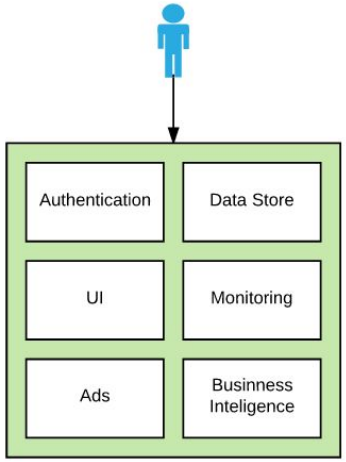
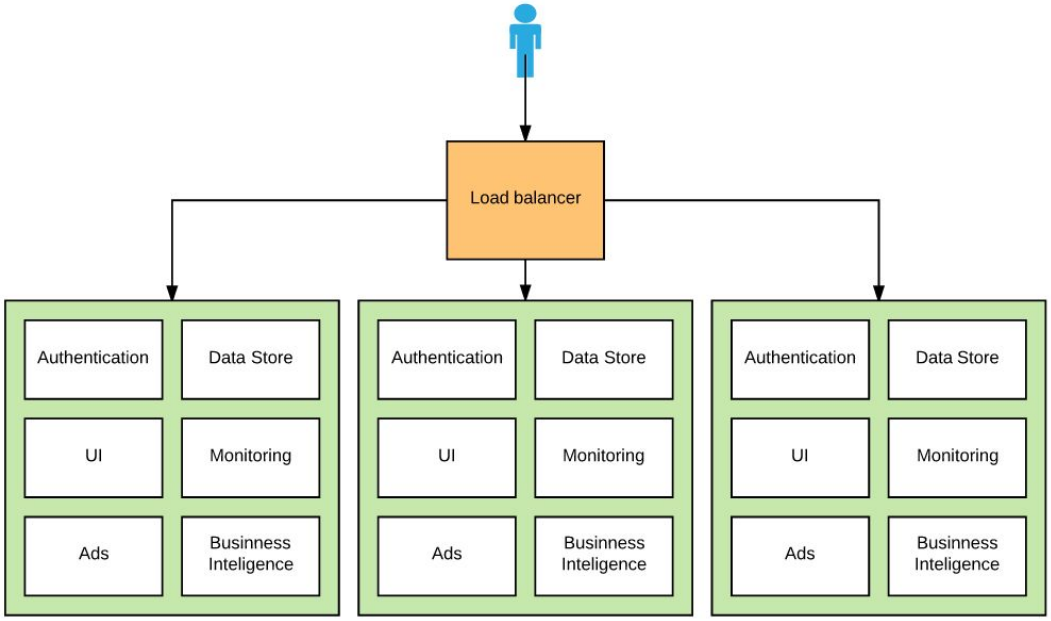
Mikroserwisy I/O bound

w poszukiwaniu wydajności
22.05.2017

Mikroserwisy



Mikroserwisy



CPU bound

- Szybkość przetwarzania ograniczona szybkością CPU
- Przyspieszenie:
  - Szybszy algorytm
  - Szybszy procesor
  - Więcej procesorów

IO bound

- Szybkość przetwarzania ograniczona przepustowością IO
  - Dysk
  - Sieć
  - Urządzenia zewnętrzne
- Przyspieszenie
  - Odpowiednia architektura
  - Tuning systemu operacyjnego

(Mikro)serwisy IO bound

- Ciągły strumień zgłoszeń
- Większość czasu spędzana na oczekiwaniu na IO

Klasyczne rozwiązanie

- 1 proces per połączenie
  - Inetd
  - Apache Httpd
  - Sendmail
- Usprawnienia
  - Pula procesów
  - Wątki zamiast procesów
  - Pula wątków

Klasyczne rozwiązanie

- Zalety
  - Prosta implementacja (Odczytaj z socketa, przetwórz, odpowiedz)
  - Iluzja synchronicznego kodu
- Wady
  - Kolejkowanie zgłoszeń
  - Wydłużenie czasu oczekiwania
  - Zasoby zajęte przez długi czas
  - Kosztowne tworzenie procesów/wątków
  - Kosztowne przełączanie kontekstu
  - “C10K problem”

Rozwiązanie asynchroniczne

- Wzorce Reactor i Proactor
- 1 wątek per CPU
- Event loop
  - poll()/epoll()
  - Przetwarzanie
  - Nieblokujące IO

Rozwiązanie asynchroniczne

- Zalety
  - Dobra wydajność
- Wady
  - Bardzo złożony kod (callbacki, maszyna stanów)
  - Wymagana dyscyplina - blokujące IO niedozwolone
  - Skomplikowana architektura współczesnych komputerów
  - Konieczny dobór odpowiedniego języka programowania

Inne rozwiązania

- Seastar
- Go

Seastar

- https://github.com/scylladb/seastar
- *“SeaStar is an event-driven framework allowing you to write non-blocking, asynchronous code in a relatively straightforward manner (once understood). It is based on futures.”*

Seastar

- Cooperative micro-task scheduler
- Share-nothing SMP architecture
- Future based APIs
- Share-nothing TCP stack
- DMA-based storage APIs [*]

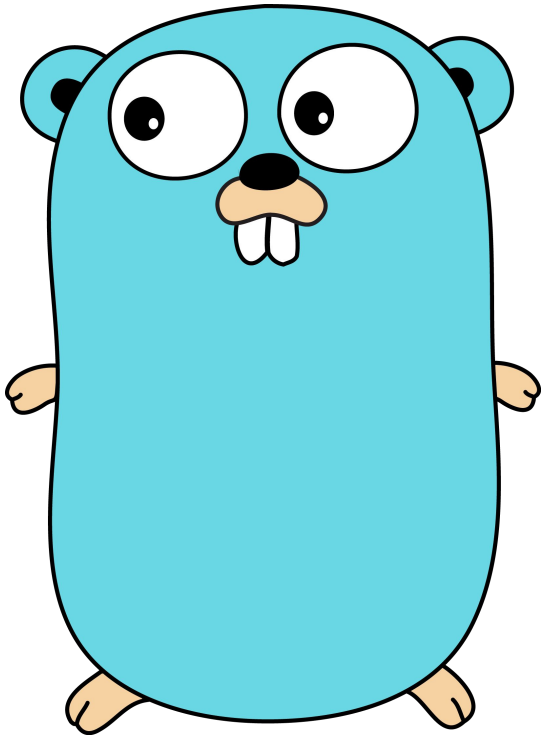
* https://github.com/scylladb/seastar/blob/master/doc/tutorial.md

Go

- https://golang.org/
- *“The Go programming language is an open source project to make programmers more productive. Go is expressive, **concise**, clean, and efficient. Its **concurrency** mechanisms make it easy to write programs that get the most out of **multicore and networked machines**, while its novel type system enables flexible and modular program construction. Go compiles quickly to machine code yet has the convenience of **garbage collection** and the power of run-time reflection. It's a fast, **statically typed, compiled language** that feels like a dynamically typed, interpreted language.”*

Go - architektura wewnętrzna

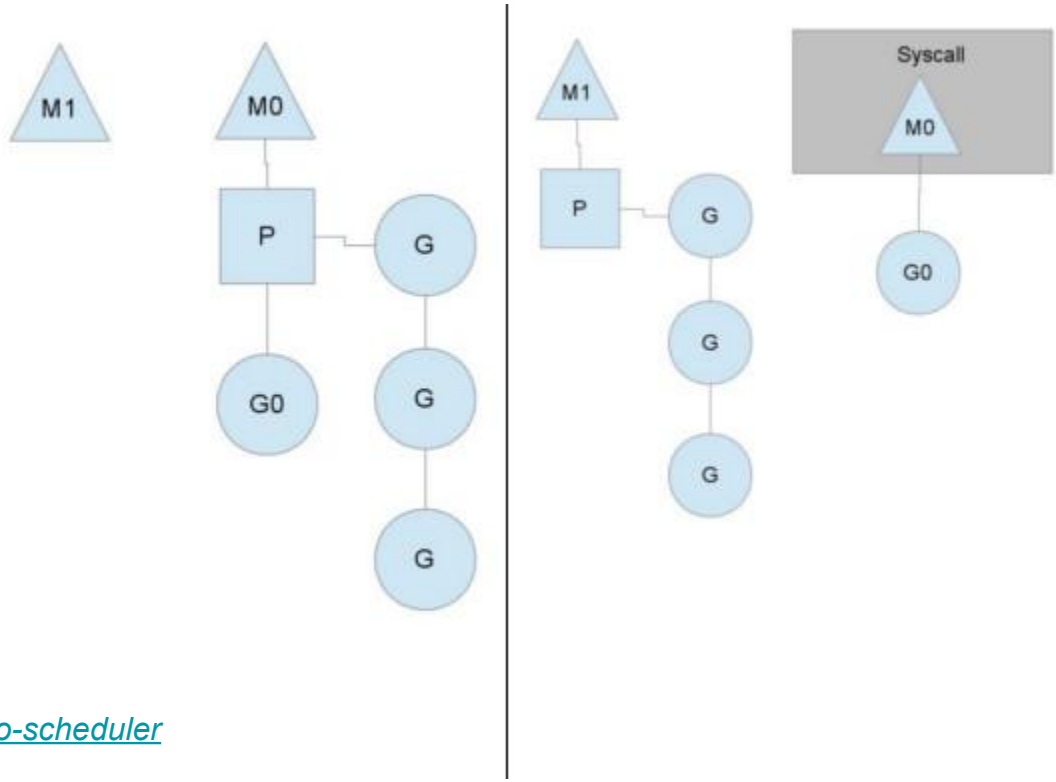
- Software scheduler
- Gorutyny
- Przełączanie gorutyn na syscallach



Go - architektura wewnętrzna

- M - machine - wątek systemowy
  - Tyle, ile rdzeni procesora
  - Czasem więcej
- P - processor - kontekst
  - Runqueue - kolejka gorutyn do wykonania
- G - gorutyna

Go - syscall



https://morsmachine.dk/go-scheduler

Http echo

- Kod na Githubie
  - https://github.com/p-kozlowski/techtalk-iobound
- Przykład

```
$ curl http://192.168.0.4:10000/users/Piotr

Hello, Piotr
```

Seastar - echo

```
class userHandler : public httpd::handler_base {
public:
     future<std::unique_ptr<reply>> handle(
                    const sstring& path,
                    std::unique_ptr<request> req,
                    std::unique_ptr<reply> rep) override {
          rep->_content = "Hello, " + req->param["name"];
          rep->done("html");
          return make_ready_future<std::unique_ptr<reply>>(std::move(rep));
     }
};
```

Seastar - echo

```
class userHandler : public httpd::handler_base {
public:
    future<std::unique_ptr<reply>> handle(
                const sstring& path,
                std::unique_ptr<request> req,
                std::unique_ptr<reply> rep) override {
        rep->_content = "Hello, " + req->param["name"];
        rep->done("html");
        return make_ready_future<std::unique_ptr<reply>>(std::move(rep));
    }
};
```

Go - echo

```
func getUser(c echo.Context) error {
	name := c.Param("name")
	return c.String(http.StatusOK, fmt.Sprintf("Hi, %s!", name))
}
```

Pomiary

- OpenStack
- Serwer
  - 8 vCPU
  - 16GB RAM
  - Dysk 160GB
- Klient
  - 4 vCPU
  - 8GB RAM
  - Dysk 80GB

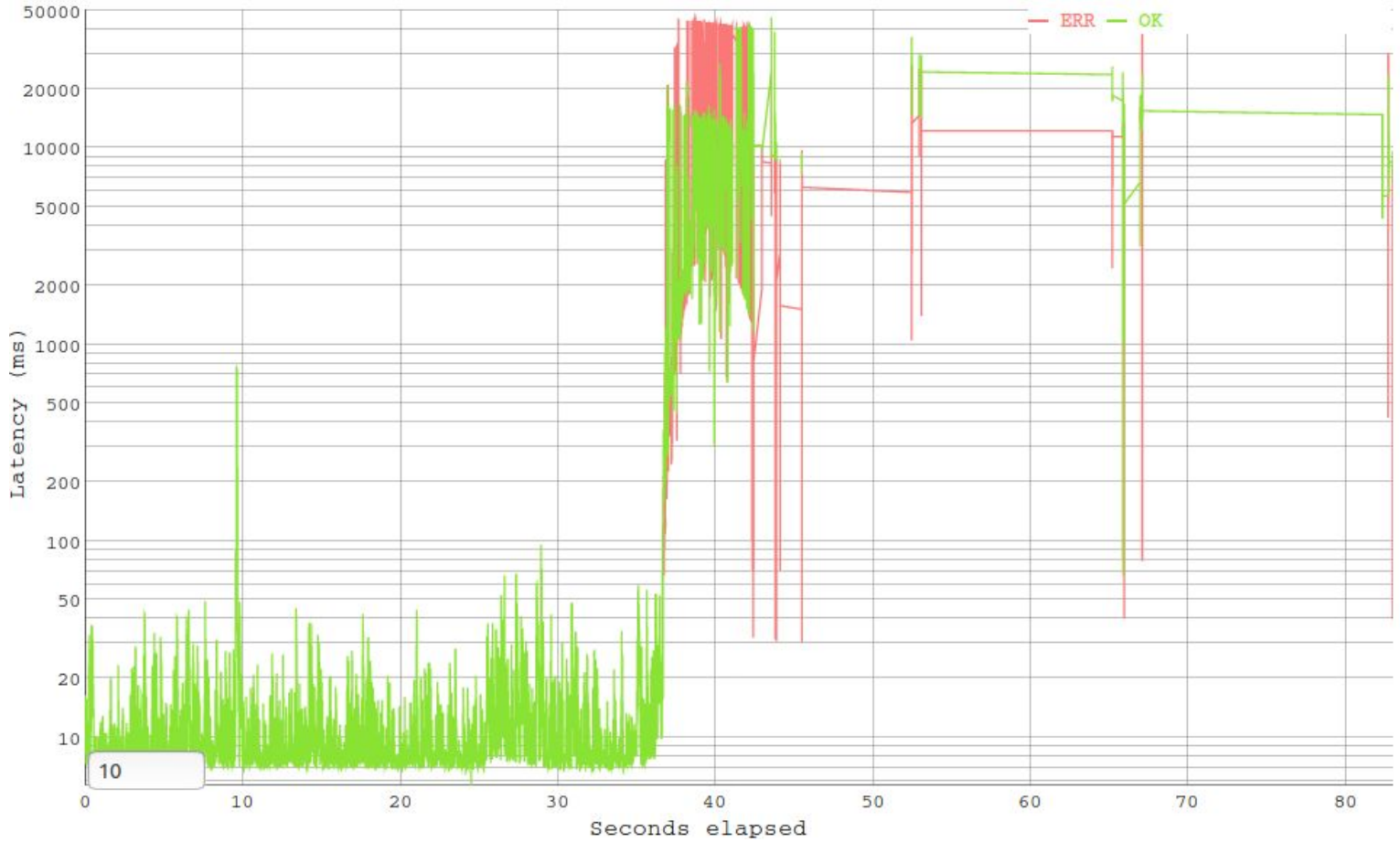
Pomiary - Vegeta

- https://github.com/tsenart/vegeta
- *“Vegeta is a versatile HTTP load testing tool built out of a need to drill HTTP services with a constant request rate. It can be used both as a command line utility and a library.”*

Vegeta attack

```
echo "GET http://192.168.0.4:10000/users/attack" |
./vegeta attack -duration=60s -rate=5000 |
tee results.bin |
./vegeta report
```

Wynik


ERR
OK
50000
20000
10000
5000
2000
1000
500
200
100
50
20
10
Latency (ms)
10
0
10
20
30
40
50
60
70
80
Seconds elapsed

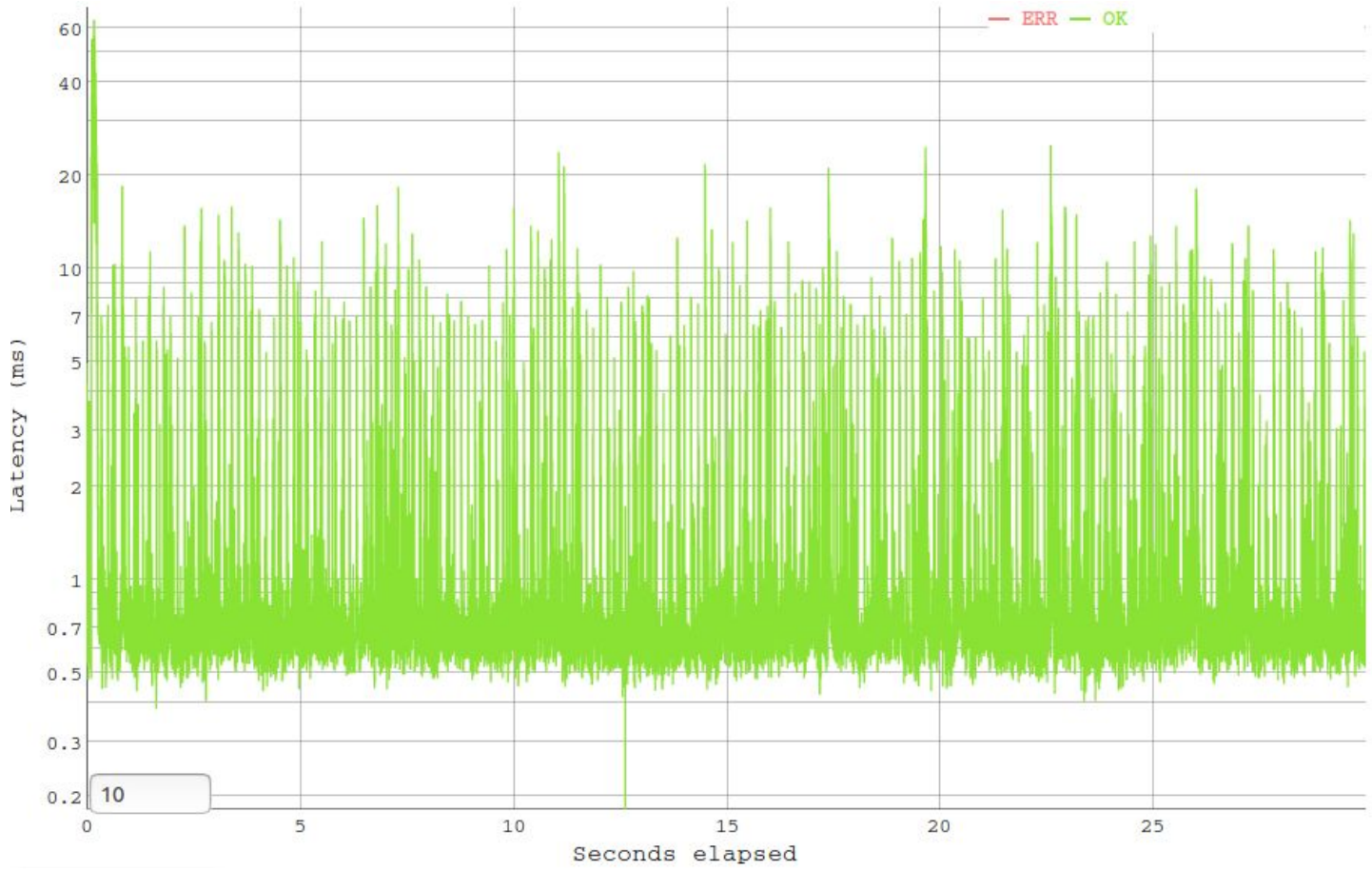
Tuning

```
# ./posix_net_conf.sh ens3

Restarting irqbalance: going to ban the following IRQ numbers: 26, 28, 27 ...

Setting mask 00000001 in /proc/irq/26/smp_affinity

Setting mask 00000001 in /proc/irq/28/smp_affinity

Setting mask 00000001 in /proc/irq/27/smp_affinity

Setting mask 000000fe in /sys/class/net/ens3/queues/rx-0/rps_cpus

Setting net.core.rps_sock_flow_entries to 32768

Setting limit 32768 in /sys/class/net/ens3/queues/rx-0/rps_flow_cnt

Setting mask 000000ff in /sys/class/net/ens3/queues/tx-0/xps_cpus
```

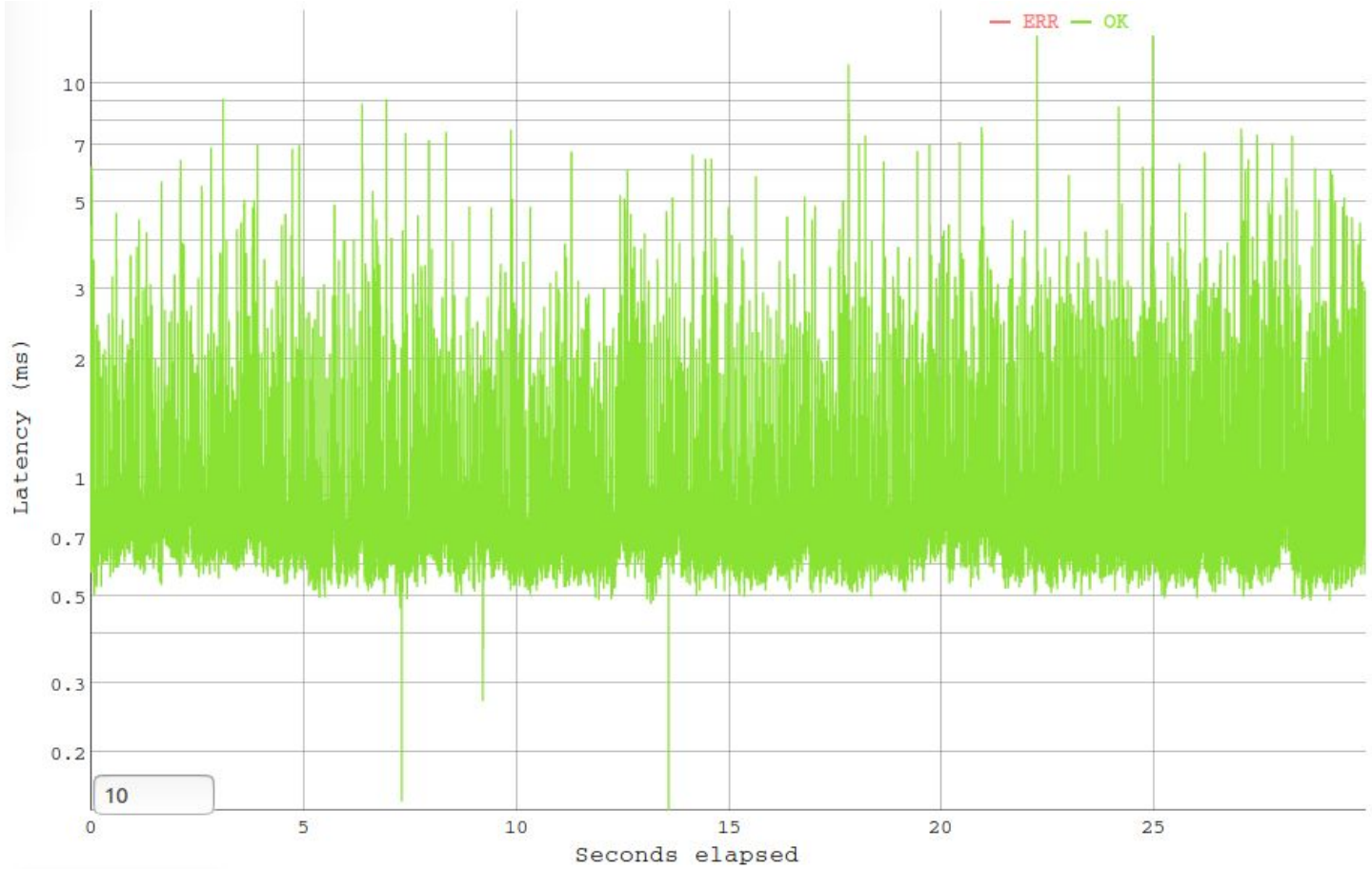
Tuning, ciąg dalszy

```
net.core.wmem_max = 25165824

net.core.rmem_max = 25165824

net.ipv4.tcp_rmem = 20480 174760 25165824

net.ipv4.tcp_wmem = 20480 174760 25165824

net.ipv4.tcp_window_scaling = 1

net.ipv4.tcp_timestamps = 1

net.ipv4.tcp_sack = 1

net.core.netdev_max_backlog = 30000

net.ipv4.tcp_max_syn_backlog = 32768

net.core.somaxconn = 32768
```

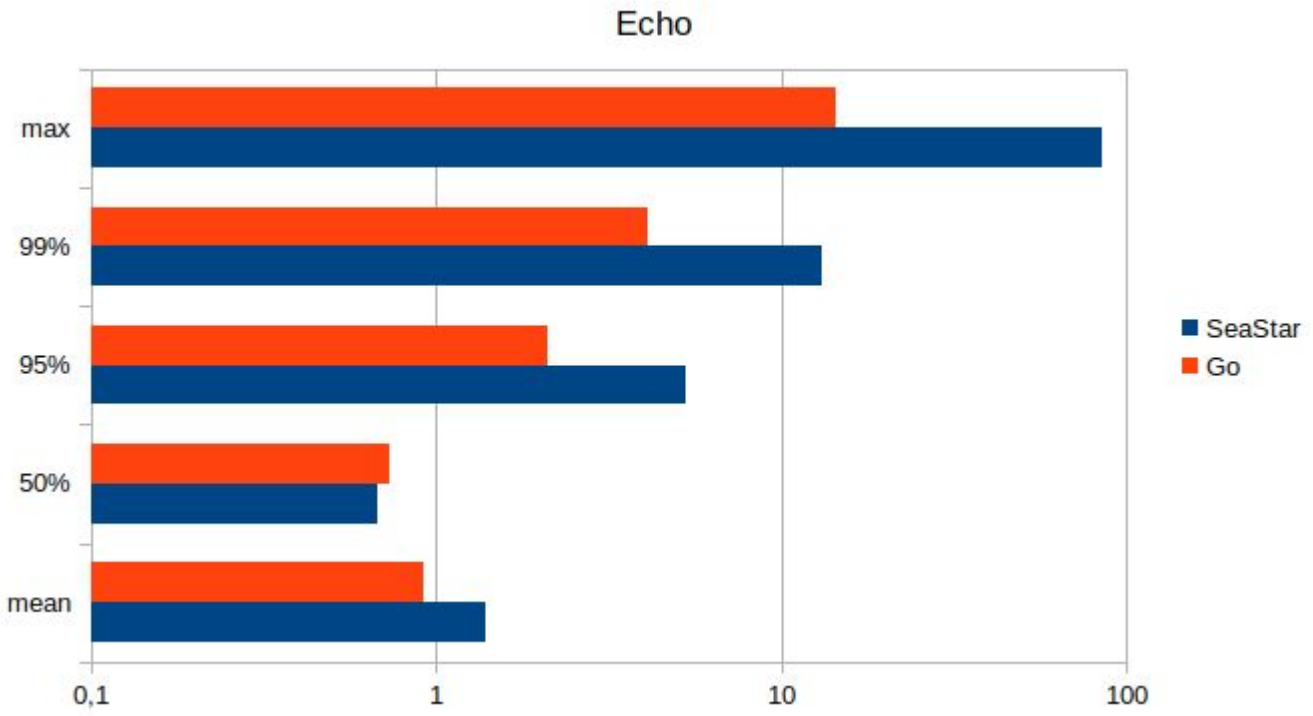
Wynik - SeaStar



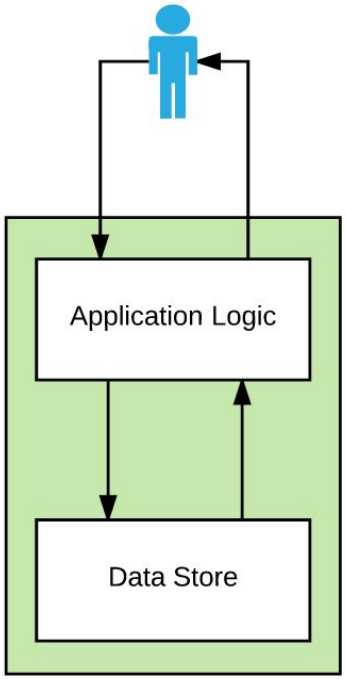
Wynik - Go


ERR
OK
Latency (ms)
10
7
5
3
2
1
0.7
0.5
0.3
0.2
10
0
5
10
15
20
25
Seconds elapsed

Wynik



Testowy serwis



SeaStar - service

```
future<std::unique_ptr<reply>> handle(const sstring& path,
        std::unique_ptr<request> req, std::unique_ptr<reply> rep) override {
        auto name = req->param["name"];
        return connect(this->dataStoreAddr).then([name, req = std::move(req), rep = std::move(rep)](connected_socket s) mutable {
                return do_with(std::move(s), [name, rep = std::move(rep)](connected_socket& s) mutable {
                        return do_with(s.output(), [name, &s, rep = std::move(rep)](auto& os) mutable {
                                auto f = os.write(name+sstring("\n"));
                                return f.then([&s, &os, rep = std::move(rep)]() mutable {
                                        auto f = os.flush();
                                        return f.then([&s, &os,rep = std::move(rep)]() mutable {
                                                return do_with(s.input(), [rep = std::move(rep)](auto& in) mutable {
                                                        auto f = in.read();
                                                        return f.then([rep = std::move(rep)](temporary_buffer<char> buf) mutable {
                                                                rep->_content = sstring("Hello, ") + sstring(buf.get(), buf.size());
                                                                rep->done("html");
                                                                return make_ready_future<std::unique_ptr<reply>>(std::move(rep));
                                                        });
                                                });
                                        });
                                });
                        });
                });
        });
}
```

SeaStar - service

```
future<std::unique_ptr<reply>> handle(const sstring& path,
        std::unique_ptr<request> req, std::unique_ptr<reply> rep) override {
        auto name = req->param["name"];
        return connect(this->dataStoreAddr).then([name, req = std::move(req), rep = std::move(rep)](connected_socket s) mutable {
                return do_with(std::move(s), [name, rep = std::move(rep)](connected_socket& s) mutable {
                        return do_with(s.output(), [name, &s, rep = std::move(rep)](auto& os) mutable {
                                auto f = os.write(name+sstring("\n"));
                                return f.then([&s, &os, rep = std::move(rep)]() mutable {
                                        auto f = os.flush();
                                        return f.then([&s, &os,rep = std::move(rep)]() mutable {
                                                return do_with(s.input(), [rep = std::move(rep)](auto& in) mutable {
                                                        auto f = in.read();
                                                        return f.then([rep = std::move(rep)](temporary_buffer<char> buf) mutable {
                                                                rep->_content = sstring("Hello, ") + sstring(buf.get(), buf.size());
                                                                rep->done("html");
                                                                return make_ready_future<std::unique_ptr<reply>>(std::move(rep));
                                                        });
                                                });
                                        });
                                });
                        });
                });
        });
}
```

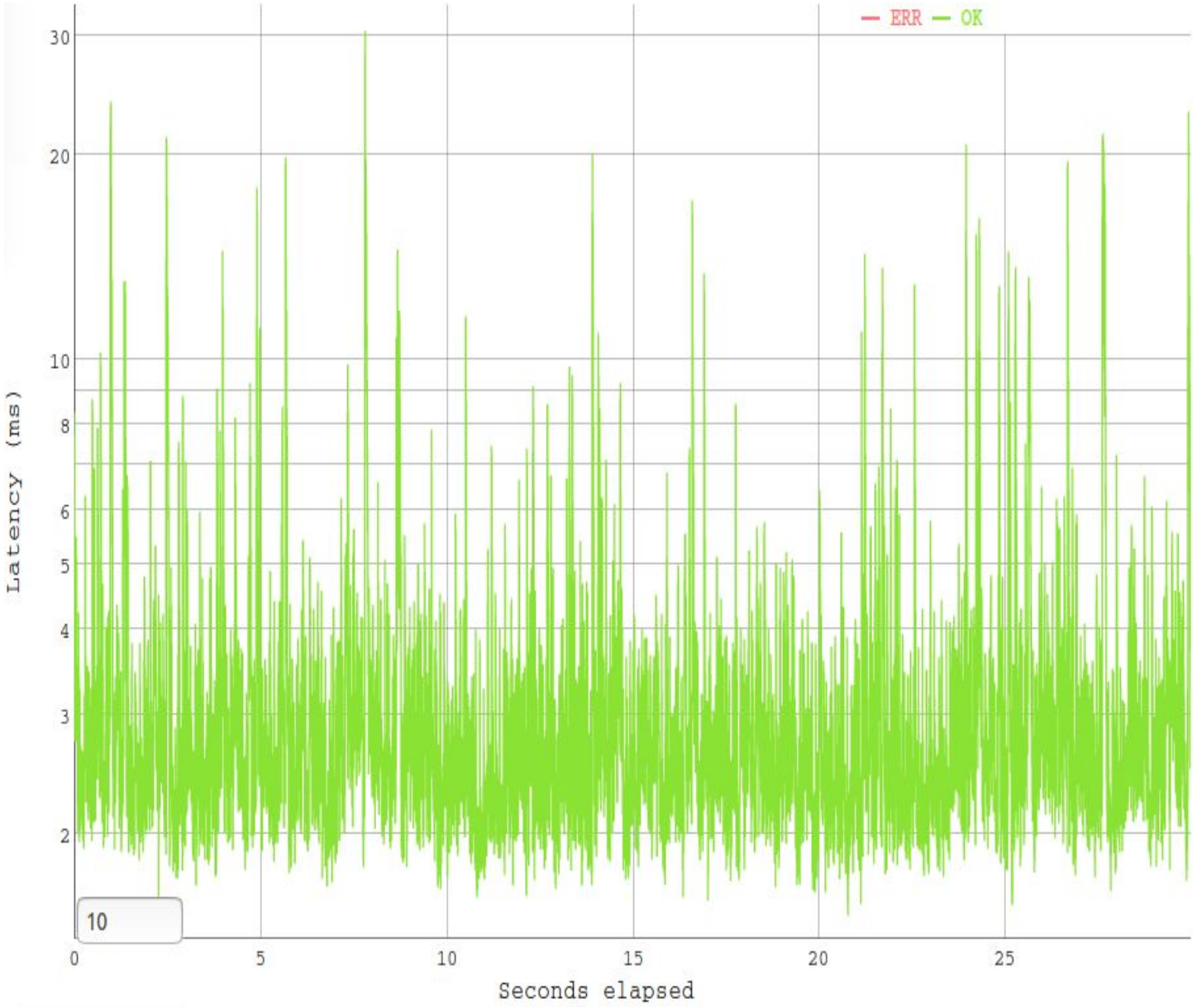
Go - service

```
func getUser(c echo.Context) error {
	conn, err := net.Dial("tcp", datastoreAddr)
	if err != nil {
		return c.String(http.StatusInternalServerError, "Error connecting to the datastore")
	}
	defer conn.Close()

	name := c.Param("name")
	fmt.Fprintf(conn, "%s\n", name)

	greeting, err := bufio.NewReader(conn).ReadString('\n')
	if err != nil {
		return c.String(http.StatusInternalServerError, "Error reading from the datastore")
	}
	return c.String(http.StatusOK, fmt.Sprintf("Hi, %s!", strings.TrimSpace(greeting)))
}
```

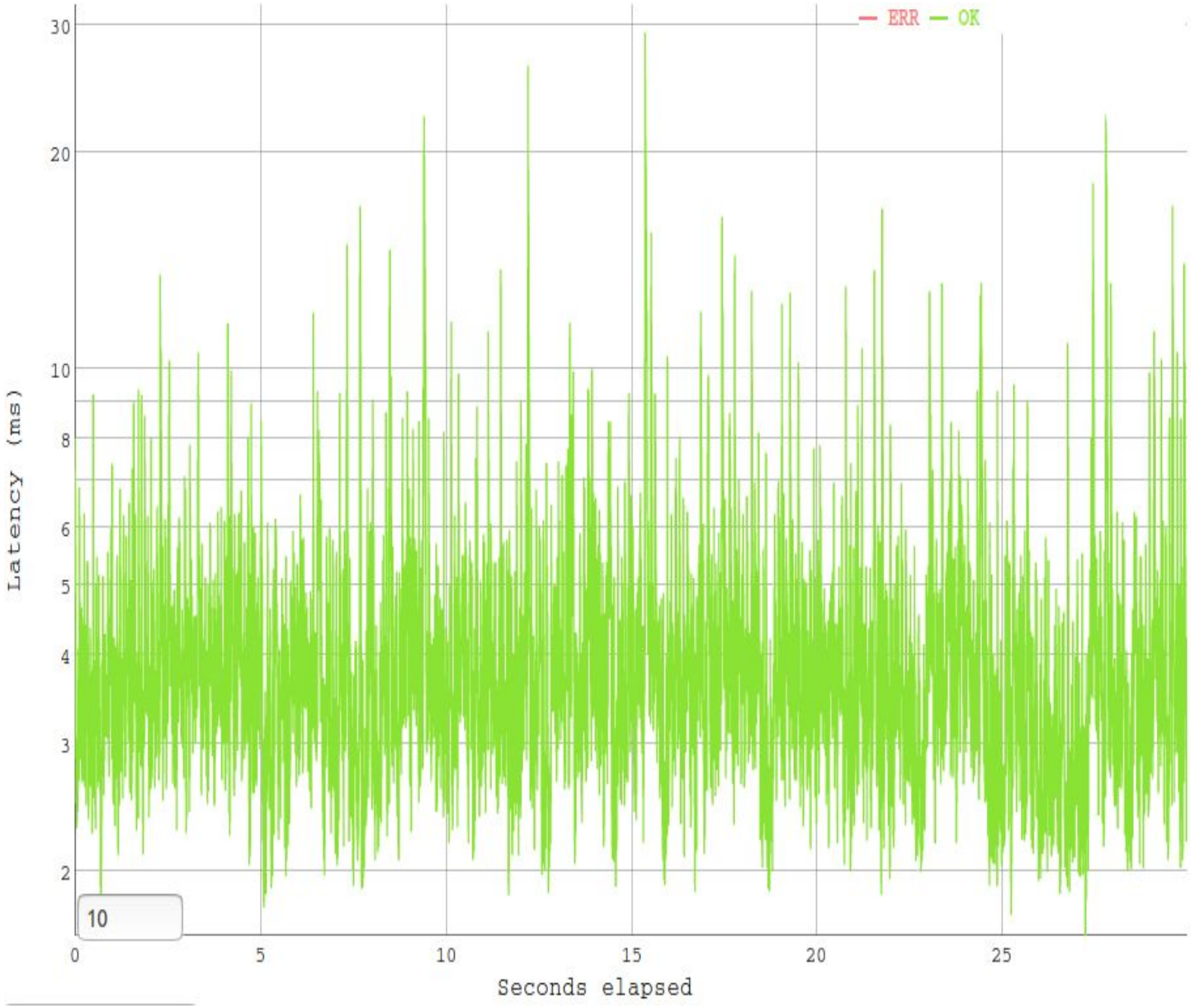
Test case 1

- 5000 zapytań/s
- 0ms opóźnienia

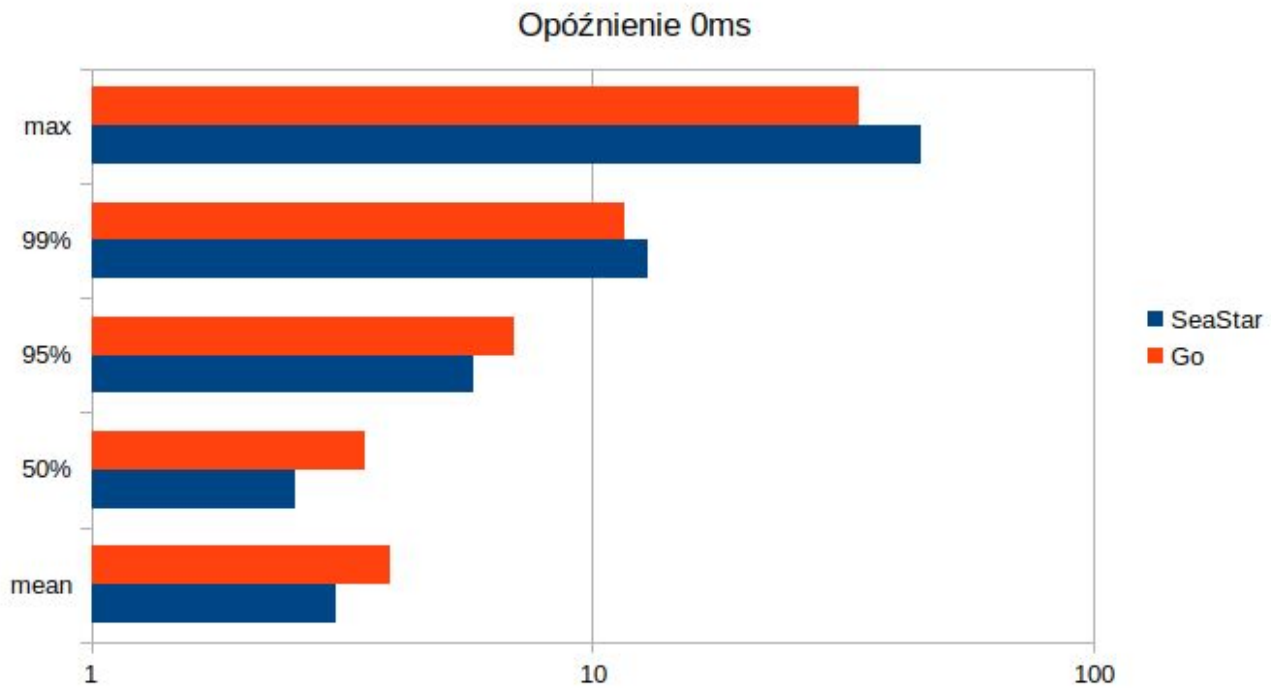
Wynik - SeaStar



Wynik - Go


ERR
OK
Latency (ms)
30
20
10
8
6
5
4
3
2
10
0
5
10
15
20
25
Seconds elapsed

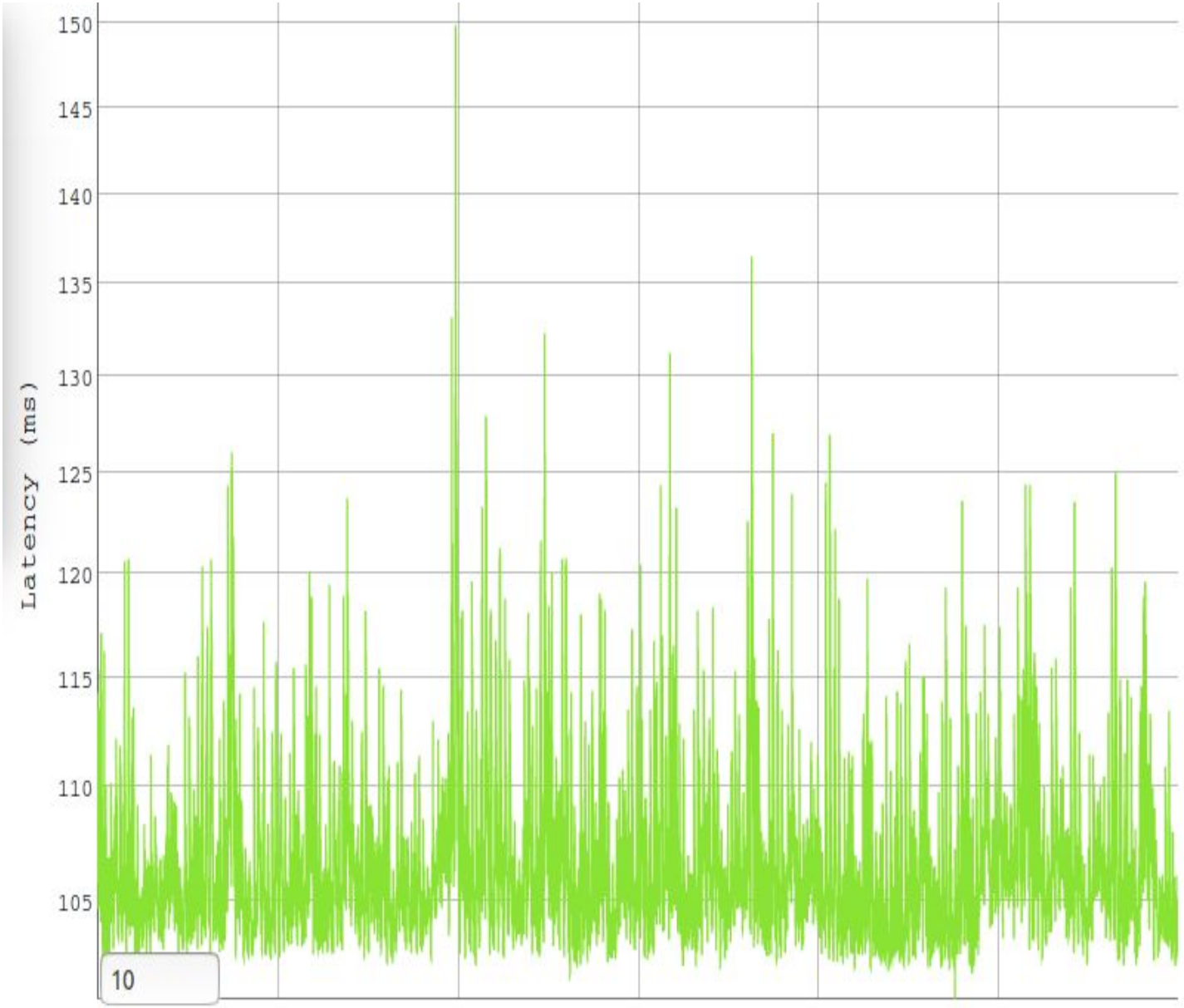
Wynik



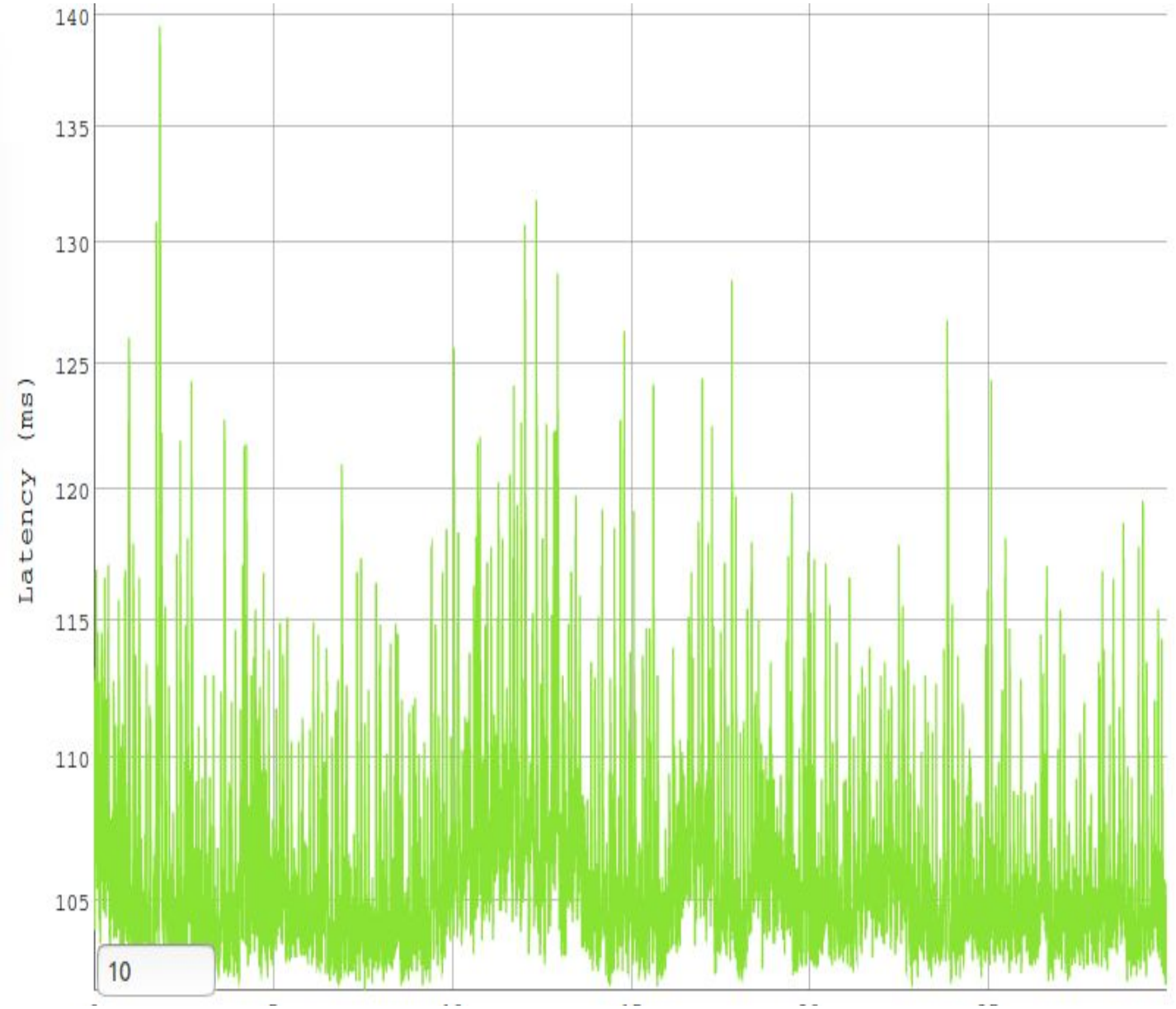
Test case 2

- 5000 zapytań/s
- 100ms opóźnienia
- ok. 500 połączeń równolegle

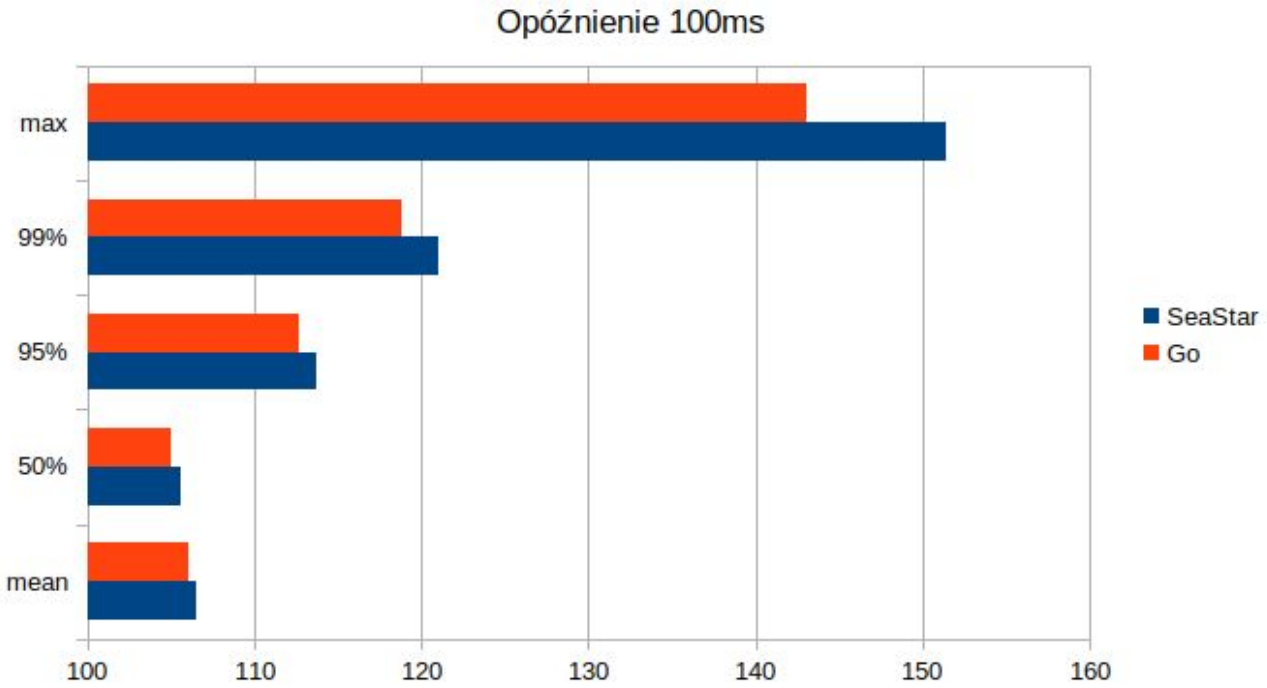
Wynik - SeaStar



Wynik - Go



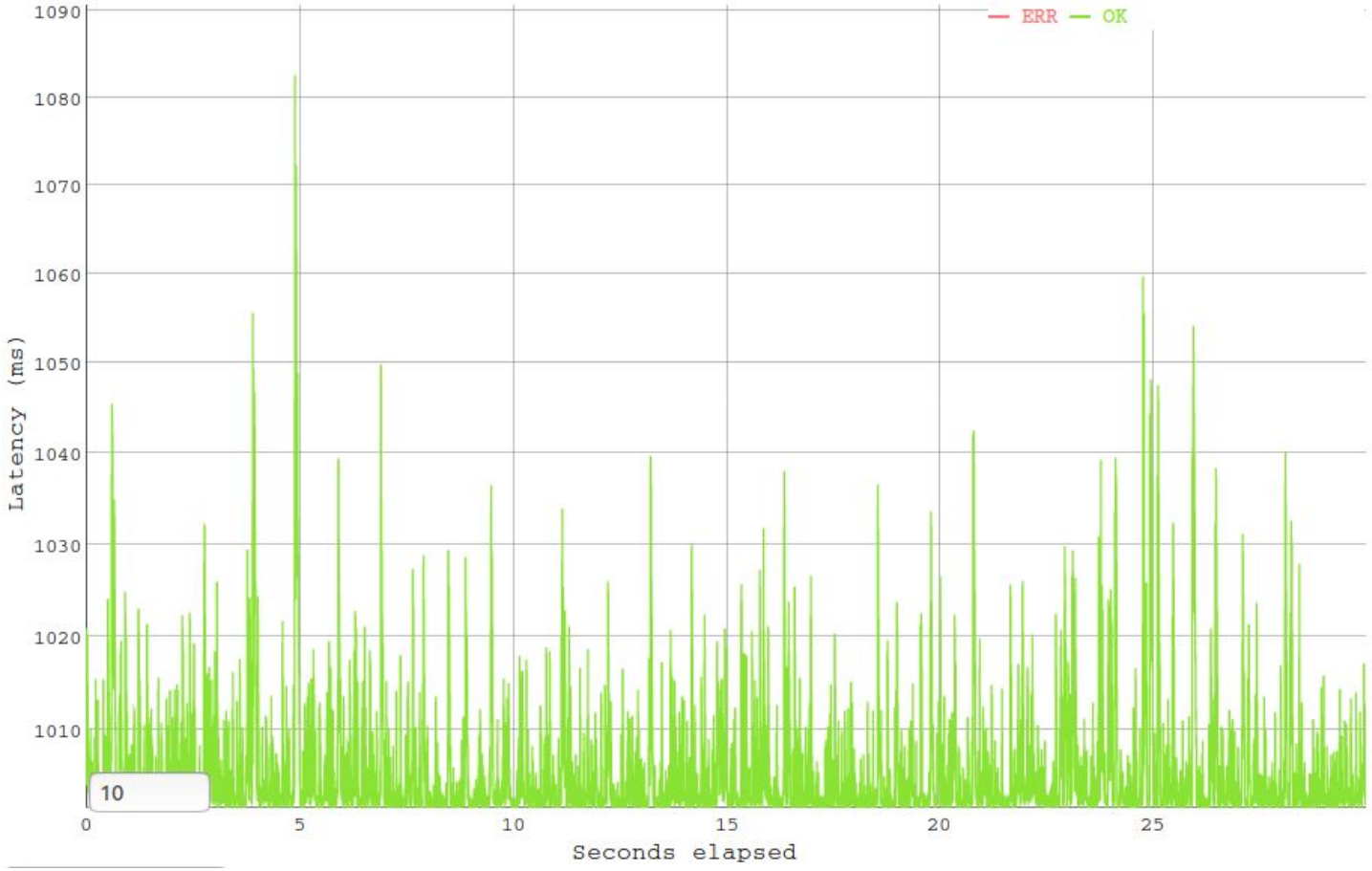
Wynik


Opóźnienie 100ms
max
99%
95%
50%
mean
100
110
120
130
140
150
160
SeaStar
Go

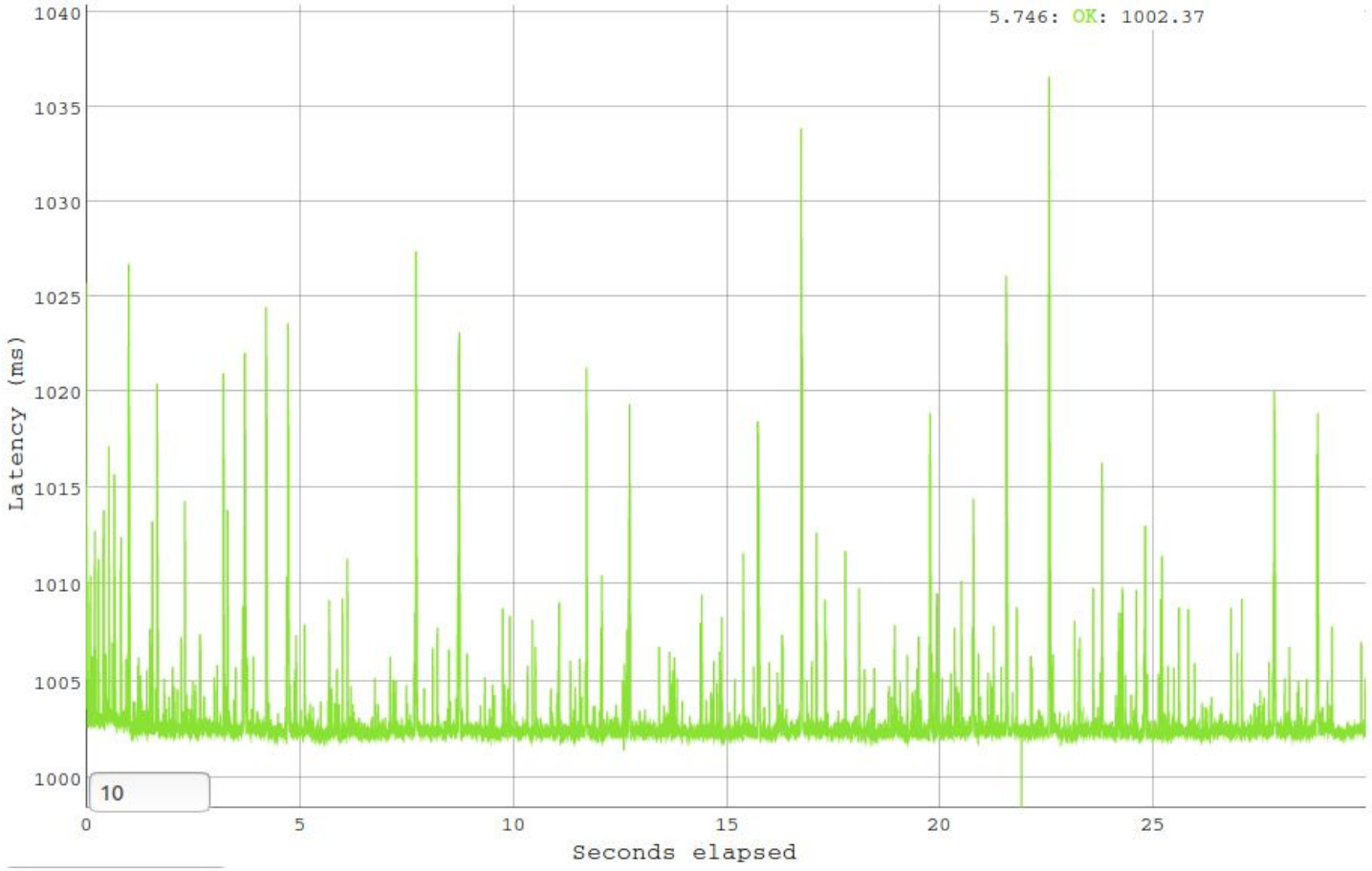
Test case 3

- 2500 zapytań/s
- 1000ms opóźnienia
- ok. 2500 połączeń równolegle

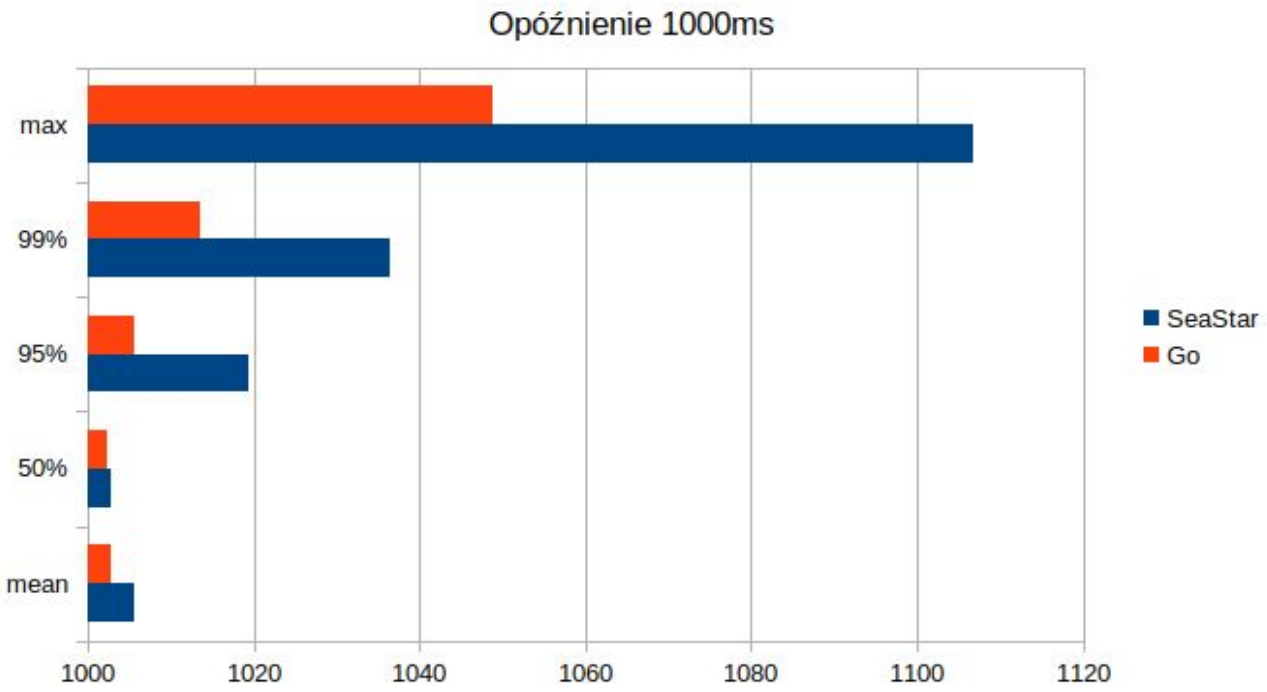
Wynik - SeaStar



Wynik - Go


5.746: OK: 1002.37
1040
1035
1030
1025
1020
1015
1010
1005
1000
Latency (ms)
10
0
5
10
15
20
25
Seconds elapsed

Wynik


Opóźnienie 1000ms
max
99%
95%
50%
mean
1000
1020
1040
1060
1080
1100
1120
SeaStar
Go

Porównanie

- Szybkość: Go
  - uwaga na low-latency
- Narzędzia:
  - Go: wbudowane
  - C++: dostępne
- Język:
  - Subiektywne
- Społeczność

Co dalej?

- NUMA
- Stos TCP/IP w warstwie użytkownika
  - Go: https://github.com/google/netstack
  - Seastar: native

Podsumowanie

- Nie ufaj intuicji, ufaj pomiarom

Pytania?

codilime®
CREATING VALUE

Dziękuję za uwagę