>
<td></td>
</tr>
</tbody>
</table>
Case 1

The largest hurdle
Case 1

The largest hurdle
- Recreating production traffic
Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
Case 1

The largest hurdle

• Recreating production traffic
  • The rate of messages
  • The variety of metrics
    • The random distribution of customers, zones, records and POPs
Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software
Case 1

The largest hurdle

• Recreating production traffic
  • The rate of messages
  • The variety of metrics
    • The random distribution of customers, zones, records and POPs
  • First approach → build some load testing software
    • Randomly set customer, zones, records and POPs
Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software
    - Randomly set customer, zones, records and POPs
    - Spray a bunch of messages at the message queue
Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software
    - Randomly set customer, zones, records and POPs
    - Spray a bunch of messages at the message queue
  - Second approach → have our message queue copy production messages into the test environment
Case 1

The largest hurdle

• **Recreating production traffic**
  • The rate of messages
  • The variety of metrics
    • The random distribution of customers, zones, records and POPs
  • First approach → build some load testing software
    • Randomly set customer, zones, records and POPs
    • Spray a bunch of messages at the message queue
  • Second approach → have our message queue copy production messages into the test environment
    • This required a lot of configuration
Case 1

The largest hurdle

- Recreating production traffic
  - The rate of messages
  - The variety of metrics
    - The random distribution of customers, zones, records and POPs
  - First approach → build some load testing software
    - Randomly set customer, zones, records and POPs
    - Spray a bunch of messages at the message queue
  - Second approach → have our message queue copy production messages into the test environment
    - This required a lot of configuration
    - We were able to throttle the amount of traffic well
Case 1

What Worked
Case 1

What Worked

- Pprof + Go’s built-in profiling tools ← Incredibly valuable
Case 1

What Worked

- Pprof + Go’s built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions
Case 1

What Worked

- Pprof + Go's built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions
- eBPF
Case 1

What Worked

- Pprof + Go’s built-in profiling tools ← Incredibly valuable
- Benchmark tests around important functions
- eBPF
- Operational metrics collecting
<table>
<thead>
<tr>
<th>What Worked</th>
<th>What Did Not Work</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Pprof + Go’s built-in profiling tools ← Incredibly valuable</td>
<td></td>
</tr>
<tr>
<td>• Benchmark tests around important functions</td>
<td></td>
</tr>
<tr>
<td>• eBPF</td>
<td></td>
</tr>
<tr>
<td>• Operational metrics collecting</td>
<td></td>
</tr>
</tbody>
</table>

**Case 1**
Case 1

What Worked

• Pprof + Go’s built-in profiling tools ← Incredibly valuable
• Benchmark tests around important functions
• eBPF
• Operational metrics collecting

What Did Not Work

• Being able to reproduce a production level load in a dev or testing environment
Case 1

What Worked

• Pprof + Go’s built-in profiling tools ← Incredibly valuable
• Benchmark tests around important functions
• eBPF
• Operational metrics collecting

What Did Not Work

• Being able to reproduce a production level load in a dev or testing environment
• Keeping a base comparison when making changes for the sake of performance
Case 1

What Worked

- Pprof + Go's built-in profiling tools – incredibly valuable
- Benchmark tests around important functions
- eBPF
- Operational metrics collecting

What Did Not Work

- Being able to reproduce a production level load in a dev or testing environment
- Keeping a base comparison when making changes for the sake of performance
- Should’ve changed the initial service more iteratively, instead of a full rewrite immediately
Case 2: Public REST api slowing down
Case 2

Initial Issue
Case 2

Initial Issue

- Response times of our public api service are increasing and some were timing out
Case 2

Initial Issue

- Response times of our public api service are increasing and some were timing out
- A planned large influx of records to our database had put a much greater amount of load on our database
Case 2

Initial Issue

- Response times of our public api service are increasing and some were timing out
- A planned large influx of records to our database had put a much greater amount of load on our database
- Not all of our queries were optimized
Case 2
Case 2

Steps Taken
Case 2

Steps Taken

1. Instrument the IO of the api process
Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
from nsone.util.tracing import ExportedTracingContext

TRACER = ExportedTracingContext(__name__)

def request(host, route, body):
    with TRACER.trace("{}\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\cdot\·
Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code
Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code

2. Profile on-CPU for potential blocking code
Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code

2. Profile on-CPU for potential blocking code
   - Twisted uses async io
Case 2

Steps Taken

1. **Instrument the IO of the api process**
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code

2. **Profile on-CPU for potential blocking code**
   - Twisted uses async io
   - Large blocks would hint towards blocking code
Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code

2. Profile on-CPU for potential blocking code
   - Twisted uses async io
   - Large blocks would hint towards blocking code

3. Utilize instrumentation around our database queries
Case 2

Steps Taken

1. Instrument the IO of the api process
   - Operational Metrics → How often is the api process querying the database? How often is a specific query being executed?
   - Distributed tracing → With small additions to our existing code, we could see how much time a block of code, or a round trip out to another service was taking
   - Combined both methods to provide monotonic averages of latency for spans of code

2. Profile on-CPU for potential blocking code
   - Twisted uses async io
   - Large blocks would hint towards blocking code

3. Utilize instrumentation around our database queries

4. Based on the instrumentation, we identified the slow routes and queries
Case 2

Results
Case 2

Results

• Through the use of the previously mentioned tools, we were able to identify specific routes, and queries that needed to be optimizes
Case 2

Results

• Through the use of the previously mentioned tools, we were able to identify specific routes, and queries that needed to be optimized.
• Some of those queries we were able to remove all together.
Case 2

Results

- Through the use of the previously mentioned tools, we were able to identify specific routes, and queries that needed to be optimized
- Some of those queries we were able to remove all together
- We were able to observe these improvements the moment they went to production
Case 2

Results

• Through the use of the previously mentioned tools, we were able to identify specific routes, and queries that needed to be optimizes
• Some of those queries we were able to remove all together
• We were able to observe these improvements the moment they went to production
• There were definitely some queries that given more time, would be ideal to move to a different, better suited database
Lessons Learned
Lessons Learned

- Logs and Metrics are invaluable
• **Lessons Learned**

  • Logs and Metrics are invaluable
    • Why? - You get so much visibility into your processes and it's rather simple to implement
Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it’s rather simple to implement
- Distributing Tracing is awesome, and the more the better
Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it's rather simple to implement
- Distributing Tracing is awesome, and the more the better
- Profilers provide a huge amount of insight into your code and you environment
• **Lessons Learned**

  • Logs and Metrics are invaluable
    • Why? - You get so much visibility into your processes and it's rather simple to implement
  • Distributing Tracing is awesome, and the more the better
  • Profilers provide a huge amount of insight into your code and you environment
  • Pprof visualizations plus the amount of info and flexibility eBPF provides would be great, but currently takes a lot of steps
• **Lessons Learned**

• Logs and Metrics are invaluable
  • Why? - You get so much visibility into your processes and it’s rather simple to implement
• Distributing Tracing is awesome, and the more the better
• Profilers provide a huge amount of insight into your code and you environment
• Pprof visualizations plus the amount of info and flexibility eBPF provides would be great, but currently takes a lot of steps
  • I’ve been working on a project to make this easier in my spare time [https://github.com/cpg1111/pprof-ebpf](https://github.com/cpg1111/pprof-ebpf)
• **Lessons Learned**

• Logs and Metrics are invaluable
  • Why? - You get so much visibility into your processes and it’s rather simple to implement

• Distributing Tracing is awesome, and the more the better

• Profilers provide a huge amount of insight into your code and your environment

• Pprof visualizations plus the amount of info and flexibility eBPF provides would be great, but currently takes a lot of steps
  • I’ve been working on a project to make this easier in my spare time [https://github.com/cpg1111/pprof-ebpf](https://github.com/cpg1111/pprof-ebpf)

• Being able to keep your changes small and more iterative will yield you greater results in the long run.
Lessons Learned

- Logs and Metrics are invaluable
  - Why? - You get so much visibility into your processes and it's rather simple to implement
- Distributing Tracing is awesome, and the more the better
- Profilers provide a huge amount of insight into your code and you environment
- Pprof visualizations plus the amount of info and flexibility eBPF provides would be great, but currently takes a lot of steps
  - I've been working on a project to make this easier in my spare time
    https://github.com/cpg1111/pprof-ebpf
- Being able to keep your changes small and more iterative will yield you greater results in the long run.
- Having a baseline to compare changes for performance to is a must
Thank You