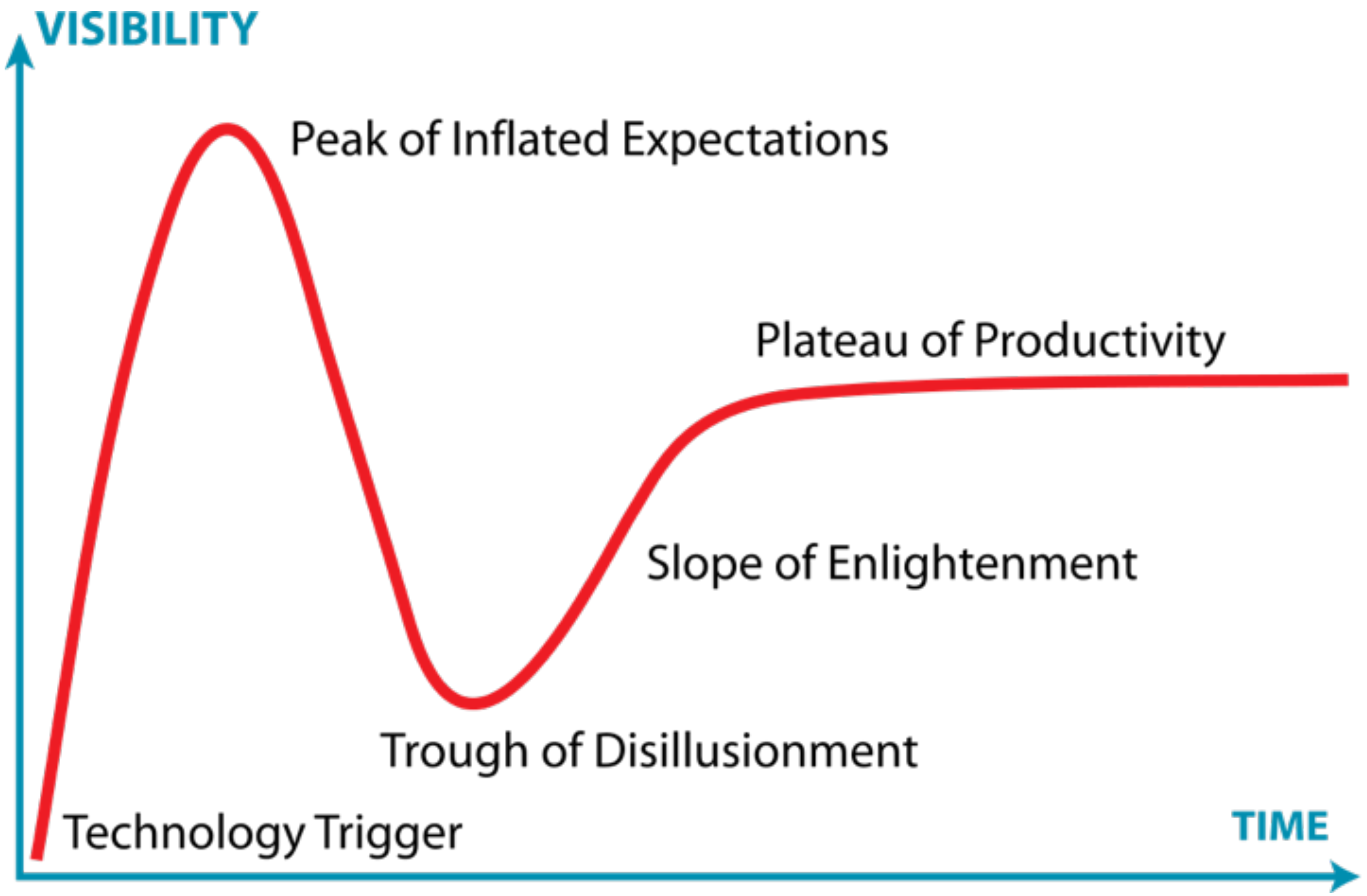
Performance of synchronous and asynchronous I/O in network applications

parallel 2015

Hubert Schmid



Hype Cycle

What performance can you expect from synchronous and asynchronous I/O in network applications?

Agenda

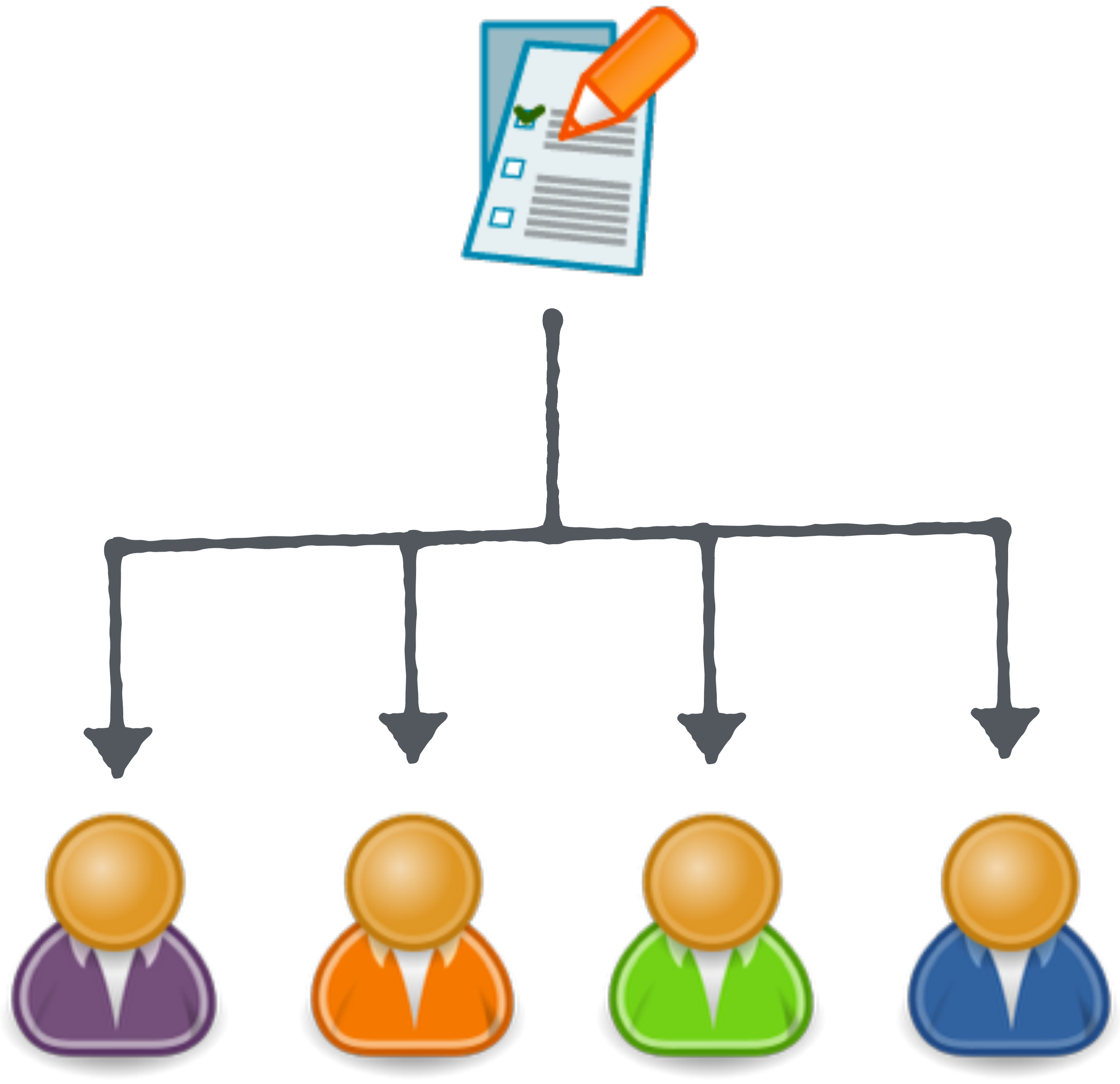
Introduction

Example, Benchmark, Results

Strengths, Weaknesses, Hybrids

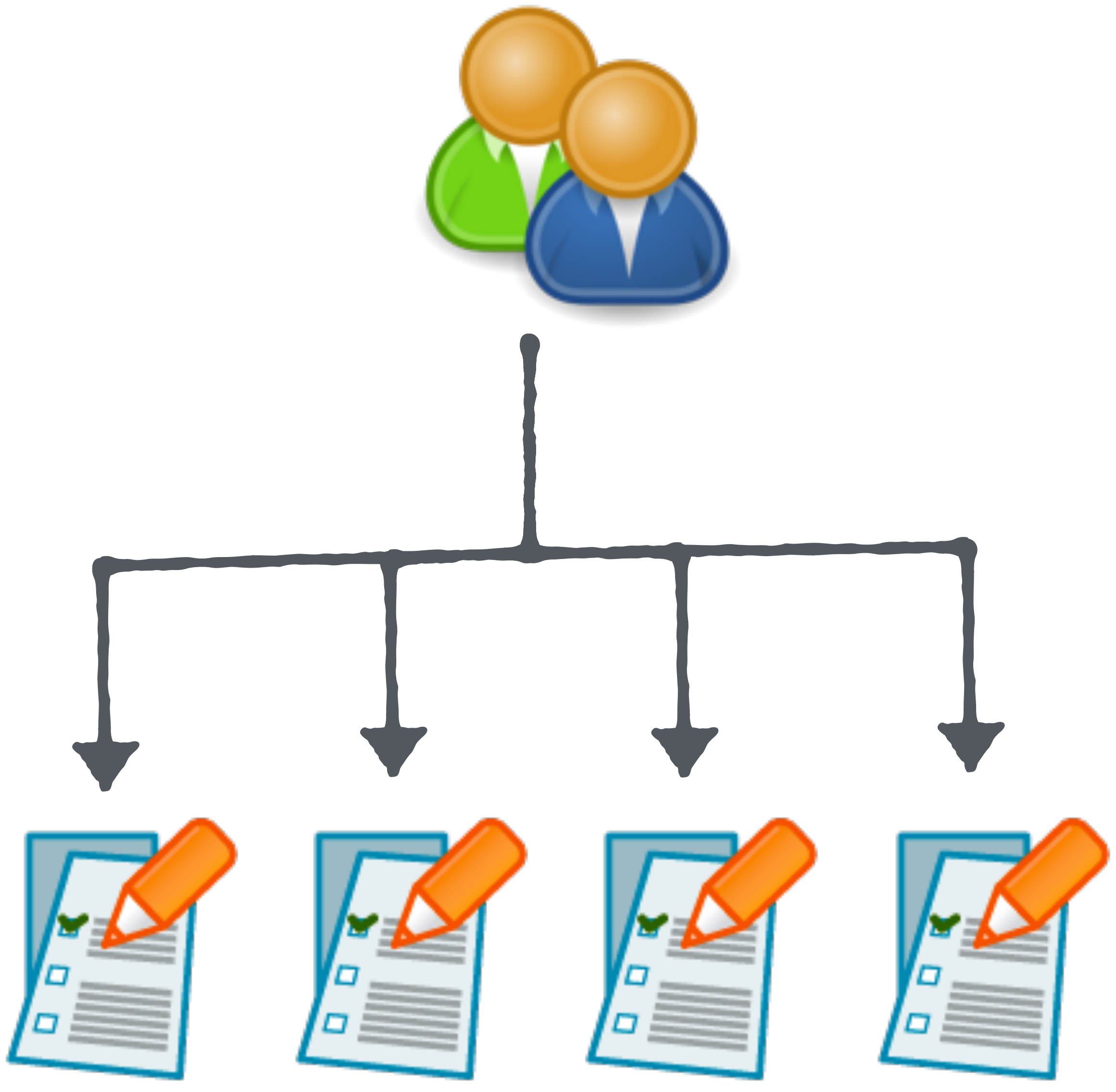
Summary

Parallelism



optimisation of throughput time

Concurrency



optimisation of efficiency

Terminology

Synchronous Model:

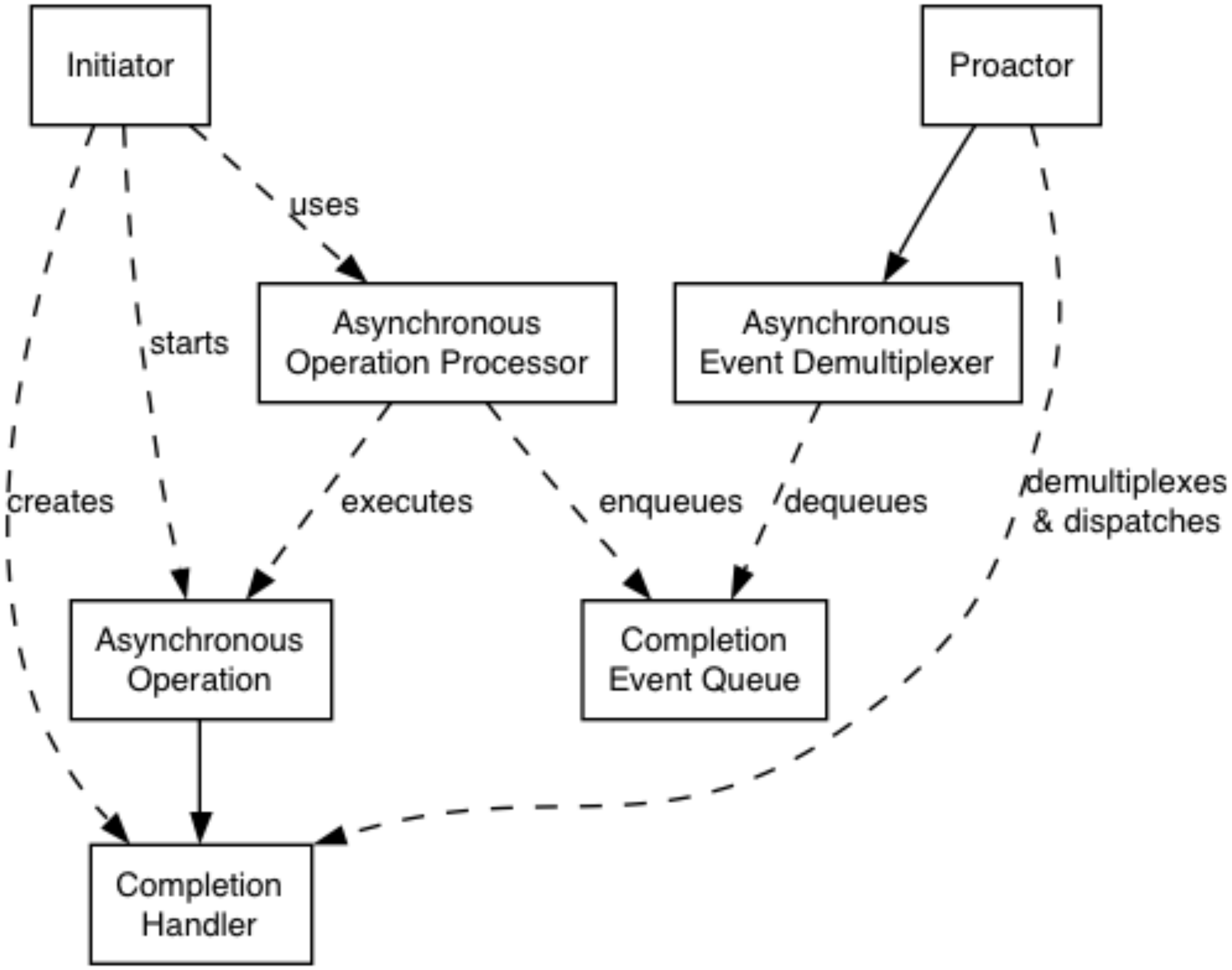
- blocking I/O
- multi-threaded

Asynchronous Model:

- non-blocking I/O
- proactor pattern
  - notify-on-completion
  - notify-on-readiness

Out of scope:

- asynchronism in kernel, hardware, network



Proactor Design Pattern
(Documentation of Boost.Asio 1.57)

Example, Benchmark, Results

Reverse-Echo-Server

- tangible
- simple
- effective
- low-level
- realistic

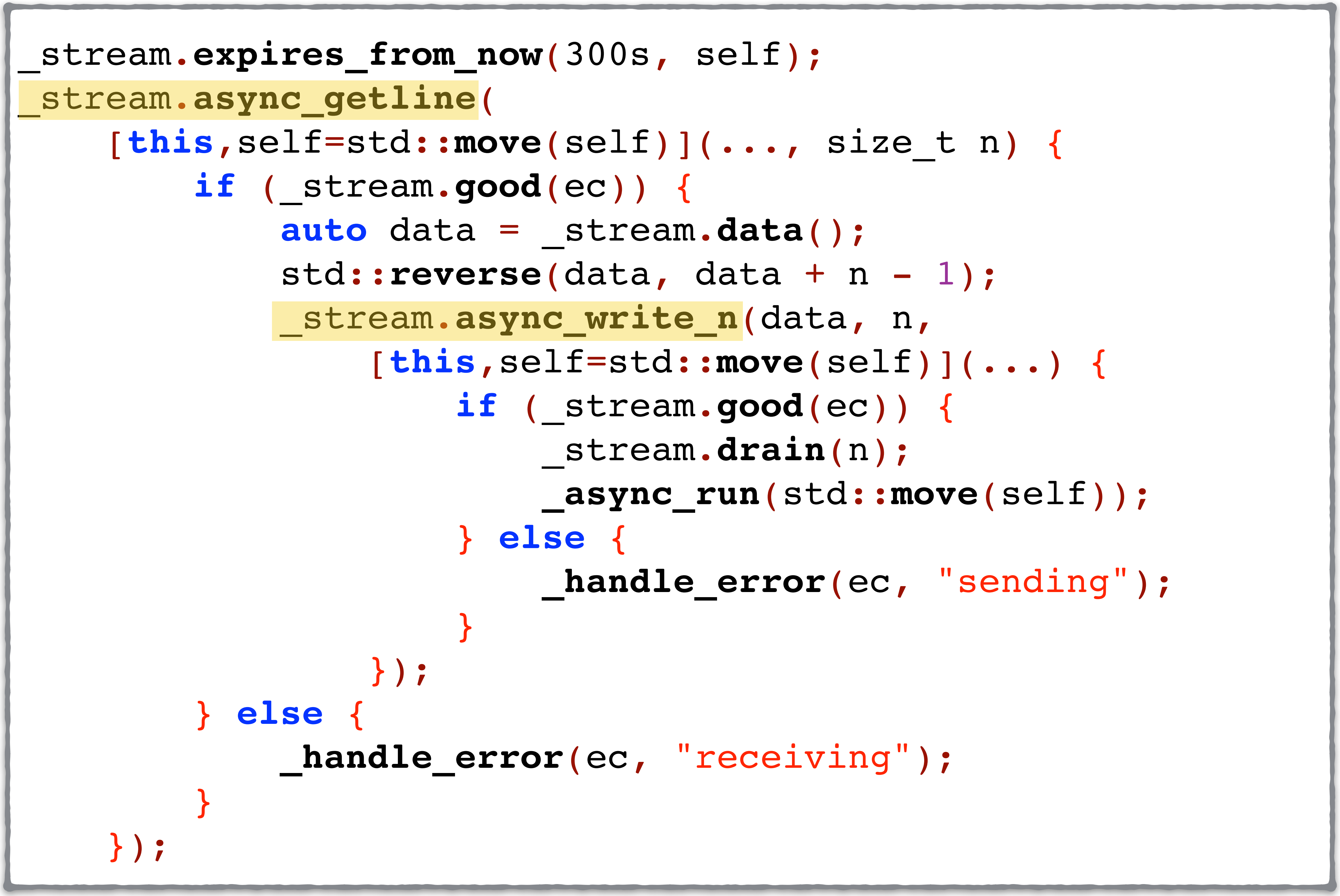
Synchronous Implementation

```
try {
    auto timeout = 300s;
    deadline deadline(timeout);
    while (auto n = _stream.getline(deadline)) {
        auto data = _stream.data();
        std::reverse(data, data + n - 1);
        _stream.write_n(data, n, deadline);
        _stream.drain(n);
        deadline.expires_from_now(timeout);
    }
    if (_stream.available())
        throw std::runtime_err
    }
} catch (...) {
    _handle_error();
}
```

```
::recv(_sock, ...);
if (errno == EAGAIN) {
    ::poll(_sock, _timer);
    ::recv(_sock, ...);
}
```

```
::send(_sock, ...);
if (errno == EAGAIN) {
    ::poll(_sock, _timer);
    ::send(_sock, ...);
}
```

Asynchronous Implementation

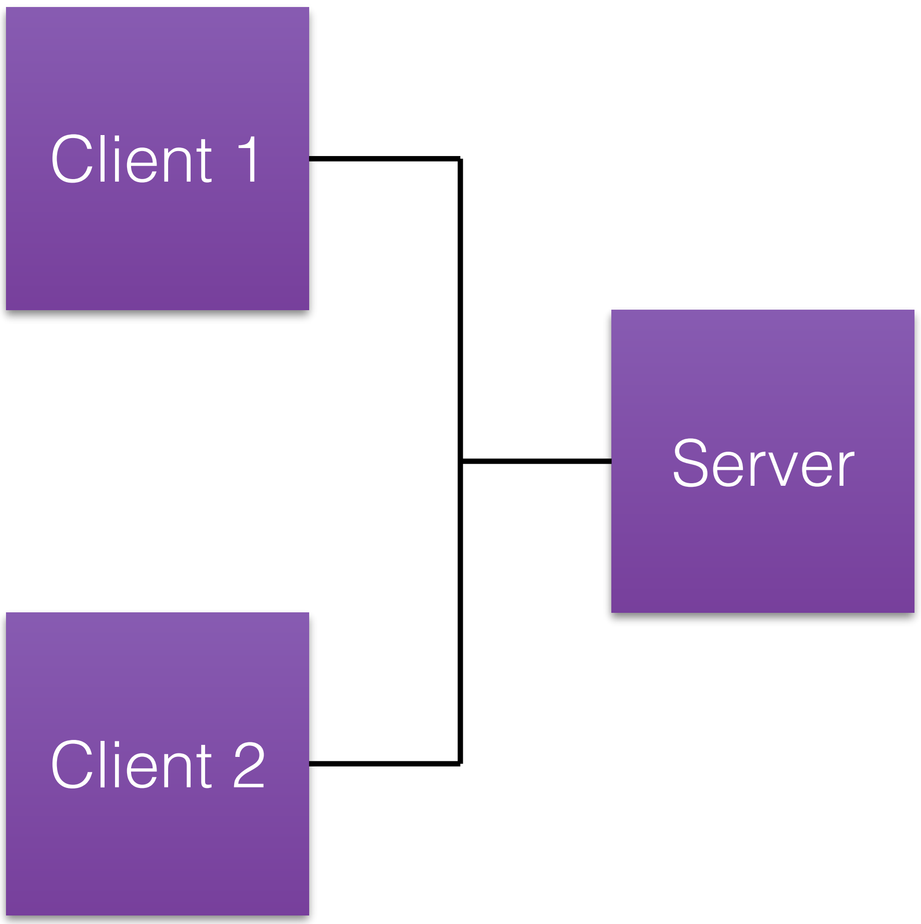


```
_stream.expires_from_now(300s, self);
_stream.async_getline(
    [this,self=std::move(self)](..., size_t n) {
        if (_stream.good(ec)) {
            auto data = _stream.data();
            std::reverse(data, data + n - 1);
            _stream.async_write_n(data, n,
                [this,self=std::move(self)](...) {
                    if (_stream.good(ec)) {
                        _stream.drain(n);
                        _async_run(std::move(self));
                    } else {
                        _handle_error(ec, "sending");
                    }
                });
        } else {
            _handle_error(ec, "receiving");
        }
    });
```

Test candidates and setup

- Async Single-Core (single-threaded)
- Async Multi-Core (thread per core)
- Sync Multi-Core (thread per connection)

Measurement from 2 systems with each 500,000 simultaneous TCP connections



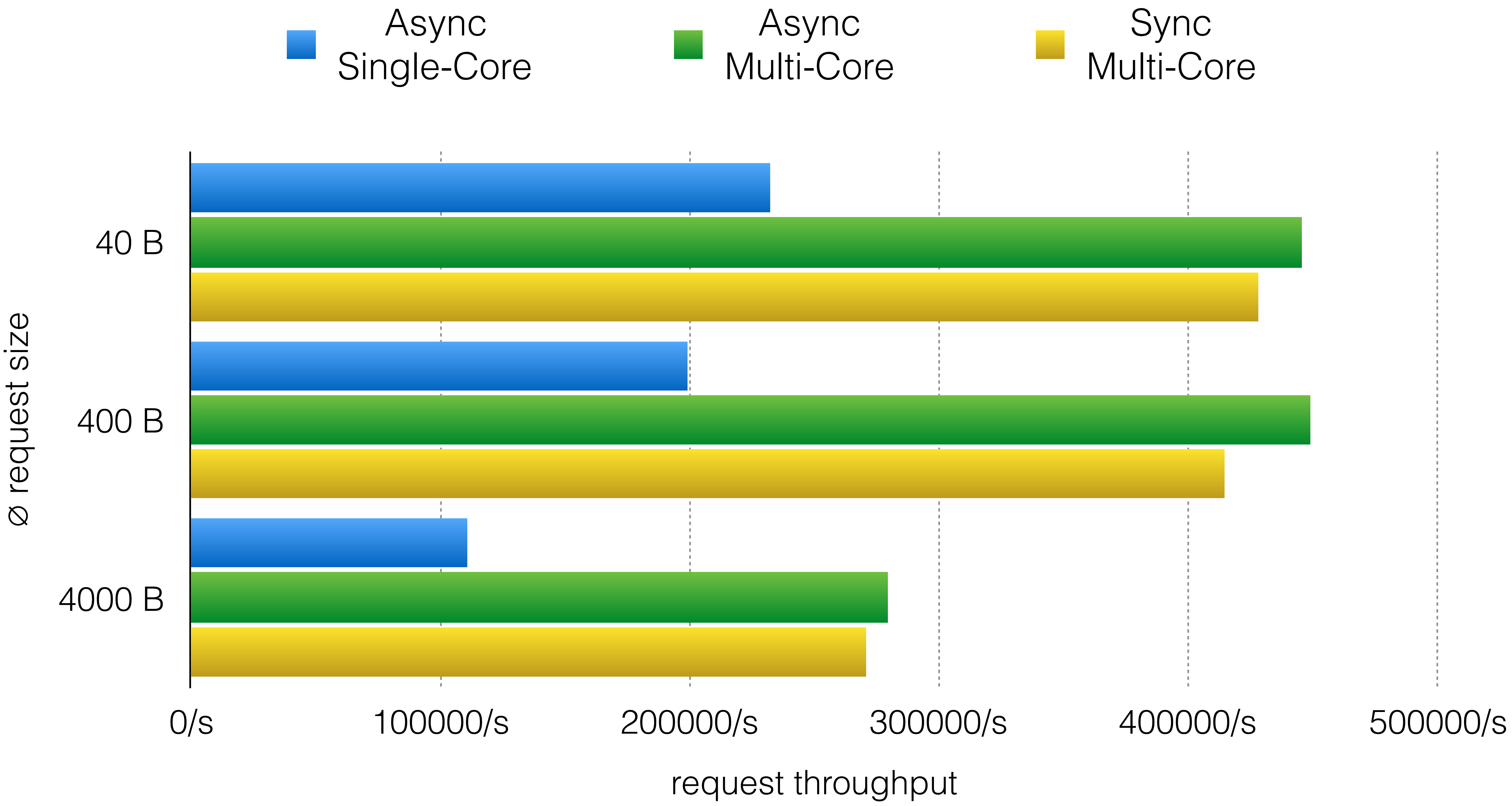
Systems

- 2x Intel Xeon E5-2666 10C
- Intel NIC 10 GbE (2x Multi-Queue)
- Linux 3.16 (Debian jessie/testing)

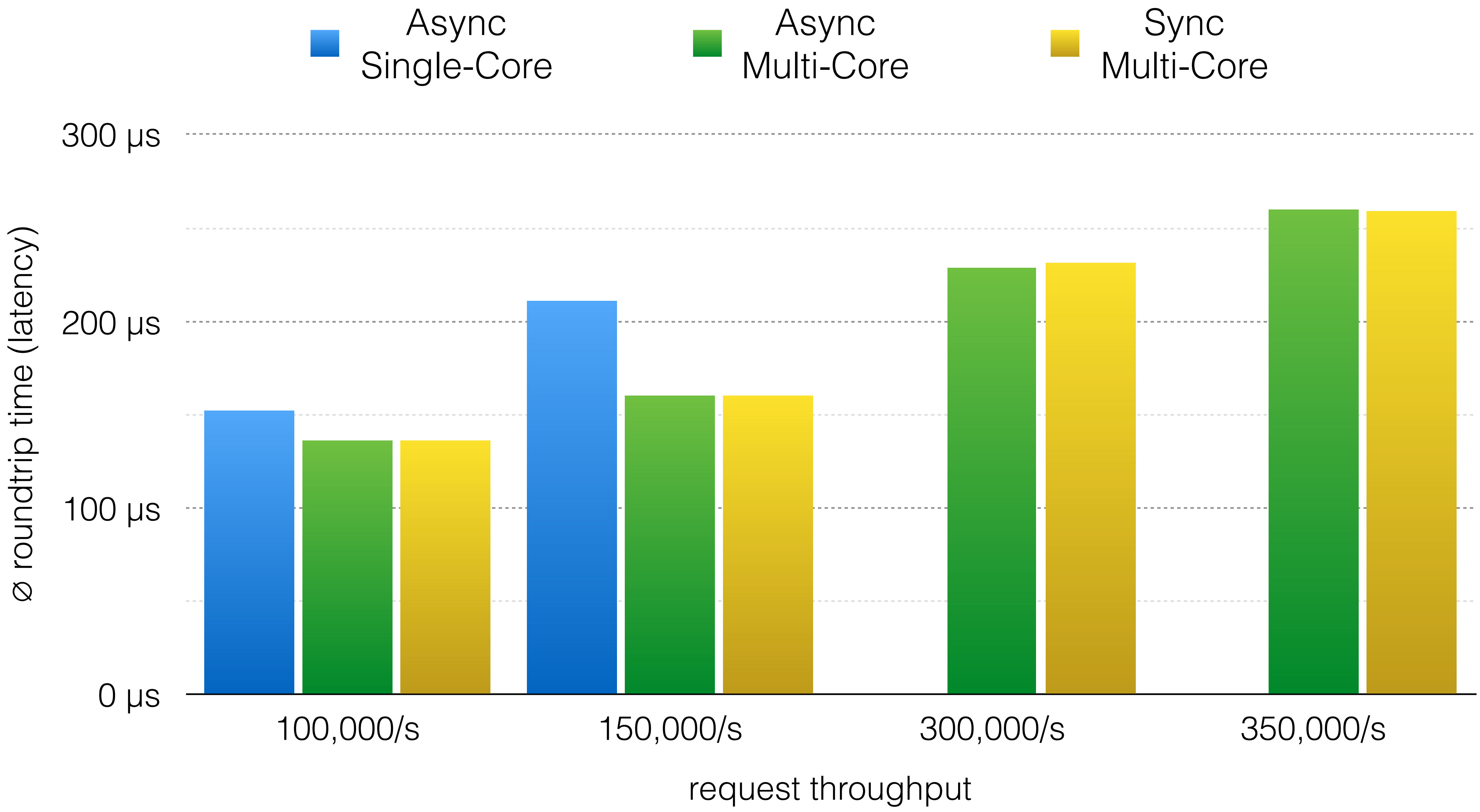
Network

- 10 Gb Ethernet (shared)

Throughput

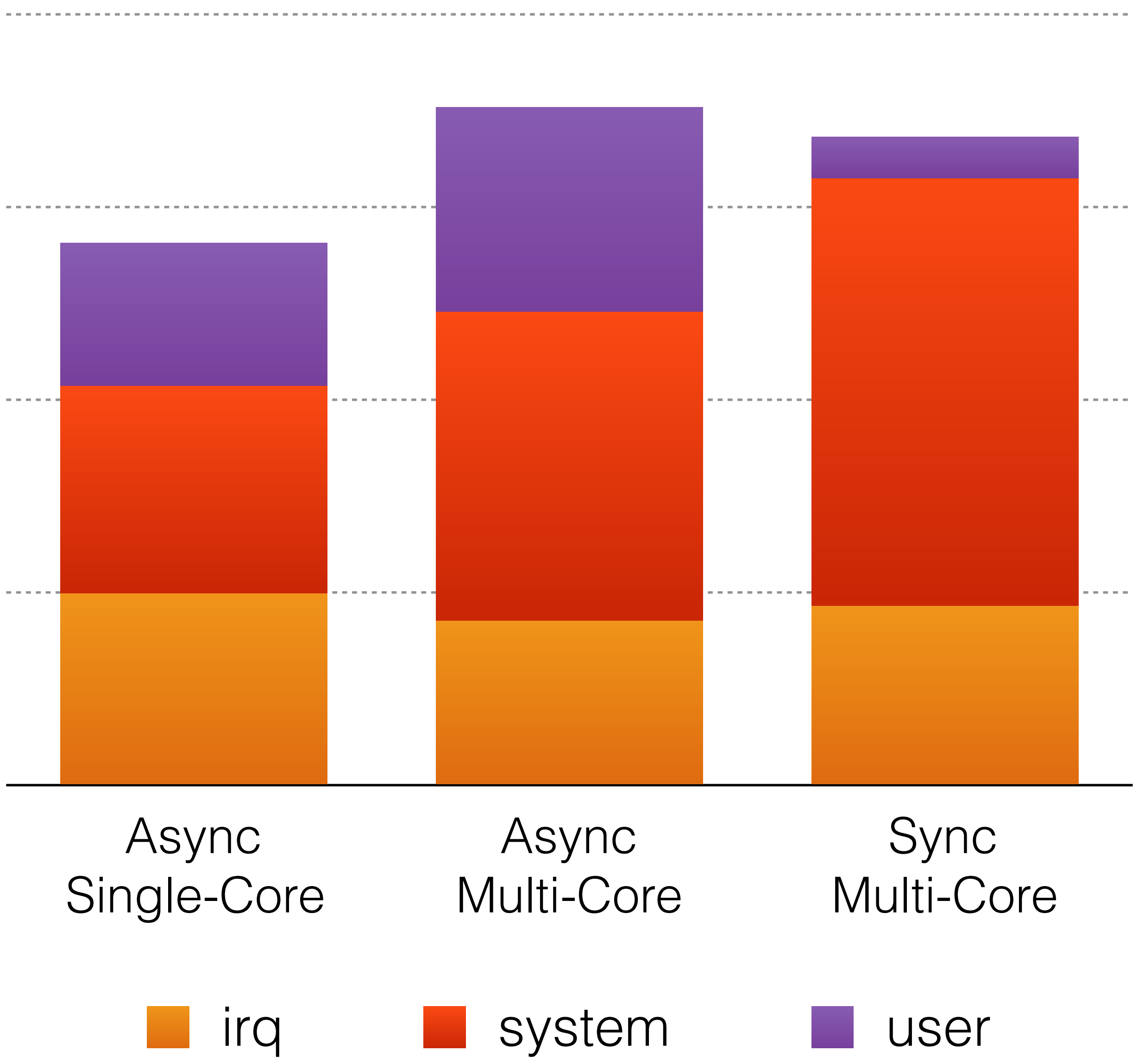

Async Single-Core
Async Multi-Core
Sync Multi-Core
⌀ request size
40 B
400 B
4000 B
0/s
100000/s
200000/s
300000/s
400000/s
500000/s
request throughput

Latency

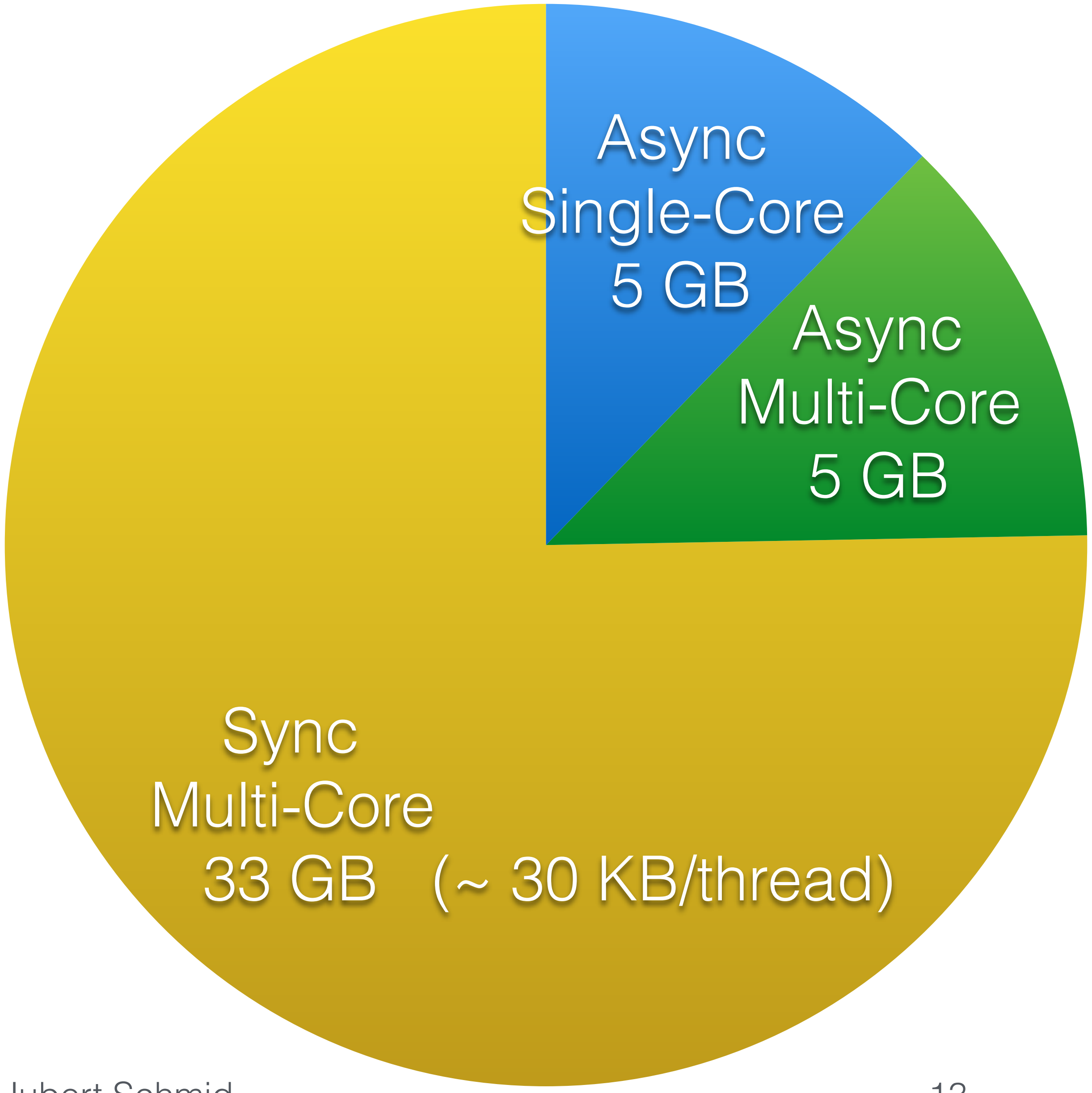

Async Single-Core
Async Multi-Core
Sync Multi-Core
300 µs
200 µs
100 µs
0 µs
⌀ roundtrip time (latency)
100,000/s
150,000/s
300,000/s
350,000/s
request throughput

Resource Usage

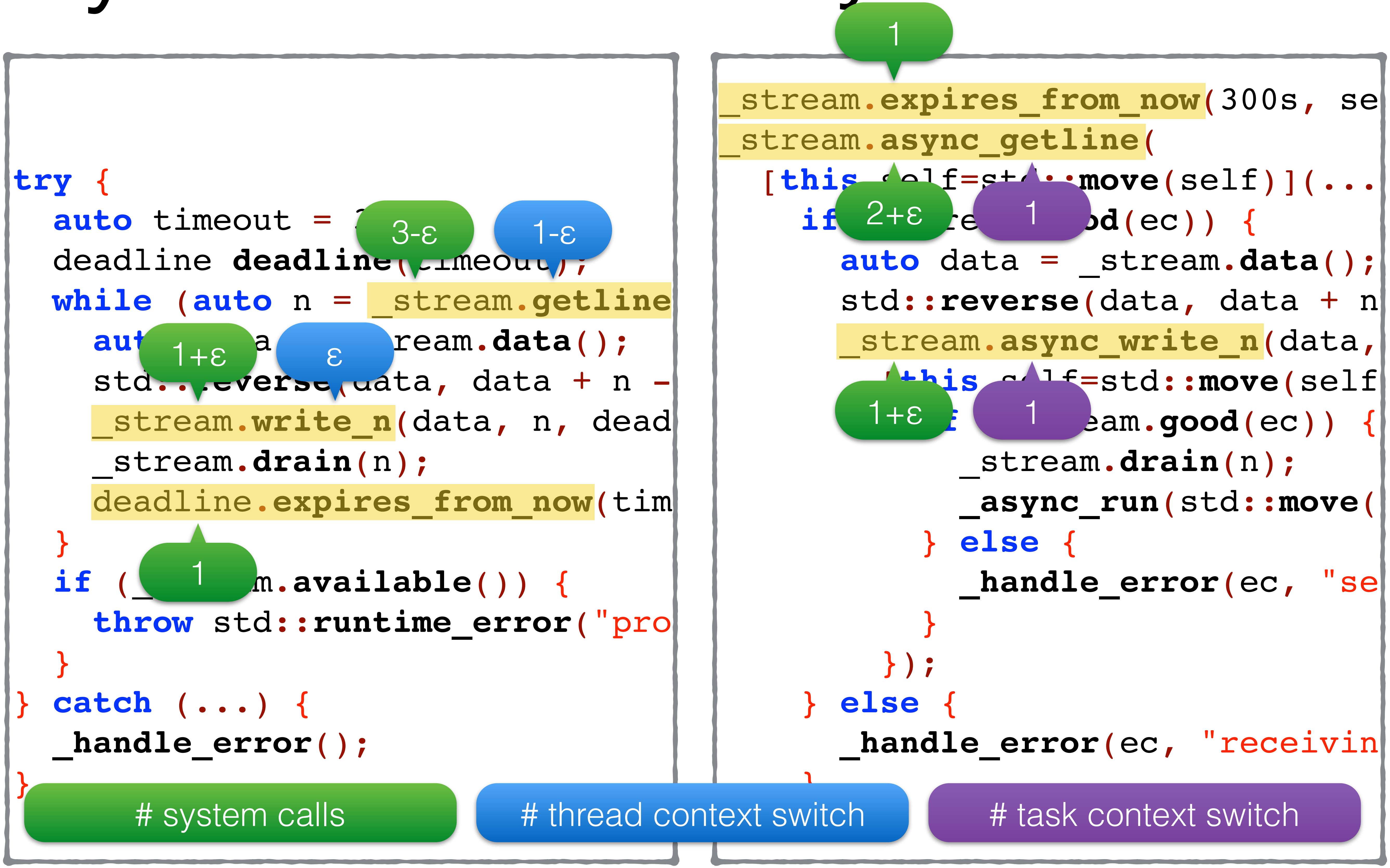
CPU



RAM



synchronous asynchronous



```
try {
  auto timeout = 3
  deadline deadline(timeout);
  while (auto n = _stream.getline
    auto data = _stream.data();
    std::reverse(data, data + n -
    _stream.write_n(data, n, dead
    _stream.drain(n);
    deadline.expires_from_now(tim
  }
  if (_stream.available()) {
    throw std::runtime_error("pro
  }
} catch (...) {
  _handle_error();
}
```

```
_stream.expires_from_now(300s, se
_stream.async_getline(
  [this, self=std::move(self)](...
    if (_stream.good(ec)) {
      auto data = _stream.data();
      std::reverse(data, data + n
      _stream.async_write_n(data,
        [this, self=std::move(self
          if (_stream.good(ec)) {
            _stream.drain(n);
            _async_run(std::move(
          } else {
            _handle_error(ec, "se
          }
        });
    } else {
      _handle_error(ec, "receivin
    }
```

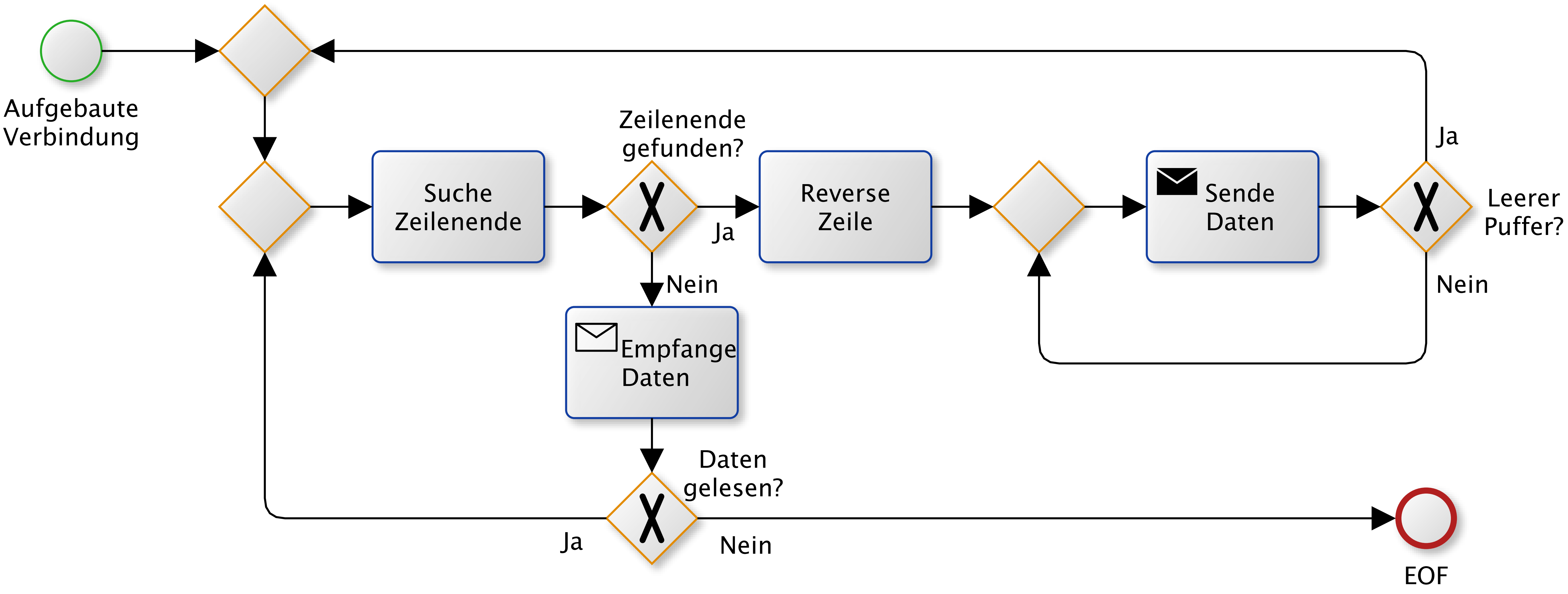
Results

- network saturation with multi-core
- similar usage of network stack
- different RAM usage
- actual optimisations are unrelated to programming model (e.g. receive-side-scaling with NIC support)

Strengths, Weaknesses, Hybrids

Asynchronous Strengths

- platform support
- kernel bypass
- customisation
- non-sequential flows

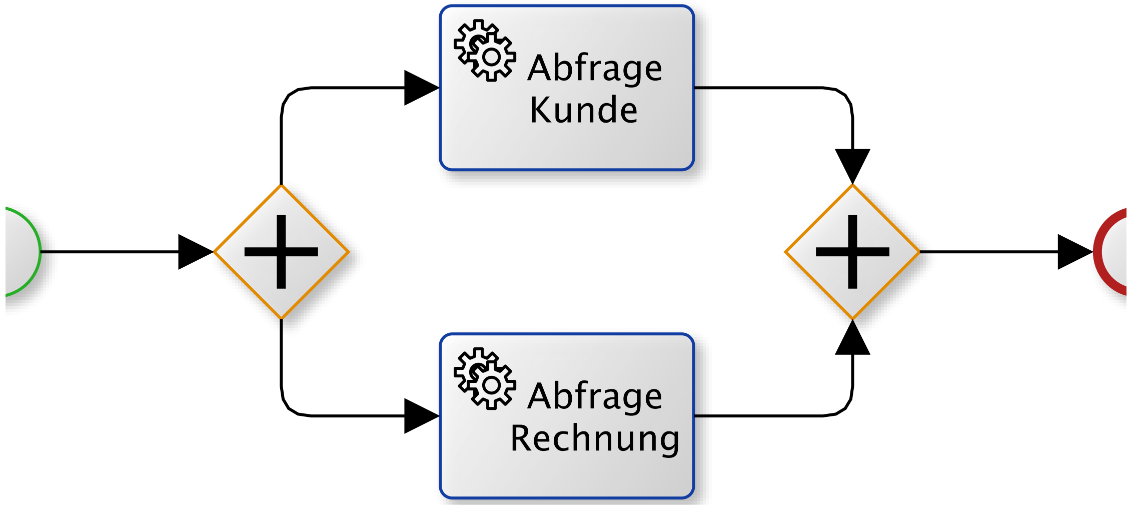


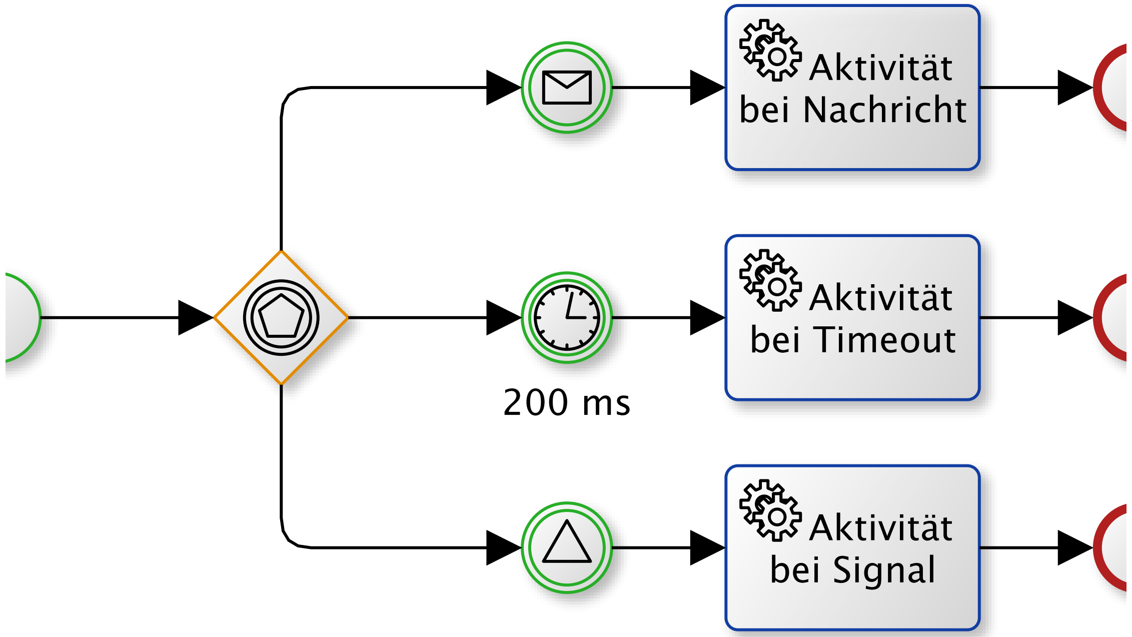
sequence flow diagram

Reverse-Echo-Server is fully sequential

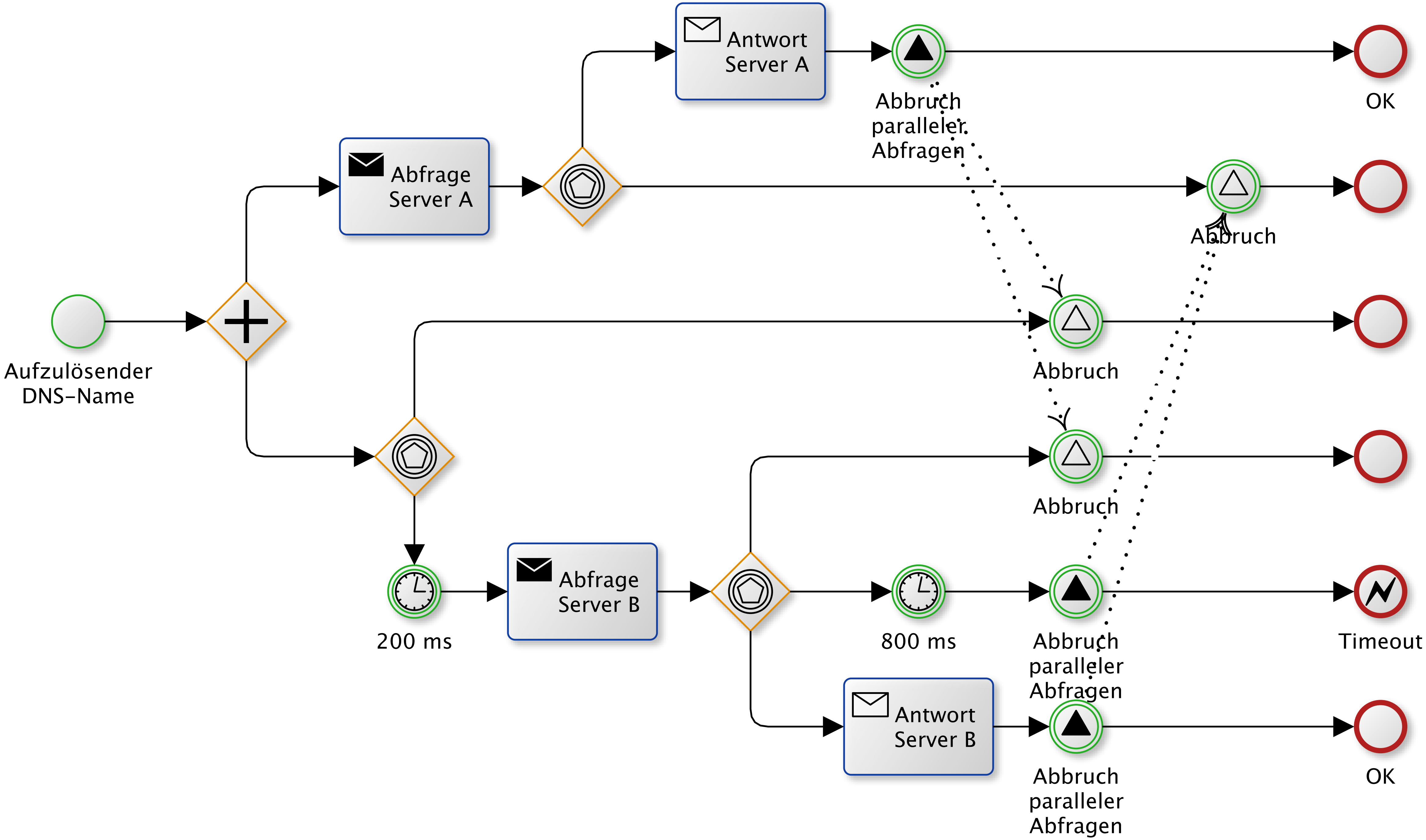
Non-sequential flows

- parallel forks (overlapping I/O)
- event-based decision gateways
- control of parallel execution paths





Example DNS Query



Asynchronous Weaknesses

operating systems

- incomplete interface (kernel, core libraries)
- virtual memory

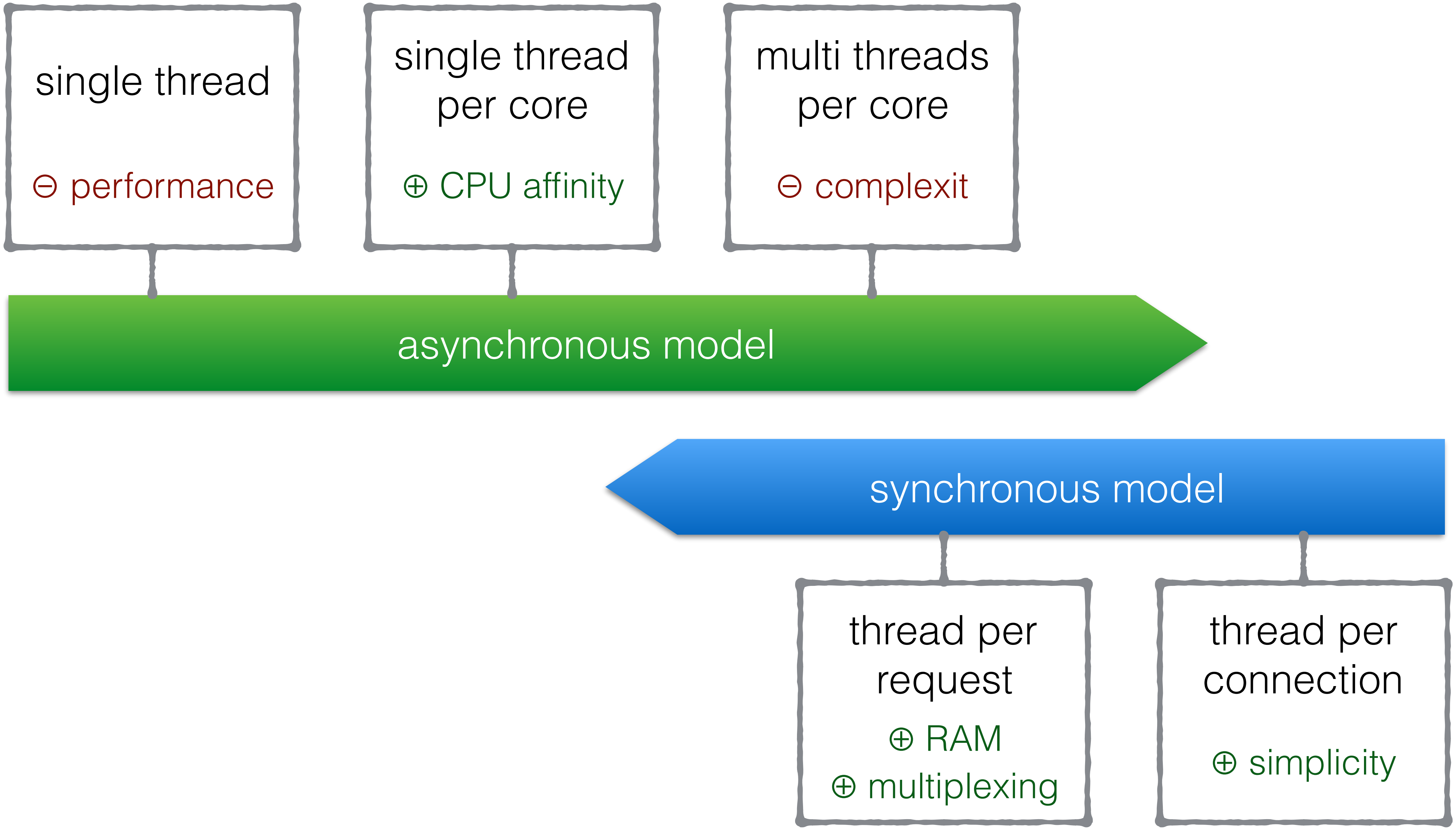
CPU-bound flows

- no preemption (causing latency)

third party libraries

- incomplete, incompatible

Mitigation



single thread

⊖ performance

single thread per core

⊕ CPU affinity

multi threads per core

⊖ complexit

asynchronous model

synchronous model

thread per request

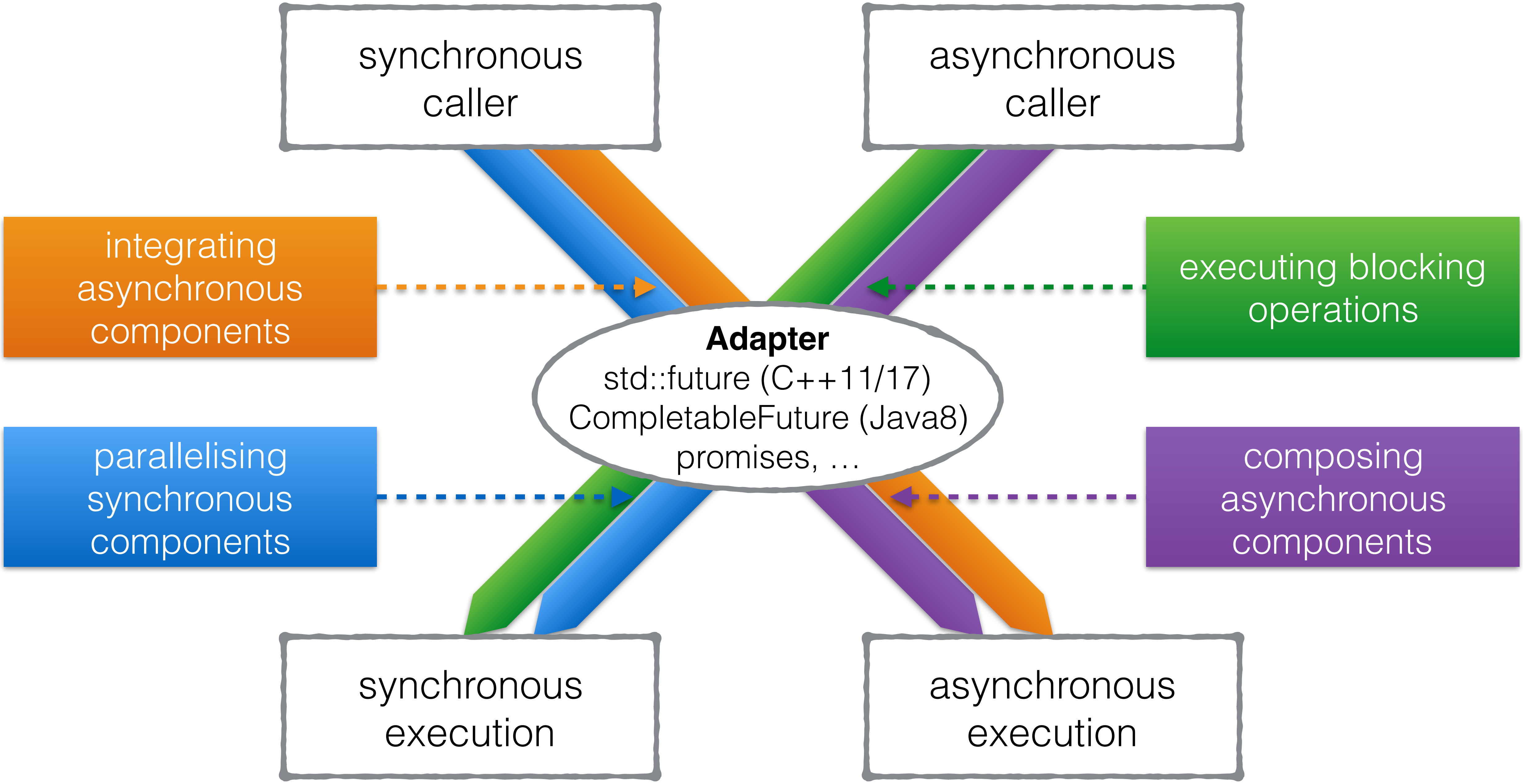
⊕ RAM

⊕ multiplexing

thread per connection

⊕ simplicity

Interplay



synchronous caller

asynchronous caller

integrating asynchronous components

executing blocking operations

Adapter
std::future (C++11/17)
CompletableFuture (Java8)
promises, …

parallelising synchronous components

composing asynchronous components

synchronous execution

asynchronous execution

Summary

- Multi-Core
- RAM and platform peculiarities
- non-sequential flows
- blocking operations
- hybrids

Sources

- "The C10K problem" by Dan Kegel
- "Scaling in the Linux Networking Stack" by Tom Herbert and Willem de Bruijn
- "Comparing the performance of web server architectures." by Pariag, David, et el.
- "Hype-Zyklus nach Gartner Inc." by "Idotter" is licensed under CC BY-SA 3.0
- "Arcadia pocket watch 1859" by "Suebun" is licensed under CC BY-SA 3.0
- "An icon in the Tango! theme style: Blue person. Green person. Orange person. Purple person." by "Inductiveload" is released into public domain.
- "Emblem persons blue green" by "Con-struct" is licensed under CC BY-SA 3.0
- "View refresh" by "The people from the Tango! project" is released into public domain.
- "Korganizer Icon" by "Umut Pulat" is licensed under GNU Lesser General Public License
- "Proactor design pattern" by "Christopher M. Kohlhoff" is licensed under Boost Software License 1.0
- "AE.svg" by "C.Thure" is released into public domain
- "Question mark made of question marks" by "Khaydock" is licensed under CC BY-SA 3.0