Whoops!
I Rewrote it in Rust

Brian Martin (he/him)
Software Engineer at Twitter

2021
Rust
CONF

Brian Martin (he/him)

Software Engineer at Twitter

- At Twitter for 7 years, using Rust for 6 years
- Open source benchmarking, telemetry agent, cache server
- Volunteer Search and Rescue

Caching

- distributed key-value storage
    - `std::collections::HashMap` over network

Caching

- high throughput, low latency
  - 100k requests/s on a single core!
  - < 1 millisecond latency

Caching

- store frequently accessed items
  - protect database from high request rates

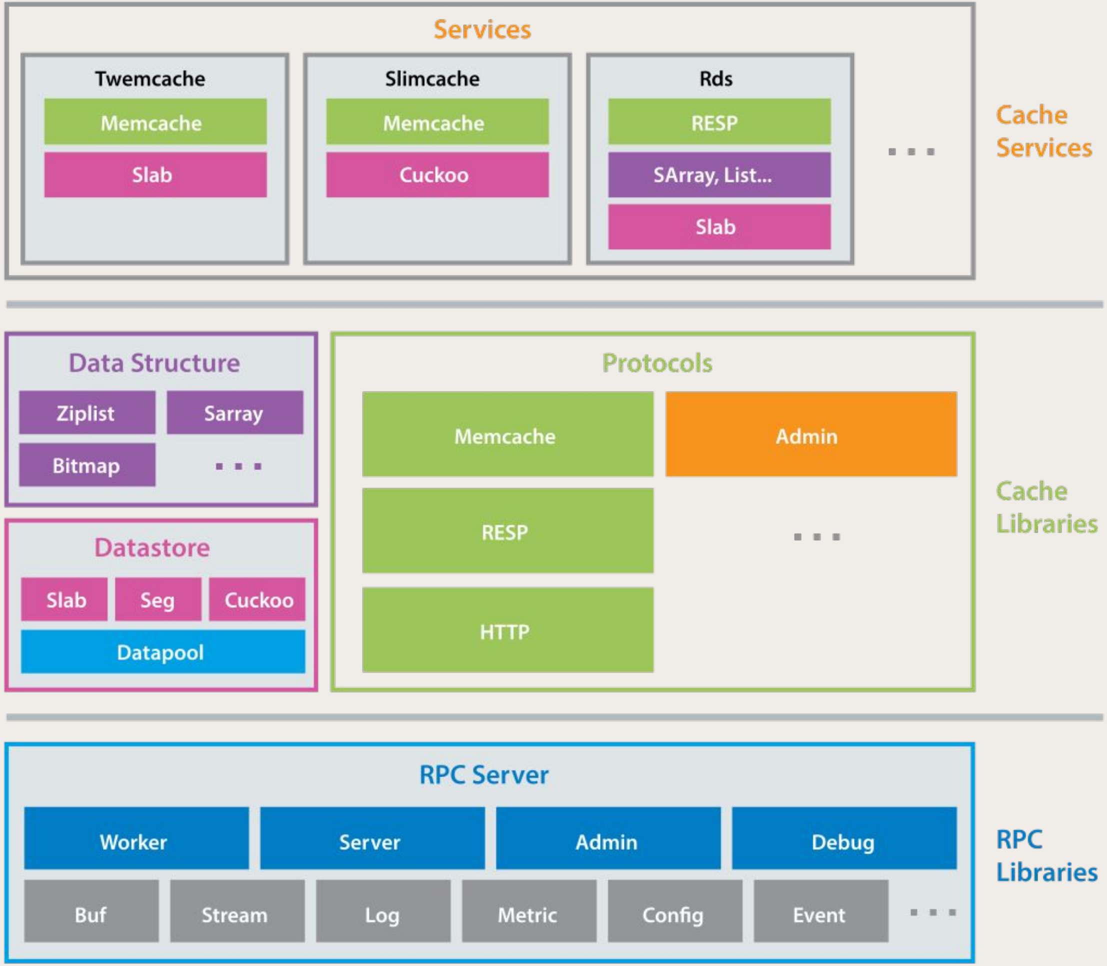
Caching

- several open source solutions
  - Memcached
  - Redis

Pelikan

- open source caching framework
- single codebase
- multiple solutions

http://pelikan.io



Services

Twemcache
Memcache
Slab

Slimcache
Memcache
Cuckoo

Rds
RESP
SArray, List...
Slab

. . .

Cache Services

Data Structure
Ziplist
Sarray
Bitmap
. . .

Datastore
Slab
Seg
Cuckoo
Datapool

Protocols
Memcache
Admin
RESP
. . .
HTTP

Cache Libraries

RPC Server
Worker
Server
Admin
Debug
Buf
Stream
Log
Metric
Config
Event
. . .

RPC Libraries

Use Rust to:

- add Transport Layer Security (TLS)
- match performance of C implementation

Previous Efforts

Rust Storage in Pelikan: 2018

Engineer wants to add storage to Pelikan and write the library in Rust

- Used the C framework with FFI to use Rust Storage
- First commit of Rust to Pelikan!

Rust Server in Pelikan: 2019

Engineer wants to use Rust for server code

- Tokio / async server
- Reuse C components for Storage / Parser / Buffers / Metrics / …
- Proved that we could use Rust for larger roles within Pelikan

Performance Testing

Apply synthetic workloads with ratelimit to measure latency at a specific request rate

Goal:

- maximize throughput
- latency below 1ms at p999 / 99.9th percentile

p999 / 99.9th percentile

- 0.1% of requests > p999

p999 / 99.9th percentile

- fanout = 1:
  - 0.1% of requests > p999
- fanout = 10:
  - 1% of requests > p999

p999 / 99.9th percentile

- fanout = 1:
  - 0.1% of requests > p999
- fanout = 10:
  - 1% of requests > p999
- fanout = 100:
  - 10% of requests > p999

p999 / 99.9th percentile

- fanout = 1:
  - 0.1% of requests > p999
- fanout = 10:
  - 1% of requests > p999
- fanout = 100:
  - 10% of requests > p999

Multiple cache requests for a higher-level request causes us to see tail latency more often than you think!

Pelikan Twemcache in Rust: 2019

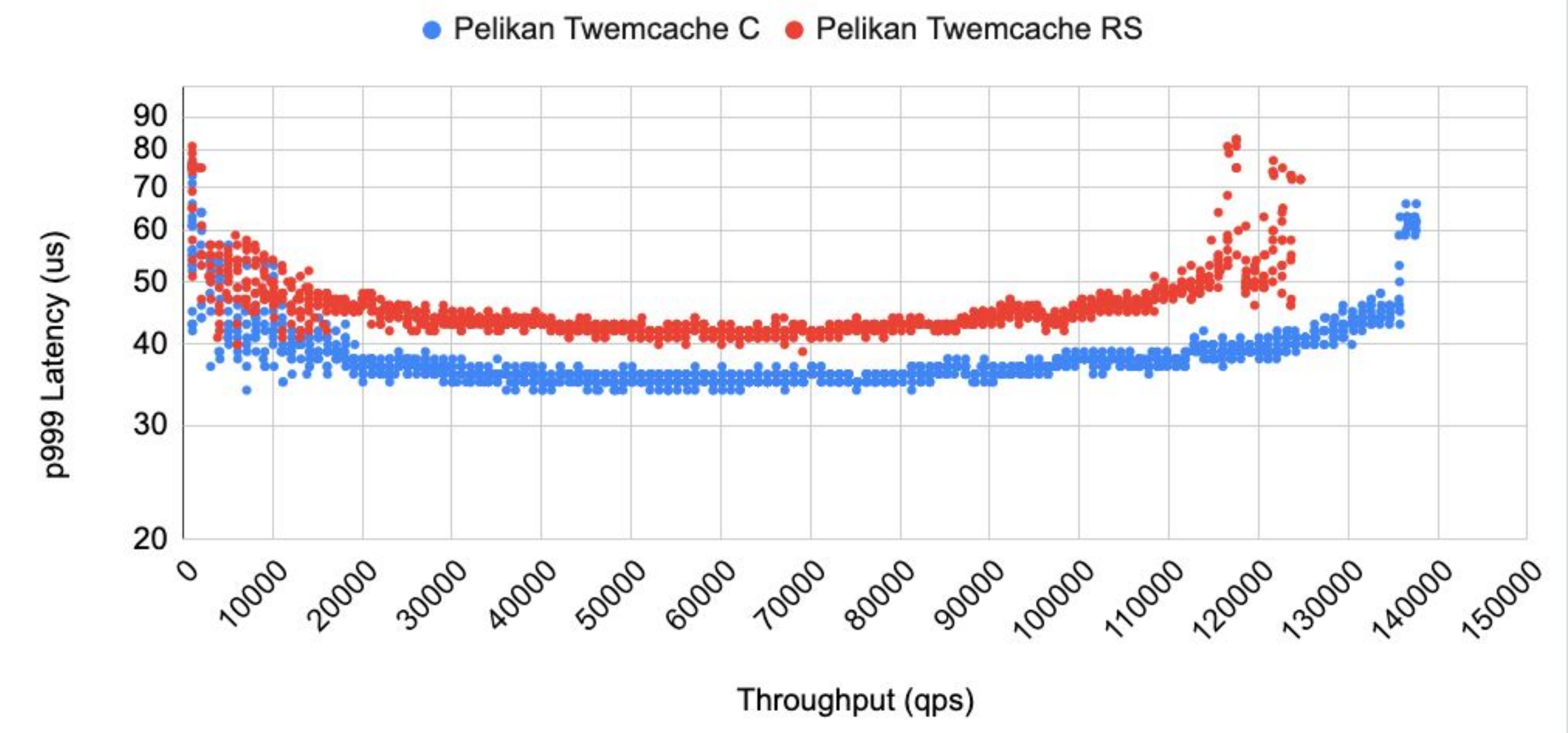
- Memcached compatible
- uses Tokio / async for networking
- uses C storage library

Can we use this to add TLS?

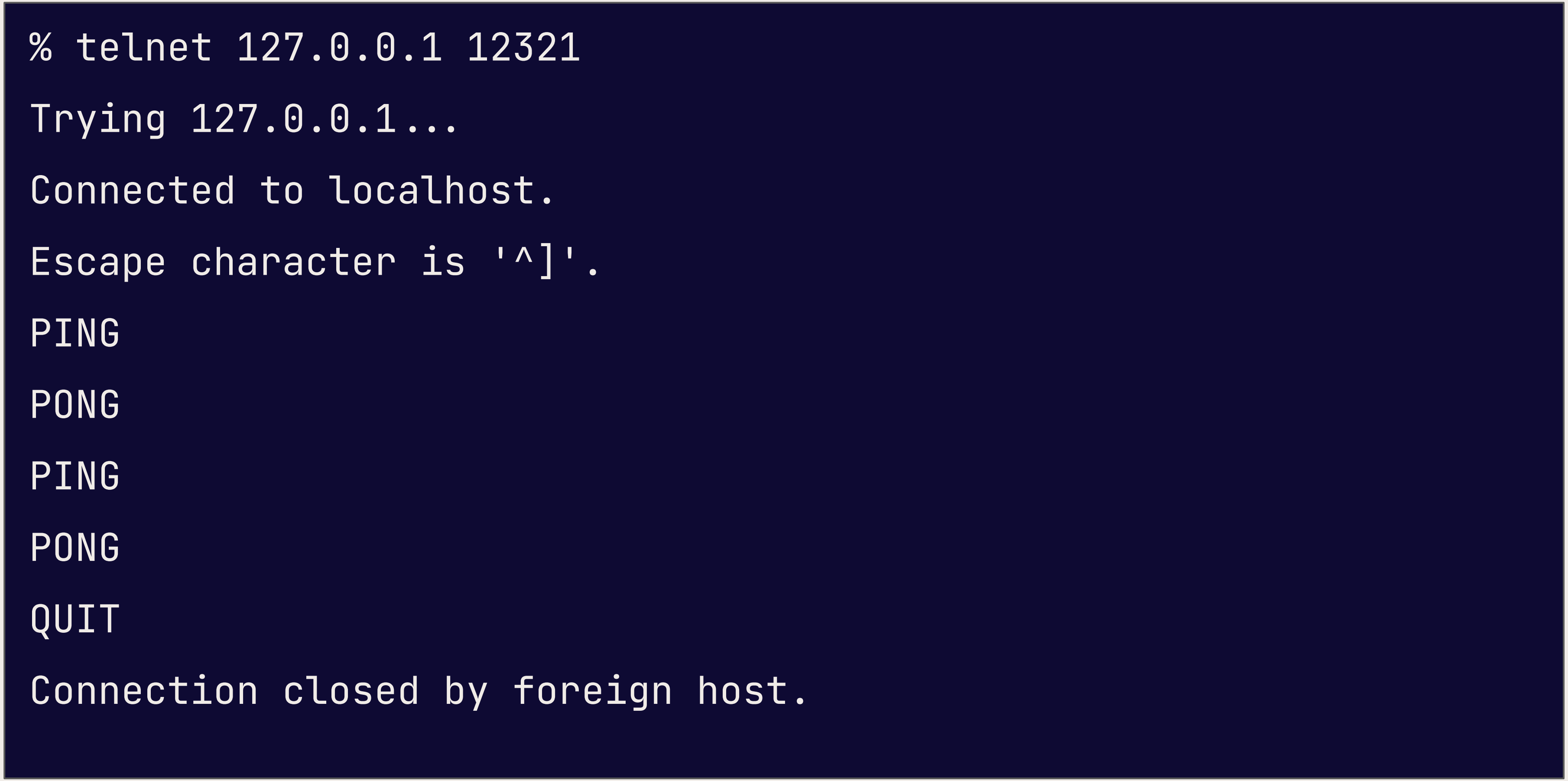
Performance Problems

- Throughput **10-15%** slower
  - need more instances to match throughput requirements
- Latency **25-30%** higher (@ p999 / 99.9th Percentile)
  - could cause timeouts/errors in production

Pelikan Twemcache: C vs Rust



Pingserver



```
% telnet 127.0.0.1 12321
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
PING
PONG
PING
PONG
QUIT
Connection closed by foreign host.
```

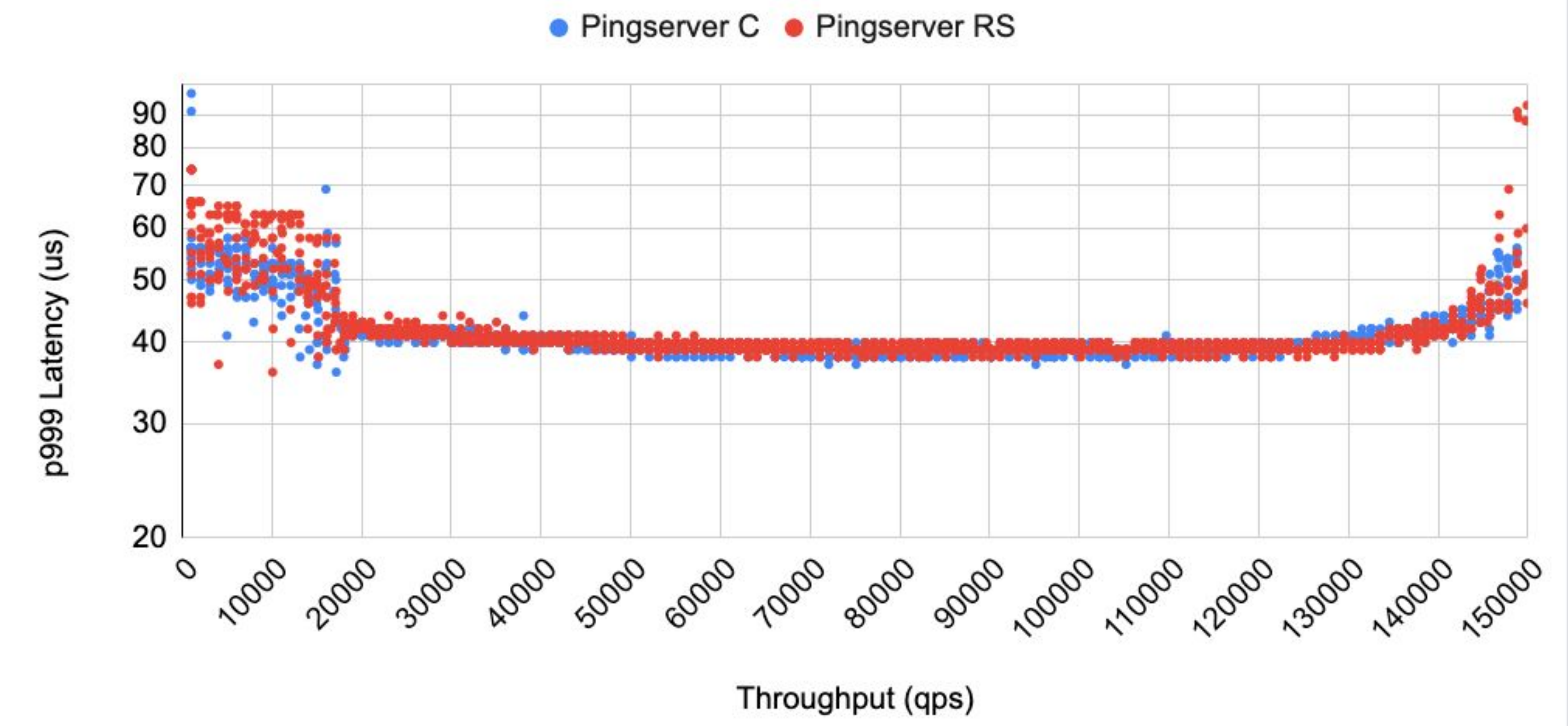
Why Pingserver?

- prove out performance of design
- buffers / parsers / metrics / logging
- can add TLS support

Pingserver v2

- replace `tokio` with `mio`
- replace most C components with Rust
  - metrics / logging / buffers / …
- TLS using `boringssl`

Pingserver: C vs Rust



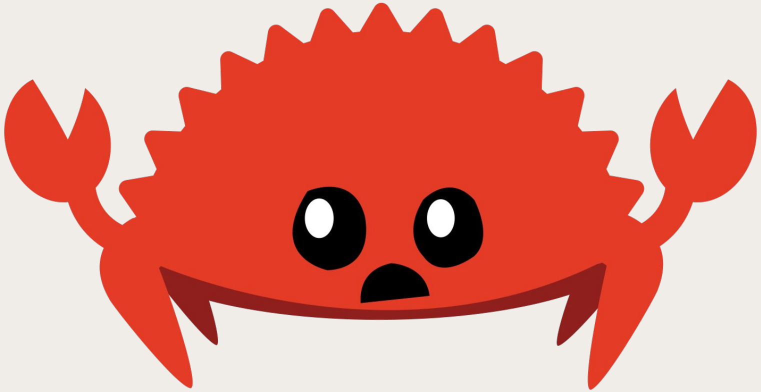
Pelikan Twemcache

Prototype

- Memcached protocol compatible
- wrapped `std::collections::HashMap` as temporary storage
- benchmarking looked good
- next step: FFI for C storage library

Oh no!

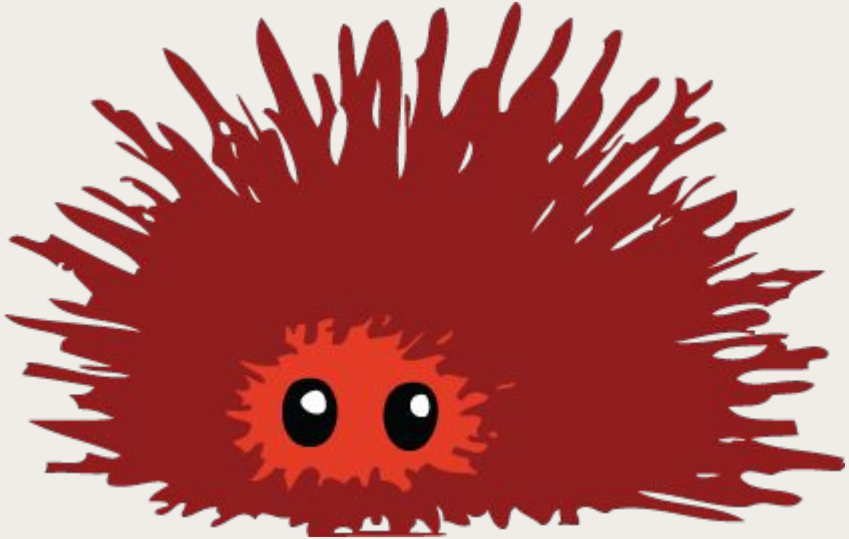
- metrics / logging causing problems



Rewrite it in Rust?

No. Don’t do it.

- self-referential data structures
- memory allocations!
- linked lists!
- pointers everywhere!
- unsafe { }

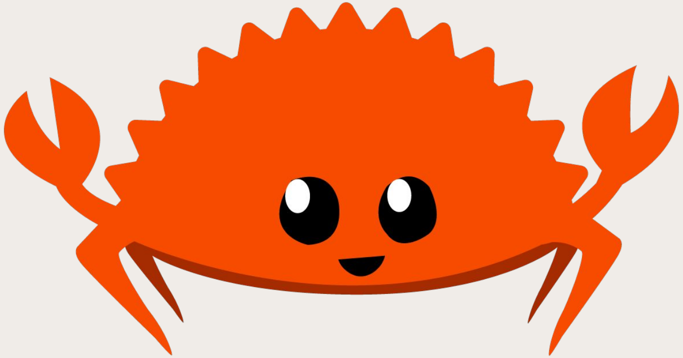


But... maybe yes?

- start with single-threaded
- only hashtable in Rust?
- use C slab allocator?

Whoops! All Rust

- implemented new research storage design
    - easier to port to Rust
- single-threaded
    - easy mode
- managing all memory allocations in Rust



Rewrite Downsides

- cost ~2 months of work
  - duplicating previous efforts
- not exact feature parity
  - more work to be done

Rewrite Benefits

- new storage design
- added new ideas to storage library
- helped make it more production ready

Rust Benefits

“empowering everyone to build reliable and efficient software”

- high performance
- code with confidence in reliability
- awesome language features and tools
- zero cost abstraction

Rust Benefits

“The Rust implementation finally made Pelikan modules what they ought to be, but couldn’t before, due to the limitation of the C language. This feels exactly right and is just as fast”

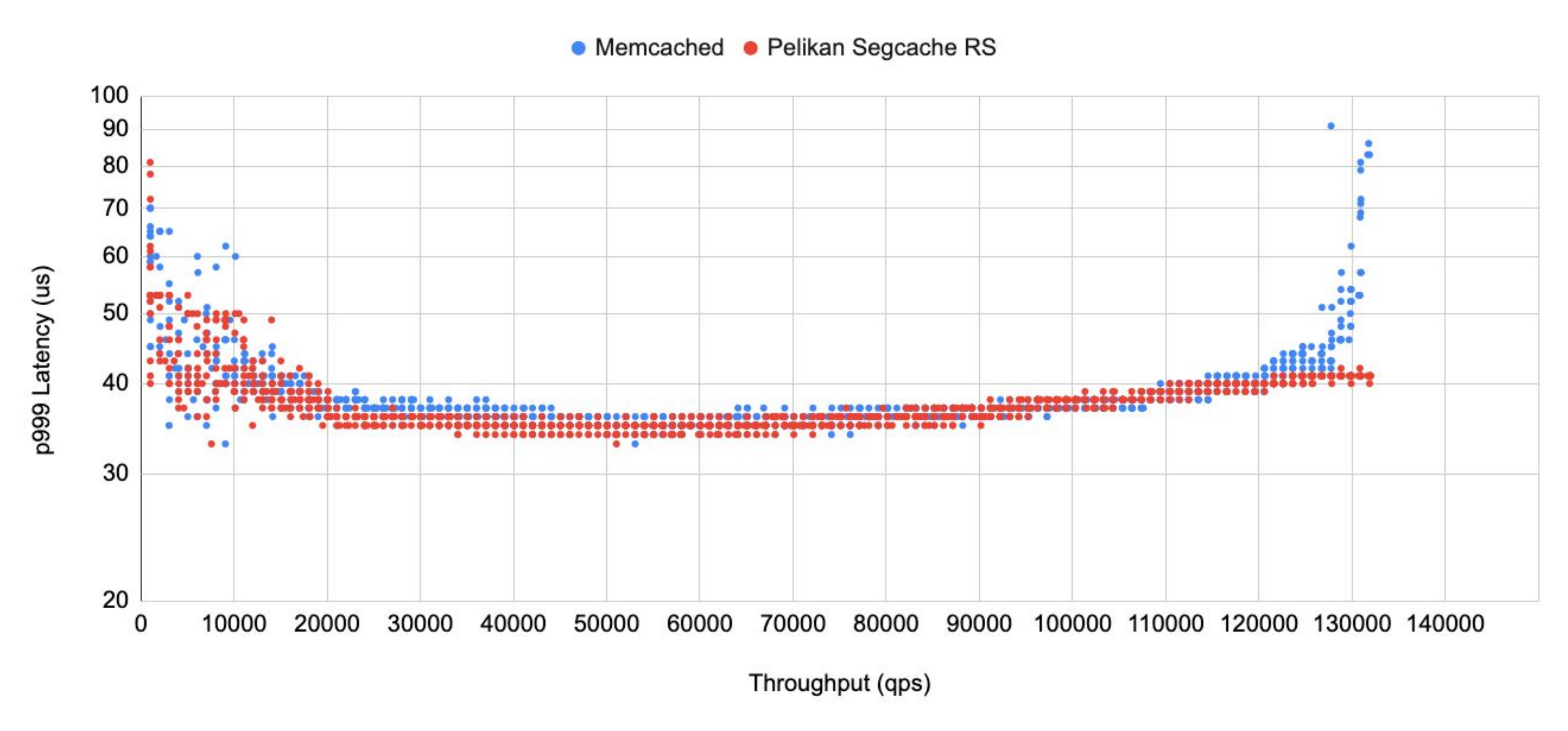
- Yao (@thinkingfish)

Rust Tooling Benefits

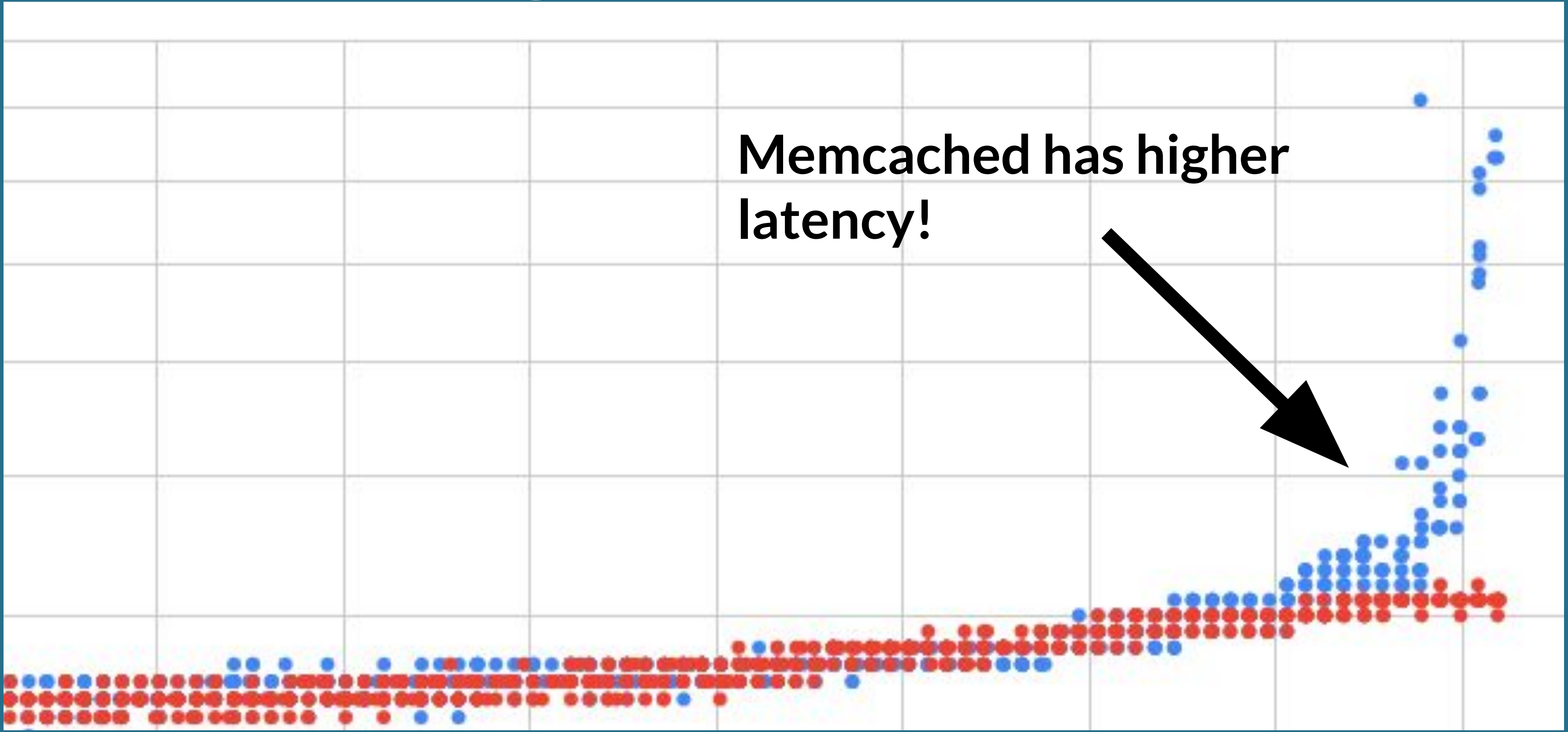
- cargo bench & criterion
  - microbenchmarking of critical components
- cargo fuzz
  - easy to add fuzz testing for protocol library

```
// Segcache is Memcache protocol + Seg storage


type Parser = MemcacheRequestParser;

type Request = MemcacheRequest;

type Response = MemcacheResponse;

type Storage = Seg;

...


let process_builder = ProcessBuilder::<Storage, Parser, Request,
Response>::new(...);
```

Memcached vs Segcache RS



Memcached vs Segcache RS



Memcached has higher latency!

Conclusions

Next Steps

Path to Production:

- feature complete
- more testing
- production canary
- deployment

Future Work:

- multi-threaded performance
- optimization
- Redis replacement
- io_uring
- ...

Rewriting has costs and benefits

Costs

- Extra time would have caused missed deadlines
- Duplicating work that’s been paid for

Benefits

- Easier to work with an all Rust codebase
- No more cmake!!!
- New ideas got added

C and Rust are both very fast

Profiling and benchmarking helped get us match the performance of the C implementation.

Rust ecosystem has awesome tools

- cargo
- fuzzers
- benchmarks
- rustfmt
- clippy

Pelikan has an exciting future with Rust

Lots of fun work to come!

Thanks!