Asynchronous C++

History of Time

→ *github.com/stevejims/acpp*

Steve Simpson
steve@stackhpc.com
www.stackhpc.com

Stack
HPC

Overview

1) Background
2) I/O (Is Hard)
3) Blocking
4) Theading
5) Select
6) Epoll
7) Callbacks
8) Futures
9) Coroutines
10) Summary

Background

Background

Systems Software Engineer

C, C++, Python

Background

Bristol, UK

Thriving technology industry

Background

Bristol, UK

The best place to live in the UK!

Background

BBC | Sign in | News | Sport | Weather | iPlayer | TV | Radio

NEWS

Home | UK | World | Business | Politics | Tech | Science | Health | Education | Entertainmen

England | Regions

Bristol named best place to live in Britain in 2017

19 March 2017 | England — Share

The "cool, classy and supremely creative" city beat off stiff competition to top the list

The internet says!

Background


BBC | Sign in | News | Sport | Weather | iPlayer | TV | Radio

NEWS

Home | UK | World | Business | Politics | Tech | Science | Health | Education | Entertainme

England | Regions

Bristol named best place to live in Britain in 2017

19 March 2017 | England

Share

GETTY IMAGES

The "cool, classy and supremely creative" city beat off stiff competition to top the list

The internet says!

It must be true…

Background

The internet says!

It must be true…

Background

The “cool, classy and supremely creative” city

Background

“

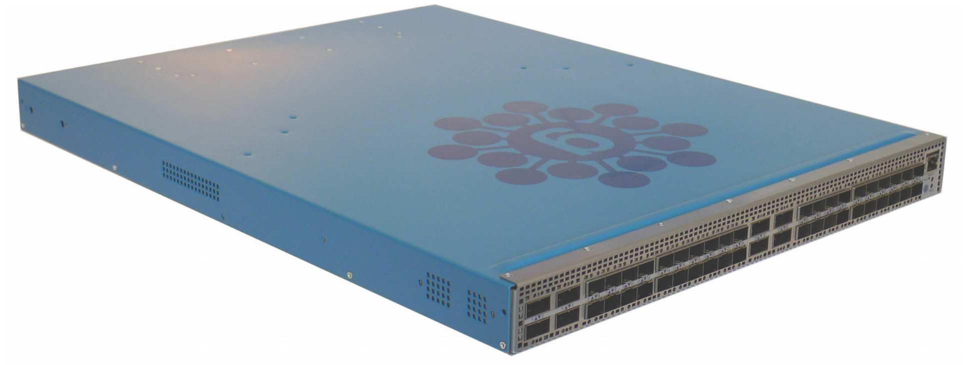
The “cool, classy and supremely creative” city

beat off stiff competition to top the list

”

Background

- **Gnodal**
  - 10GbE Ethernet
  - ASIC Verification
  - Embedded Firmware
- **JustOne Database**
  - Agile “Big Data” RMDBS
  - Based on PostgreSQL
  - Storage Team Lead



JustOne™

Background

Stack
HPC

Consultancy for HPC on OpenStack

Multi-tenant massively parallel workloads

Monitoring complex infrastructure

Background

UNIVERSITY OF CAMBRIDGE

Working with University of Cambridge

2016: Deployed HPC OpenStack cluster

Medical research; brain image processing

Background

Cloud orchestration platform

IaaS through API and dashboard

Multi-tenancy throughout

Network, Compute, Storage

Background

Operational visibility is critical

OpenStack is a complex, distributed application

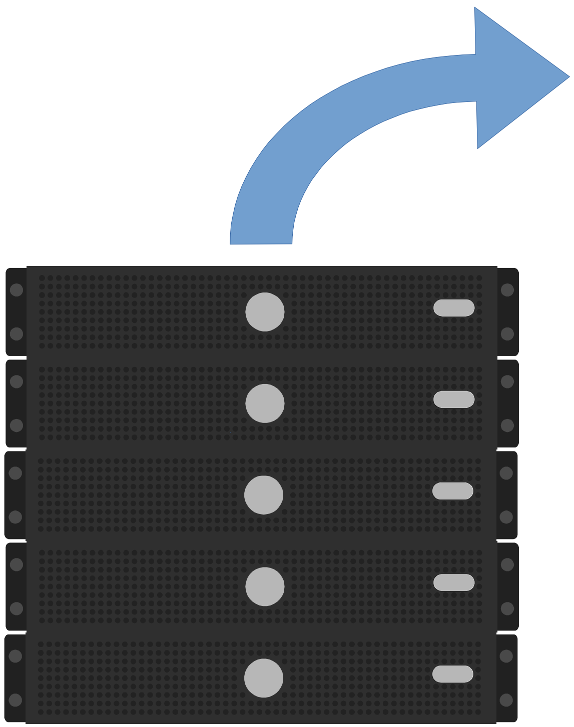
Background

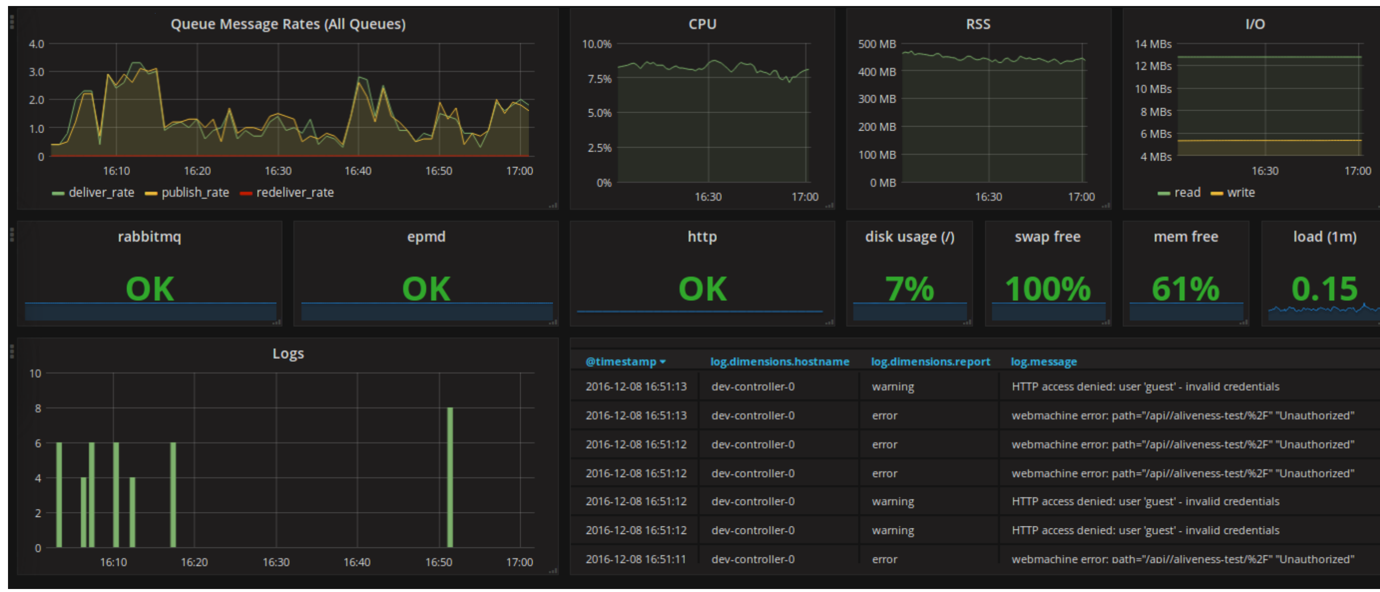
Operational visibility is critical

OpenStack is a complex, distributed application

...to run your complex, distributed applications

Background - Monitoring



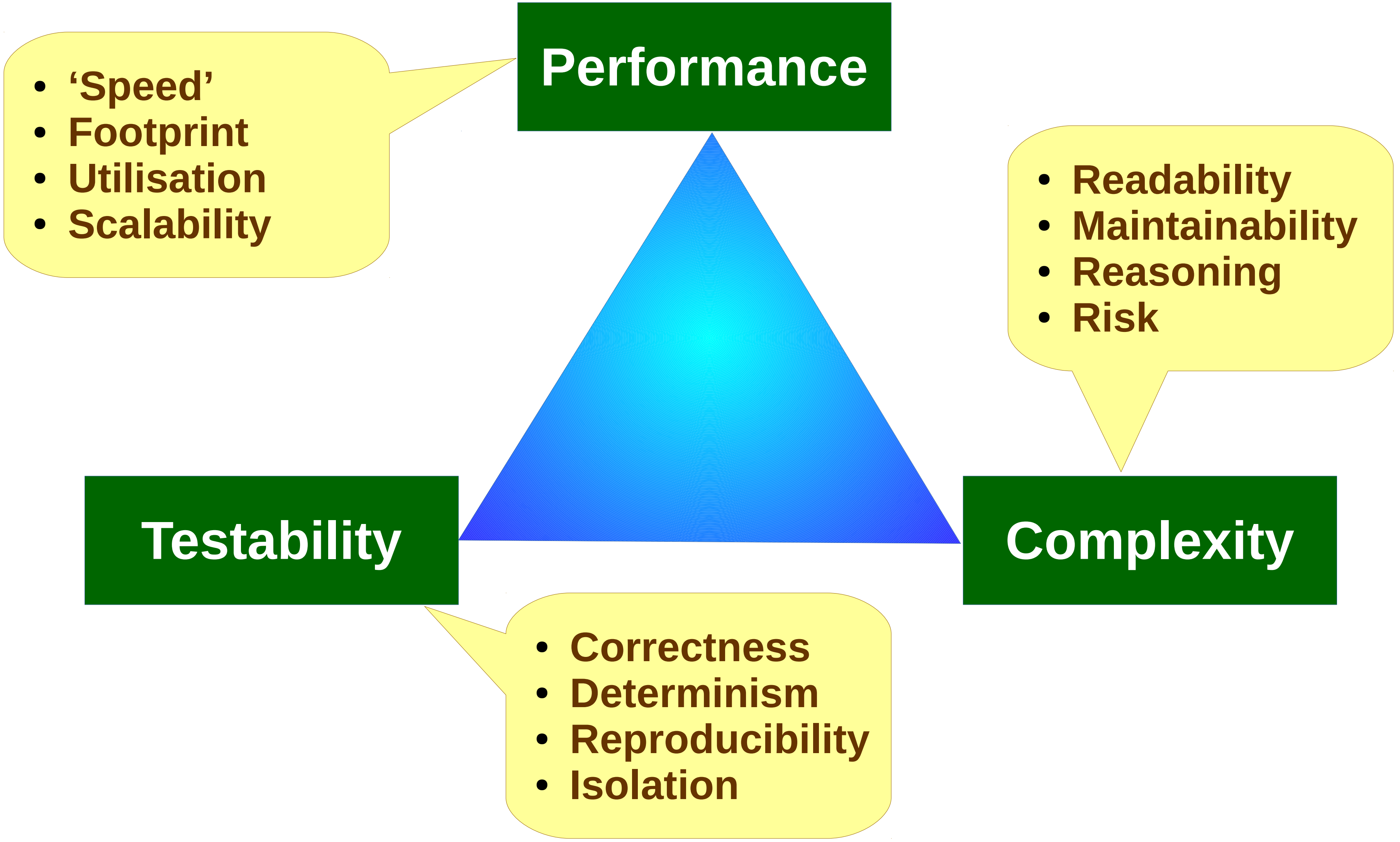


Gain visibility into the operation of the hardware and software

e.g. web site, database, cluster, disk drive

I/O
(Is Hard)

I/O – Trade Offs

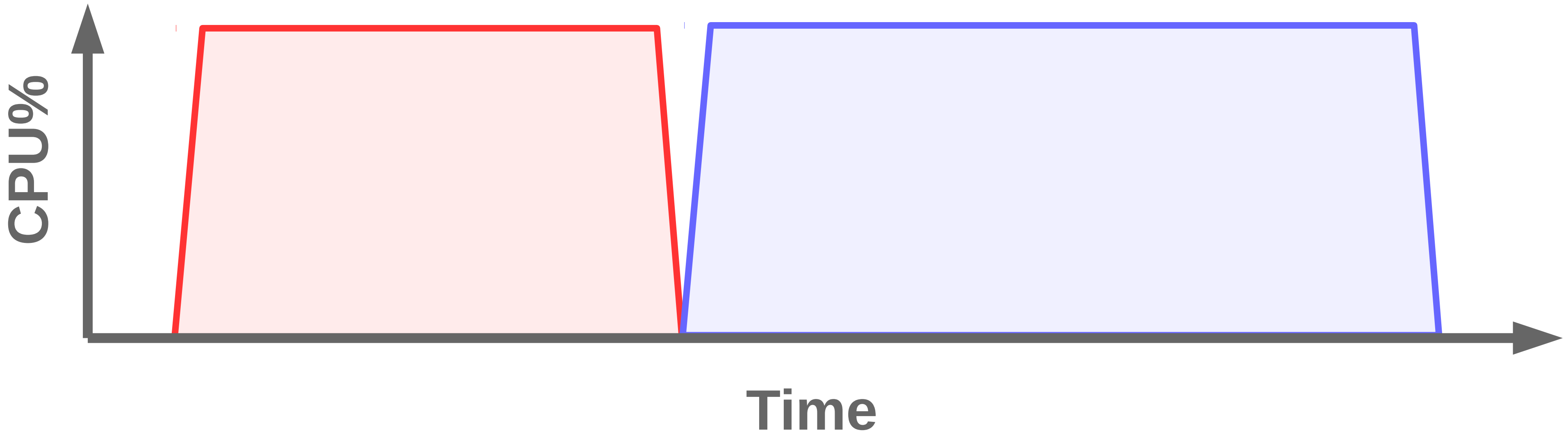


I/O – Utilisation

- **No I/O?**
  - Lucky!
- **Example; focus on CPU utilisation**

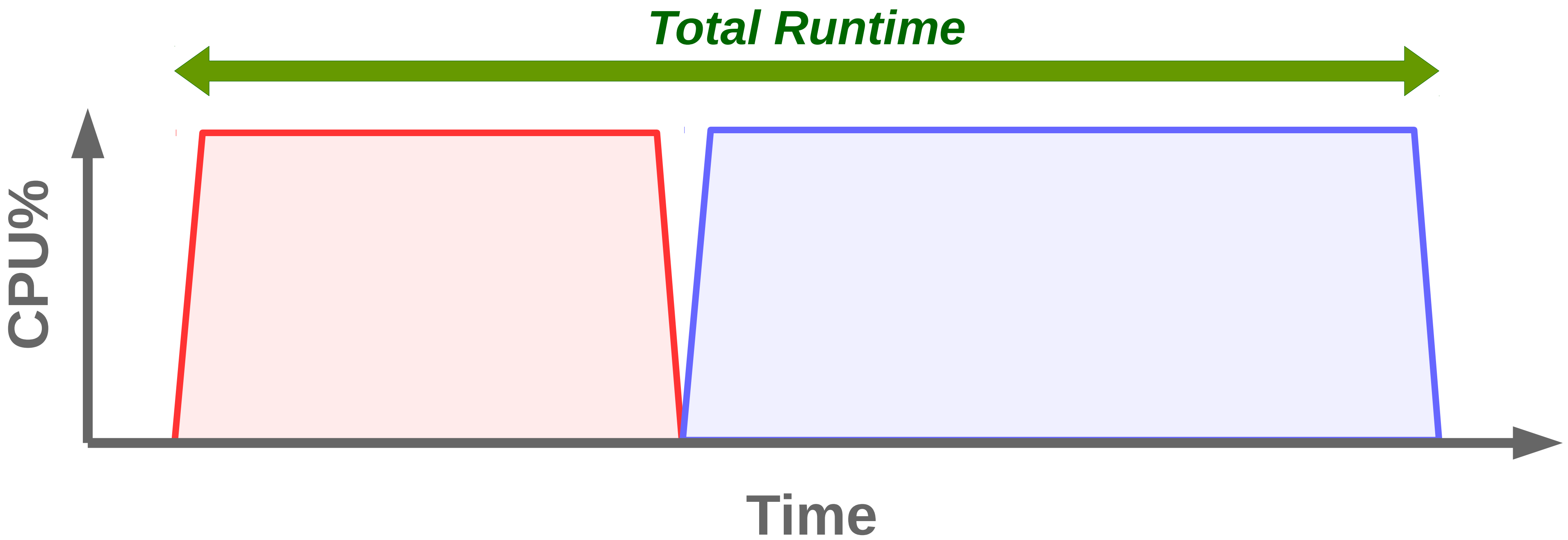
I/O – Utilisation

- **Program performing two tasks**
  - On a single CPU; One after another



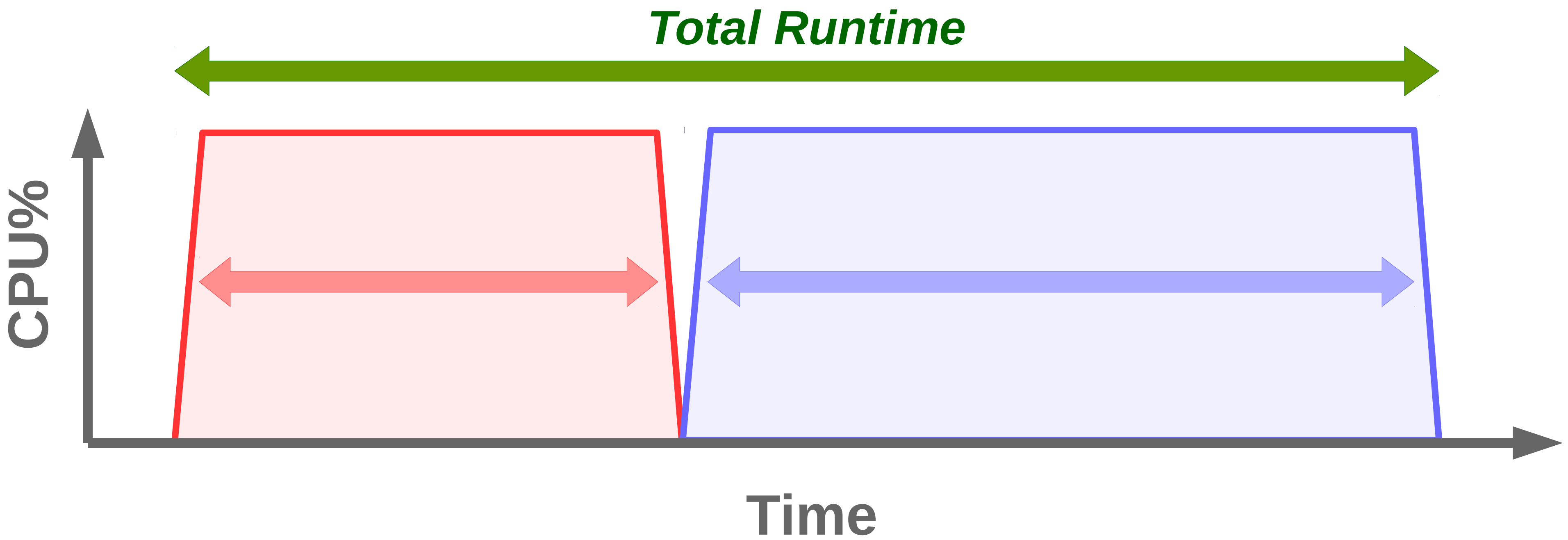
I/O – Utilisation

- **Total runtime gauges performance**
  - Time to execute each task



I/O – Utilisation

- **How to improve performance?**
  - Must optimise runtime of tasks

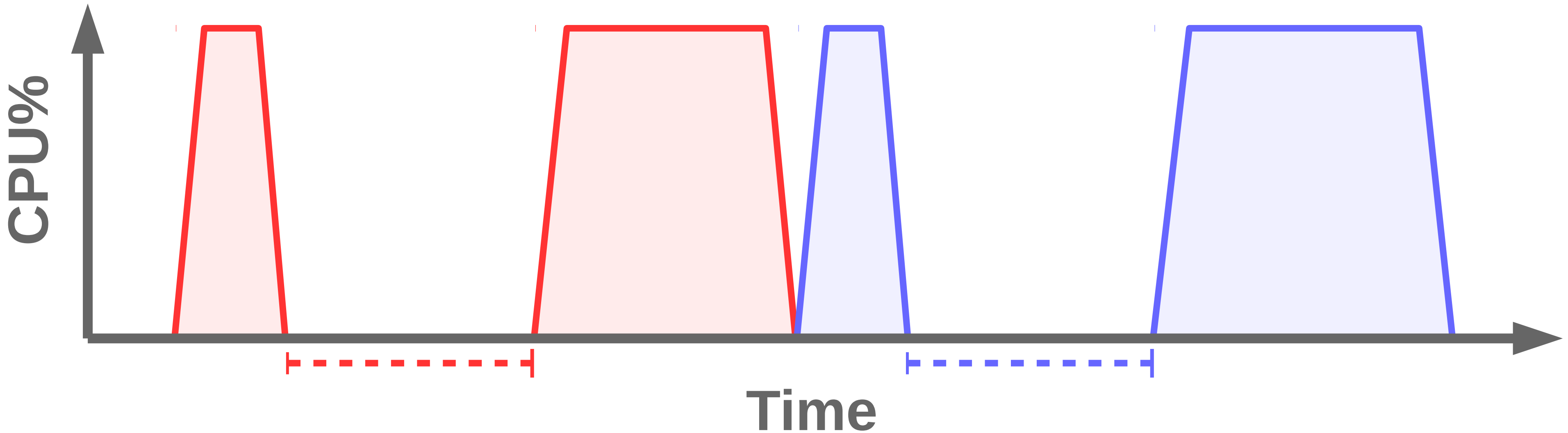


I/O – Utilisation

- **Slightly different example (with I/O)**

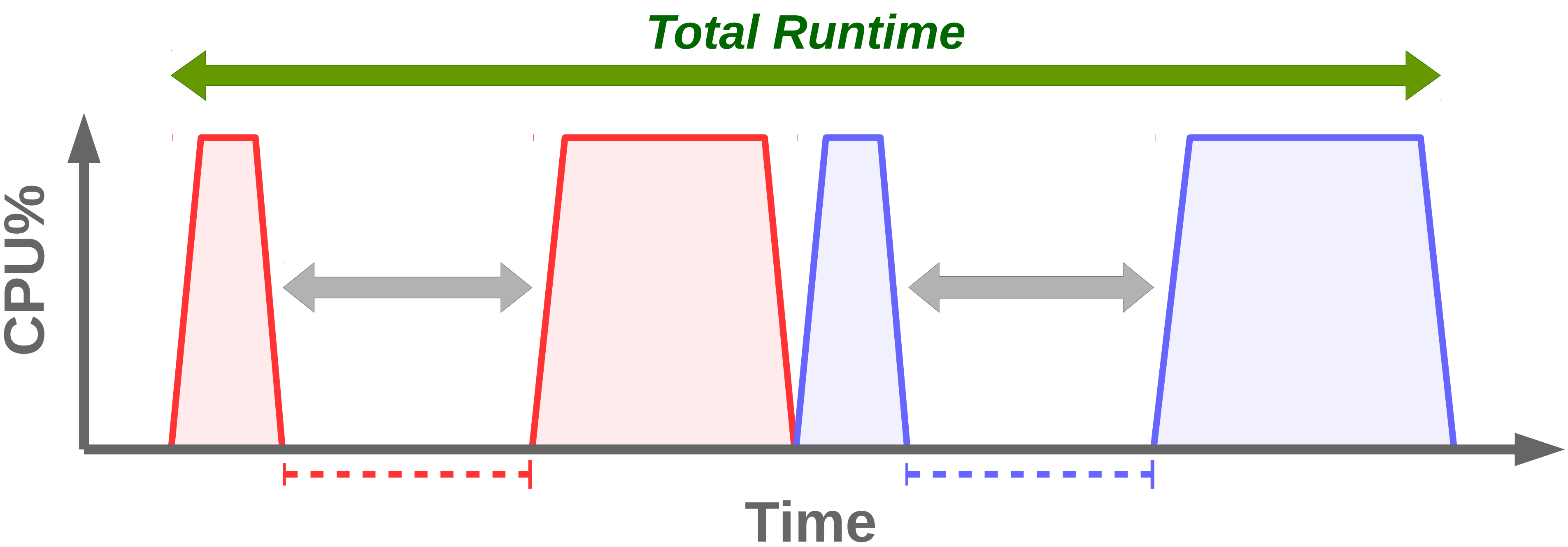
I/O – Utilisation

- **Tasks must now fetch some data first**
  - Either from a disk or a network connection
  - Can’t complete processing until data available



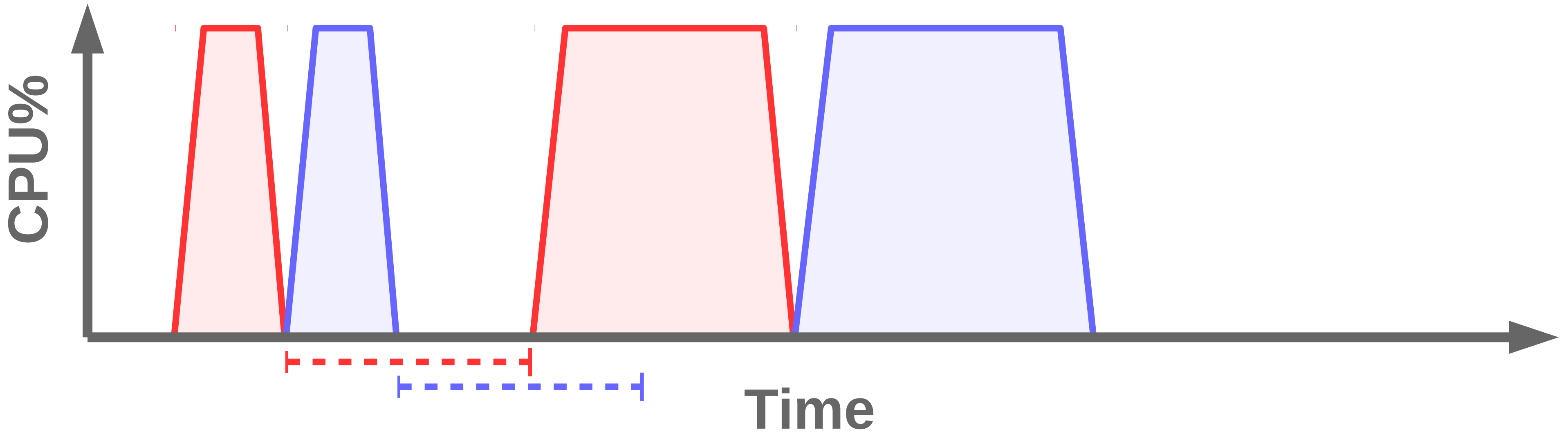
I/O – Utilisation

- **CPU time wasted waiting for data**
  - Ideally we could do work in those gaps



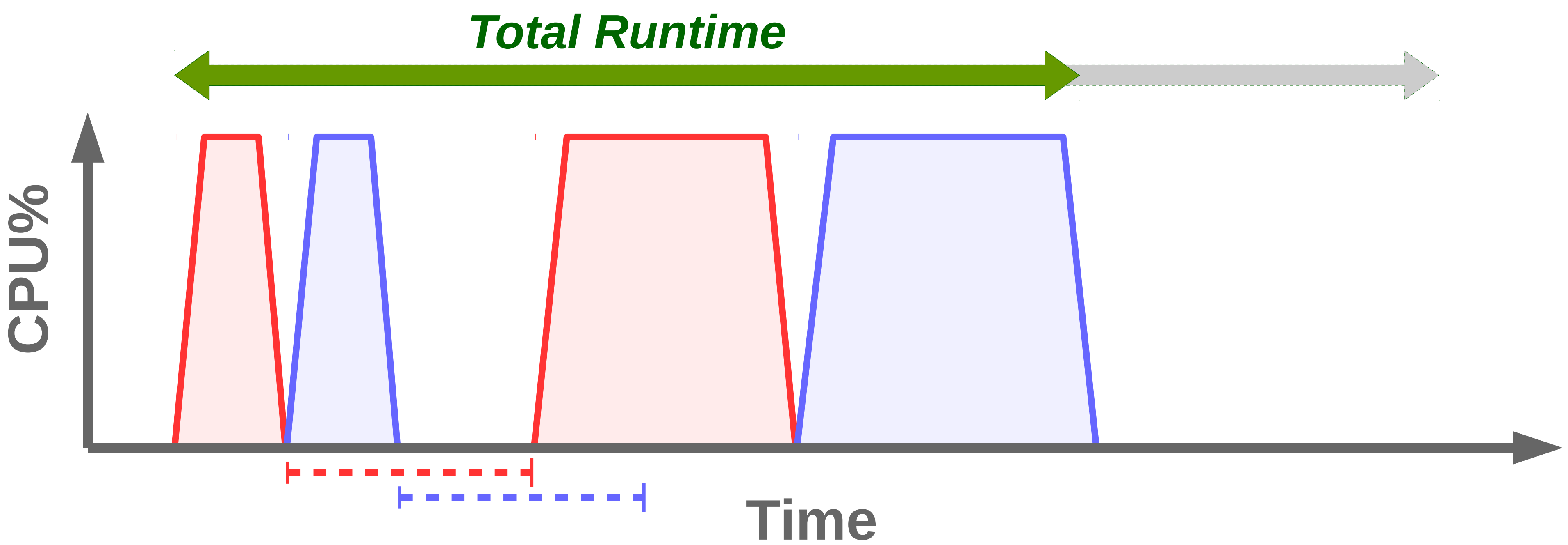
I/O – Utilisation

- **Start second task waiting for first**
  - Processing second task can start much sooner



I/O – Utilisation

- **Total runtime is reduced**
  - Due to increasing CPU utilisation



I/O – Utilisation

- **This is not necessarily *parallelism***
- **This is *concurrency***

  - When we discuss performance in this talk, it will focus on issues of concurrency on a single CPU
  - Improving I/O performance using *parallelism* (with multiple CPUs cores) is an entire other talk

I/O – Complexity

- **Desire for concurrency**
- **Root of much complexity**
  - Particularly in regards to code readability
  - Order of events is no longer serialised
- **Our code should at least read serially**
  - Given an event (e.g. disk read completing)
  - Able to read code executed as a consequence
  - Not as simple as it sounds

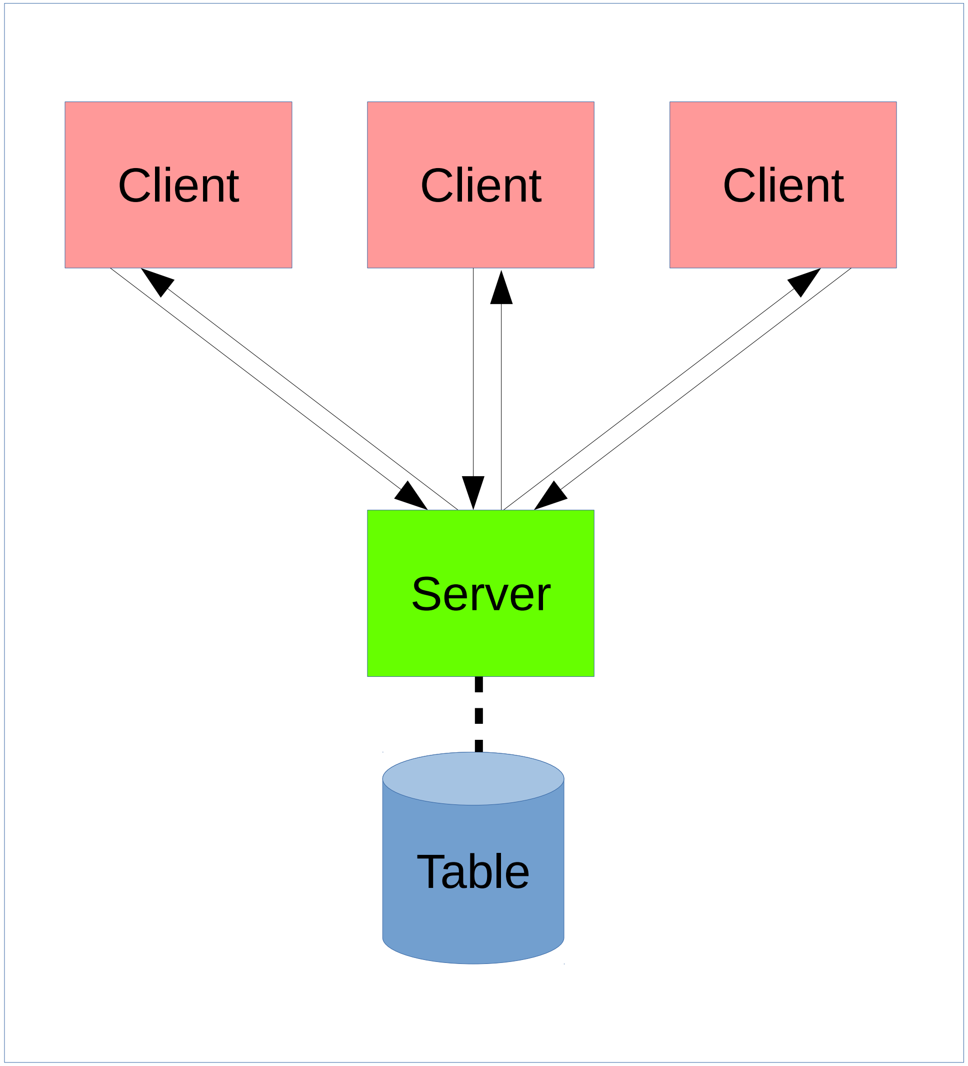
I/O - Testability

- **Reproducible testing is hard**
  - I/O is often unpredictable (web site?)
  - Logic can be tested with sufficient mocking
  - Still need to test external interactions
- **Concurrency just makes it harder**
  - Events can execute in different orders
  - Should we exhaustively test each path?
  - Or settle for something representative

I/O – Example Problem

- **Focus on network I/O**
- **Most prevalent and most explored**
- **Async. disk I/O is contentious topic**
  - Little agreement on the correct way to do it
- **Local socket connections (Unix Sockets)**
  - TCP overhead uninteresting
- **Basic example**
  - Little more than a lookup table
  - Think of it as the worlds simplest database

I/O – Example Problem



- Typical multi-user server example
- Key-value data to share between multiple clients
- No requests which modify the data

I/O – Performance Testing

- **How many operations per second**
  - Messages received and responses sent
- **How many concurrent operations**
  - Concurrent connections from multiple clients

- **50,000 connections, 10 requests each**
- **Vary number of concurrent connections**
  - 1, 5, 10, 50, 100, 500, 1000, 5000, 10000, 50000

I/O – Supporting Code

- “net” library
- Don’t use it!
- For educational purposes
- Wrappers around system calls
- Useful for exploring fundamental concepts, with minimal associated code noise (e.g. system calls)

I/O – Supporting Code

```
class socket {
public:
    void listen();
    socket accept();

    void send(string_view);
    optional<string> recv();

    // ...
};


// Server

socket bind_tcp(const string &port);
socket bind_local(const string &path);


// Client

socket connect_tcp(const string &hostname,
                   const string &port);
socket connect_local(const string &path);
```

- **Socket class**
  - Wraps C functions
  - Listen / Accept
  - Send / Recv
- **Bind sockets**
  - Local sockets used due to low overhead
- **Connect sockets**

I/O – Supporting Code

github.com/stevejims/acpp

Blocking

Blocking

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  for (;;) {
    auto conn = sock.accept();

    while (auto req = conn.recv()) {
      auto result = t.lookup(*req);
      conn.send(result);
    }
  }
}
```

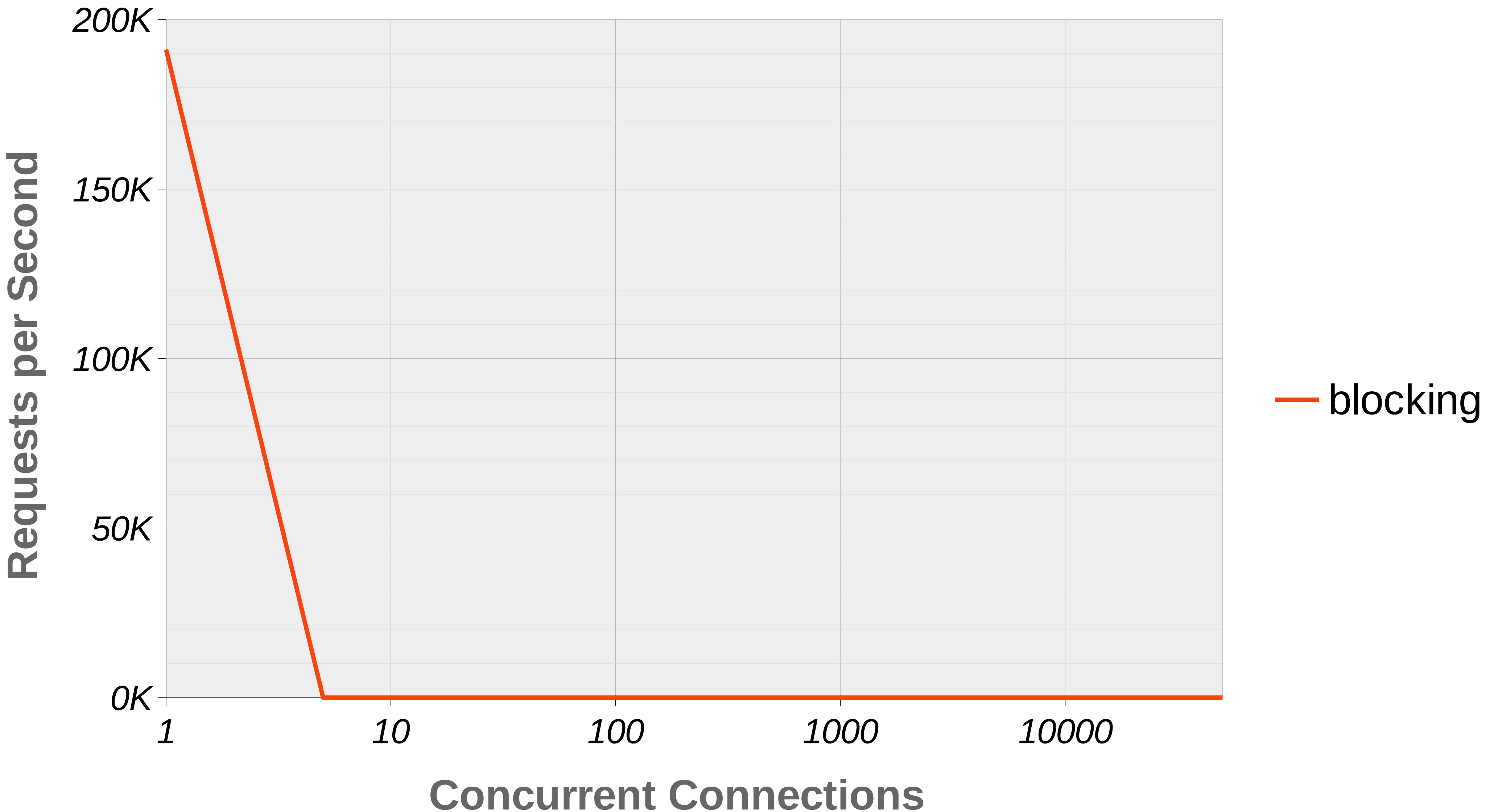
- Create resource
- Start listening
- Accept connections
- Receive messages
- ..until disconnect
- Lookup result
- Send response

Blocking – Performance

```cpp
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  for (;;) {
    auto conn = sock.accept();

    while (auto req = conn.recv()) {
      auto result = t.lookup(*req);
      conn.send(result);
    }
  }
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	2,616	191.1

Blocking – Performance



Blocking – Good?

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  for (;;) {
    auto conn = sock.accept();

    while (auto req = conn.recv()) {
      auto result = t.lookup(*req);
      conn.send(result);
    }
  }
}
```

- ✔ As simple as it gets!
- ✔ Clear event order
- ✔ Maximally efficient
- ✔ Just wraps syscalls
- ✔ I/O easily mocked

Blocking – Bad?

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  for (;;) {
    auto conn = sock.accept();

    while (auto req = conn.recv()) {
      auto result = t.lookup(*req);
      conn.send(result);
    }
  }
}
```

- ✗ **One connection only**
- ✗ **Low CPU utilisation**
  Idle pending receive
- ✗ **Inflexible**
  Second listener?

Select

Select - Overview

```
class fdset {
public:
    void add(int fd);
    void del(int fd);
    bool readable(int fd) const;

    // ...
};

fdset select(const fdset &);
```

- **Available since Linux 2.0**
  - Circa 1996
- **Takes a set of sockets**
  - Specified by bit set of integers - *“fdset”* (file descriptor set)
  - Clears bits for sockets which are not ready
  - Must test relevant bit to check if socket ready

Select

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  auto conns = std::vector<socket>();
  auto all_fds = fdset();
  all_fds.add(sock.fd());

  for (;;) {
    auto ready_fds = select(all_fds);

    if (ready_fds.readable(sock.fd())) {
      conns.emplace_back(sock.accept());
      all_fds.add(conns.back().fd());
    }
    for (auto it=conns.begin(); it!=conns.end();) {
      if (ready_fds.readable(it->fd())) {
        if (auto req = it->recv()) {
          auto result = t.lookup(*req);
          it->send(result);
        }
        else {
          all_fds.del(it->fd());
          it = conns.erase(it); continue;
        }
      }
      ++it;
    }
  }
}
```

- **1/2**
  - Listener set-up
  - State
  - Select
- **2/2**
  - Accept handling
  - Receive handling

Select – 1/2

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  auto conns = std::vector<socket>();
  auto all_fds = fdset();
  all_fds.add(sock.fd());

  for (;;) {
    auto ready_fds = select(all_fds);
```

- **Socket state**
- **All sockets *fdset***
  - Add listener
- **Event loop**
- **Select from *fdset***
  - Returns *fdset* of ready sockets

Select – 2/2

```
  if (ready_fds.readable(sock.fd())) {
    conns.emplace_back(sock.accept());
    all_fds.add(conns.back().fd());
  }
  for (auto it=conns.begin();
       it!=conns.end();) {
    if (ready_fds.readable(it->fd())) {
      if (auto req = it->recv()) {
        auto result = t.lookup(*req);
        it->send(result);
      }
      else {
        all_fds.del(it->fd());
        it = conns.erase(it); continue;
      }
    }
    ++it;
  }
}
```

- **Listener ready?**
  - Store socket
  - Add to ***all*** ***fdset***
- **Connection ready?**
  - Receive message
  - Get & send result
- **Connection closed?**
  - Remove from fdset
  - Release socket

Select – Performance

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  auto conns = std::vector<socket>();
  auto all_fds = fdset();
  all_fds.add(sock.fd());

  for (;;) {
    auto ready_fds = select(all_fds);

    if (ready_fds.readable(sock.fd())) {
      conns.emplace_back(sock.accept());
      all_fds.add(conns.back().fd());
    }
    for (auto it=conns.begin(); it!=conns.end();) {
      if (ready_fds.readable(it->fd())) {
        if (auto req = it->recv()) {
          auto result = t.lookup(*req);
          it->send(result);
        }
        else {
          all_fds.del(it->fd());
          it = conns.erase(it); continue;
        }
      }
      ++it;
    }
  }
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	3,120	160.3

Blocking

1	2,616	191.1

Select – Performance

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  auto conns = std::vector<socket>();
  auto all_fds = fdset();
  all_fds.add(sock.fd());

  for (;;) {
    auto ready_fds = select(all_fds);

    if (ready_fds.readable(sock.fd())) {
      conns.emplace_back(sock.accept());
      all_fds.add(conns.back().fd());
    }
    for (auto it=conns.begin(); it!=conns.end();) {
      if (ready_fds.readable(it->fd())) {
        if (auto req = it->recv()) {
          auto result = t.lookup(*req);
          it->send(result);
        }
        else {
          all_fds.del(it->fd());
          it = conns.erase(it); continue;
        }
      }
      ++it;
    }
  }
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	3,120	160.3
5	3,457	144.6
10	3,369	148.4
50	4,313	115.9
100	6,292	79.5
500	25,596	19.5
1,000	52,008	9.6

Select – Performance

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  auto conns = std::vector<socket>();
  auto all_fds = fdset();
  all_fds.add(sock.fd());

  for (;;) {
    auto ready_fds = select(all_fds);

    if (ready_fds.readable(sock.fd())) {
      conns.emplace_back(sock.accept());
      all_fds.add(conns.back().fd());
    }
    for (auto it=conns.begin(); it!=conns.end();) {
      if (ready_fds.readable(it->fd())) {
        if (auto req = it->recv()) {
          auto result = t.lookup(*req);
          it->send(result);
        }
        else {
          all_fds.del(it->fd());
          it = conns.erase(it); continue;
        }
      }
      ++it;
    }
  }
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	3,120	160.3
5	3,457	144.6
10	3,369	148.4
50	4,313	115.9
100	6,292	79.5
500	25,596	19.5
1,000	52,008	9.6
5,000	X	X
10,000	X	X
50,000	X	X

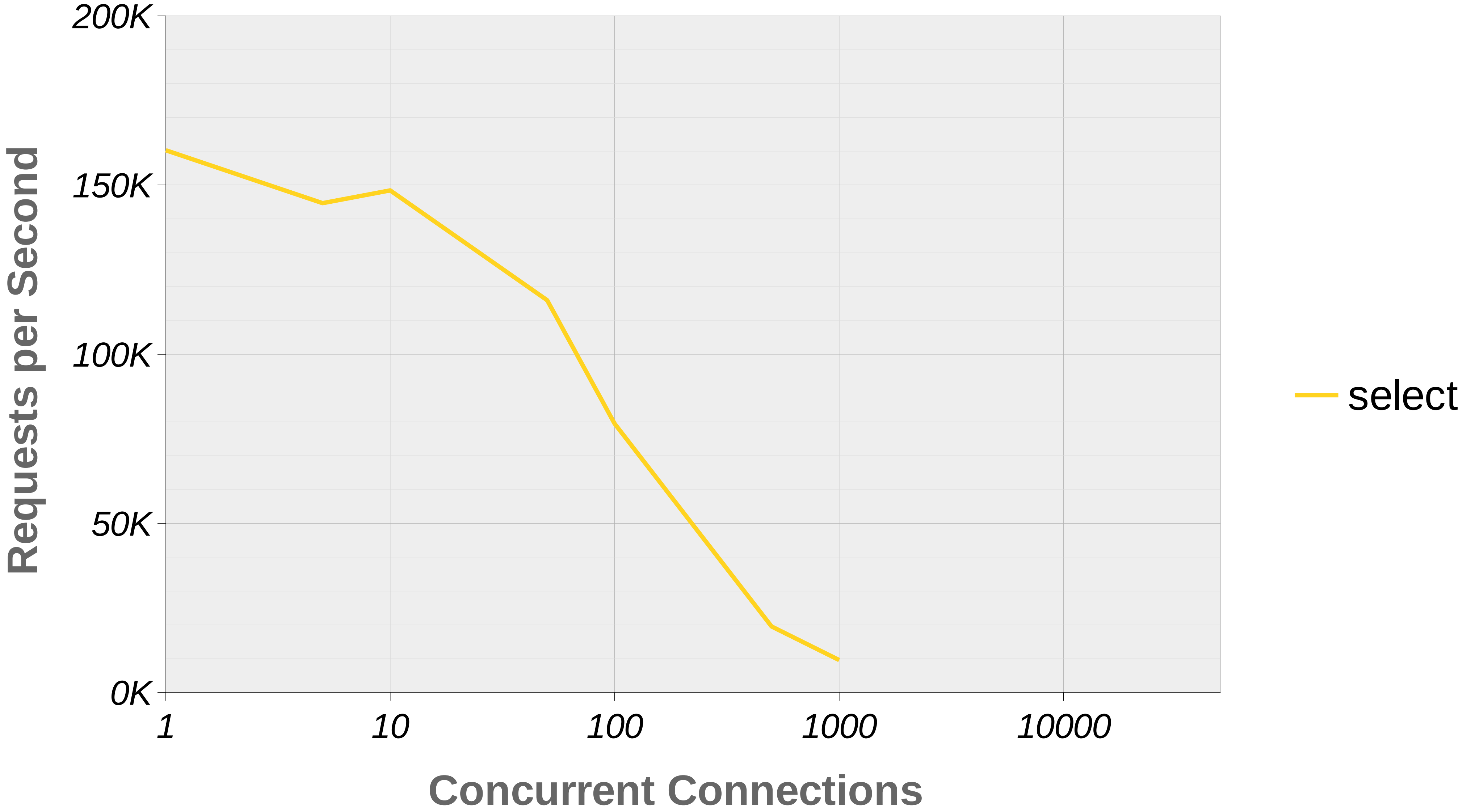
Select – Performance

- **Compile time size of “fdset”**
- **There is an alternative - “poll”**
- **But suffers from same scaling issues**

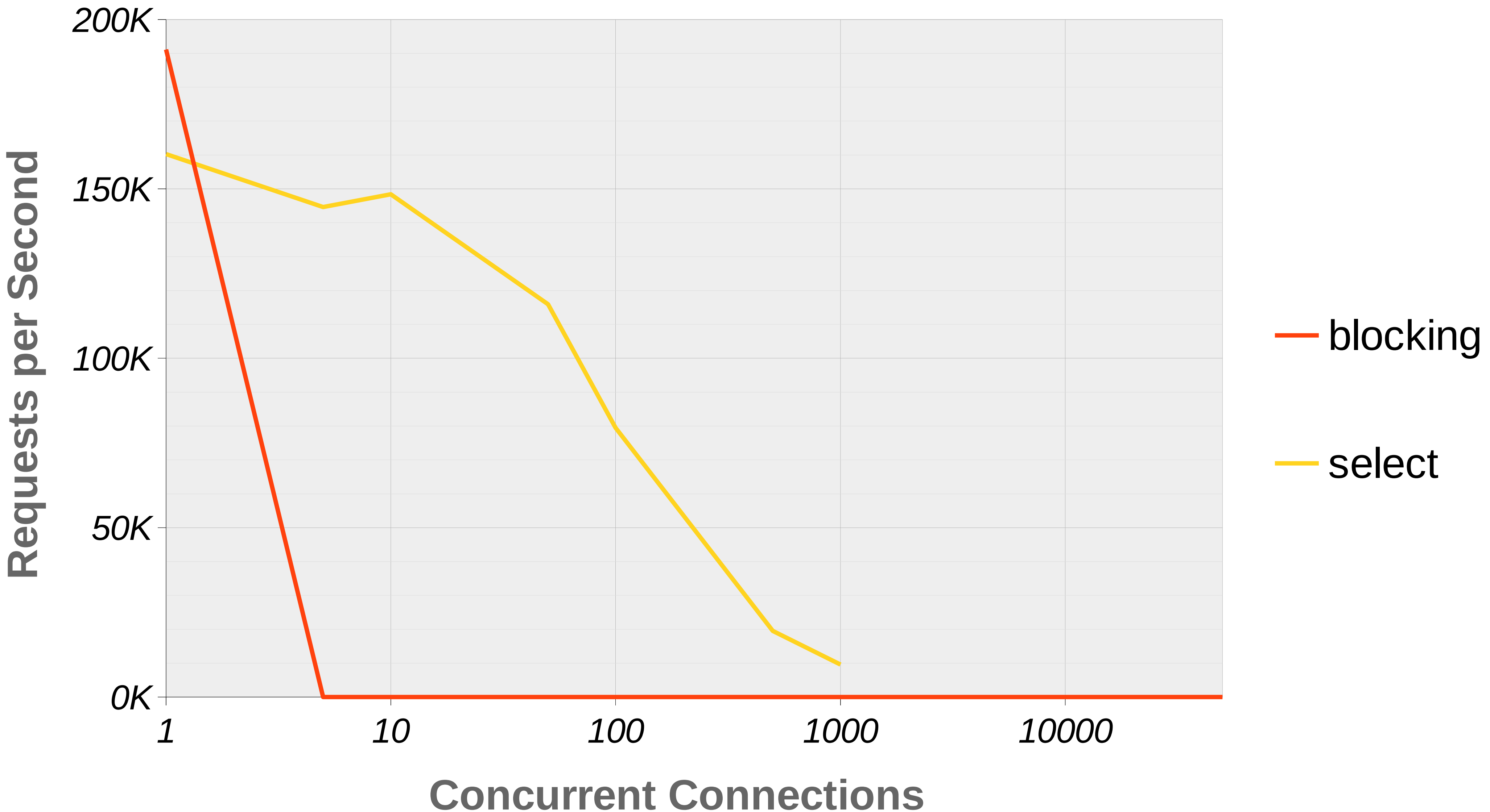
```
/* Number of descriptors that can fit in an `fd_set'.  */
#define __FD_SETSIZE        1024
```

bits/typesizes.h

Select – Performance



Select – Performance



Select – Good?

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  auto conns = std::vector<socket>();
  auto all_fds = fdset();
  all_fds.add(sock.fd());

  for (;;) {
    auto ready_fds = select(all_fds);

    if (ready_fds.readable(sock.fd())) {
      conns.emplace_back(sock.accept());
      all_fds.add(conns.back().fd());
    }
    for (auto it=conns.begin(); it!=conns.end();) {
      if (ready_fds.readable(it->fd())) {
        if (auto req = it->recv()) {
          auto result = t.lookup(*req);
          it->send(result);
        }
        else {
          all_fds.del(it->fd());
          it = conns.erase(it); continue;
        }
      }
      ++it;
    }
  }
}
```

- ✔ **Multiple concurrent connections**
- ✔ **Efficient for few connections**

Select – Bad?

```cpp
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  auto conns = std::vector<socket>();
  auto all_fds = fdset();
  all_fds.add(sock.fd());

  for (;;) {
    auto ready_fds = select(all_fds);

    if (ready_fds.readable(sock.fd())) {
      conns.emplace_back(sock.accept());
      all_fds.add(conns.back().fd());
    }
    for (auto it=conns.begin(); it!=conns.end();) {
      if (ready_fds.readable(it->fd())) {
        if (auto req = it->recv()) {
          auto result = t.lookup(*req);
          it->send(result);
        }
        else {
          all_fds.del(it->fd());
          it = conns.erase(it); continue;
        }
      }
      ++it;
    }
  }
}
```

- ✗ Control flow changed entirely
- ✗ Not trivial to follow order of events
- ✗ Complex state management
- ✗ Awful performance scaling number of connections

Threading

Threading - Overview

```
#include <thread>

using std::move;
using std::thread;

template <typename F>
void spawn(F f)
{
    auto t = thread(move(f));
    t.detach();
}
```

- **Available since Linux 2.0**
  - LinuxThreads
  - Circa 1996
- **Improved greatly in Linux 2.6**
  - NPTL
  - Circa 2002
- **ISO C++ since 2011**

Threading - Overview

```cpp
#include <thread>

using std::move;
using std::thread;

template <typename F>
void spawn(F f)
{
    auto t = thread(move(f));
    t.detach();
}
```

- **Simply put: Run a function asynchronously**
- **CPU time shared between threads**
- **OS switches between threads**
- **Even if function is blocked**

Threading

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        () mutable {
    for (;;) {
      auto conn = sock.accept();

      spawn([&t, conn=move(conn)]
            () mutable {
        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  pause();
}
```

- **Very similar**
- **Thread to accept new connections**
- **Thread to receive messages for each connection**
- **Sleep forever to prevent main exit**
  - Kernel switches between threads

Threading – Compare to Blocking

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        () mutable {
    for (;;) {
      auto conn = sock.accept();

      spawn([&t, conn=move(conn)]
            () mutable {
        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  pause();
}
```

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();


    for (;;) {
      auto conn = sock.accept();



        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }

     }



}
```

Threading – Performance

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        () mutable {
    for (;;) {
      auto conn = sock.accept();

      spawn([&t, conn=move(conn)]
            () mutable {
        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  pause();
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	2,752	181.7

Select

1	3,120	160.3

Threading – Performance

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        () mutable {
    for (;;) {
      auto conn = sock.accept();

      spawn([&t, conn=move(conn)]
            () mutable {
        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  pause();
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	2,752	181.7
5	3,282	152.3
10	3,597	139.0
50	3,816	131.0
100	3,893	128.4
500	4,715	106.1
1,000	5,015	99.7
5,000	5,127	97.5
10,000	5,970	83.8

Threading – Performance

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        () mutable {
    for (;;) {
      auto conn = sock.accept();

      spawn([&t, conn=move(conn)]
            () mutable {
        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  pause();
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	2,752	181.7
5	3,282	152.3
10	3,597	139.0
50	3,816	131.0
100	3,893	128.4
500	4,715	106.1
1,000	5,015	99.7
5,000	5,127	97.5
10,000	5,970	83.8
50,000	x	x

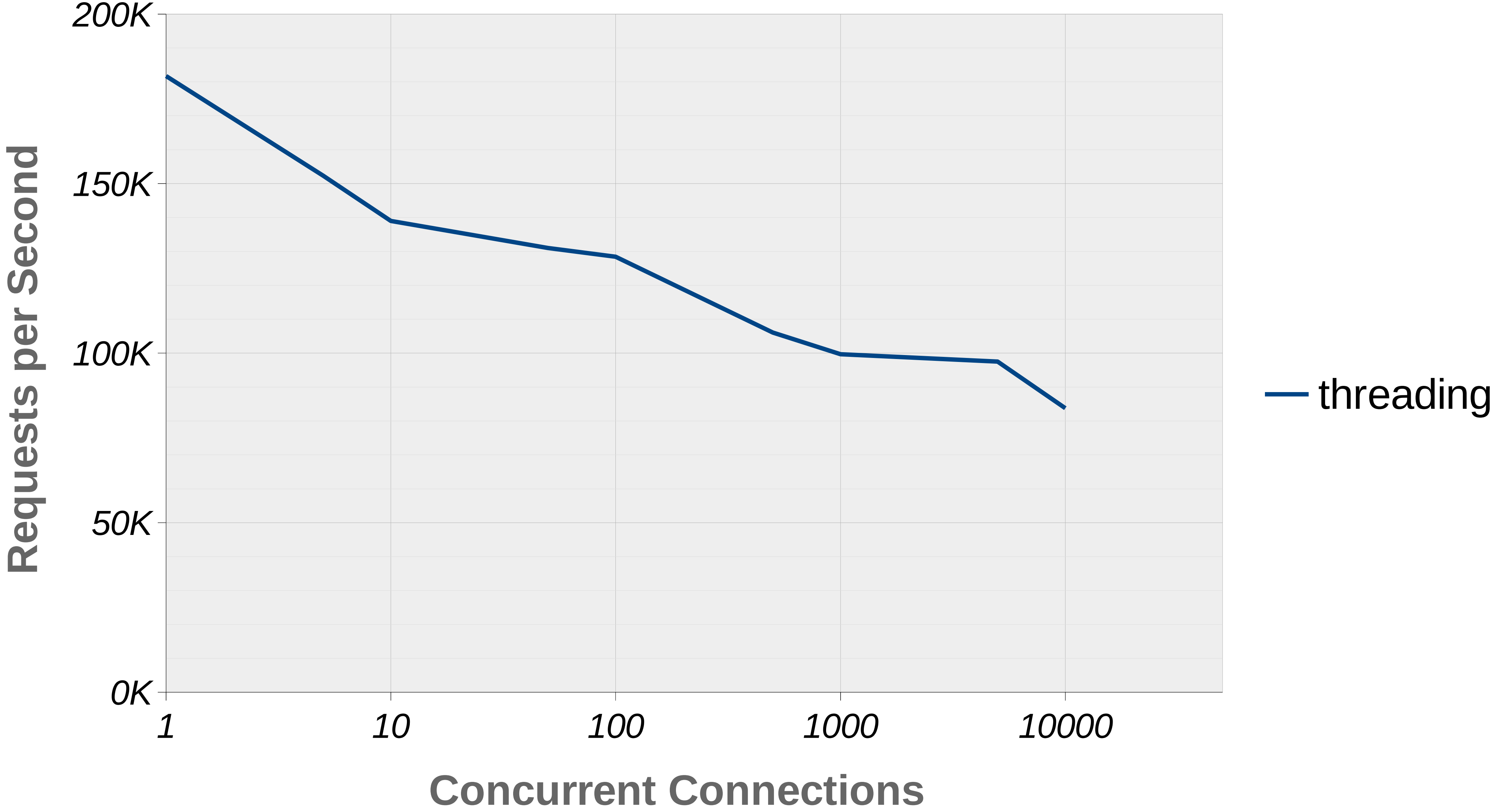
Threading - Performance

- **50,000 Threads?**

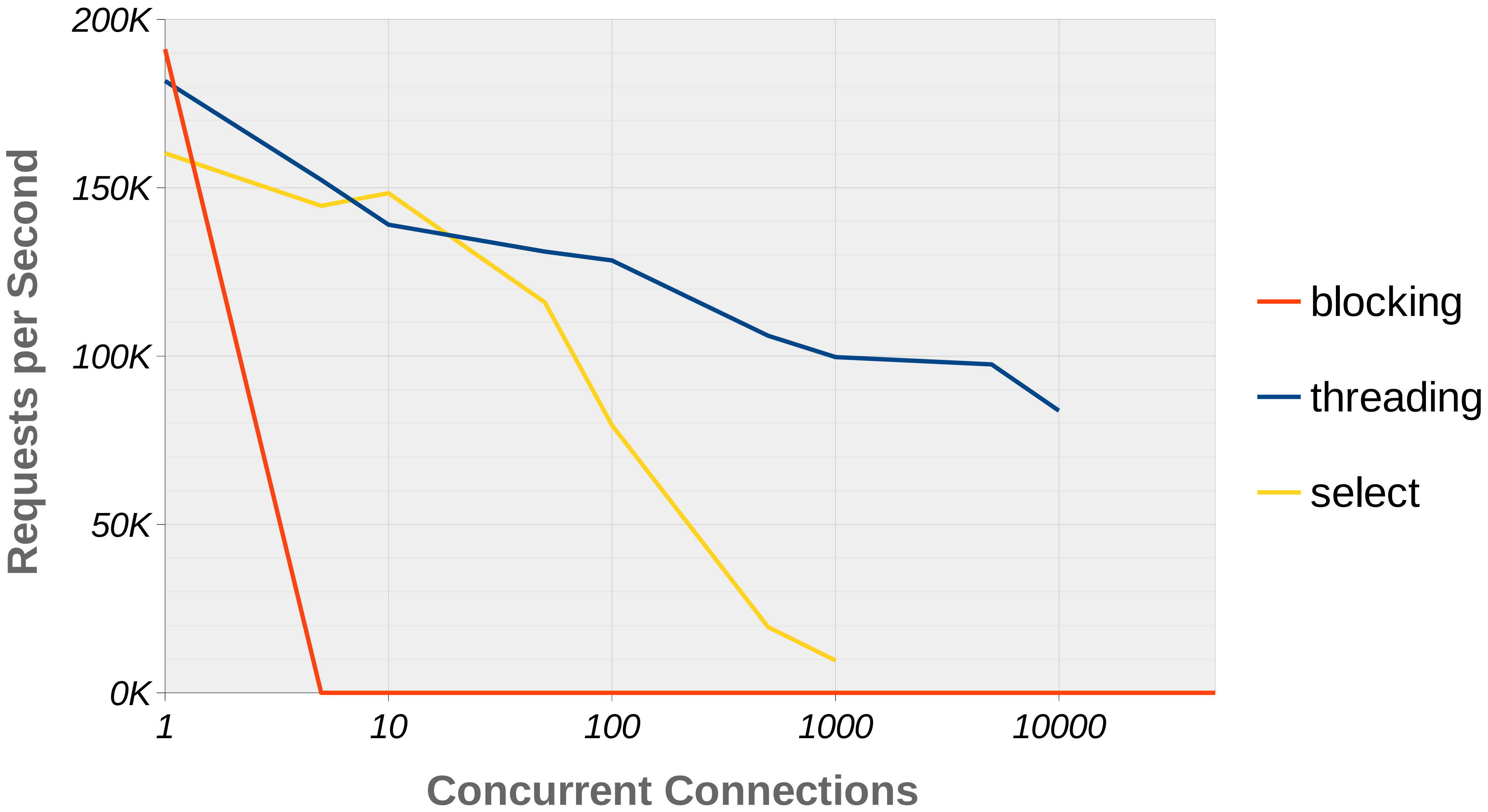
```
terminate called after throwing an instance of
'std::system_error'
  what():  Resource temporarily unavailable
```

(Limit depends on OS / Memory / etc..)

Threading - Performance



Threading - Performance



Threading – Good?

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        () mutable {
    for (;;) {
      auto conn = sock.accept();

      spawn([&t, conn=move(conn)]
            () mutable {
        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  pause();
}
```

- ✔ **Multiple concurrent connections**
- ✔ **Very readable**
  - ✔ Similar to blocking
- ✔ **Request handling reasonably efficient**
- ✔ **Easy to add new server listeners**

Threading – Bad?

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        () mutable {
    for (;;) {
      auto conn = sock.accept();

      spawn([&t, conn=move(conn)]
            () mutable {
        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  pause();
}
```

- ✗ **Shared state must be thread-safe**
- ✗ **Easy to introduce race conditions**
- ✗ **Exhaustive testing near impossible**
- ✗ **Thread creation hurts performance**
- ✗ **Context switching hurts performance**

Epoll

Epoll

(BSD: kqueue)

(Windows: IOCP)

Epoll - Overview

```
class epoll {
public:
    void add(int fd);
    void del(int fd);
    int wait();

    // ....
};
```

- **Available since Linux 2.6 (2.5.44)**
  - Circa 2003
- **Kernel resource Set of sockets**
  - Specified by *fd* (file descriptor)
  - Indicates which are ready (e.g. readable)
  - Specifically *<u>which</u>* socket(s) are ready

Epoll

```
int main()
{
    const auto t = table();
    auto sock = bind_local("socket");
    sock.listen();

    auto conns = unordered_map<int, socket>();
    epoll poll;
    poll.add(sock.fd());

    for (;;) {
        const auto fd = poll.wait();

        if (fd == sock.fd()) {
            auto conn = sock.accept();
            poll.add(conn.fd());
            conns[conn.fd()] = move(conn);
        }
        else {
            auto &conn = conns.at(fd);
            if (auto req = conn.recv()) {
                auto result = t.lookup(*req);
                conn.send(result);
            }
            else {
                poll.del(fd);
                conns.erase(fd);
            }
        }
    }
}
```

- **1/2**
  - Listener set-up
  - State
  - Event waiting
- **2/2**
  - Accept handling
  - Receive handling

Epoll – 1/2

```
const auto t = table();
auto sock = bind_local("socket");
sock.listen();

auto conns = unordered_map<int, socket>();
epoll poll;
poll.add(sock.fd());

for (;;) {
    const auto fd = poll.wait();
```

- **Create epoll**
  - Add listener *fd*
- **Event loop**
- **Wait on epoll**
  - Returns the *fd* of ready socket
- **Socket state**
  - Lookup by *fd*

Epoll – 2/2

```
    if (fd == sock.fd()) {
        auto conn = sock.accept();
        poll.add(conn.fd());
        conns[conn.fd()] = move(conn);
    }
    else {
        auto &conn = conns.at(fd);
        if (auto req = conn.recv()) {
            auto result = t.lookup(*req);
            conn.send(result);
        }
        else {
            poll.del(fd);
            conns.erase(fd);
        }
    }
}
```

- **Listener ready?**
  - Add to epoll set
  - Store socket by fd
- **Connection ready**
  - Get socket by fd
  - Receive message
  - Get & send result
- **Connection closed?**
  - Remove from epoll
  - Release socket

Epoll – Performance

```
int main()
{
    const auto t = table();
    auto sock = bind_local("socket");
    sock.listen();

    auto conns = unordered_map<int, socket>();
    epoll poll;
    poll.add(sock.fd());

    for (;;) {
        const auto fd = poll.wait();

        if (fd == sock.fd()) {
            auto conn = sock.accept();
            poll.add(conn.fd());
            conns[conn.fd()] = move(conn);
        }
        else {
            auto &conn = conns.at(fd);
            if (auto req = conn.recv()) {
                auto result = t.lookup(*req);
                conn.send(result);
            }
            else {
                poll.del(fd);
                conns.erase(fd);
            }
        }
    }
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	3,096	161.5
5	3,030	165.0
10	3,040	164.5

Threading

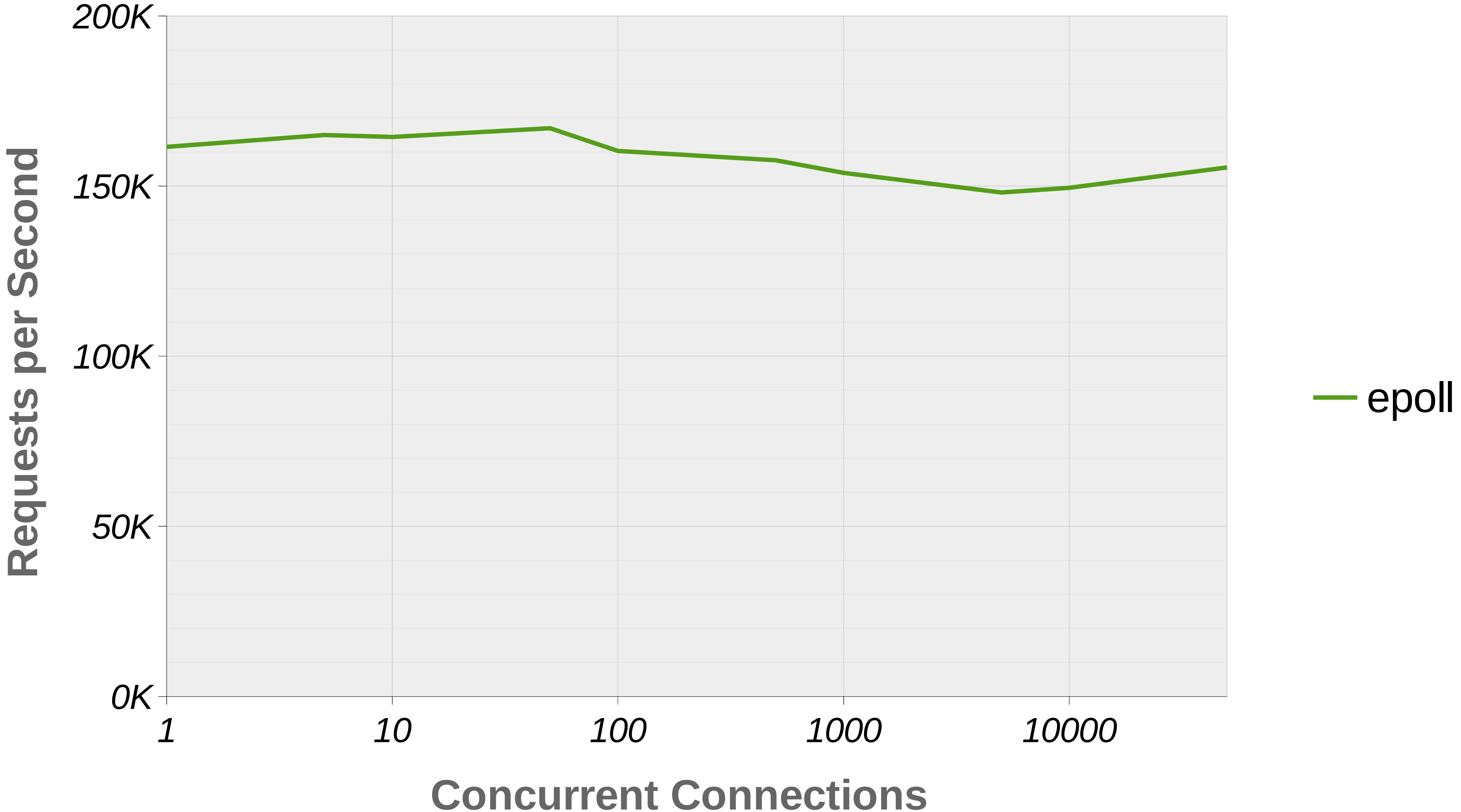
1	2,752	181.7
5	3,282	152.3
10	3,597	139.0

Epoll – Performance

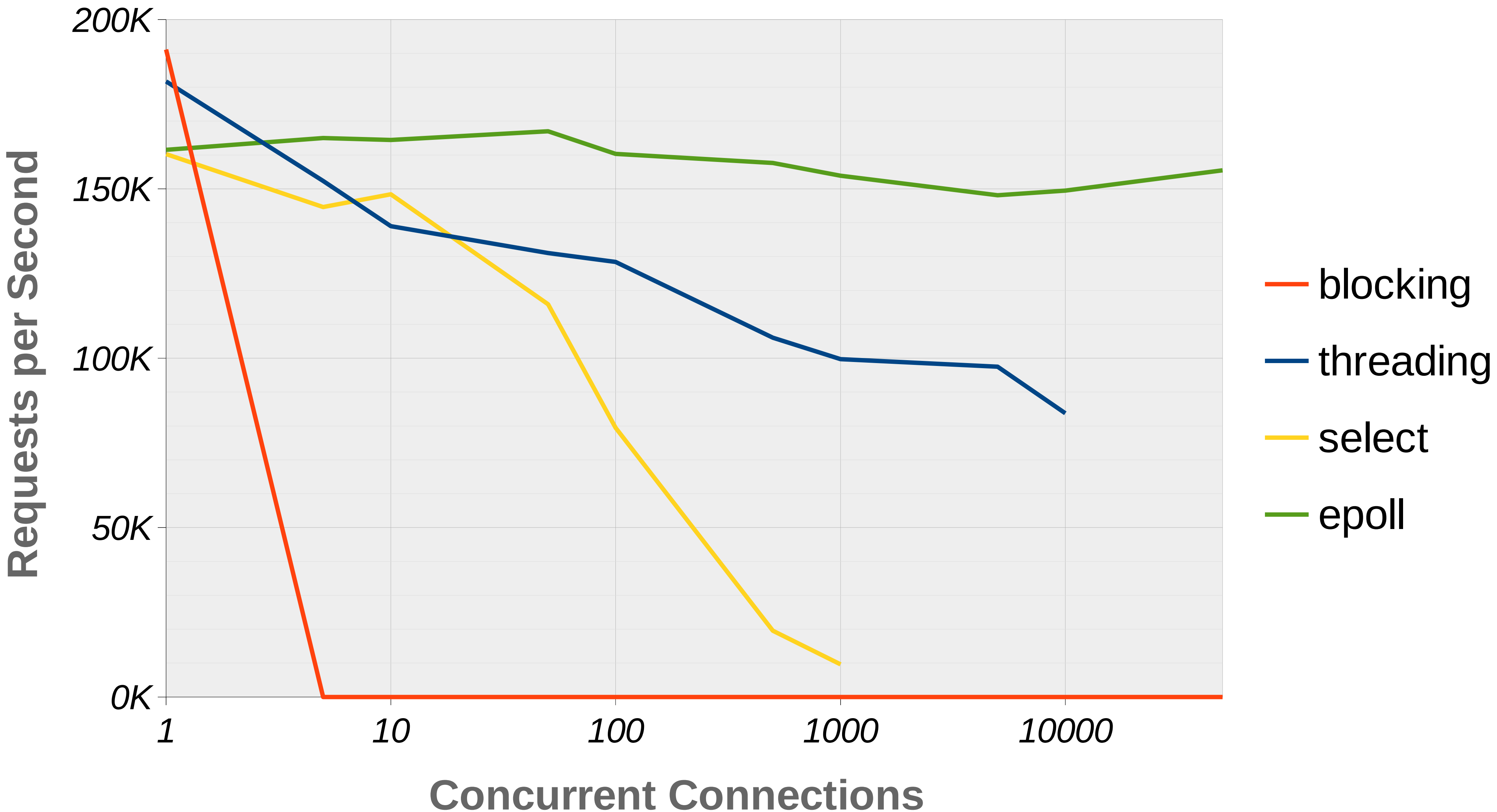
```
int main()
{
    const auto t = table();
    auto sock = bind_local("socket");
    sock.listen();

    auto conns = unordered_map<int, socket>();
    epoll poll;
    poll.add(sock.fd());

    for (;;) {
        const auto fd = poll.wait();

        if (fd == sock.fd()) {
            auto conn = sock.accept();
            poll.add(conn.fd());
            conns[conn.fd()] = move(conn);
        }
        else {
            auto &conn = conns.at(fd);
            if (auto req = conn.recv()) {
                auto result = t.lookup(*req);
                conn.send(result);
            }
            else {
                poll.del(fd);
                conns.erase(fd);
            }
        }
    }
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	3,096	161.5
5	3,030	165.0
10	3,040	164.5
50	2,994	167.0
100	3,119	160.3
500	3,172	157.6
1,000	3,249	153.9
5,000	3,376	148.1
10,000	3,345	149.5
50,000	3,216	155.5

Epoll - Performance



Epoll - Performance



Epoll – Good?

```
int main()
{
    const auto t = table();
    auto sock = bind_local("socket");
    sock.listen();

    auto conns = unordered_map<int, socket>();
    epoll poll;
    poll.add(sock.fd());

    for (;;) {
        const auto fd = poll.wait();

        if (fd == sock.fd()) {
            auto conn = sock.accept();
            poll.add(conn.fd());
            conns[conn.fd()] = move(conn);
        }
        else {
            auto &conn = conns.at(fd);
            if (auto req = conn.recv()) {
                auto result = t.lookup(*req);
                conn.send(result);
            }
            else {
                poll.del(fd);
                conns.erase(fd);
            }
        }
    }
}
```

- ✔ Multiple concurrent connections
- ✔ Single threaded
- ✔ No race conditions
- ✔ Efficient scaling to many connections

Epoll – Bad?

```
int main()
{
    const auto t = table();
    auto sock = bind_local("socket");
    sock.listen();

    auto conns = unordered_map<int, socket>();
    epoll poll;
    poll.add(sock.fd());

    for (;;) {
        const auto fd = poll.wait();

        if (fd == sock.fd()) {
            auto conn = sock.accept();
            poll.add(conn.fd());
            conns[conn.fd()] = move(conn);
        }
        else {
            auto &conn = conns.at(fd);
            if (auto req = conn.recv()) {
                auto result = t.lookup(*req);
                conn.send(result);
            }
            else {
                poll.del(fd);
                conns.erase(fd);
            }
        }
    }
}
```

- ✗ Control flow changed entirely
- ✗ Not trivial to follow order of events
- ✗ Complex state management

Callbacks

Callbacks - Overview

- **“Genre” of libraries**
  - **C:** libevent, libev, libuv
  - **C++:** Boost ASIO
  - **Python:** Twisted, asyncio
  - **Javascript:** Promises
- **Event loop / “reactor” pattern**
- **Usually wraps an OS-level primitive**
  - select / epoll / kqueue / IOCP

Callbacks – More Epoll

```
class epoll {
public:
    void add(int fd);
    void del(int fd);
    int wait();

    // ....
};
```

```
class epoll_ptr {
public:
    void add(int fd, void *ptr);
    void del(int fd);
    void *wait();

    // ....
};
```

- **Epoll is more flexible than shown**
- **Can specify an arbitrary 8 bytes**
  - Such as a pointer
  - ...to a functor?
- **Returns it instead**
- **Avoid *fd* lookup**

Callbacks – In Theory (1)

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  sock.accept([&t] (auto conn) {
    conn.recv([&t, &conn] (auto req) {
      if (req) {
        auto result = t.lookup(*req);
        conn.send(result);
      }
    });
  });

  loop();
}
```

- **Pass functor into action to invoke on completion**

Callbacks – In Theory (1)

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  sock.accept([&t] (auto conn) {
    conn.recv([&t, &conn] (auto req) {
      if (req) {
        auto result = t.lookup(*req);
        conn.send(result);
      }
    });
  });

  loop();
}
```

- **Pass functor into action to invoke on completion**
- **Code compiles but will crash**
- **Socket needed in & outside lambda**
- **Capture reference of stack variable**

Callbacks – In Theory (2)

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  sock.accept([&t] (auto s) {
    auto conn =
        make_shared<socket>(move(s));
    conn->recv([&t, conn] (auto req) {
      if (req) {
        auto result = t.lookup(*req);
        conn->send(result);
      }
    });
  });

  loop();
}
```

- **Fixed by holding socket in shared_ptr**
- **No crash, but code is still deficient**
- **Will only ever:**
  - Process one message
  - Accept one connection
- **How do we loop?**

Callbacks – In Theory (3)

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  function<void(socket)> on_accept =
  [&t, &sock, &on_accept] (auto s) {
    auto conn =
      make_shared<socket>(move(s));

    function<void(optional<string>)> on_recv =
    [&t, conn, &on_recv] (auto req) {
      if (req) {
        auto result = t.lookup(*req);
        conn->send(result);
        conn->recv(on_recv);
      }
    };
    conn->recv(on_recv);
    sock.accept(on_accept);
  };
  sock.accept(on_accept);
  loop();
}
```

- **Callbacks have to become recursive**
- **Pass the callbacks into themselves**
  - Must be reference
- **Works for on_accept**
- **Crashes for on_recv**
- **Capturing variable on stack (again) which goes out of scope**

Asynchronous Lambda
+
Reference Capture

= :(*[1][2]*

[1] **usually**
[2] **probably**

Callbacks – Working Code!

```
struct on_recv {
  const table &t;
  shared_ptr<socket> conn;
  void operator()(optional<string> req) const {
    if (req) {
      auto result = t.lookup(*req);
      conn->send(result);
      conn->recv(on_recv{t, conn});
    }
  }
};
struct on_accept {
  const table &t;
  socket &sock;
  void operator()(socket s) const {
    auto conn = make_shared<socket>(move(s));
    conn->recv(on_recv{t, conn});
    sock.accept(on_accept{t, sock});
  }
};
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();
  sock.accept(on_accept{t, sock});
  loop();
}
```

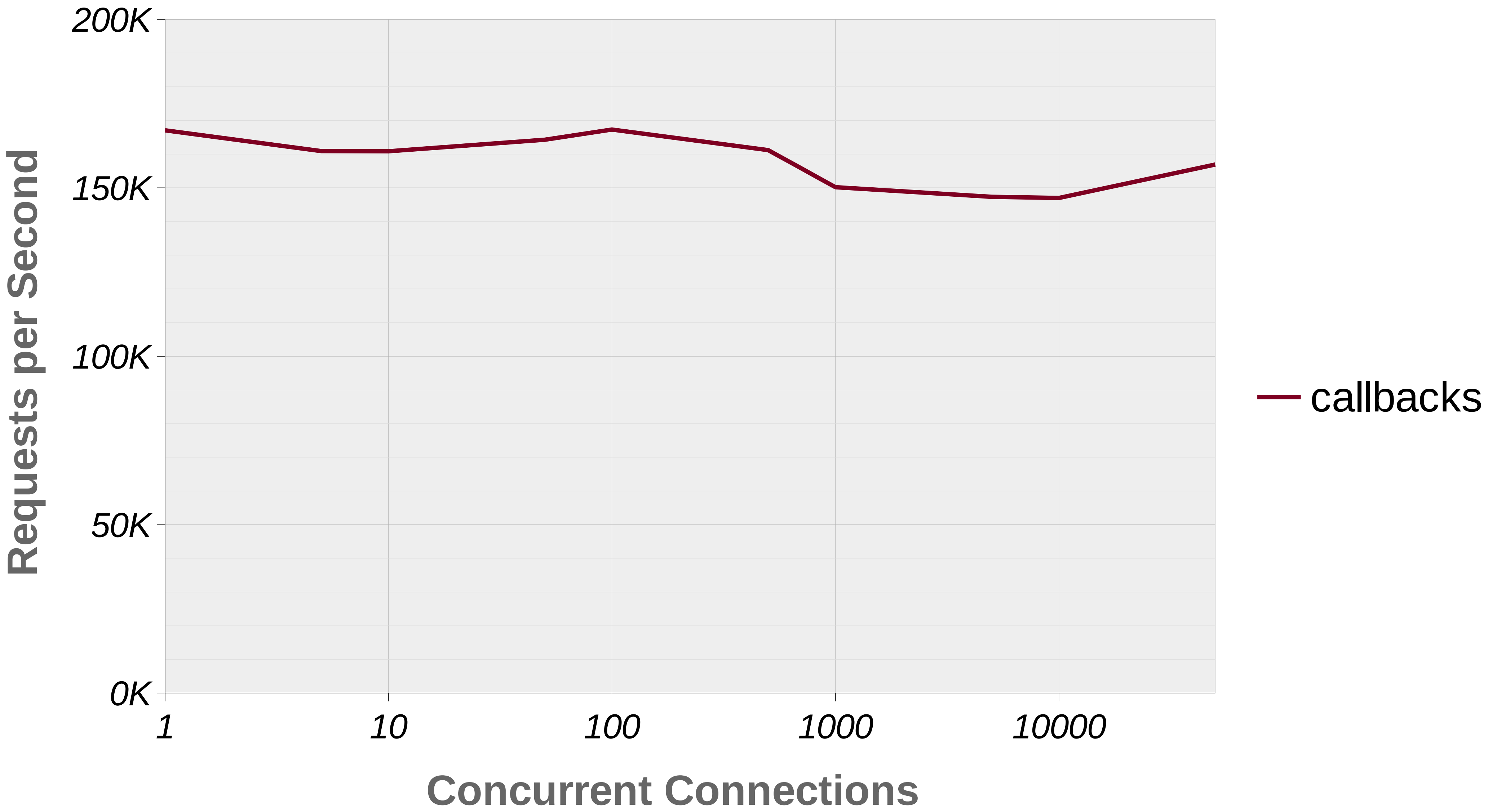
- **Normal setup**
- **Run event loop**
- **Accept connections**
- **Receive messages**
- **Notice how we read the code bottom to top...**

Callbacks – Performance

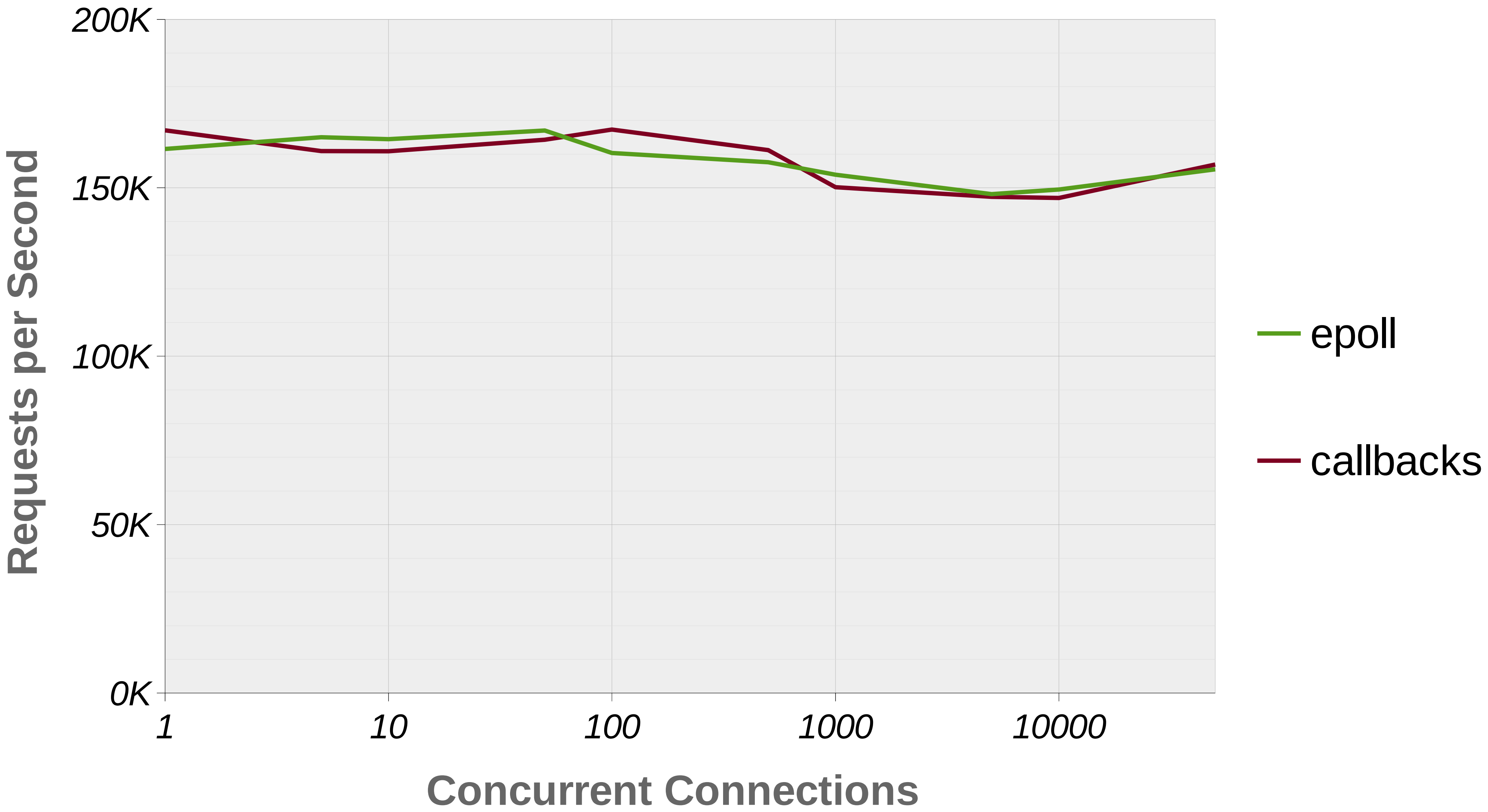
```
struct on_recv {
  const table &t;
  shared_ptr<socket> conn;
  void operator()(optional<string> req) const {
    if (req) {
      auto result = t.lookup(*req);
      conn->send(result);
      conn->recv(on_recv{t, conn});
    }
  }
};
struct on_accept {
  const table &t;
  socket &sock;
  void operator()(socket s) const {
    auto conn = make_shared<socket>(move(s));
    conn->recv(on_recv{t, conn});
    sock.accept(on_accept{t, sock});
  }
};
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();
  sock.accept(on_accept{t, sock});
  loop();
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	2,994	167.0
5	3,107	160.9
10	3,109	160.8
50	3,044	164.3
100	2,989	167.3
500	3,102	161.2
1,000	3,330	150.2
5,000	3,394	147.3
10,000	3,402	147.0
50,000	3,187	156.9

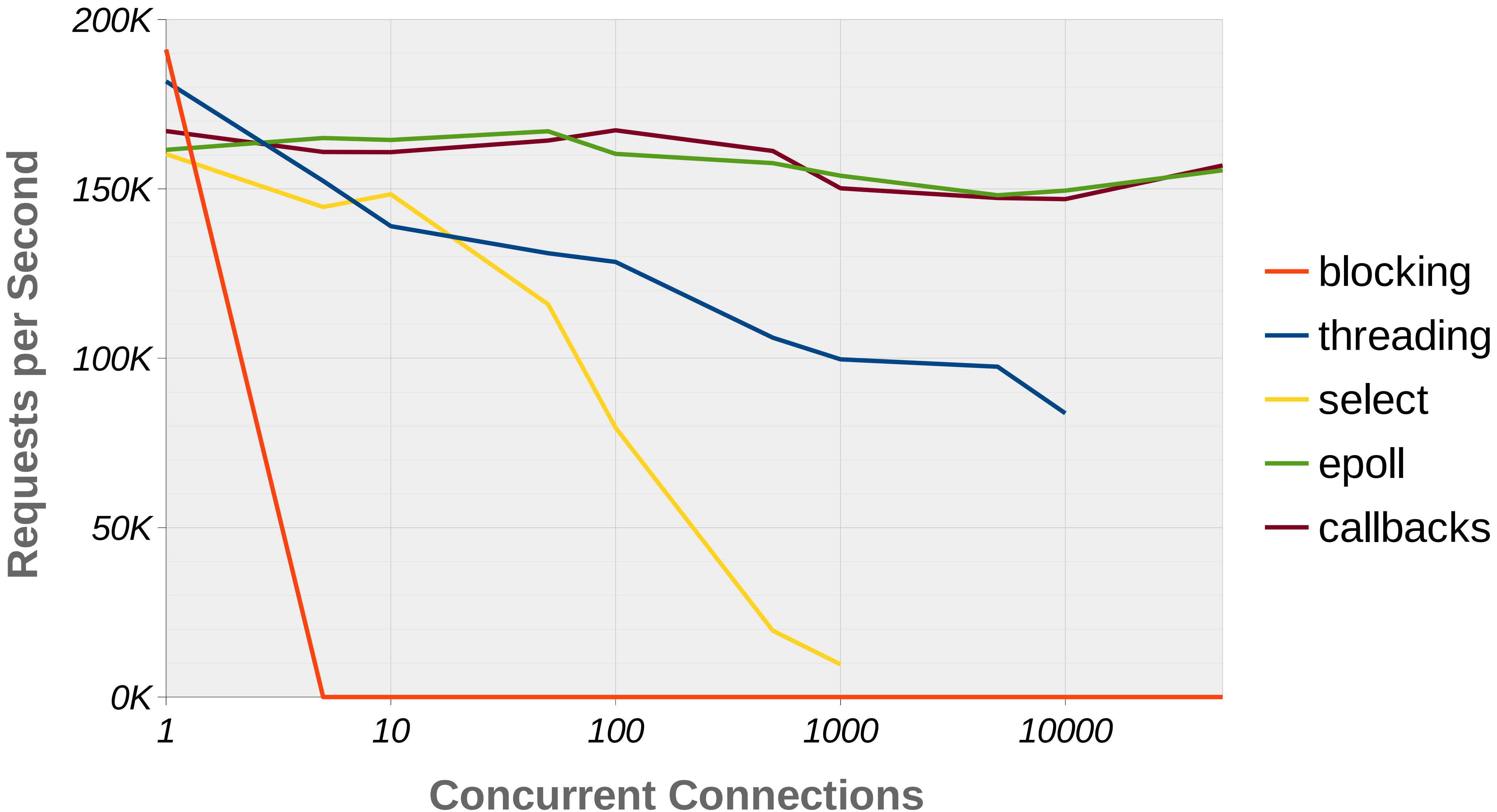
Callbacks - Performance



Callbacks - Performance



Callbacks - Performance



Callbacks – Good?

```cpp
struct on_recv {
  const table &t;
  shared_ptr<socket> conn;
  void operator()(optional<string> req) const {
    if (req) {
      auto result = t.lookup(*req);
      conn->send(result);
      conn->recv(on_recv{t, conn});
    }
  }
};
struct on_accept {
  const table &t;
  socket &sock;
  void operator()(socket s) const {
    auto conn = make_shared<socket>(move(s));
    conn->recv(on_recv{t, conn});
    sock.accept(on_accept{t, sock});
  }
};
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();
  sock.accept(on_accept{t, sock});
  loop();
}
```

- ✔ **All the advantages of using epoll directly**
  - ✔ **Multiple connections**
  - ✔ **Single threaded**
  - ✔ **No race conditions**
  - ✔ **Efficient scaling**
- ✔ **Describing each action is clearer & more flexible**
- ✔ **Some similarity to blocking code**

Callbacks – Bad?

```
struct on_recv {
  const table &t;
  shared_ptr<socket> conn;
  void operator()(optional<string> req) const {
    if (req) {
      auto result = t.lookup(*req);
      conn->send(result);
      conn->recv(on_recv{t, conn});
    }
  }
};
struct on_accept {
  const table &t;
  socket &sock;
  void operator()(socket s) const {
    auto conn = make_shared<socket>(move(s));
    conn->recv(on_recv{t, conn});
    sock.accept(on_accept{t, sock});
  }
};
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();
  sock.accept(on_accept{t, sock});
  loop();
}
```

- ✗ Whilst clearer, control flow is now inverted
  - ✗ Actions to follow event often must be written before initiating action
- ✗ Recursive callbacks required for looping
- ✗ Careful management of state lifetimes
- ✗ Risk of cycles due to shared_ptr usage

Futures

Futures

“Event Loop”

JavaScript
Promise

Python
Deferred

“Threading”

C++11
std::future

Java
Future

Futures – Event Loop Centric

"Event Loop"

JavaScript
Promise

Python
Deferred

- **Syntactic sugar around callbacks**
- **Subjectively more readable code**
- **Especially useful with exceptions**
  - (Not shown today)
- **Threads not needed**
  - No need for synchronisation

Futures – Threading Centric

- **Similar interface**
- **Can be blocking or non-blocking**
  - C++11: Blocking .get
  - C++?: Non-blocking?
- **Useful when thread consumes data from another**
- **Provides necessary safety (sync)**

“<u>Threading</u>”

C++11
std::future

Java
Future

Futures – Threading Centric

- **Suitable for I/O**
  - But not *necessary*
- **Threads not required for concurrent I/O**
- **Better suited to parallel compute**
  - Particularly when using thread pools

"Threading"

C++11
std::future

Java
Future

Coroutines

Coroutines - Overview

- **Concept, not library or OS feature**
  - “User-space” threads
  - Cooperatively, manually scheduled
- **Lots of confusing terminology**
  - Stackful Coroutines / Fibers
  - Stackless: Mechanically like callbacks
- **Library & Language implementations**
  - C#, EcmaScript 7, Python 3.5, Boost::Coroutine
  - In progress for ISO C++ TS (Not C++17)

Coroutines

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        (auto &yield) mutable {
    for (;;) {
      auto conn = sock.accept(yield);

      spawn([&t, conn=move(conn)]
            (auto &yield) mutable {
        while (auto req = conn.recv(yield)) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  loop();
}
```

- **Similar to Threads**
- **Coroutine to accept new connections**
- **Coroutine to receive messages for each connection**
- **Event loop**
  - Switches coroutines
  - On socket readiness

Coroutines – Compare to Threading

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        (auto &yield) mutable {
    for (;;) {
      auto conn = sock.accept(yield);

      spawn([&t, conn=move(conn)]
            (auto &yield) mutable {
        while (auto req = conn.recv(yield)) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  loop();
}
```

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        () mutable {
    for (;;) {
      auto conn = sock.accept();

      spawn([&t, conn=move(conn)]
            () mutable {
        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  pause();
}
```

Coroutines – Compare to Blocking

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        (auto &yield) mutable {
    for (;;) {
      auto conn = sock.accept(yield);

      spawn([&t, conn=move(conn)]
            (auto &yield) mutable {
        while (auto req = conn.recv(yield)) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  loop();
}
```

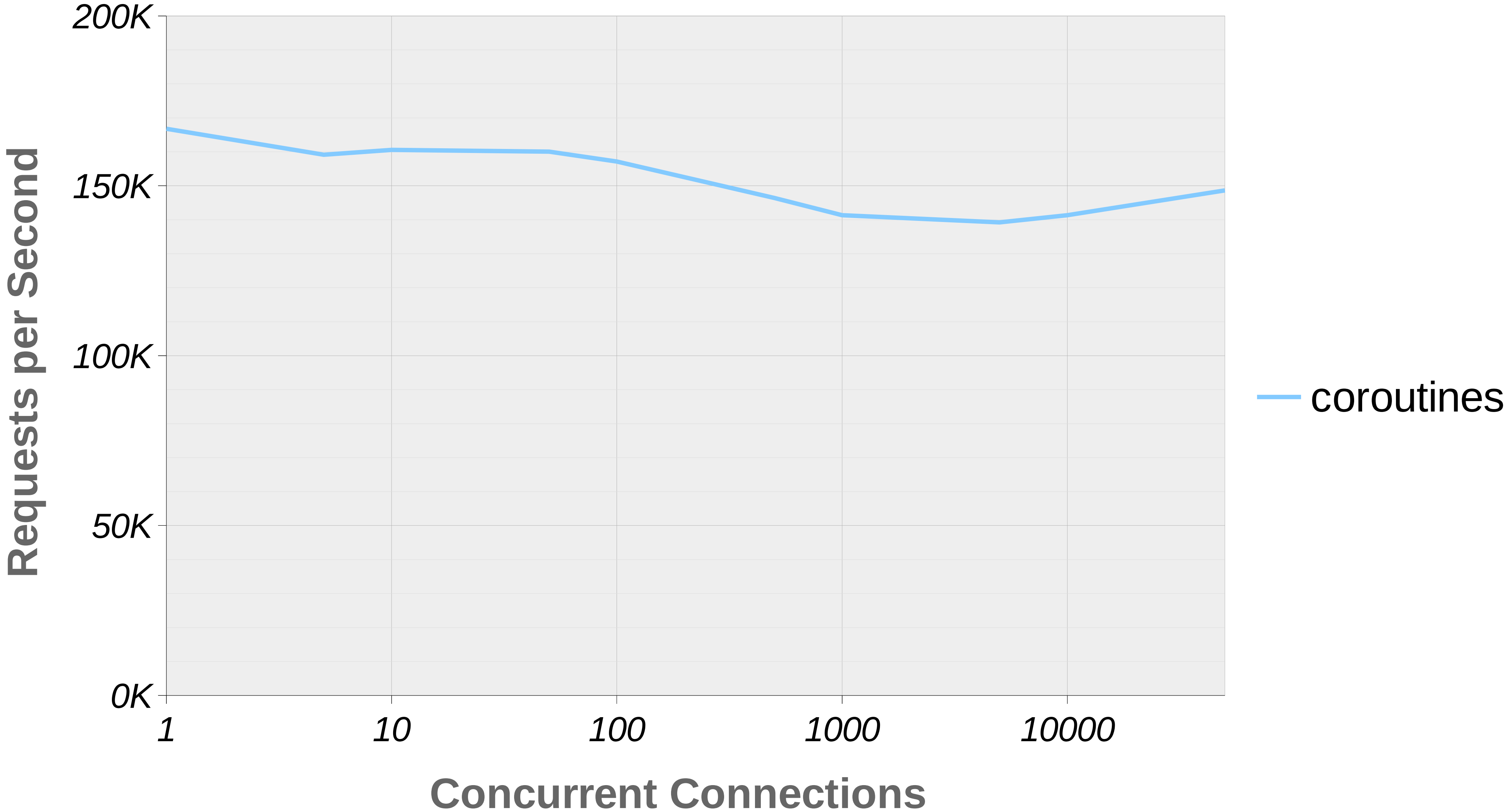
```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();



    for (;;) {
      auto conn = sock.accept();




        while (auto req = conn.recv()) {
          auto result = t.lookup(*req);
          conn.send(result);
        }

     }



}
```

Coroutines – Performance

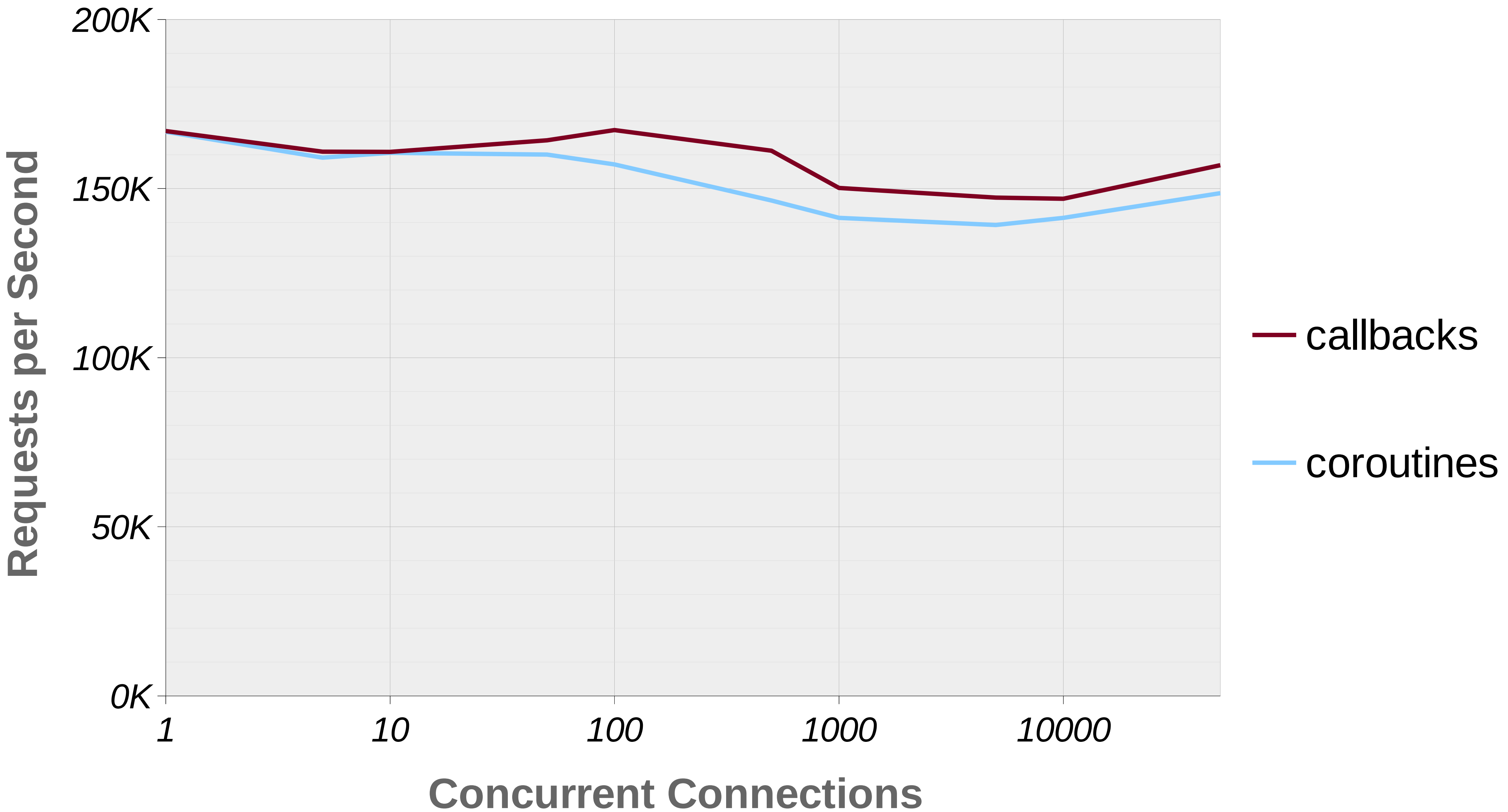
```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        (auto &yield) mutable {
    for (;;) {
      auto conn = sock.accept(yield);

      spawn([&t, conn=move(conn)]
            (auto &yield) mutable {
        while (auto req = conn.recv(yield)) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  loop();
}
```

Concurrent Connections	Execution Time (ms)	K-requests per second
1	2,997	166.8
5	3,142	159.1
10	3,114	160.6
50	3,124	160.1
100	3,182	157.2
500	3,414	146.5
1,000	3,537	141.4
5,000	3,591	139.3
10,000	3,538	141.3
50,000	3,364	148.6

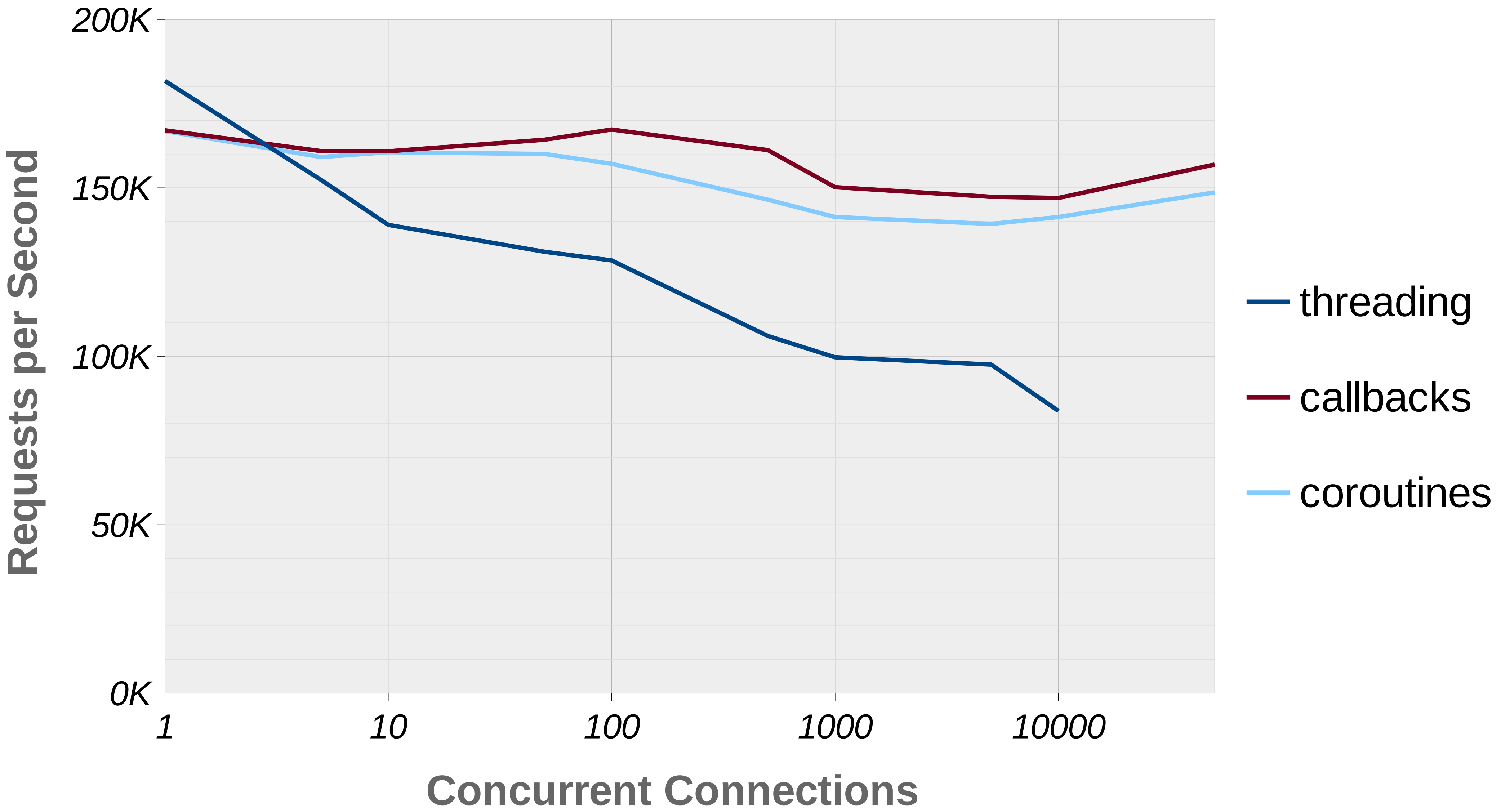
Coroutines - Performance



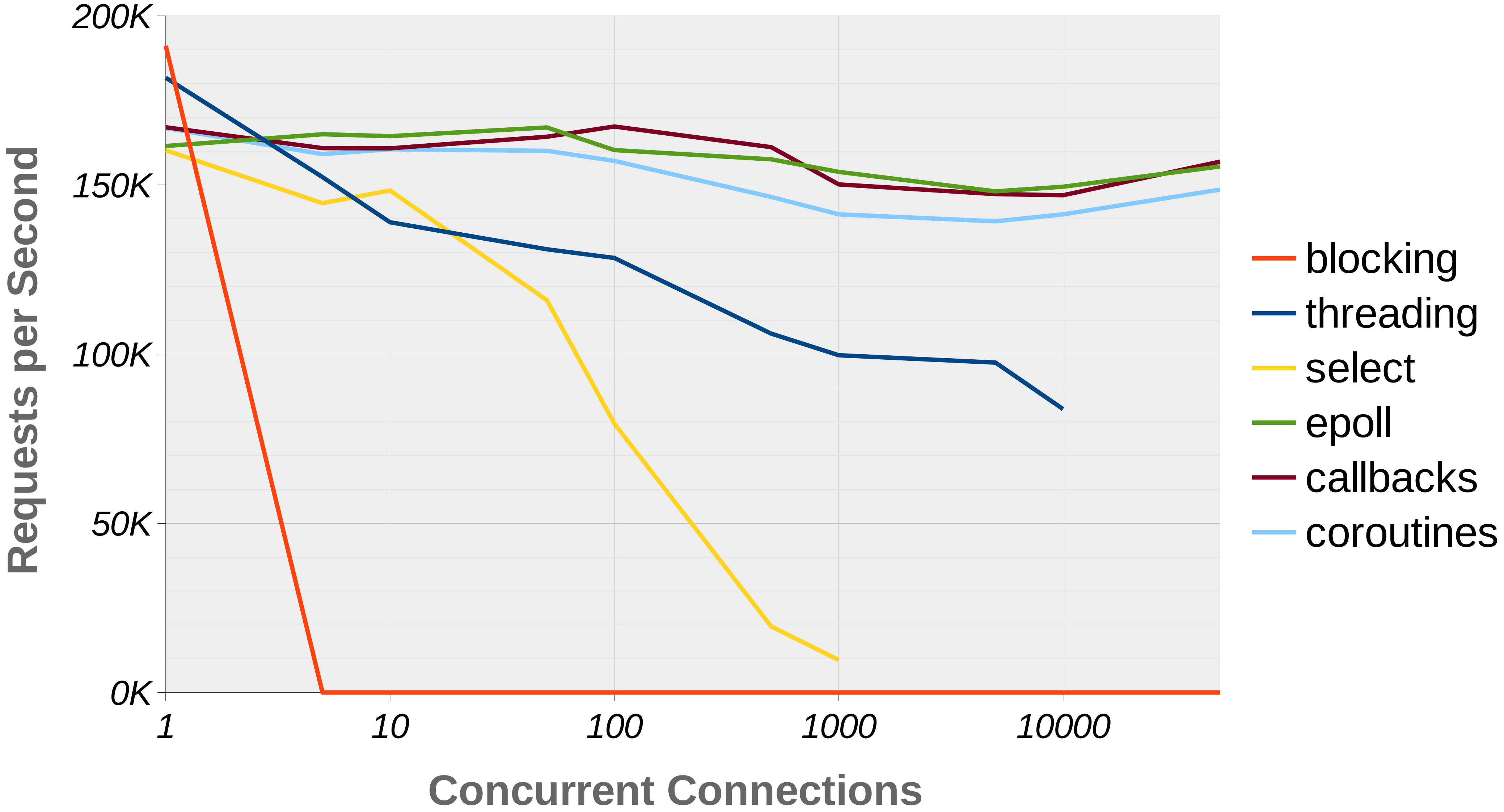
Coroutines - Performance



Coroutines - Performance



Coroutines - Performance



Coroutines – Good?

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        (auto &yield) mutable {
    for (;;) {
      auto conn = sock.accept(yield);

      spawn([&t, conn=move(conn)]
            (auto &yield) mutable {
        while (auto req = conn.recv(yield)) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  loop();
}
```

- ✔ **Advantages of threading**
  - ✔ Concurrency
  - ✔ Readability w.r.t. blocking
  - ✔ Flexibility
- ✔ **Advantages of callbacks**
  - ✔ Performance scales better than threads (#connections)
  - ✔ Avoiding race conditions
- ✔ **State management hugely simplified by having stacks**

Coroutines – Bad?

```
int main()
{
  const auto t = table();
  auto sock = bind_local("socket");
  sock.listen();

  spawn([&t, sock=move(sock)]
        (auto &yield) mutable {
    for (;;) {
      auto conn = sock.accept(yield);

      spawn([&t, conn=move(conn)]
            (auto &yield) mutable {
        while (auto req = conn.recv(yield)) {
          auto result = t.lookup(*req);
          conn.send(result);
        }
      });
    }
  });

  loop();
}
```

- ✗ **Performance slightly behind of callbacks**
- ✗ **Mechanism of coroutines is scary and non-standard**
  - ✗ Involves complex saving & restoring register state
- ✗ **Debugger support is sparse**
  - ✗ How do I get a backtrace for all running coroutines? Like: *thread apply all bt*
- ✗ **Must propagate “yield”**
  - ✗ Require “resumable” version of every function

Summary

Summary - Oversimplification

	Performance	Testability	Complexity
Blocking	✗	✓✓	✓✓
Threading	✗	✗✗	✓✓
Select	✗✗	✗✗	✗✗
Epoll	✓✓	✗✗	✗✗
Callbacks	✓✓	✓	✗
Coroutines	✓	✓	✓✓

Summary

- **Software with I/O is challenging**
  - Especially doing it efficiently
  - Most software needs to do I/O
- **This talk only scratches the surface**
  - Lots of good material on this topic
    - Boost documentation (Coroutine, Fiber, ASIO)
    - Coroutines C++ standardisation proposal papers
  - Passed over disk I/O; conceptually similar

Summary – Advice?

- **Keep it simple**
  - Choose a model and use it consistently
- **Use an established library**
- **Callbacks are... OK**
  - Be careful of object lifetime
- **Coroutines are looking promising**
  - Be aware of similarity to other mechanisms
  - Potentially long wait for standardisation

Thanks

steve@stackhpc.com

Bonus Content

- **Green Threads**
  - Threads scheduled in userspace, like Fibers
  - Try to be transparent, look like real threads
  - More common in VM languages (e.g. Java)
- **“NxM”**
  - ***M*** coroutines or tasks
  - Scheduled onto ***N*** real OS threads
  - Improve performance using multiple CPU cores