Network Monitor

ACKnowledging an observability gap

KPMG
shopify

Shopify Traffic

83k

average requests
per second

170k

peak requests
per second

Resiliency toolkit for Ruby for failing fast

resiliency | circuit-breaker | bulkheads | ruby | webscale

README.md

Semian build passing

Semian is a library for controlling access to slow or unresponsive external services to avoid cascading failures.

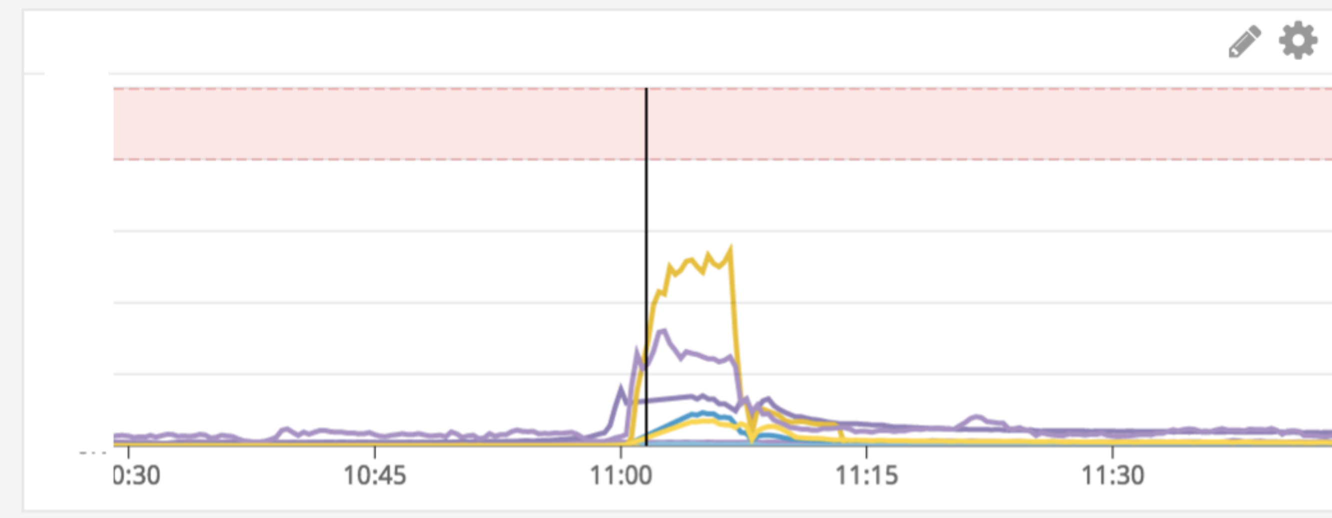
When services are down they typically fail fast with errors like `ECONNREFUSED` and `ECONNRESET` which can be rescued in code. However, slow resources fail slowly. The thread serving the request blocks until it hits the timeout for the slow resource. During that time, the thread is doing nothing useful and thus the slow resource has caused a cascading failure by occupying workers and therefore losing capacity. **Semian is a library for failing fast in these situations, allowing you to handle errors gracefully.** Semian does this by intercepting resource access through heuristic patterns inspired by Hystrix and Release It:

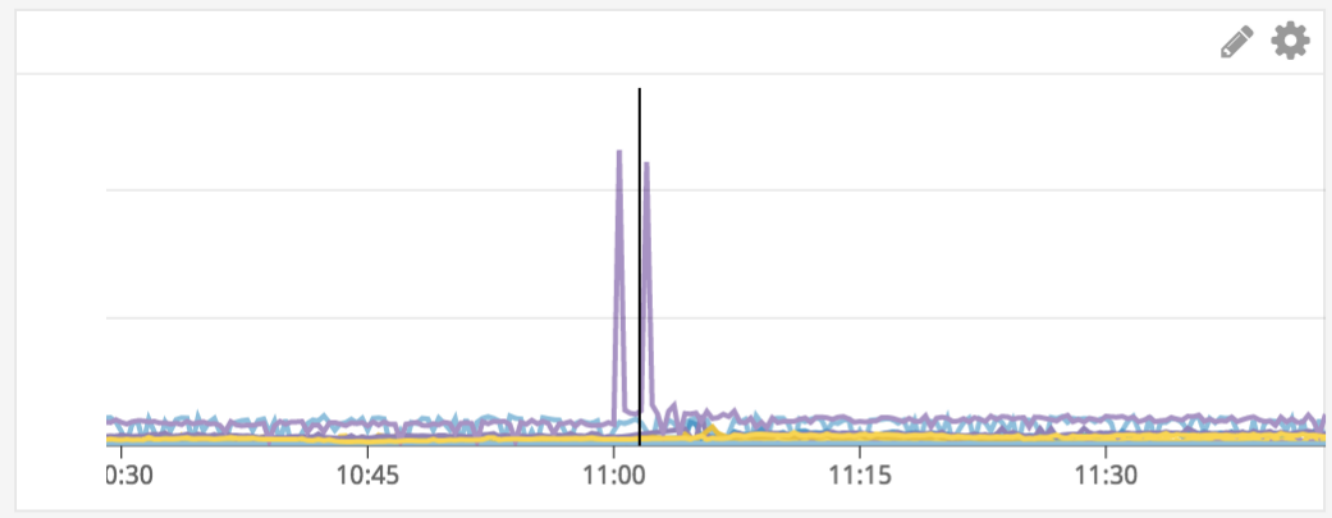
- **Circuit breaker**. A pattern for limiting the amount of requests to a dependency that is having issues.
- **Bulkheading**. Controlling the concurrent access to a single resource, access is coordinated server-wide with SysV semaphores.

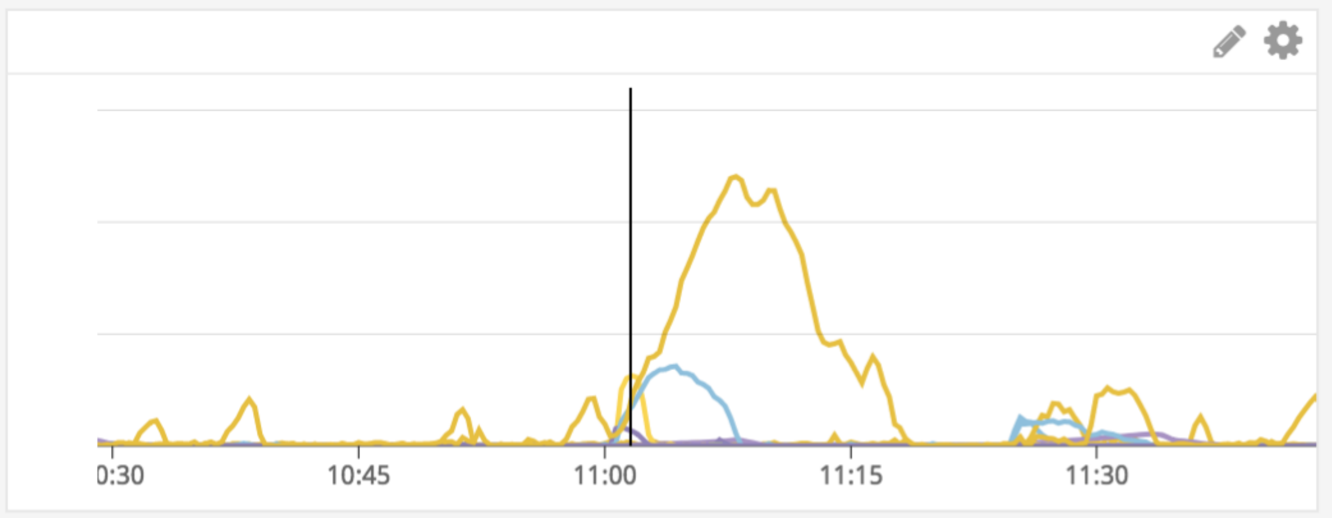
Resource drivers are monkey-patched to be aware of Semian, these are called Semian Adapters. Thus, every time resource access is requested Semian is queried for status on the resource first. If Semian, through the patterns above, deems the resource to be unavailable it will raise an exception. **The ultimate outcome of Semian is always an exception that can then be rescued for a graceful fallback**. Instead of waiting for the timeout, Semian raises straight away.

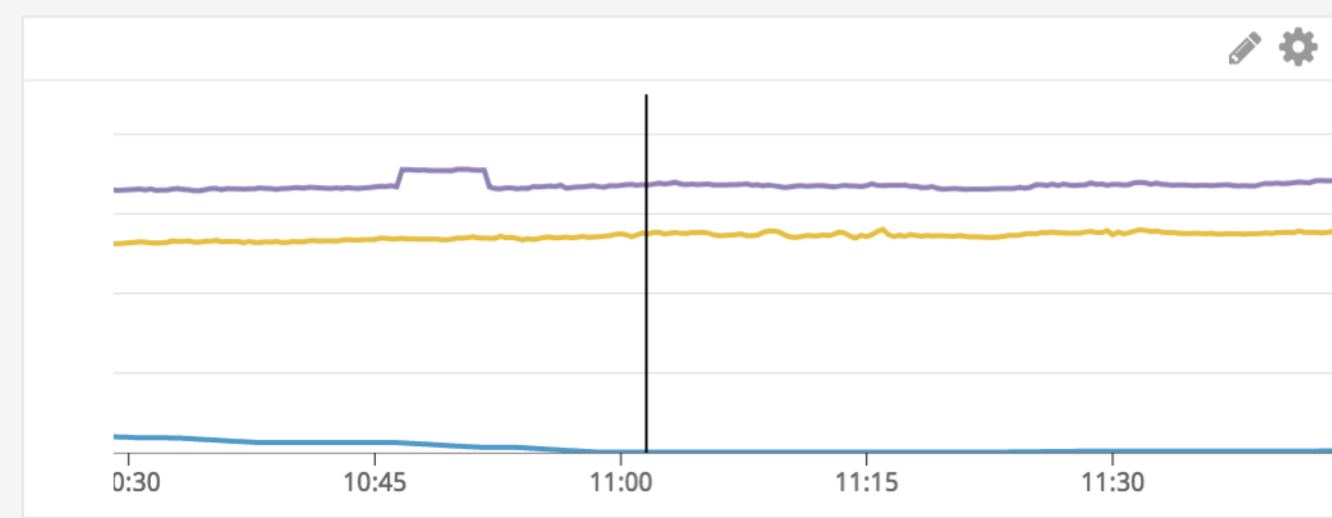
Search events to overlay...

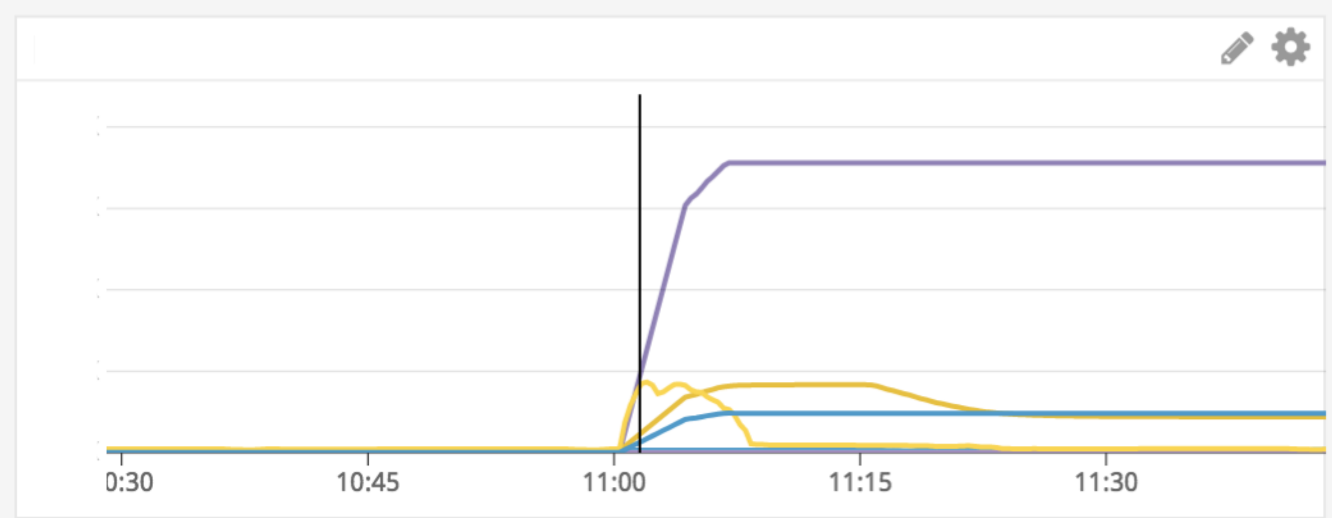
Show 74 m Oct 12, 10:29AM - Oct 12, 11:43AM

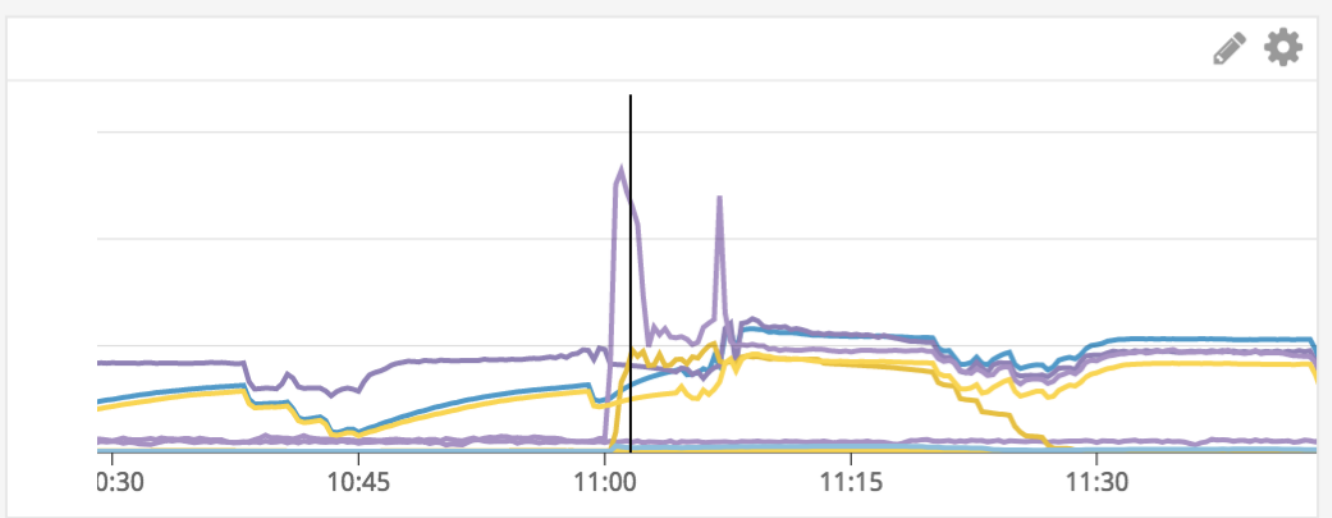


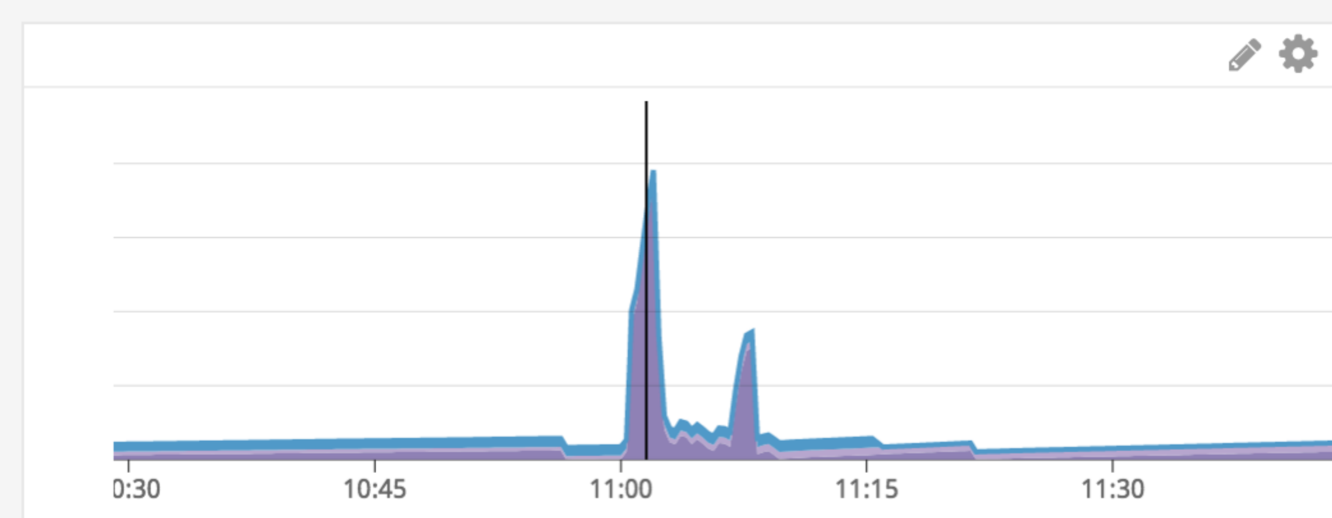


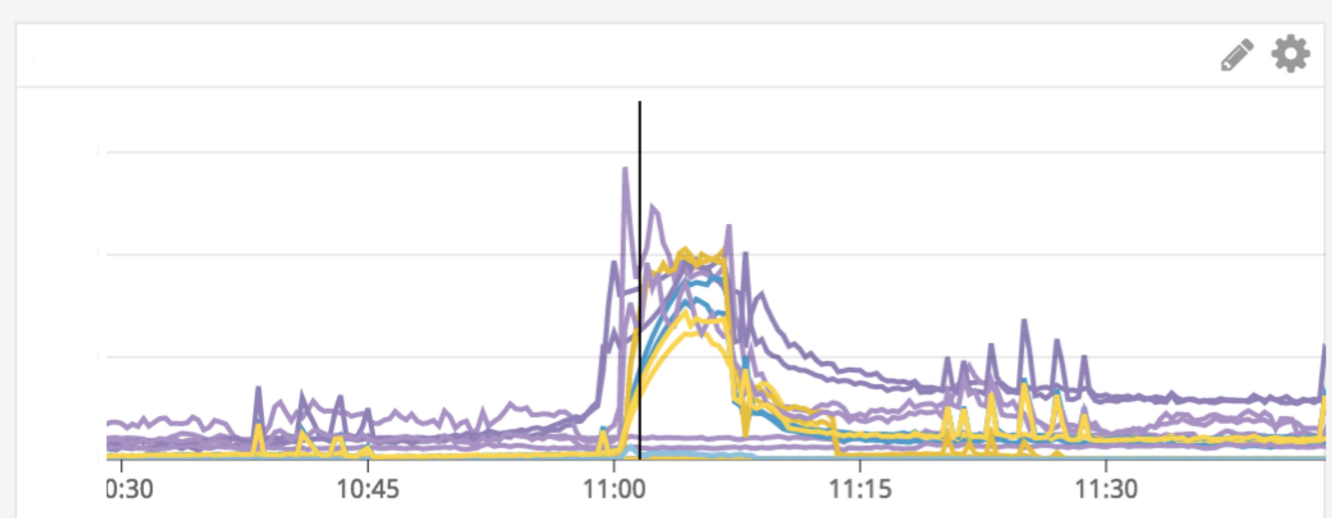


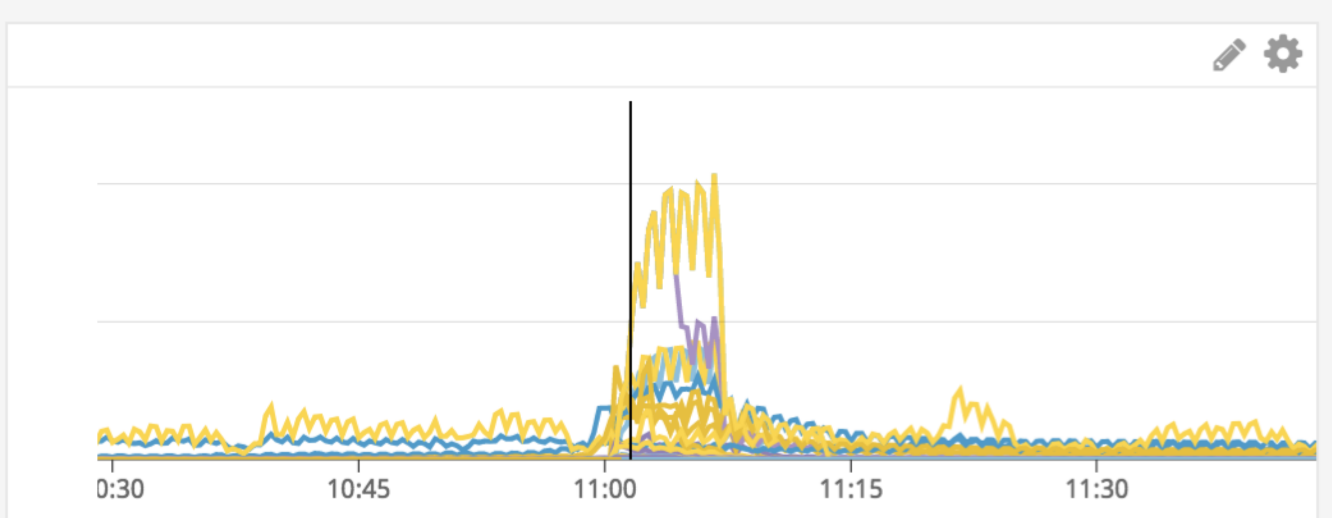


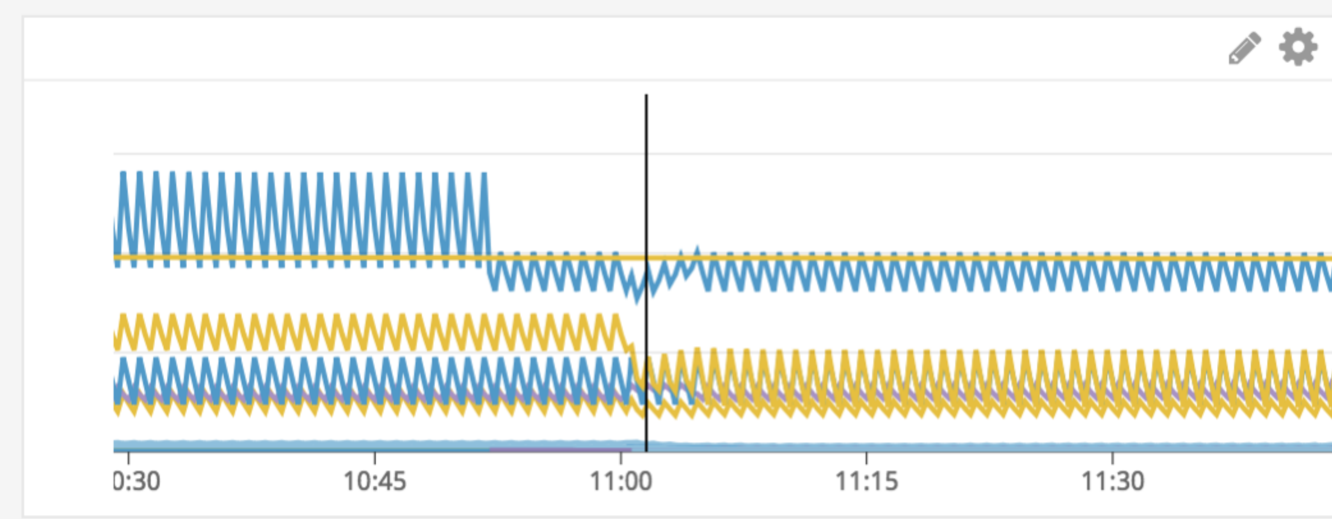


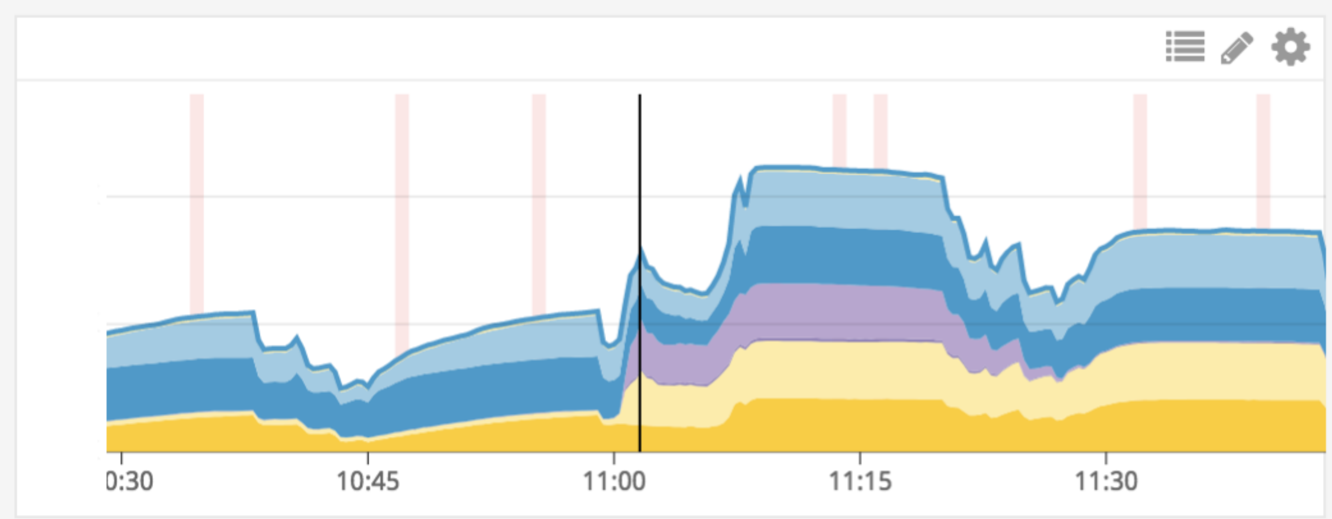


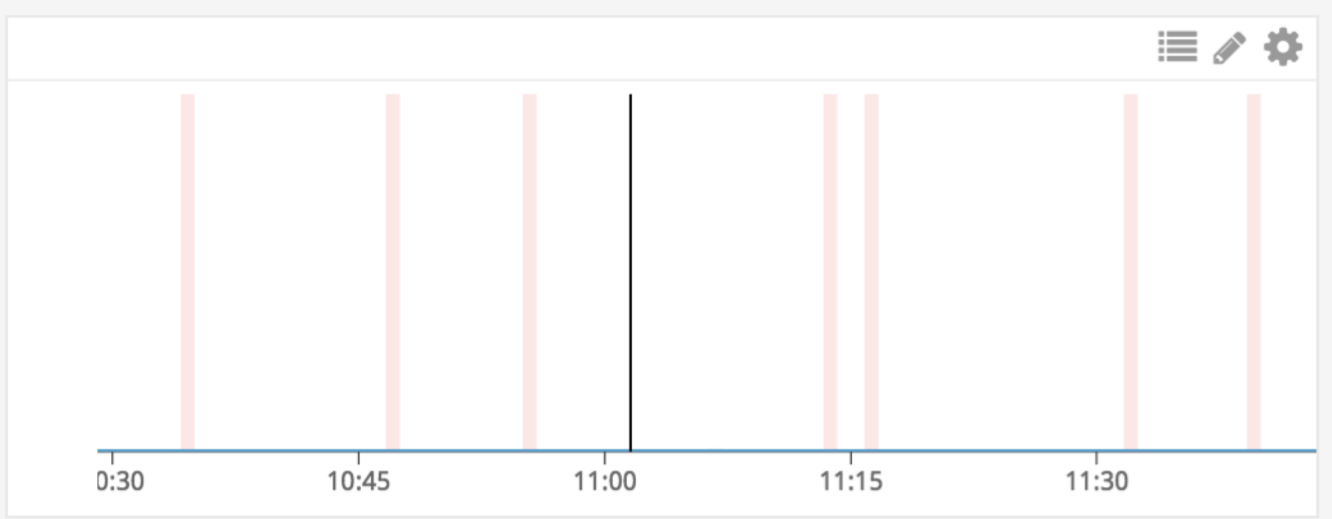


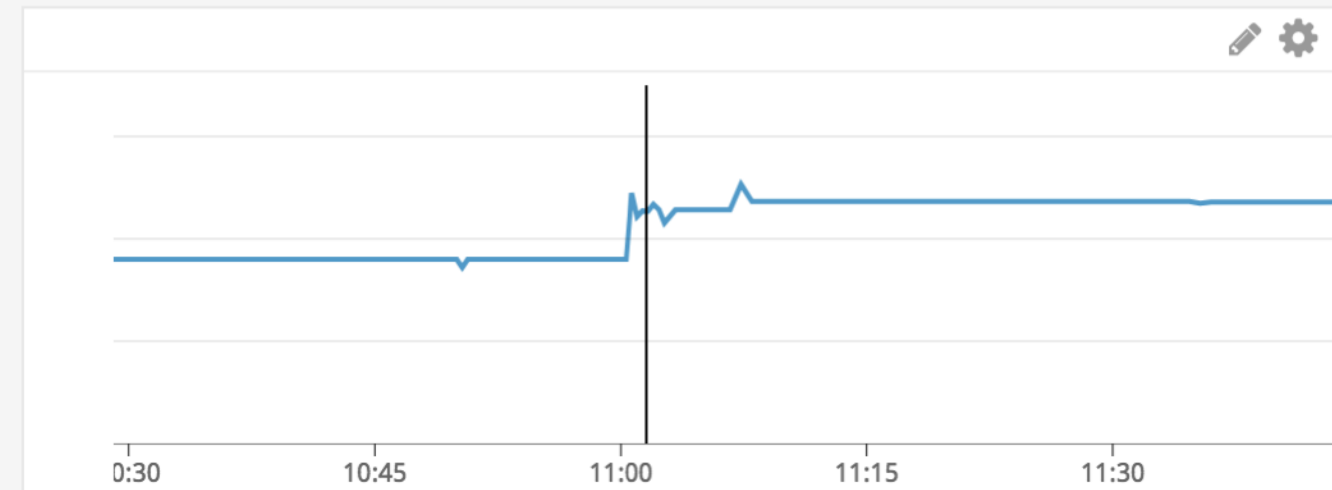


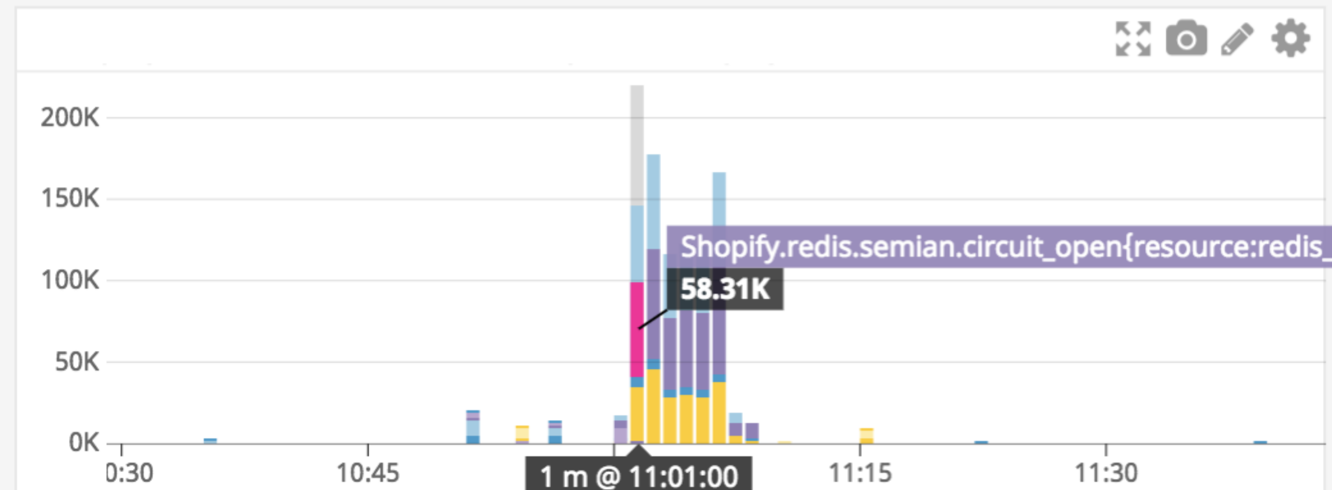












add a graph

Attribution: https://youtu.be/VNEhu7eqyVQ

Black Friday + Thanksgiving Online Revenue

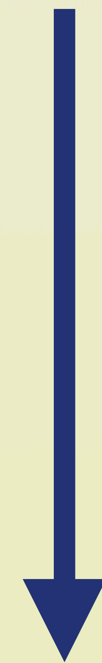
9.9	12.4
Billion USD 2018	Billion USD Estimate for 2019

Source: Adobe Analytics

b..
n..
b..
_..
p..
n..
_..
d..
d..
_..
ip..
ip..
tc..
tcp..
listSearchKey
unlinkClient
freeClient
aeProcessEvents
aeMain
main
__libc_start_main
do_sy..
entry..
__clone
n..
__..
ir..
do..
re..
l..
l..
l..
l..
l..
eva..
call
proce..
process..
d..
e..
_..
e..
s..
do..
en..
ep..
redis-server

perf

Connection management



listSearchKey
unlinkClient
freeClient
aeProcessEvents
aeMain
main
__clone
__libc_start_main
redis-server
do_sy..
entry..
tc..
tcp..
call
proce..
process..
eva..

Attribution: https://gfycat.com/deficientsprybluebird

Offsets	*Octet*	0								1								2								3							
Octet	*Bit*	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port															
4	32	Sequence number																															
8	64	Acknowledgment number																															
12	96	Data offset				Reserved 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	SYN	FIN	Window Size															
16	128	Checksum																Urgent pointer															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

Source: Wikipedia

<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><td>4</td><td>32</td><td colspan="32">Sequence number</td></tr>
<tr><td>8</td><td>64</td><td colspan="32">Acknowledgment number</td></tr>
<tr><td>12</td><td>96</td><td colspan="4">Data offset</td><td colspan="3">Reserved
0 0 0</td><td>NS</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="16">Window Size</td></tr>
<tr><td>16</td><td>128</td><td colspan="16">Checksum</td><td colspan="16">Urgent pointer</td></tr>
<tr><td>20
…</td><td>160
…</td><td colspan="32">Options (if data offset > 5. Padded at the end with "0" bytes if necessary.)
…</td></tr>
</table>

Source: Wikipedia

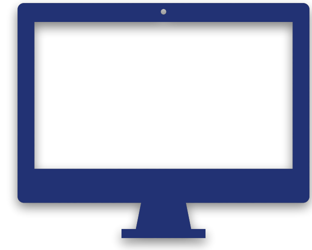
Offsets	*Octet*	0								1								2								3							
Octet	*Bit*	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	0	Source port																Destination port															
4	32	**Sequence number**																															
8	64	Acknowledgment number																															
12	96	Data offset				Reserved 0 0 0			NS	CWR	ECE	URG	ACK	PSH	RST	**SYN**	FIN	Window Size															
16	128	Checksum																Urgent pointer															
20 ...	160 ...	Options (if *data offset* > 5. Padded at the end with "0" bytes if necessary.) ...																															

Source: Wikipedia

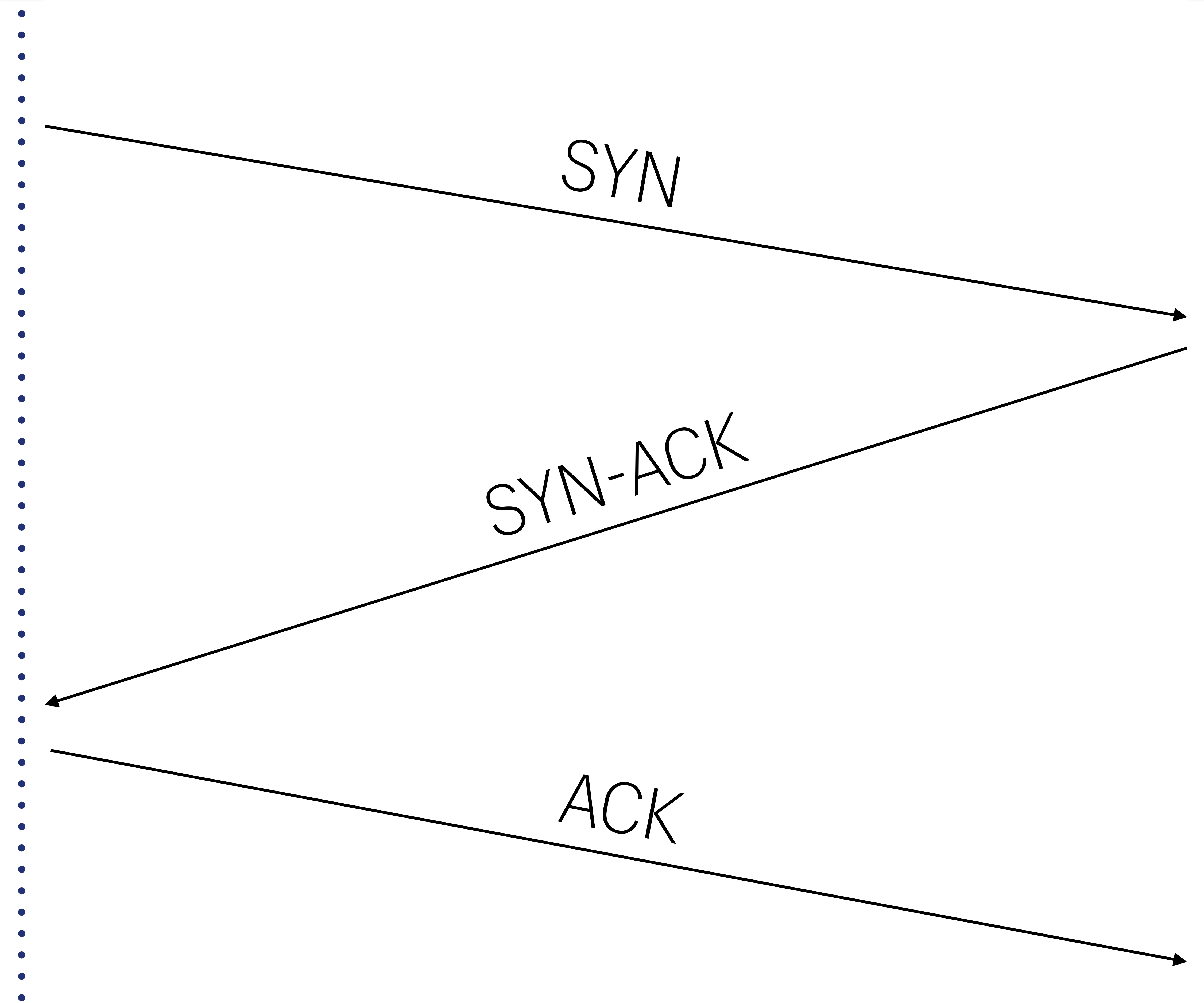
<table>
<tr><th>Offsets</th><th>Octet</th><th colspan="8">0</th><th colspan="8">1</th><th colspan="8">2</th><th colspan="8">3</th></tr>
<tr><th>Octet</th><th>Bit</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>0</td><td>0</td><td colspan="16">Source port</td><td colspan="16">Destination port</td></tr>
<tr><td>4</td><td>32</td><td colspan="32">Sequence number</td></tr>
<tr><td>8</td><td>64</td><td colspan="32">Acknowledgment number</td></tr>
<tr><td>12</td><td>96</td><td colspan="4">Data offset</td><td colspan="3">Reserved
0 0 0</td><td>NS</td><td>CWR</td><td>ECE</td><td>URG</td><td>ACK</td><td>PSH</td><td>RST</td><td>SYN</td><td>FIN</td><td colspan="16">Window Size</td></tr>
<tr><td>16</td><td>128</td><td colspan="16">Checksum</td><td colspan="16">Urgent pointer</td></tr>
<tr><td>20
...</td><td>160
...</td><td colspan="32">Options (if data offset > 5. Padded at the end with "0" bytes if necessary.)
...</td></tr>
</table>

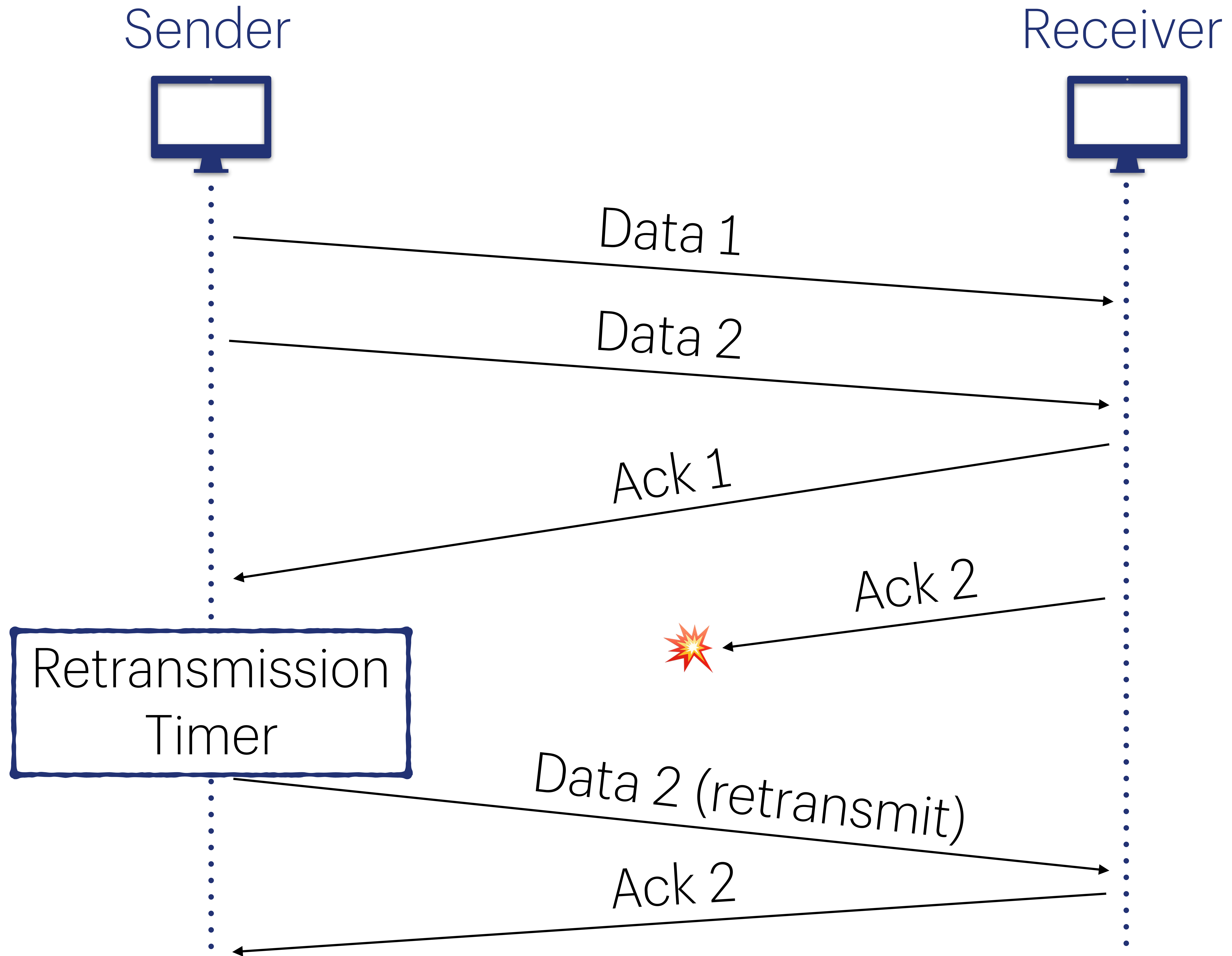
Source: Wikipedia

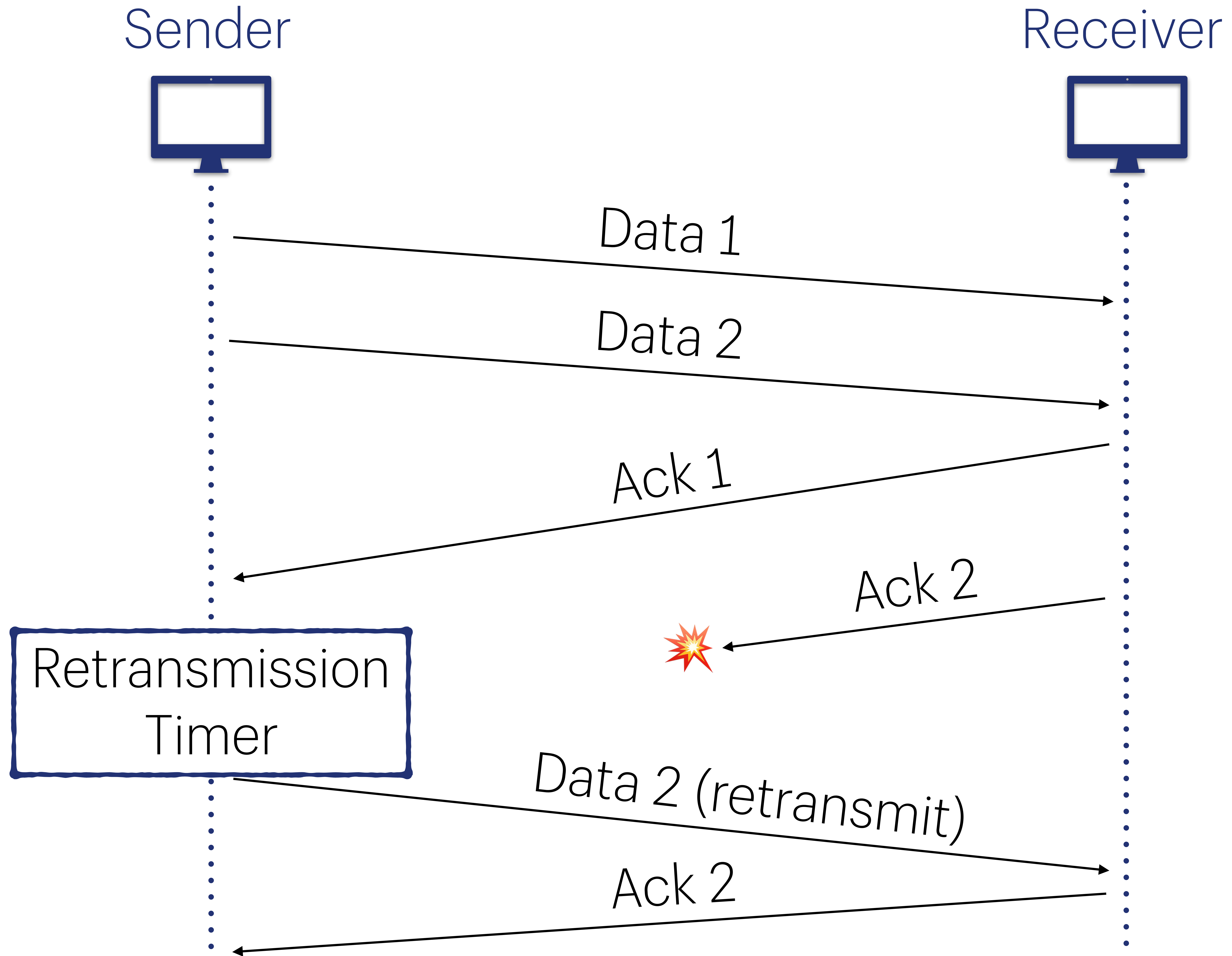
Client



Server

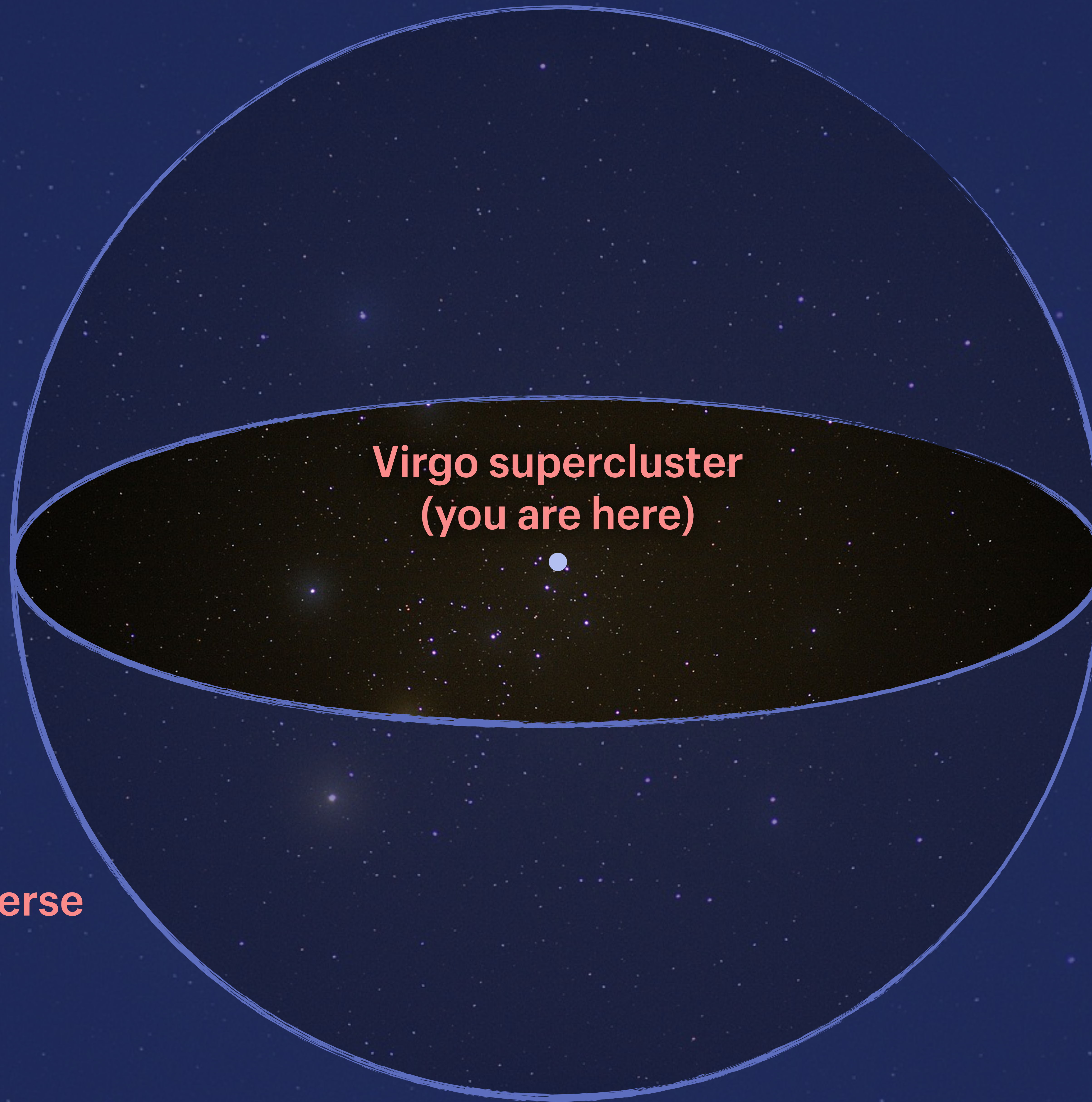



Sender
Receiver
Data 1
Data 2
Ack 1
Ack 2
Retransmission
Timer
Data 2 (retransmit)
Ack 2


Sender
Receiver
Data 1
Data 2
Ack 1
Ack 2
Retransmission Timer
Data 2 (retransmit)
Ack 2

Attribution: http://gph.is/1bgR8Yz

Attribution: https://gfycat.com/marriedlegalibis


Root cause
Virgo supercluster
(you are here)
Observable universe

Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis*

Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, Varugis Kurien†
Microsoft, †Midfin Systems
{chguo, lyuan, dxiang, yidang, rayhuang, dmaltz, zhaoyil, vinwang, bipang, stchen, linzw}@microsoft.com, vkurien@midfinsystems.com

ABSTRACT

Can we get network latency between any two servers at any time in large-scale data center networks? The collected latency data can then be used to address a series of challenges: telling if an application perceived latency issue is caused by the network or not, defining and tracking network service level agreement (SLA), and automatic network troubleshooting.

We have developed the Pingmesh system for large-scale data center network latency measurement and analysis to answer the above question affirmatively. Pingmesh has been running in Microsoft data centers for more than four years, and it collects tens of terabytes of latency data per day. Pingmesh is widely used by not only network software developers and engineers, but also application and service developers and operators.

CCS Concepts

•Networks → Network measurement; Cloud computing; Network monitoring; •Computer systems organization → Cloud computing;

Keywords

Data center networking; Network troubleshooting; Silent packet drops

1. INTRODUCTION

In today's data centers there are hundreds of thousands of servers. These servers are connected via network interface cards (NICs), switches and routers, cables and fibers, which form large-scale intra and inter data center networks. The scale of the data center networks (DCNs) is growing even larger due to the rapid development of cloud computing. On top of the physical data center infrastructure, various large-scale, distributed services are built, e.g., Search [5], distributed file systems [17] and storage [7], MapReduce [11].

These distributed services are large and evolving software systems with many components and have complex dependencies. All of these services are distributed and many of their components need to interact via the network either within a data center or across different data centers. In such large systems, software and hardware failures are the norm rather than the exception. As a result, the network team faces several challenges.

The first challenge is to determine if an issue is a network issue or not. Due to the distributed systems nature, many failures show as "network" problems, e.g., some components can only be reached intermittently, or the end-to-end latency shows a sudden increase at the 99th percentile, the network throughput degrades from 20MB/s per server to less than 5MB/s. Our experience showed that about 50% of these "network" problems are not caused by the network. However it is not easy to tell if a "network" problem is indeed caused by network failures or not.

The second challenge is to define and track network service level agreements (SLAs). Many services need the network to provide certain performance guarantees. For example, a Search query may touch thousands of servers and the performance of a Search query is determined by the last response from the slowest server.

*This work was performed when Varugis Kurien was with Microsoft.

Pingmesh: A Large-Scale System for Data Center Network Latency Measurement and Analysis*

Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, Varugis Kurien[†]
Microsoft, [†]Midfin Systems
{chguo, lyuan, dxiang, yidang, rayhuang, dmaltz, zhaoyil, vinwang, bipang, stchen, linzw}@microsoft.com, vkurien@midfinsystems.com

ABSTRACT

Can we get network latency between any two servers at any time in large-scale data center networks? The collected latency data can then be used to address a series of challenges: telling if an application perceived latency issue is caused by the network or not, defining and tracking network service level agreement (SLA), and automatic network troubleshooting.

We have developed the Pingmesh system for large-scale data center network latency measurement and analysis to answer the above question affirmatively. Pingmesh has been running in Microsoft data centers for more than four years, and it collects tens of terabytes of latency data per day. Pingmesh is widely used by not only network software developers and engineers, but also application and service developers and operators.

CCS Concepts

•Networks → Network measurement; Cloud computing; Network monitoring; •Computer systems organization → Cloud computing;

Keywords

Data center networking; Network troubleshooting; Silent packet drops

1. INTRODUCTION

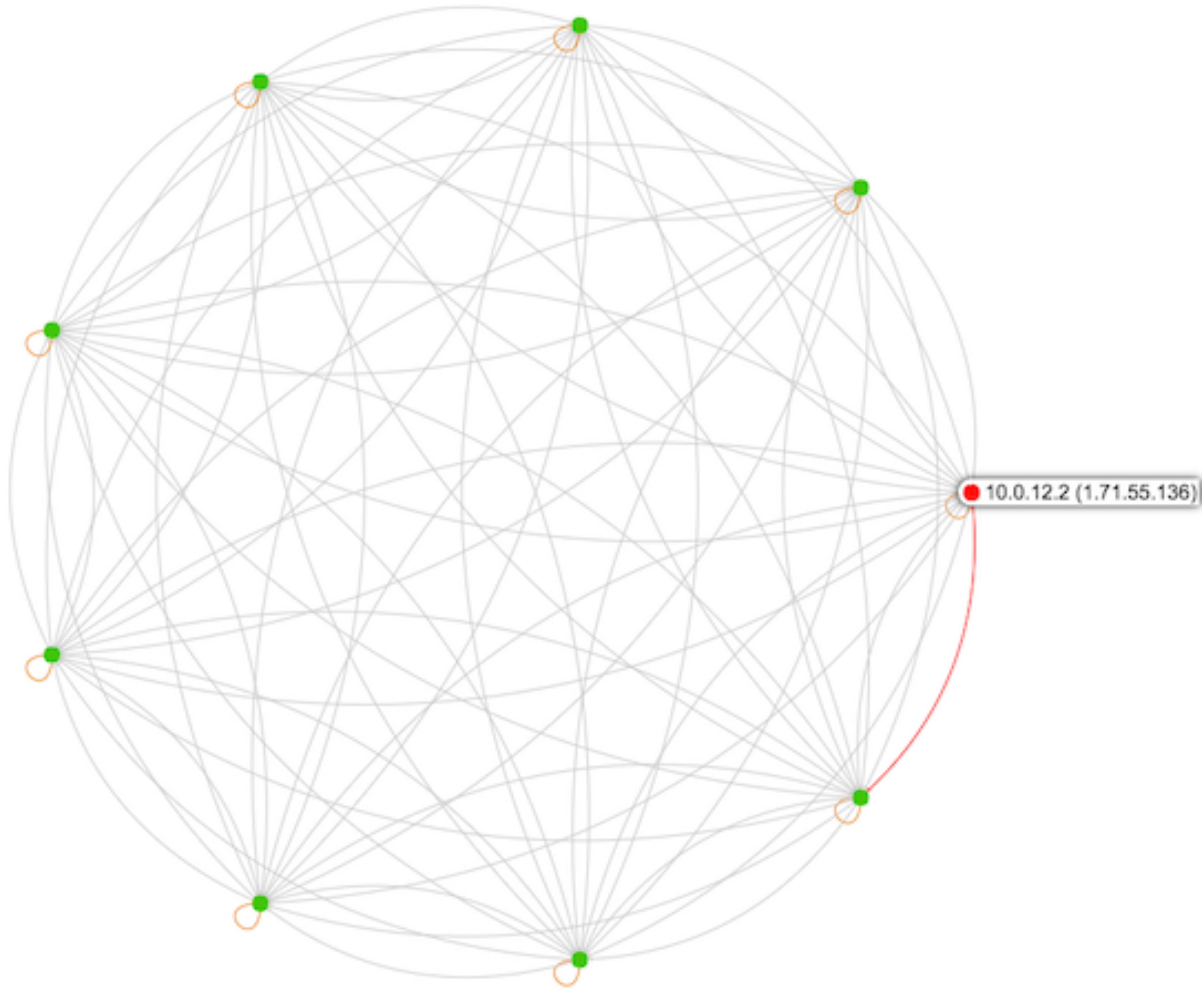
In today's data centers there are hundreds of thousands of servers. These servers are connected via network interface cards (NICs), switches and routers, cables and fibers, which form large-scale intra and inter data center networks. The scale of the data center networks (DCNs) is growing even larger due to the rapid development of cloud computing. On top of the physical data center infrastructure, various large-scale, distributed services are built, e.g., Search [5], distributed file systems [17] and storage [7], MapReduce [11].

These distributed services are large and evolving software systems with many components and have complex dependencies. All of these services are distributed and many of their components need to interact via the network either within a data center or across different data centers. In such large systems, software and hardware failures are the norm rather than the exception. As a result, the network team faces several challenges.

The first challenge is to determine if an issue is a network issue or not. Due to the distributed systems nature, many failures show as "network" problems, e.g., some components can only be reached intermittently, or the end-to-end latency shows a sudden increase at the 99th percentile, the network throughput degrades from 20MB/s per server to less than 5MB/s. Our experience showed that about 50% of these "network" problems are not caused by the network. However it is not easy to tell if a "network" problem is indeed caused by network failures or not.

The second challenge is to define and track network service level agreements (SLAs). Many services need the network to provide certain performance guarantees. For example, a Search query may touch thousands of servers and the performance of a Search query is determined by the last response from the slowest server.

*This work was performed when Varugis Kurien was with Microsoft.




```
{
  "HostIP": "10.234.15.243",
  "OK": true,
  "RoundTripTimeMS": 525,
  "response": {
    "boot_time": "2019-02-04T22:24:09.111Z"
  },
  "status-code": 200
}
```

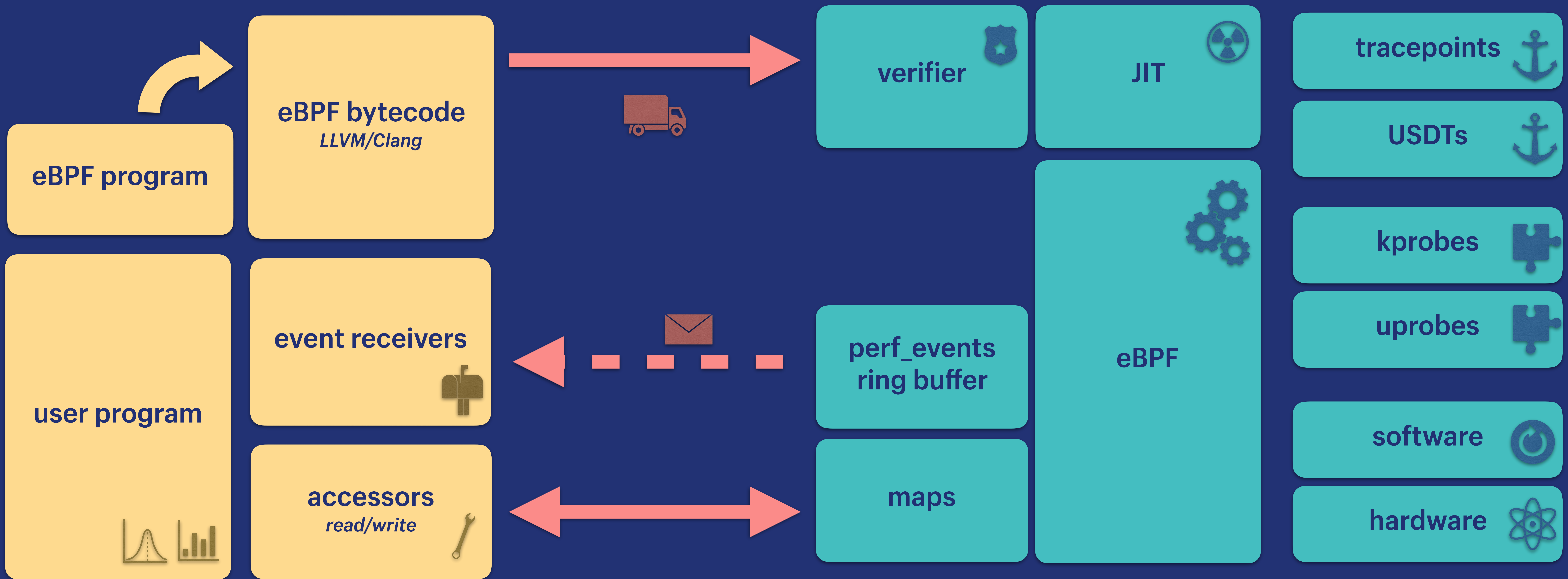
Attribution: https://gph.is/XJhNg3

REQUEST

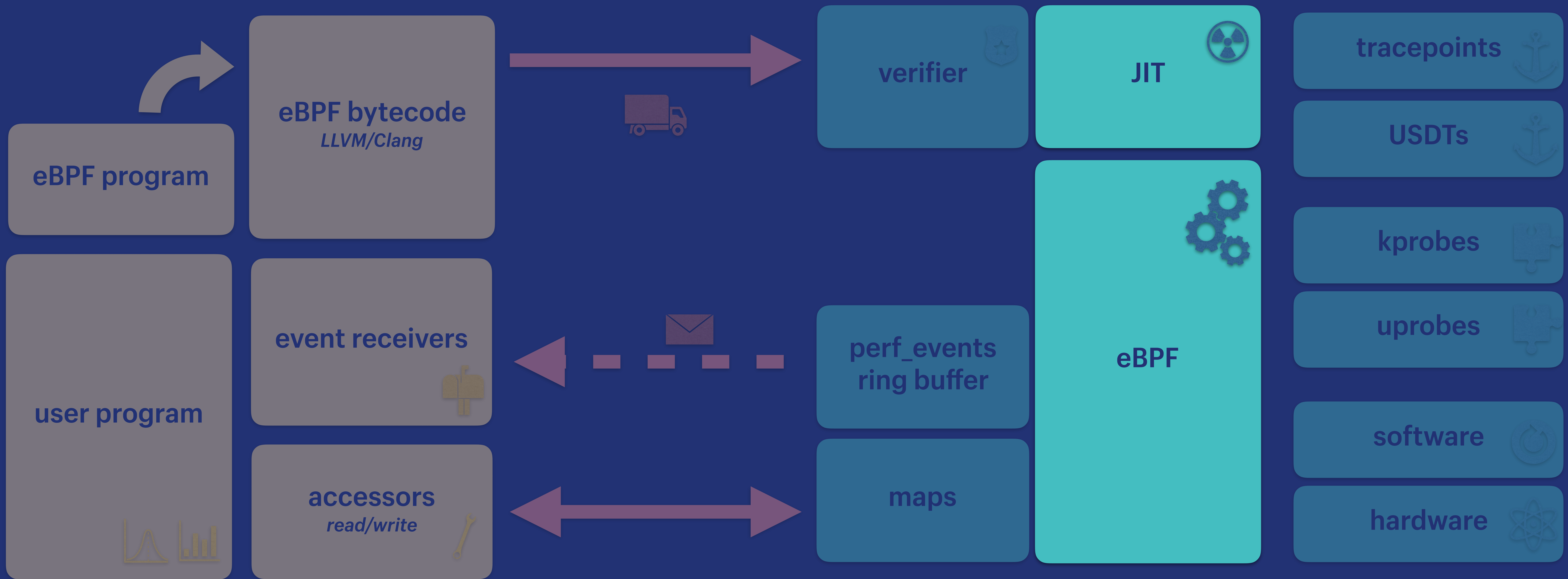
ALL THE RESOURCES

Attribution: https://knowyourmeme.com/memes/all-the-things

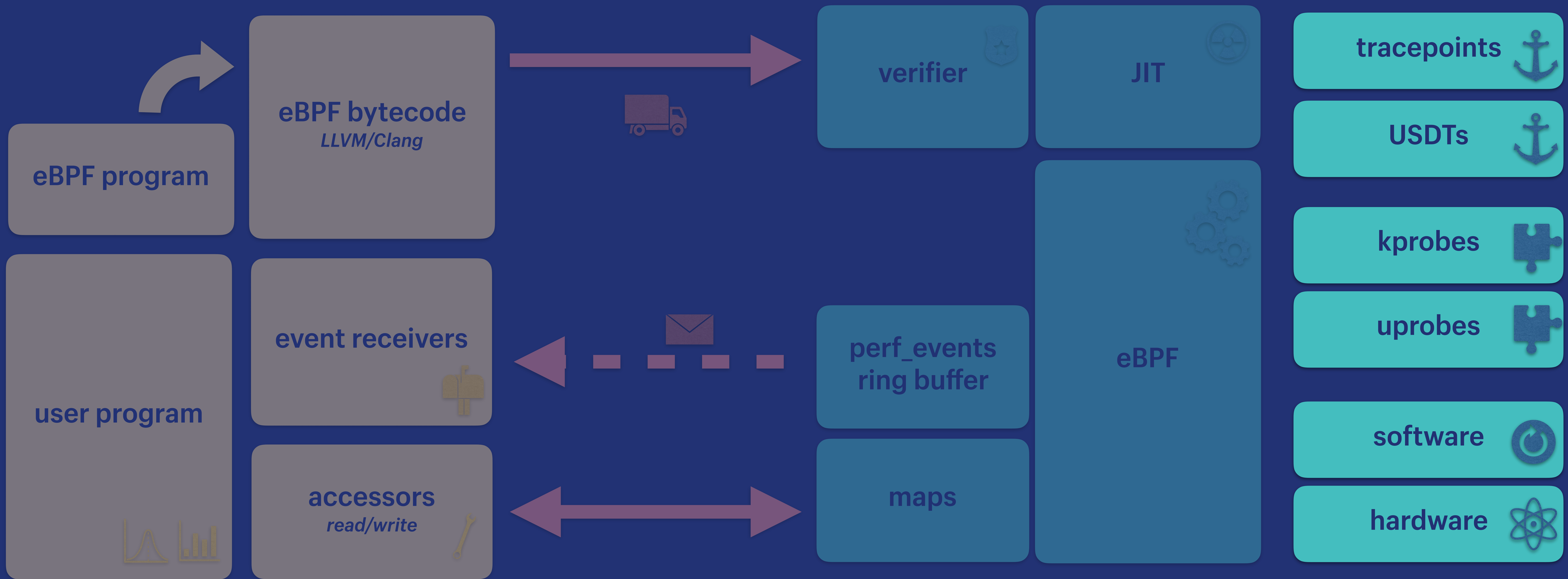
eBPF overview



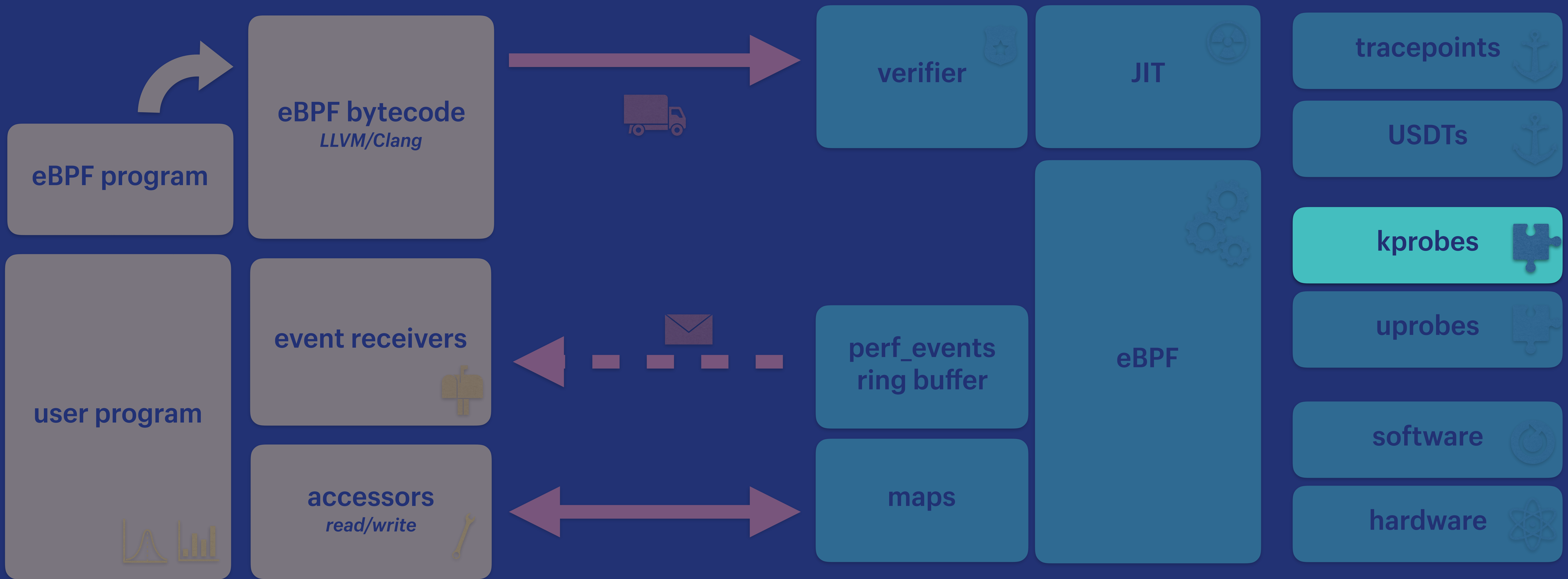
eBPF overview



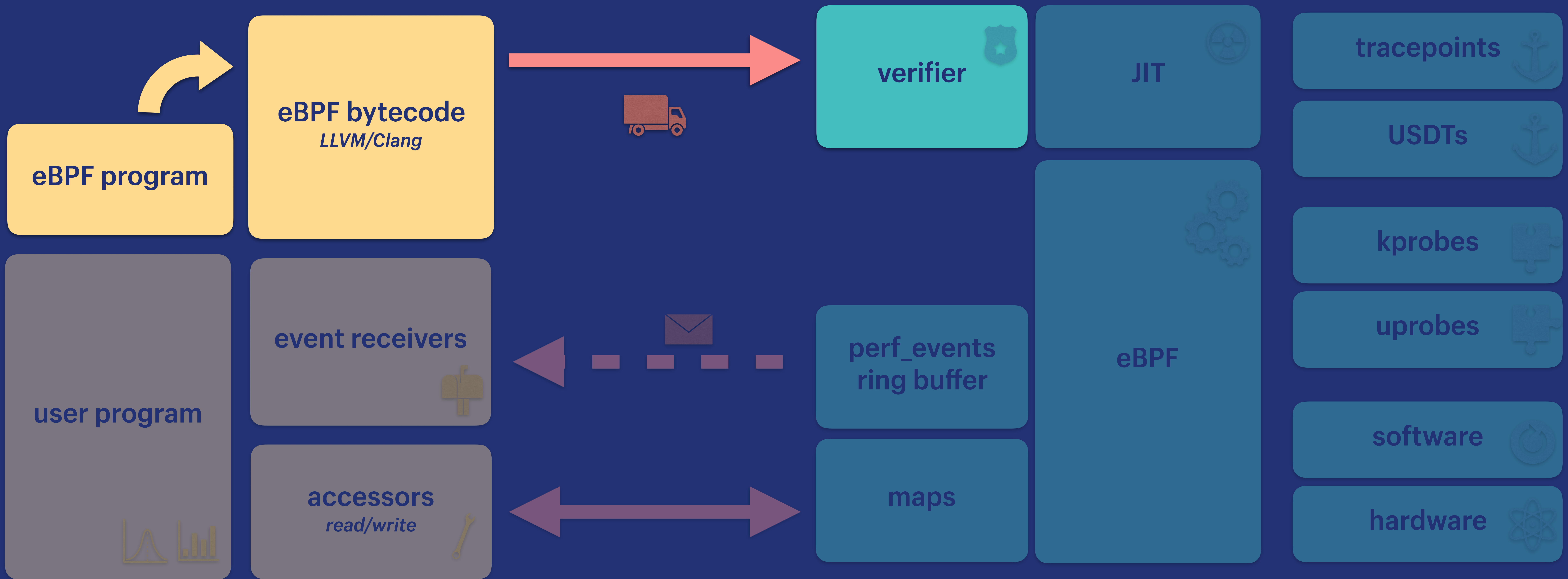
eBPF overview



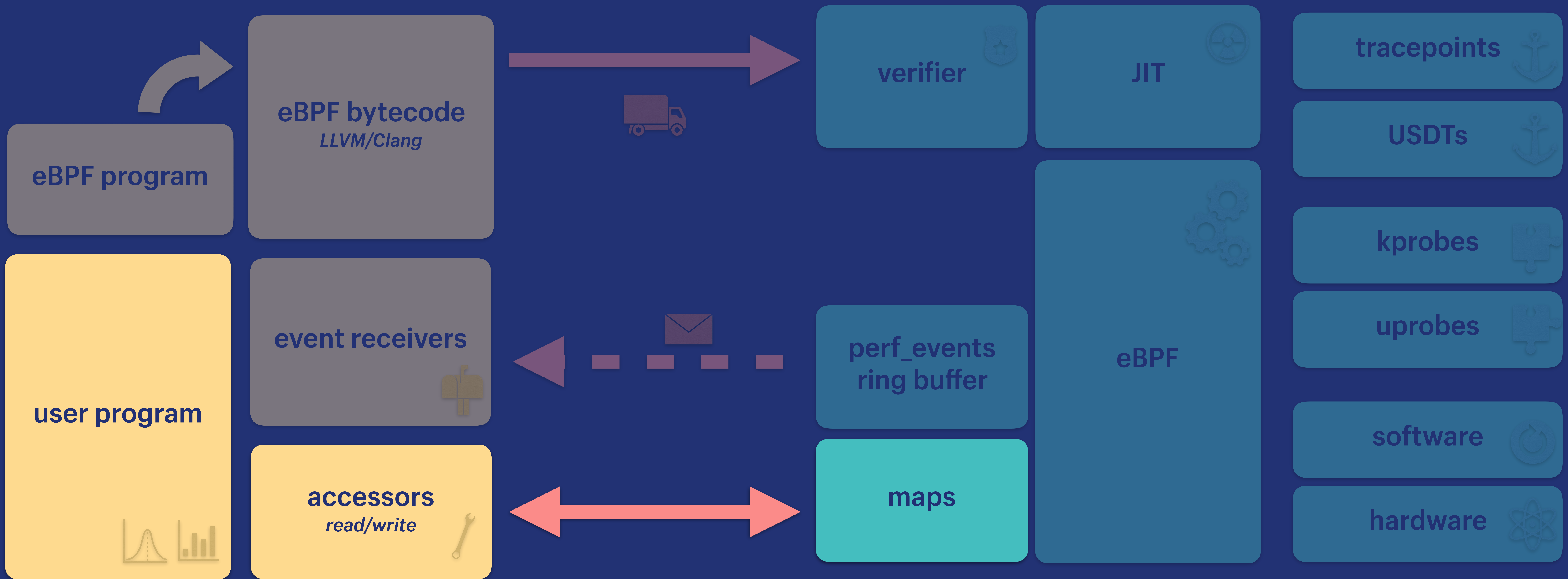
eBPF overview



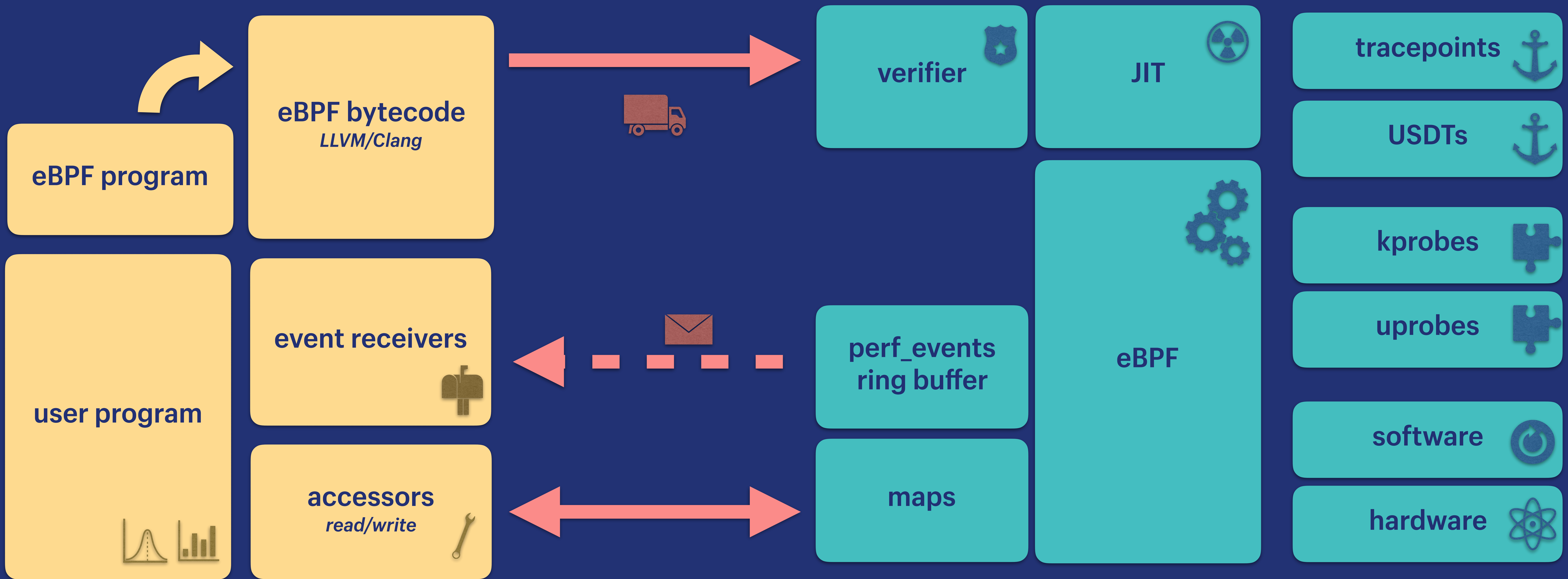
eBPF overview



eBPF overview



eBPF overview



```c
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```c
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```c
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```c
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```c
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```c
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```c
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```c
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

```c
BPF_HASH(data, u32, u64);
BPF_HASH(start, struct sock *, u64);

int kprobe__tcp_v4_connect(struct pt_regs *ctx, struct sock *sk, struct sockaddr *uaddr) {
  u64 ts_start = bpf_ktime_get_ns();
  start.update(&sk, &ts_start);
};

int kprobe__tcp_finish_connect(struct pt_regs *ctx, struct sock *sk) {
  if(sk == NULL || sk->__sk_common.skc_state != TCP_SYN_SENT) {
    return 0;
  }

  u64 *ts = start.lookup(&sk);
  if(ts == NULL) {
    return 0;
  }

  u64 now = bpf_ktime_get_ns();
  u64 delta_us = (now - *ts) / 1000ul;
  data.update(&sk->__sk_common.skc_daddr, &delta_us);

  start.delete(&sk);
  return 0;
}
```

Search or jump to... / Pull requests Issues Marketplace Explore

iovisor / bcc

Watch 461 | Star 7,507 | Fork 1,296

Code | Issues 347 | Pull requests 28 | Projects 0 | Wiki | Security | Insights

BCC - Tools for BPF-based Linux IO analysis, networking, monitoring, and more

3,152 commits | 28 branches | 19 releases | 232 contributors | Apache-2.0

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

mwesolowski and yonghong-song stackcount: fix TypeError (#2514) ... Latest commit c99c7c4 5 days ago

SPECS	spec: added shebang mangling	8 months ago
images	update tools diagram for 2019	3 months ago
snapcraft	snapcraft: fix the fetching of the most recent tag version	3 months ago
tests	Support for hardware offload (#2502)	16 days ago
tools	stackcount: fix TypeError (#2514)	5 days ago
.dockerignore	Add multiple build support styles	4 years ago
.gitignore	inject: Add support for alloc_page family of functions (#2114)	9 months ago
.travis.yml	Four parallel test runs	2 months ago
CODEOWNERS	Add /tests/ to CODEOWNERS	2 years ago
CONTRIBUTING-SCRIPTS.md	notes about tools (#2153)	8 months ago
Dockerfile.debian	Fixes for debian and ubuntu docker build	2 years ago
Dockerfile.ubuntu	Enable Ubuntu/arm64 deb packaging (#1968) Show apps	last year
FAQ.txt	Updated the FAQ with the error produced if python[2-3]-bcc isn't inst...	2 years ago
INSTALL.md	update INSTALL.md for Centos	3 months ago
LICENSE.txt	Create LICENSE.txt	4 years ago
LINKS.md	Link to article on how Circonus uses bcc	last year
QUICKSTART.md	add Quick Start Guide for bcc docker	3 years ago
README.md	update tools diagram for 2019	3 months ago

README.md

BPF Compiler Collection (BCC)

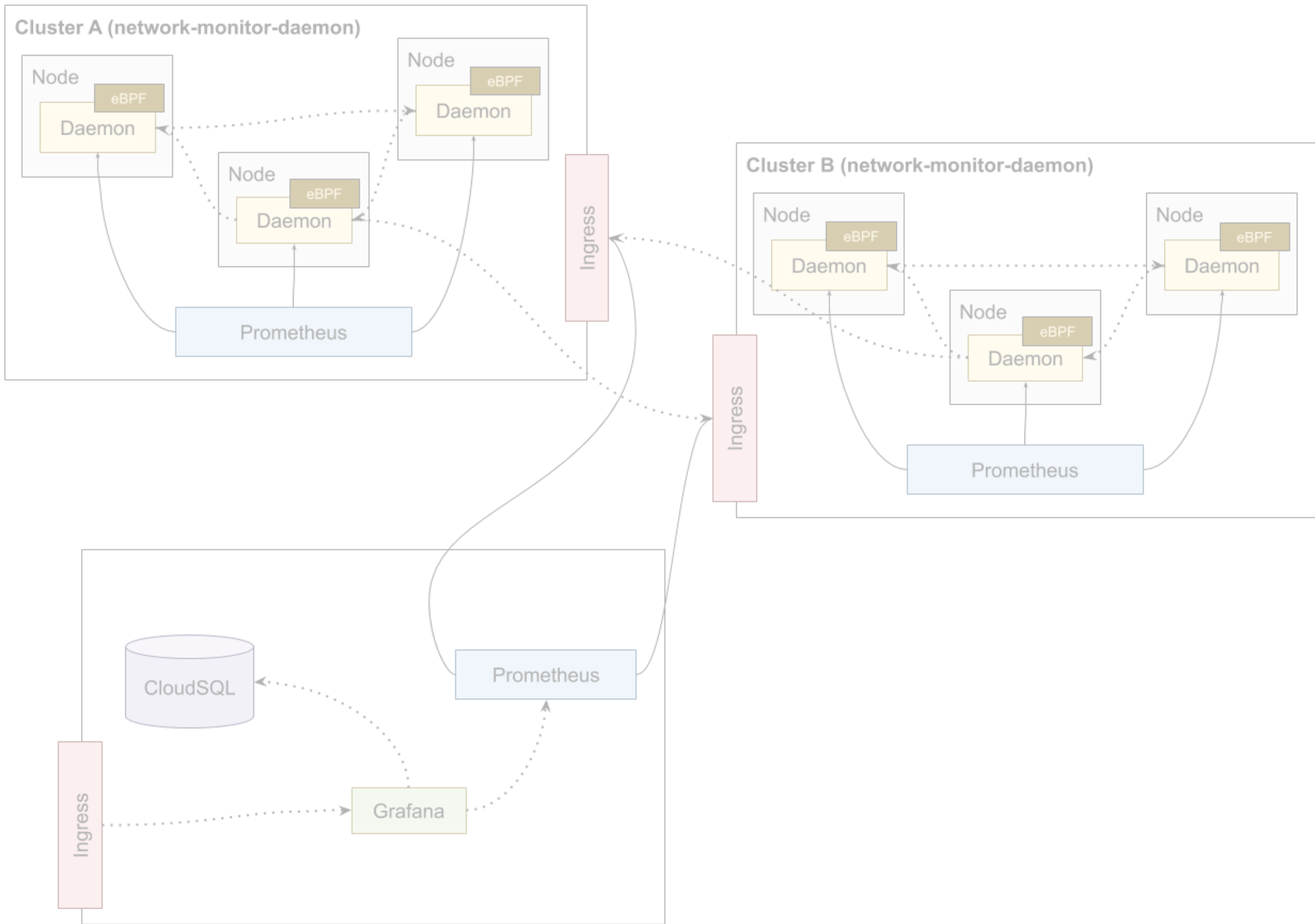
BCC is a toolkit for creating efficient kernel tracing and manipulation programs, and includes several useful tools and examples. It makes use of extended BPF (Berkeley Packet Filters), formally known as eBPF, a new feature that was first added to Linux 3.15. Much of what BCC uses requires Linux 4.1 and above.

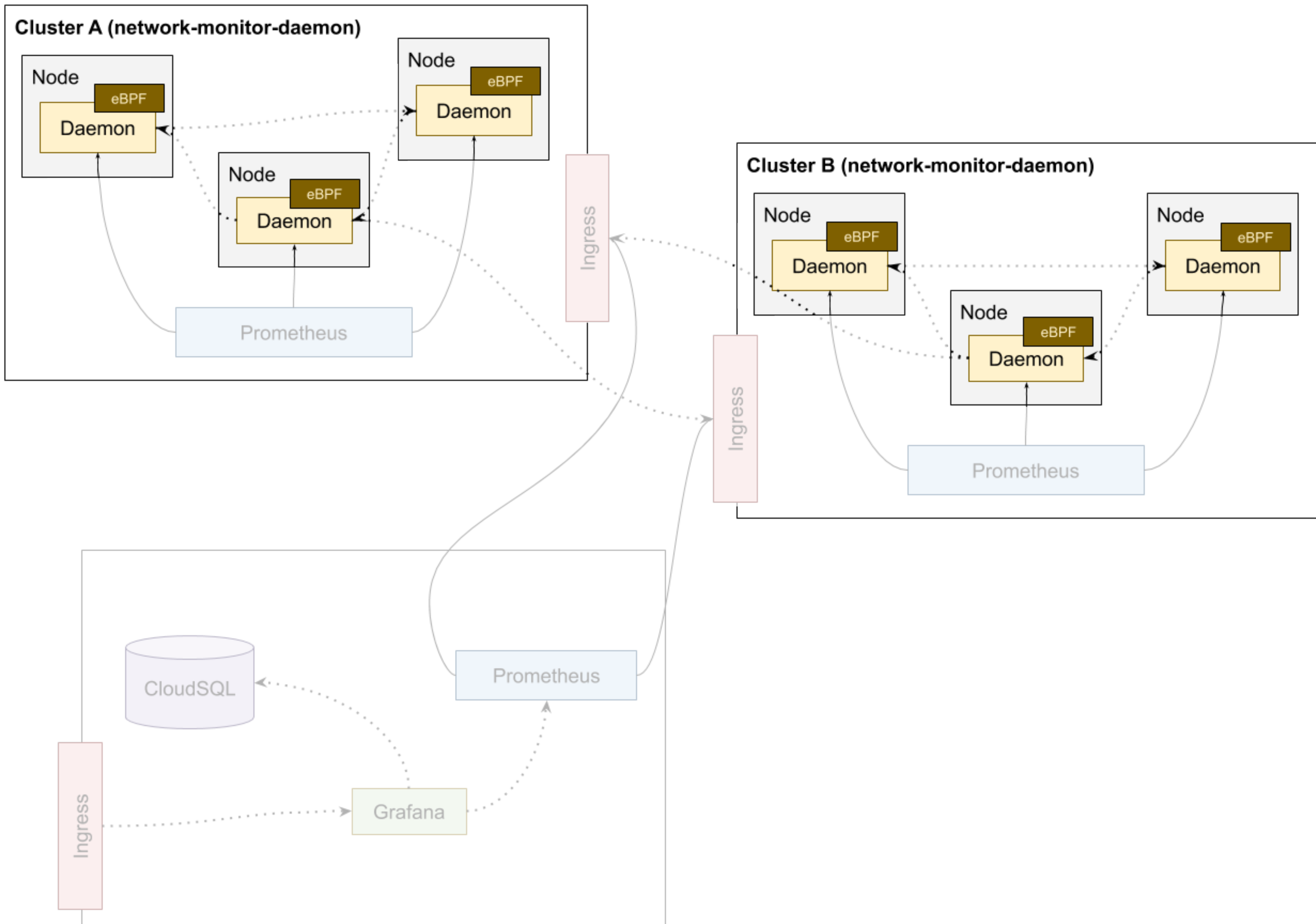
userspace max

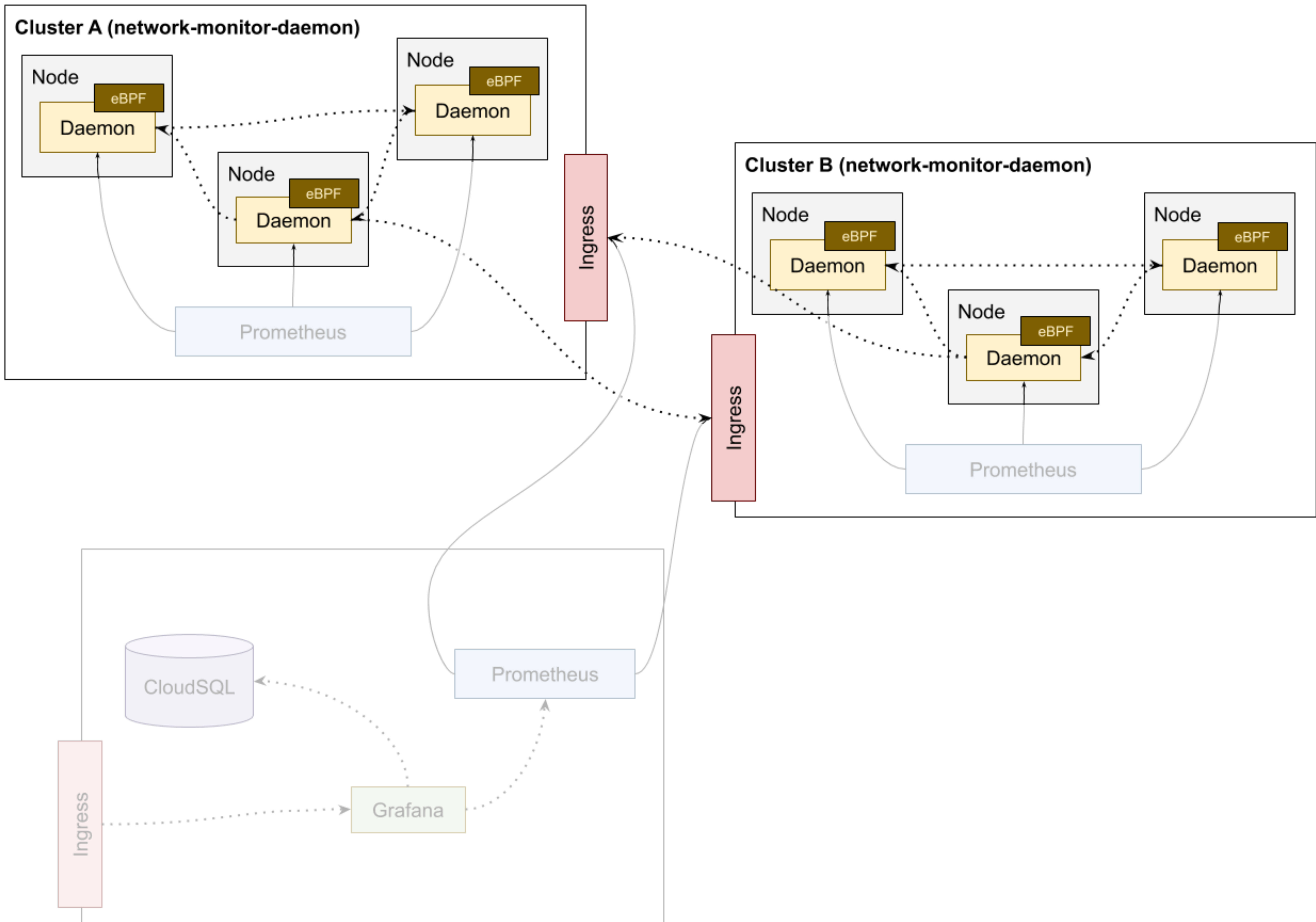
BPF max

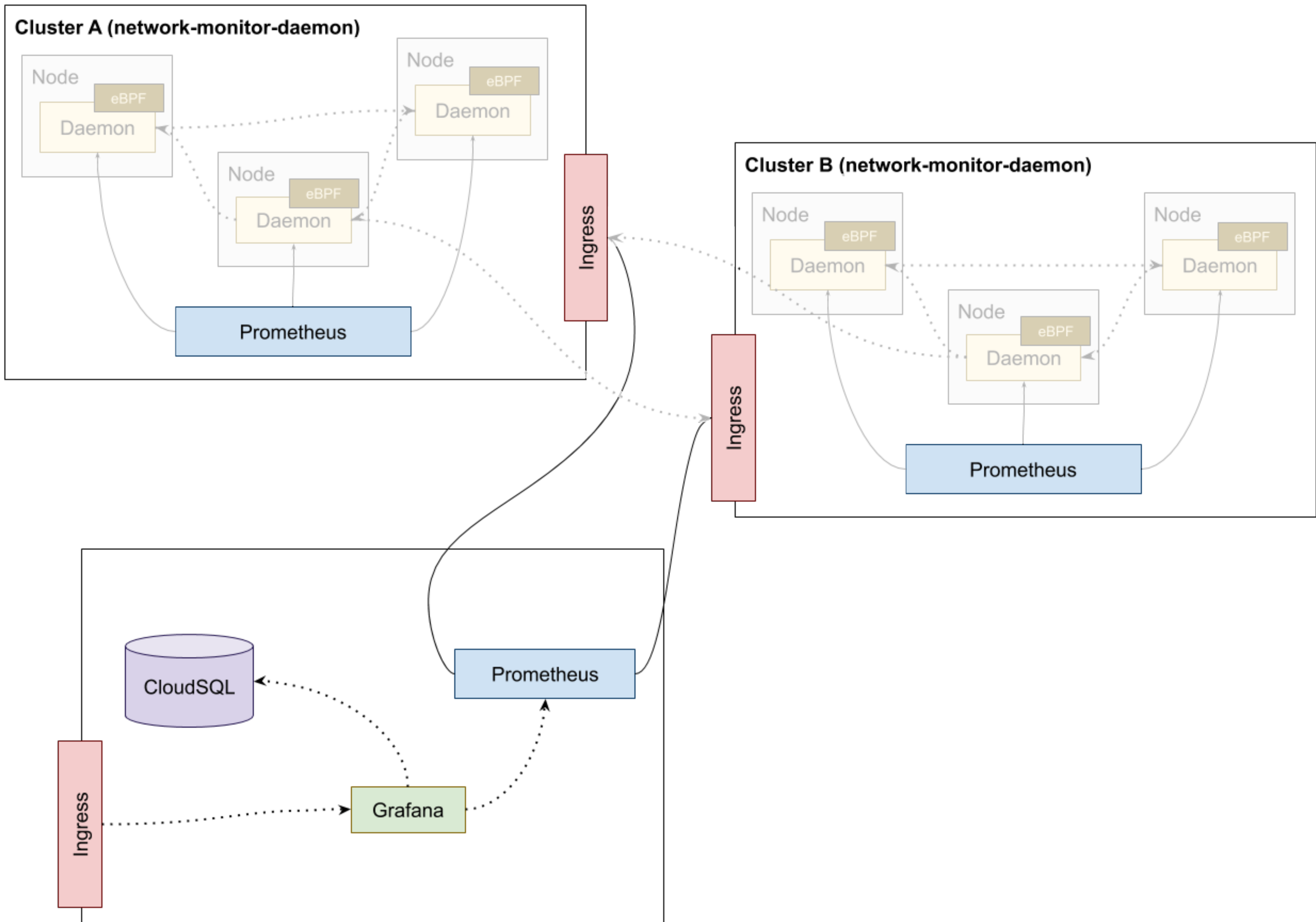
15:38 15:40 15:42 15:44 15:46 15:48 15:50 15:52 15:54 15:56 15:58 16:00 16:02 16:04 16:06

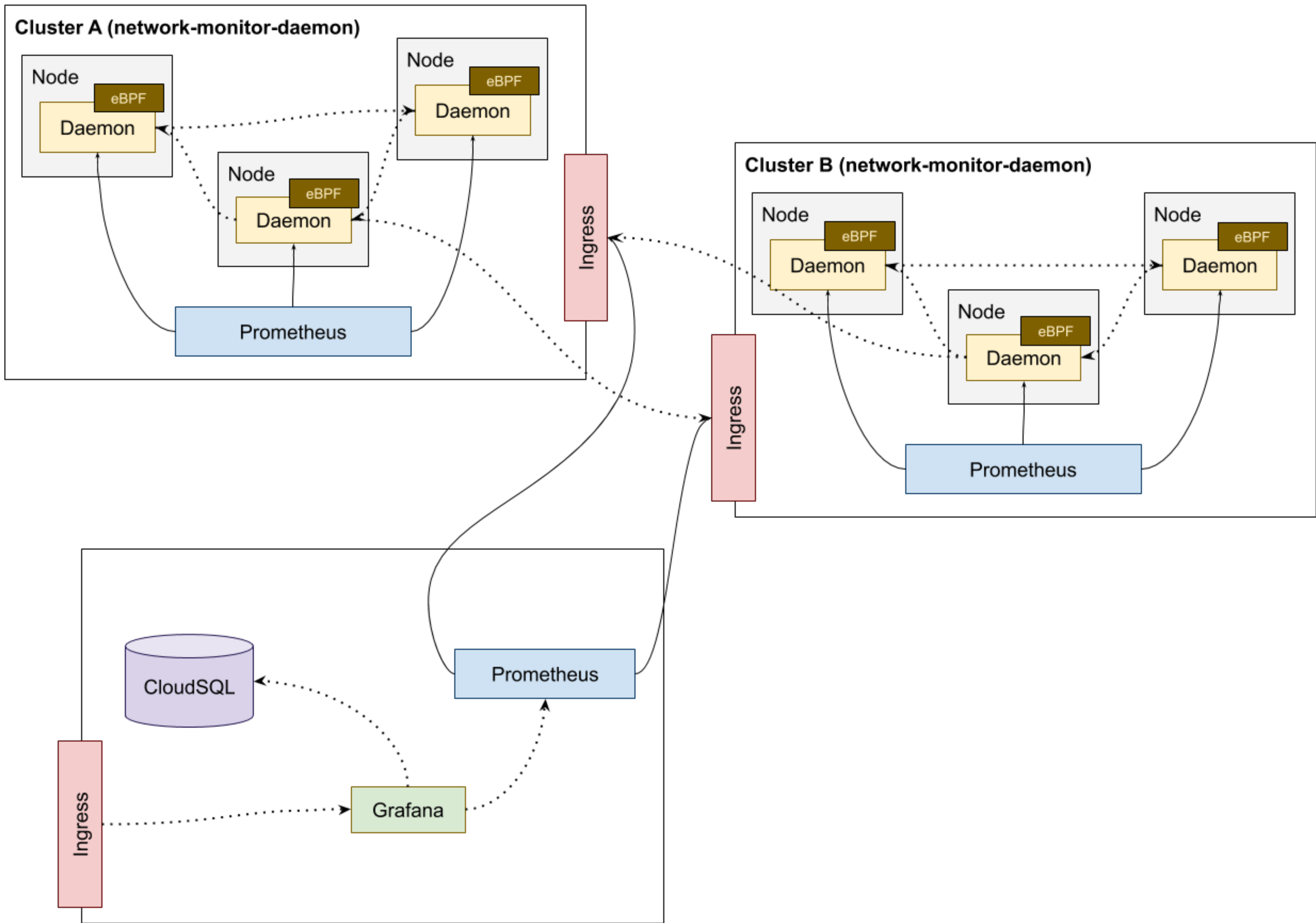
userspace max
BPF max
15:38
15:40
15:42
15:44
15:46
15:48
15:50
15:52
15:54
15:56
15:58
16:00
16:02
16:04
16:06


Cluster A (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
Cluster B (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
CloudSQL
Prometheus
Grafana
Ingress


Cluster A (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
Cluster B (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
CloudSQL
Prometheus
Grafana
Ingress


Cluster A (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
Cluster B (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
CloudSQL
Prometheus
Grafana
Ingress

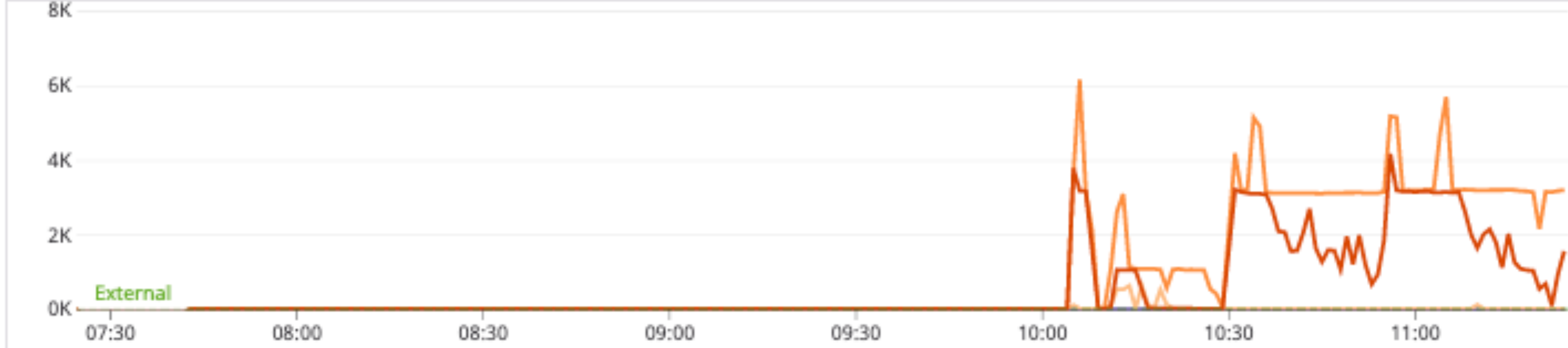

Cluster A (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
Cluster B (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
CloudSQL
Prometheus
Grafana
Ingress



Cluster A (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
Cluster B (network-monitor-daemon)
Node
eBPF
Daemon
Node
eBPF
Daemon
Node
eBPF
Daemon
Prometheus
Ingress
CloudSQL
Prometheus
Ingress
Grafana

Selected namespace health

Connection latency (ms; p95)



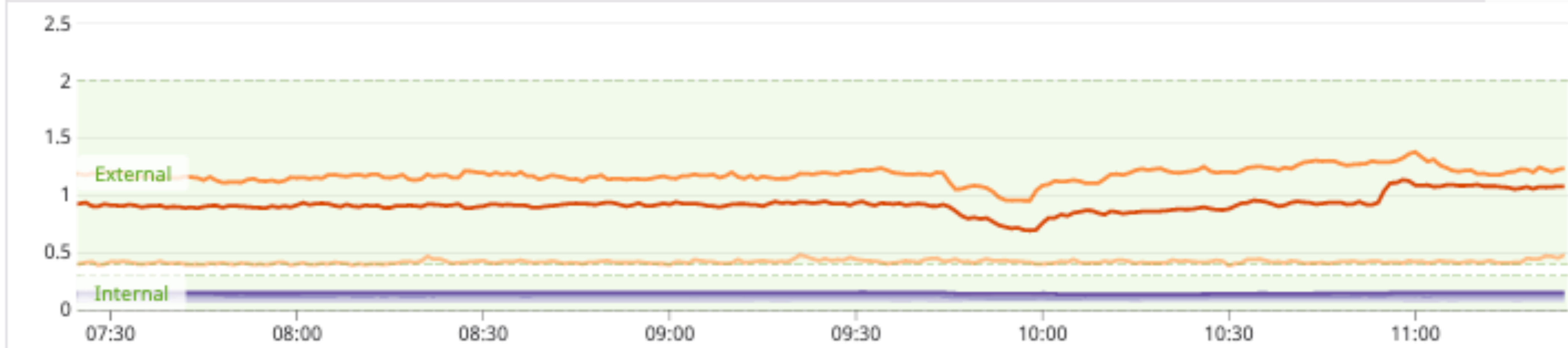
	Value	Min	Avg	Max	Metric	Tags ↓
	1.45	1.32	1.44	1.57	network_monitor.tcp_region_connection_latency_ms	external:no,source_namespace:network-monitor-daemon
	2.07	1.88	2.06	2.3	network_monitor.tcp_region_connection_latency_ms	external:no,source_namespace:network-monitor-daemon
	1.94	1.82	1.91	1.99	network_monitor.tcp_region_connection_latency_ms	external:no,source_namespace:network-monitor-daemon
	38.61	31.44	46.58	649.27	network_monitor.tcp_region_connection_latency_ms	external:yes,source_namespace:network-monitor-daemon
	3.14K	34.29	949.36	6.17K	network_monitor.tcp_region_connection_latency_ms	external:yes,source_namespace:network-monitor-daemon

Average TCP retransmission count

	Value	Sum	Metric	Tags ↓
	6.25	799.14	network_monitor.tcp_region_retransmit_count	external:no,source_namespace:network-monitor-daemon
	41.08	16.47K	network_monitor.tcp_region_retransmit_count	external:no,source_namespace:network-monitor-daemon
	119.59	19.09K	network_monitor.tcp_region_retransmit_count	external:no,source_namespace:network-monitor-daemon
	182.56	16.98K	network_monitor.tcp_region_retransmit_count	external:yes,source_namespace:network-monitor-daemon
	8.76K	211.85K	network_monitor.tcp_region_retransmit_count	external:yes,source_namespace:network-monitor-daemon

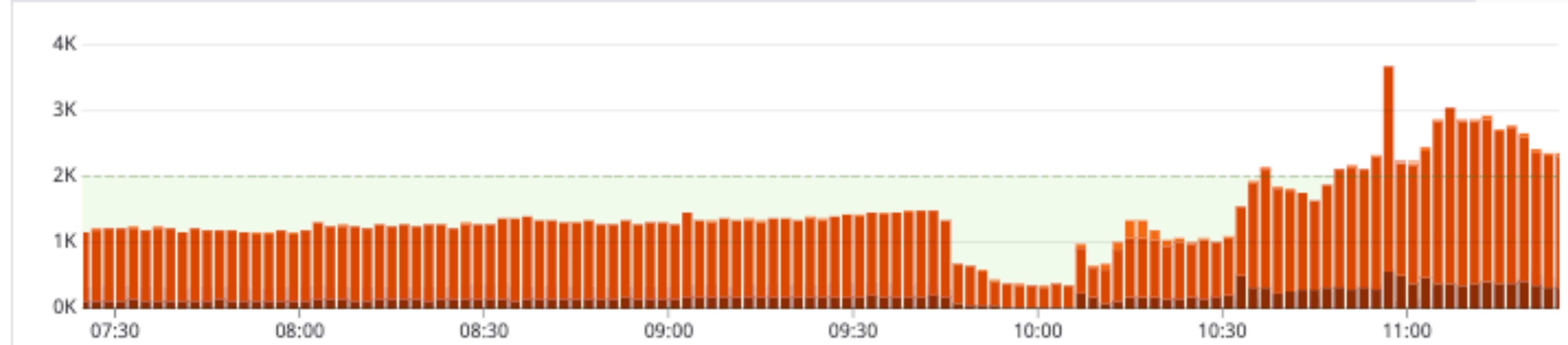
Region Health

Connection latency (ms; p50)



	Value	Min	Avg	Max	Metric	Tags ↓
	0.092	0.087	0.091	0.095	network_monitor.tcp_median_connection_latency_ms	external:no
	0.12	0.11	0.12	0.13	network_monitor.tcp_median_connection_latency_ms	external:no
	0.15	0.14	0.15	0.16	network_monitor.tcp_median_connection_latency_ms	external:no
	0.42	0.39	0.42	0.49	network_monitor.tcp_median_connection_latency_ms	external:yes
	1.24	0.96	1.18	1.38	network_monitor.tcp_median_connection_latency_ms	external:yes

Average TCP retransmission count



	Value	Sum	Metric	Tags ↓
	7.92	590.65	network_monitor.tcp_region_retransmit_count	external:no
	383.1	32.24K	network_monitor.tcp_region_retransmit_count	external:no
	77.19	5.58K	network_monitor.tcp_region_retransmit_count	external:no
	37.87	2.51K	network_monitor.tcp_region_retransmit_count	external:yes
	1.97K	148.02K	network_monitor.tcp_region_retransmit_count	external:yes

Attribution: https://youtu.be/E50-BxZq0l4

Takeaways

- How TCP works
- What eBPF is and how you can use it
- Why measuring in kernel space can give better data for things that happen quickly
- Thinking about where you could have a blind spot

Thanks!
shopify