Towards *Collaborative Performance Tuning* of Algorithmic Skeletons

Chris Cummins

THE UNIVERSITY of EDINBURGH informatics

EPSRC Centre for Doctoral Training in Pervasive Parallelism

EPSRC Engineering and Physical Sciences Research Council

http://chriscummins.cc

Uncontentious statement:

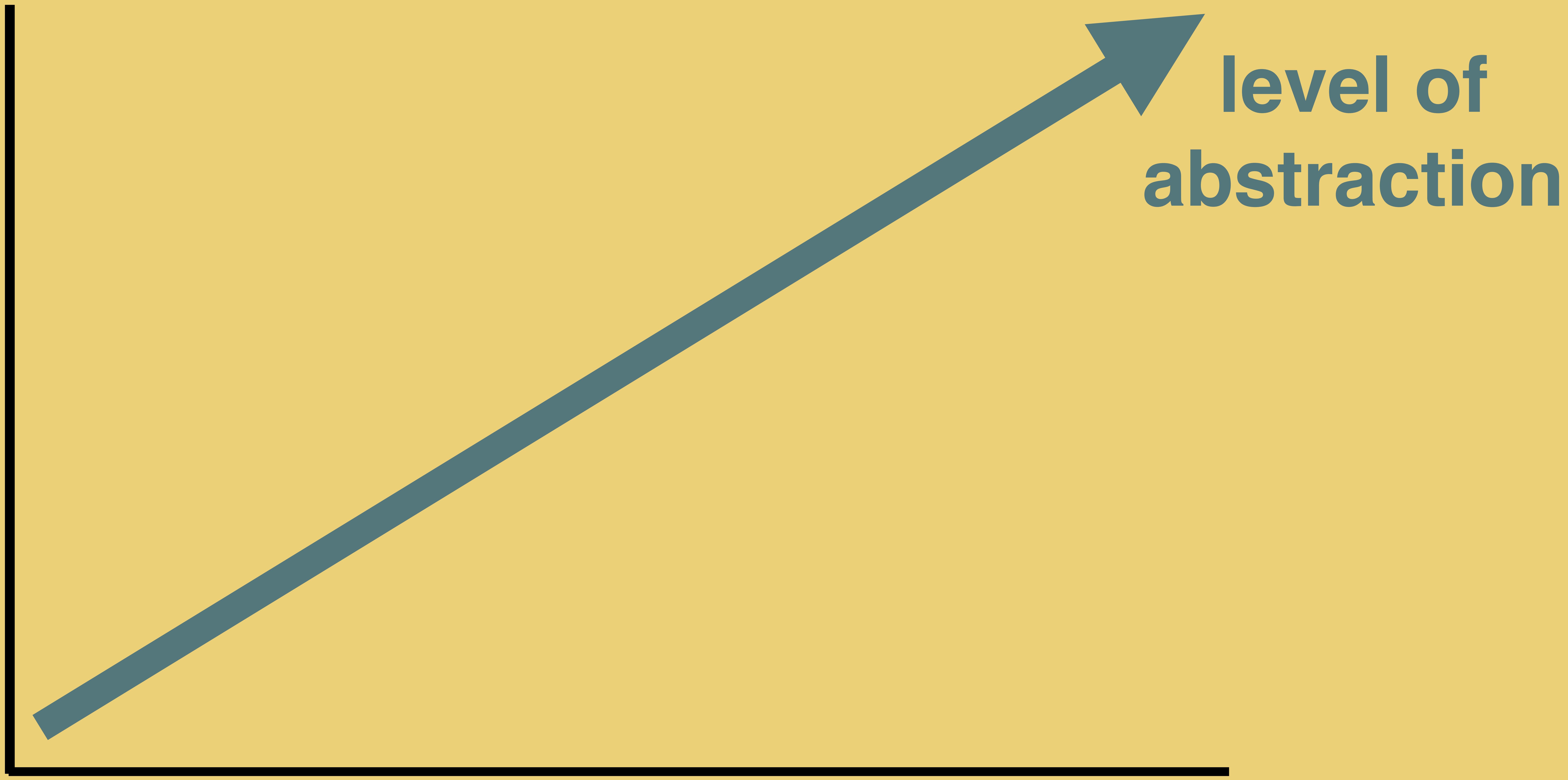
High level programming is great!

So why do application programmers resort to writing *low level* code?

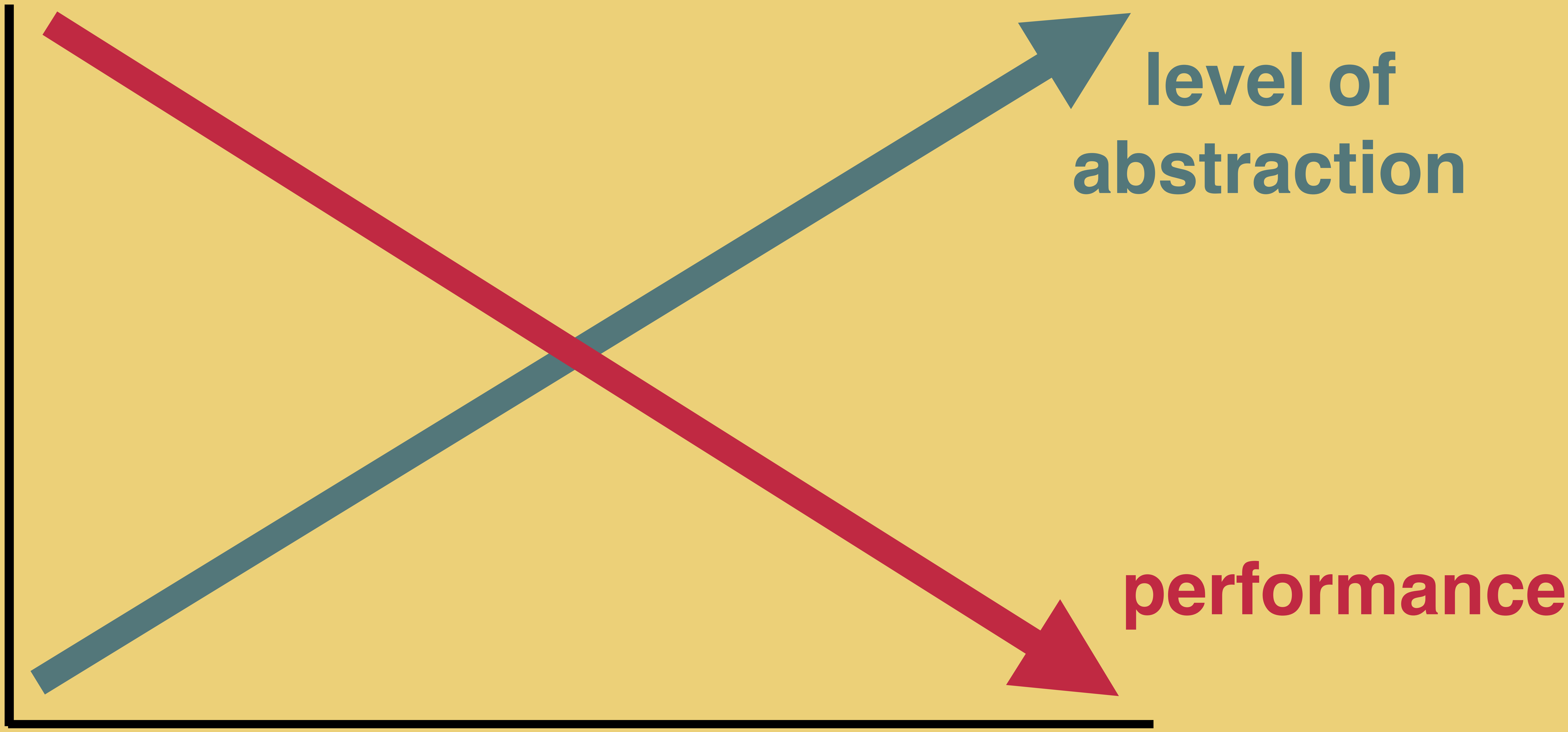
So why do application programmers resort to writing *low level* code?

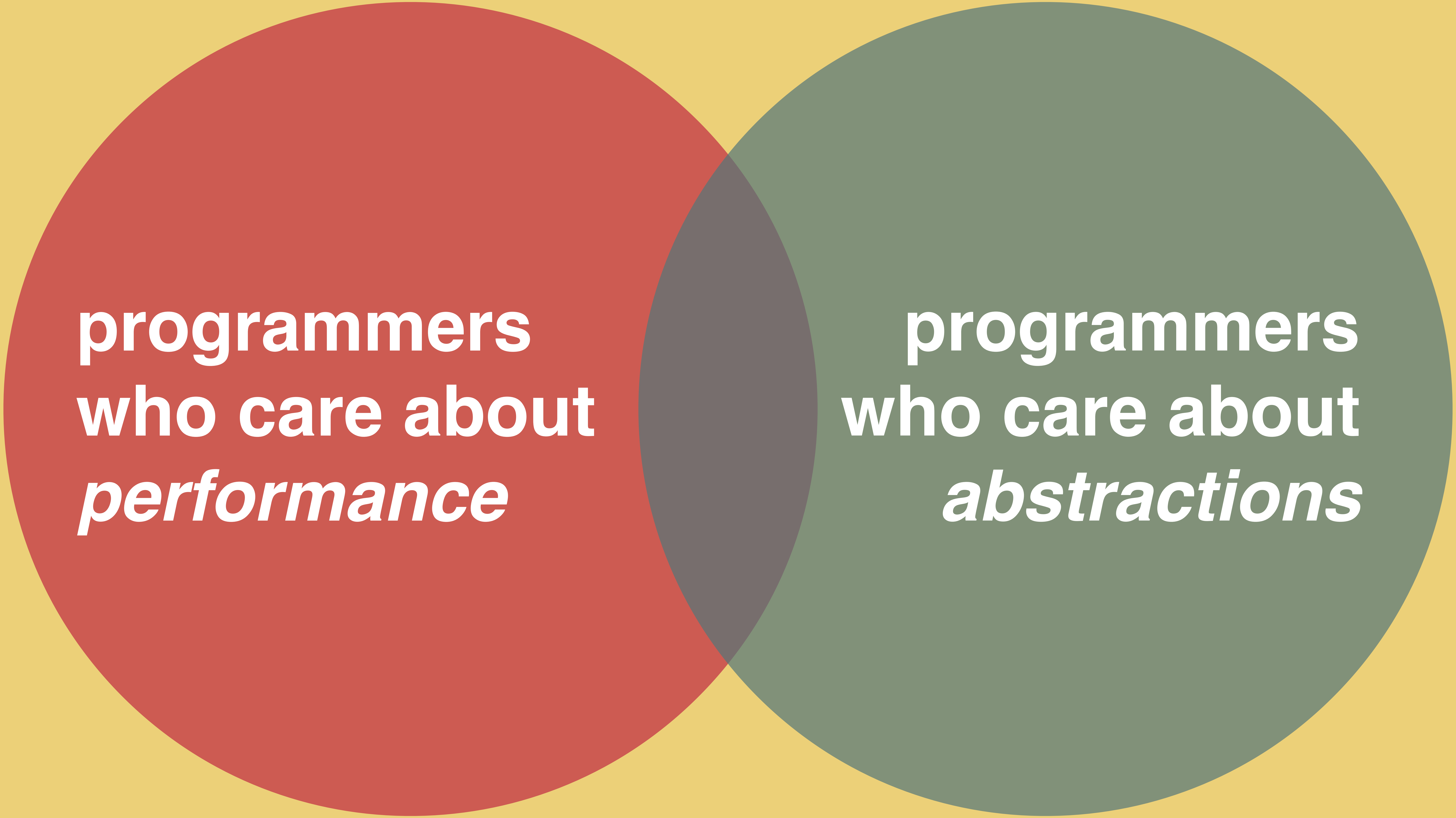
Performance!

common perception is …

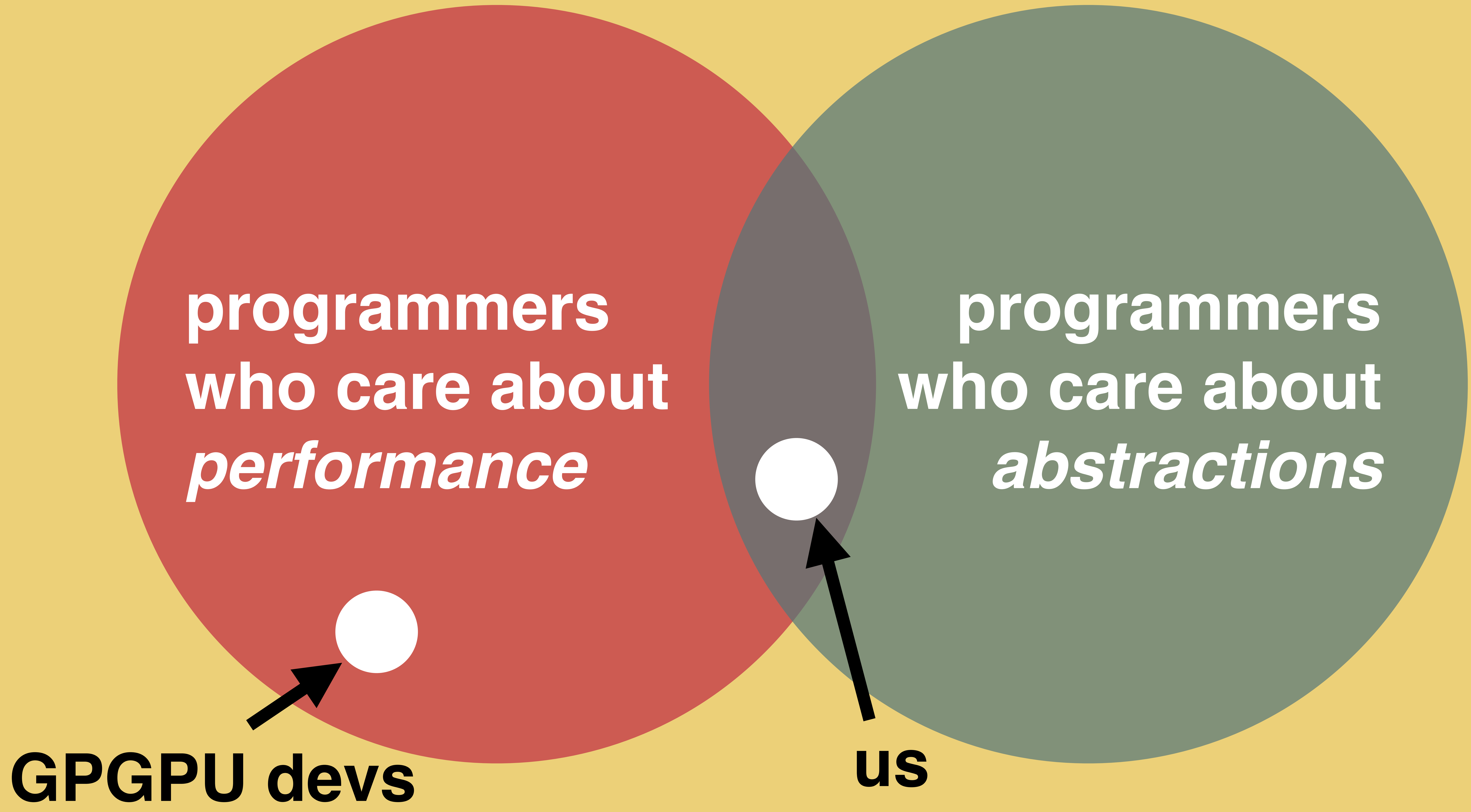


common perception is …




programmers
who care about
performance
programmers
who care about
abstractions


programmers
who care about
performance
programmers
who care about
abstractions
GPGPU devs


programmers
who care about
performance
programmers
who care about
abstractions
GPGPU devs
us

How do we break the illusion?

High level code needs to be at least *competitive* with low level

High level code needs to be at least *competitive* with low level (but faster would be nice)

Reasons for low level:

Reasons for low level: Domain-specific optimisations

Reasons for low level:
Domain-specific optimisations

Parameter tuning

Reasons for low level: Domain-specific optimisations

Parameter tuning

Parameter tuning for Algorithmic Skeletons

Parameter tuning for ~~Algorithmic~~ stencil ~~Skeletons~~ skeletons

OpenCL workgroup size

~~Parameter~~ tuning for ~~Algorithmic~~ stencil ~~Skeletons~~ skeletons

OpenCL workgroup size:

OpenCL workgroup size:

Controls decomposition of threads.

OpenCL workgroup size:

Controls decomposition of threads.

Is a 2D parameter (rows x cols).

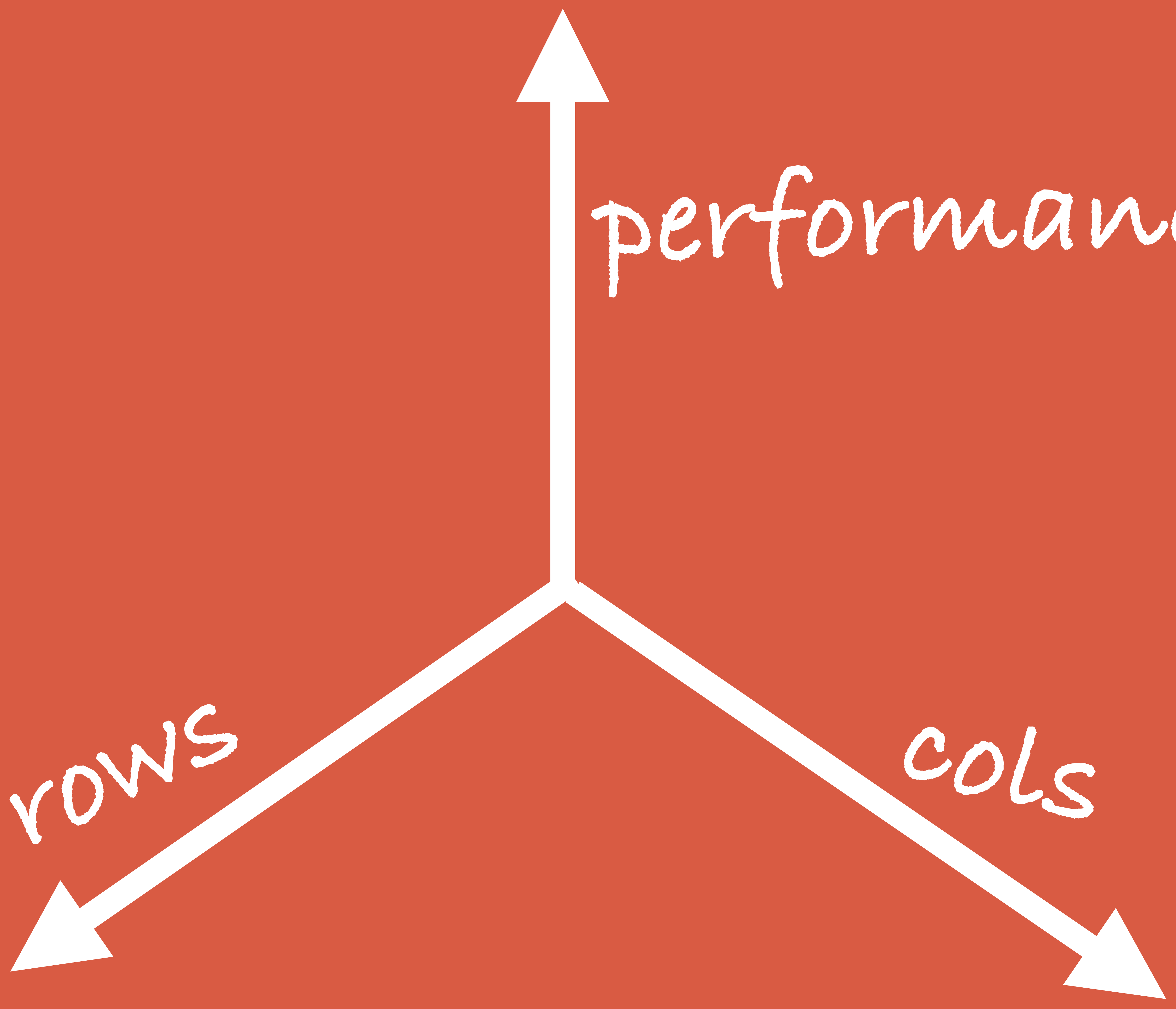
OpenCL workgroup size:

Controls decomposition of threads.

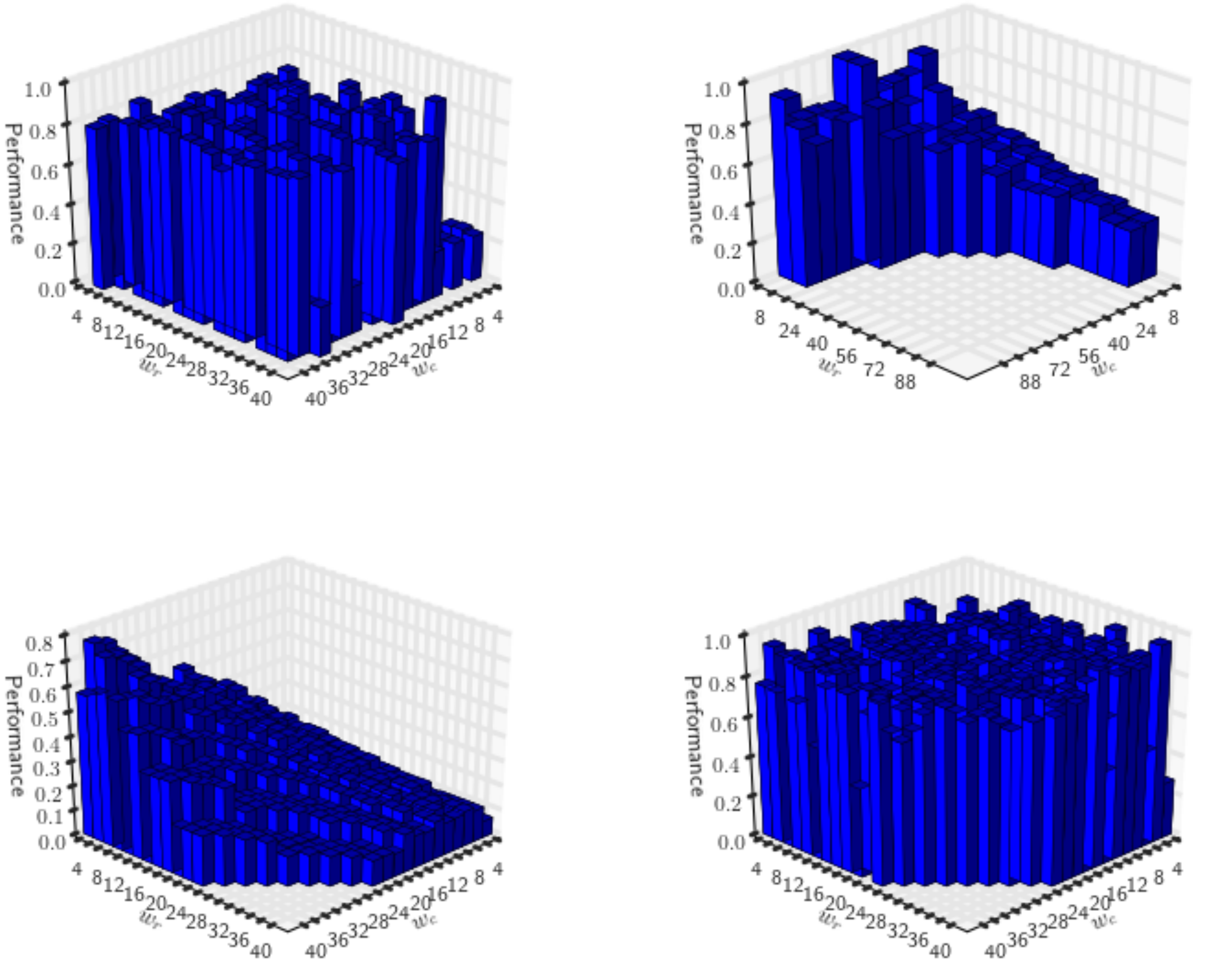
Is a 2D parameter (rows x cols).

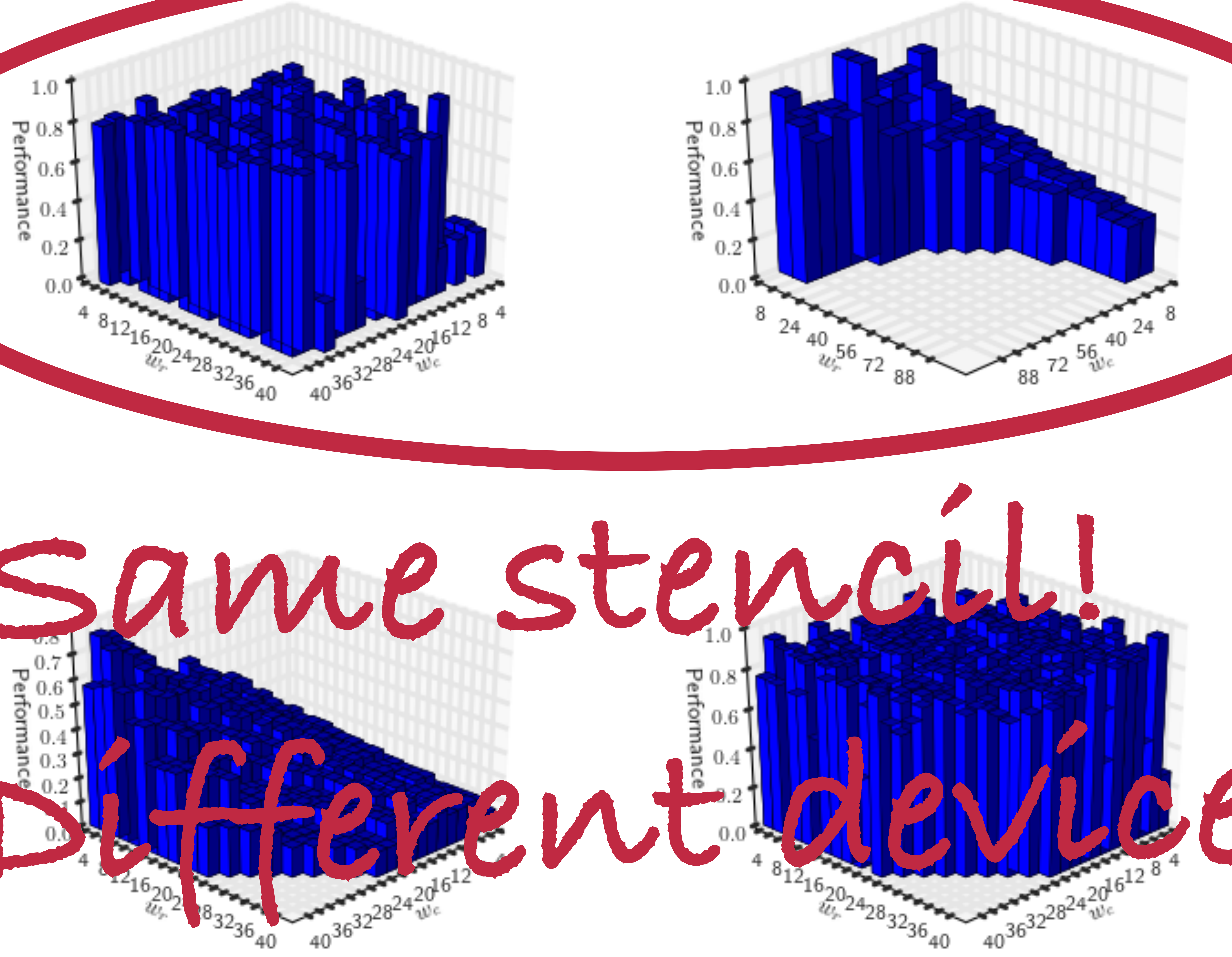
Critical to performance.

OpenCL workgroup size:



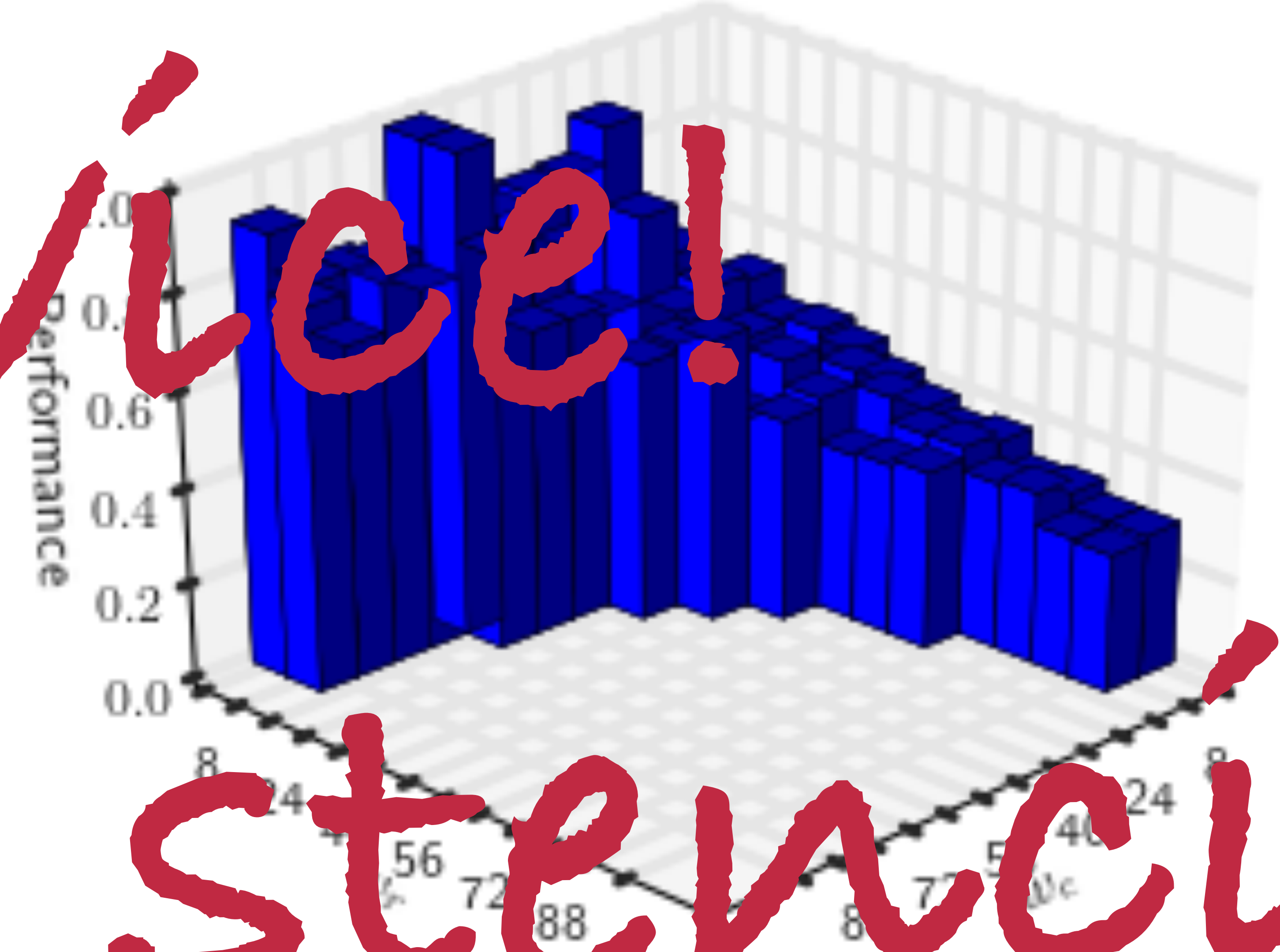
Examples

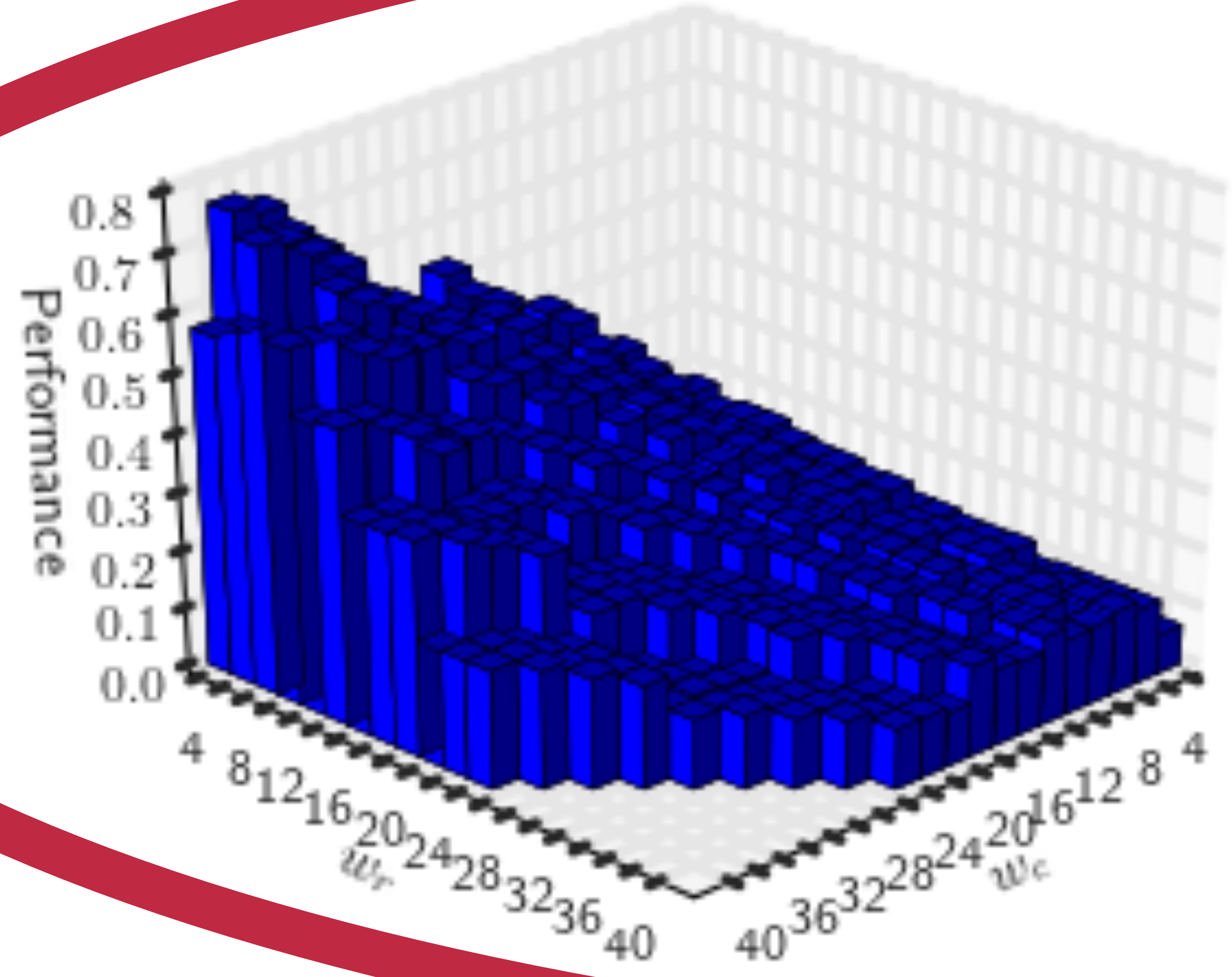

Performance
w_r
w_c
Performance
w_r
w_c
Performance
w_r
w_c
Performance
w_r
w_c

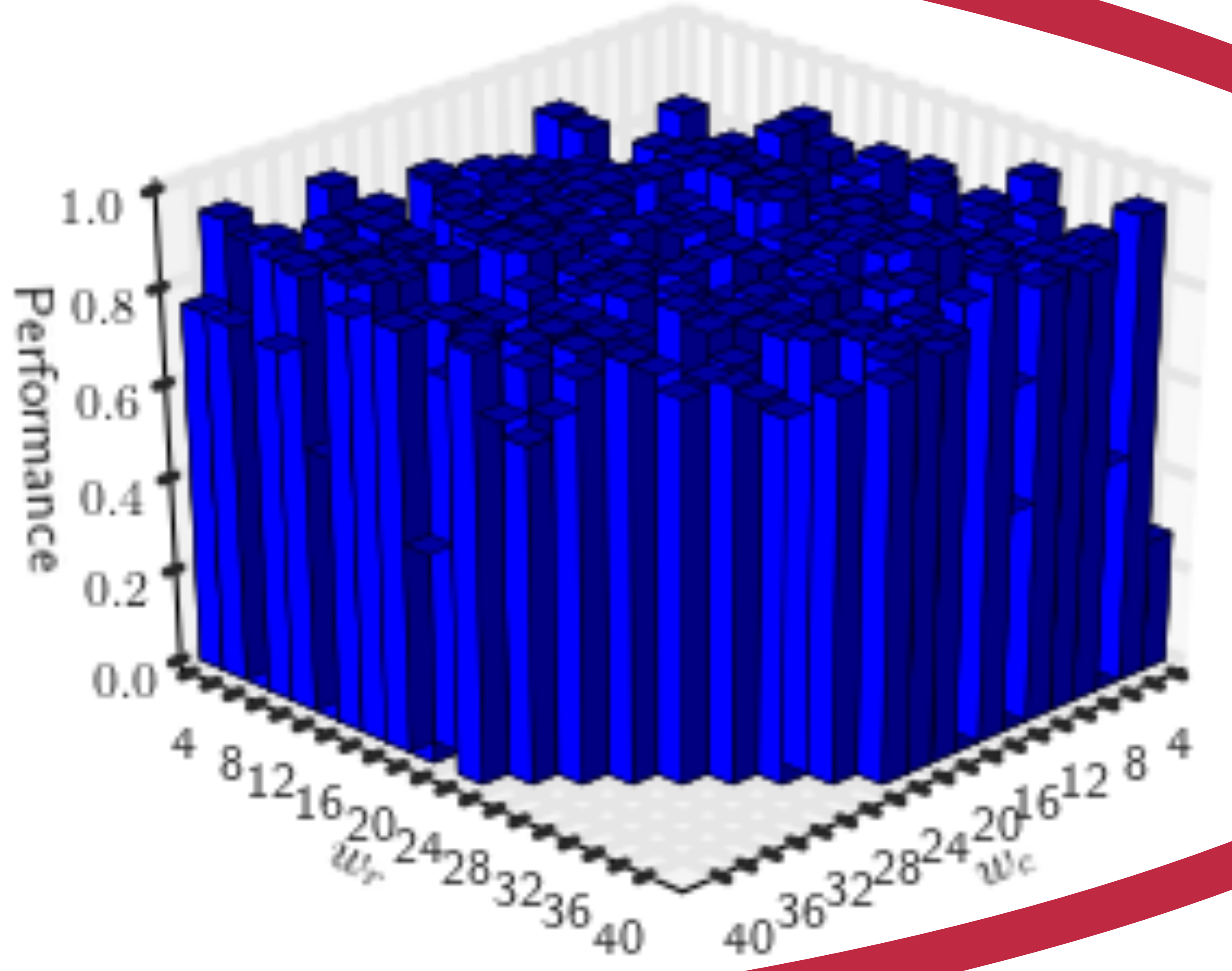


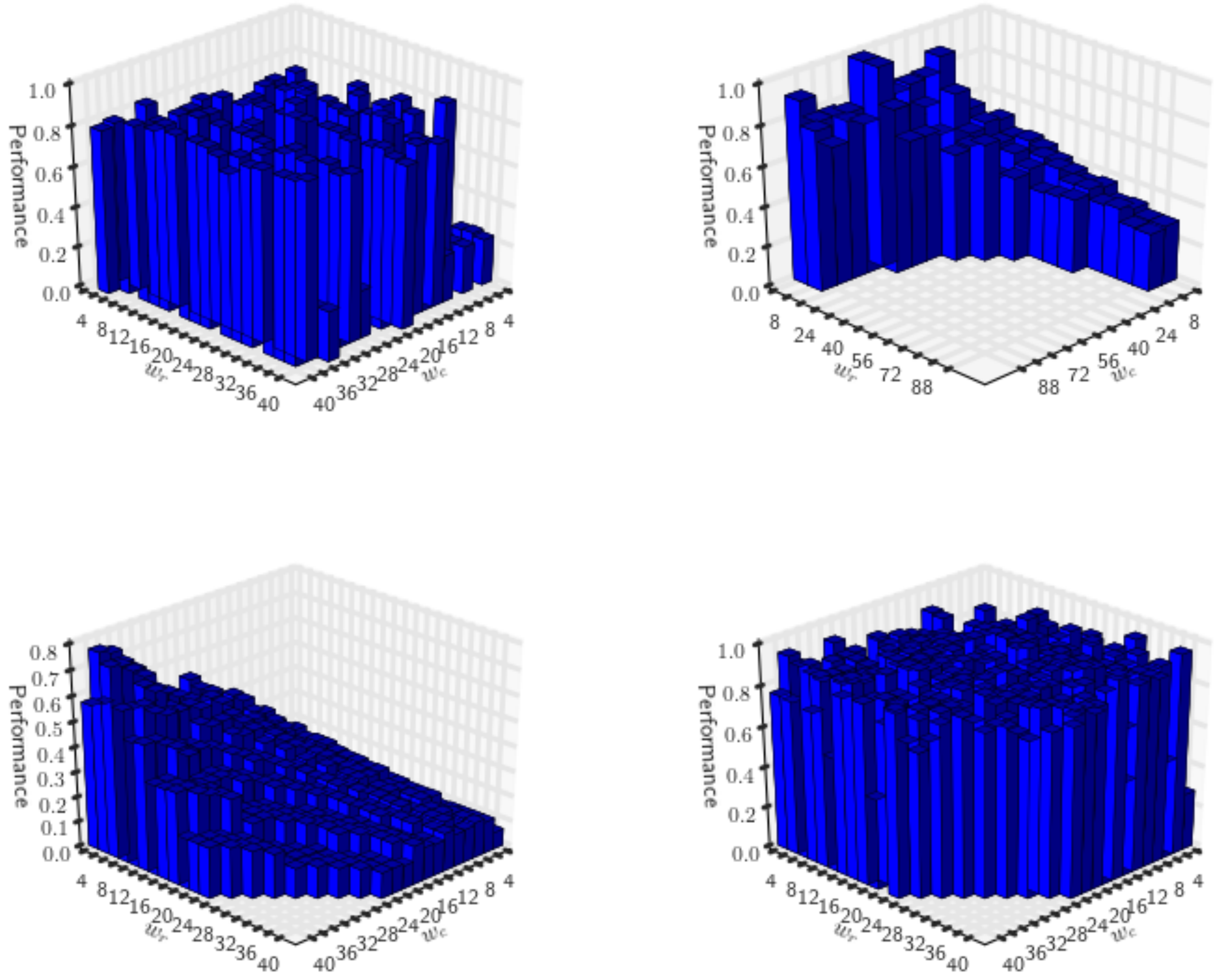
Same stencil!
Different device!

Same device!
Different stencil!









Performance
1.0
0.8
0.6
0.4
0.2
0.0
w_r
4 8 12 16 20 24 28 32 36 40
w_c
40 36 32 28 24 20 16 12 8 4
Performance
1.0
0.8
0.6
0.4
0.2
0.0
w_r
8 24 40 56 72 88
w_c
88 72 56 40 24 8
Performance
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
w_r
4 8 12 16 20 24 28 32 36 40
w_c
40 36 32 28 24 20 16 12 8 4
Performance
1.0
0.8
0.6
0.4
0.2
0.0
w_r
4 8 12 16 20 24 28 32 36 40
w_c
40 36 32 28 24 20 16 12 8 4

Choosing workgroup size depends on:

1. device
2. program
3. dataset

Let’s automate this!

Approach 1

Set a workgroup size
Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size
Execute and time program
Set a workgroup size
Execute and time program
Set a workgroup size
Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

… (continue until done / bored)

Pick the best one you tried

Set a workgroup size
Execute and time program
Set a workgroup size
Execute and time program
Set a workgroup size
Execute and time program
Set a workgroup size
Execute and time program
… (continue until done / bored)
Pick the best one you tried

(iterative compilation)

BAD!

Takes a looooong time

BAD!

Takes a loooong time

BAD!

Must be repeated for every new "x"

device

program

dataset

Approach 2

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

Set a workgroup size

Execute and time program

… (continue until done / bored)

Pick the best one you tried

Set a workgroup size
Execute and time program
Set a workgroup size
Execute and time program
Set a workgroup size
Execute and time program
Set a workgroup size
Execute and time program
… (continue until done / bored)
Pick the best one you tried

1 data point

Collect data points
Extract “features”
Train machine learning classifier

Extract “features”
Input to classifier

BETTER!

Can make *predictions* on unseen "x"

device

program

dataset

BETTER!

Can make *predictions* on unseen "x"

device

program

dataset

BETTER!

Still takes a loooong time

Can make *predictions* on unseen "x"

device

program

dataset

BETTER!

Still takes a loooong time

Requires a lot of code

Our wish list:

1. Reduce training costs
2. Reduce implementation costs
3. Minimise runtime overheads

Our Approach …

OmniTune

1. Allows *collaborative* performance tuning

Reduce training costs ✓

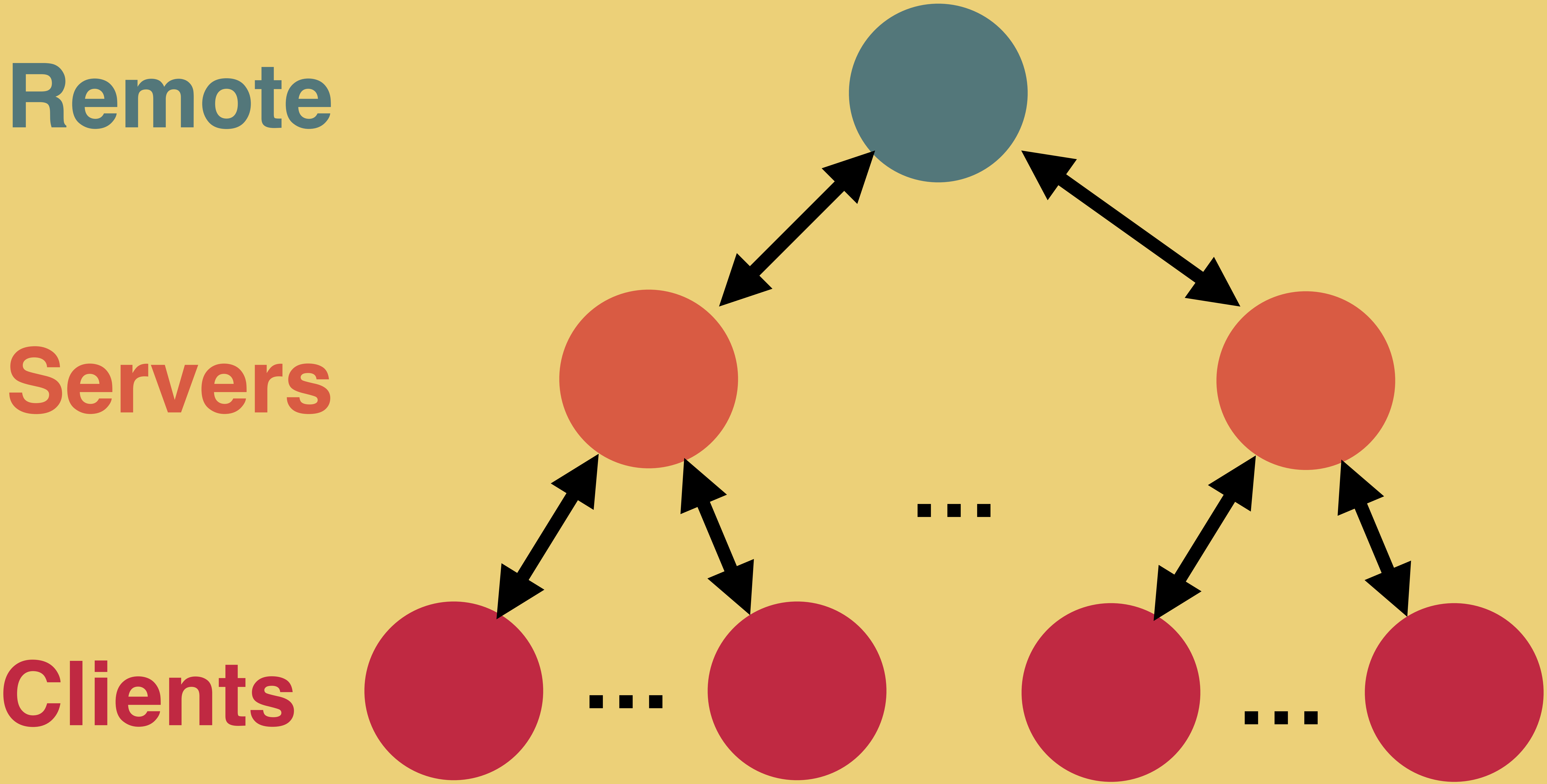
2. Provides re-usable *implementations*

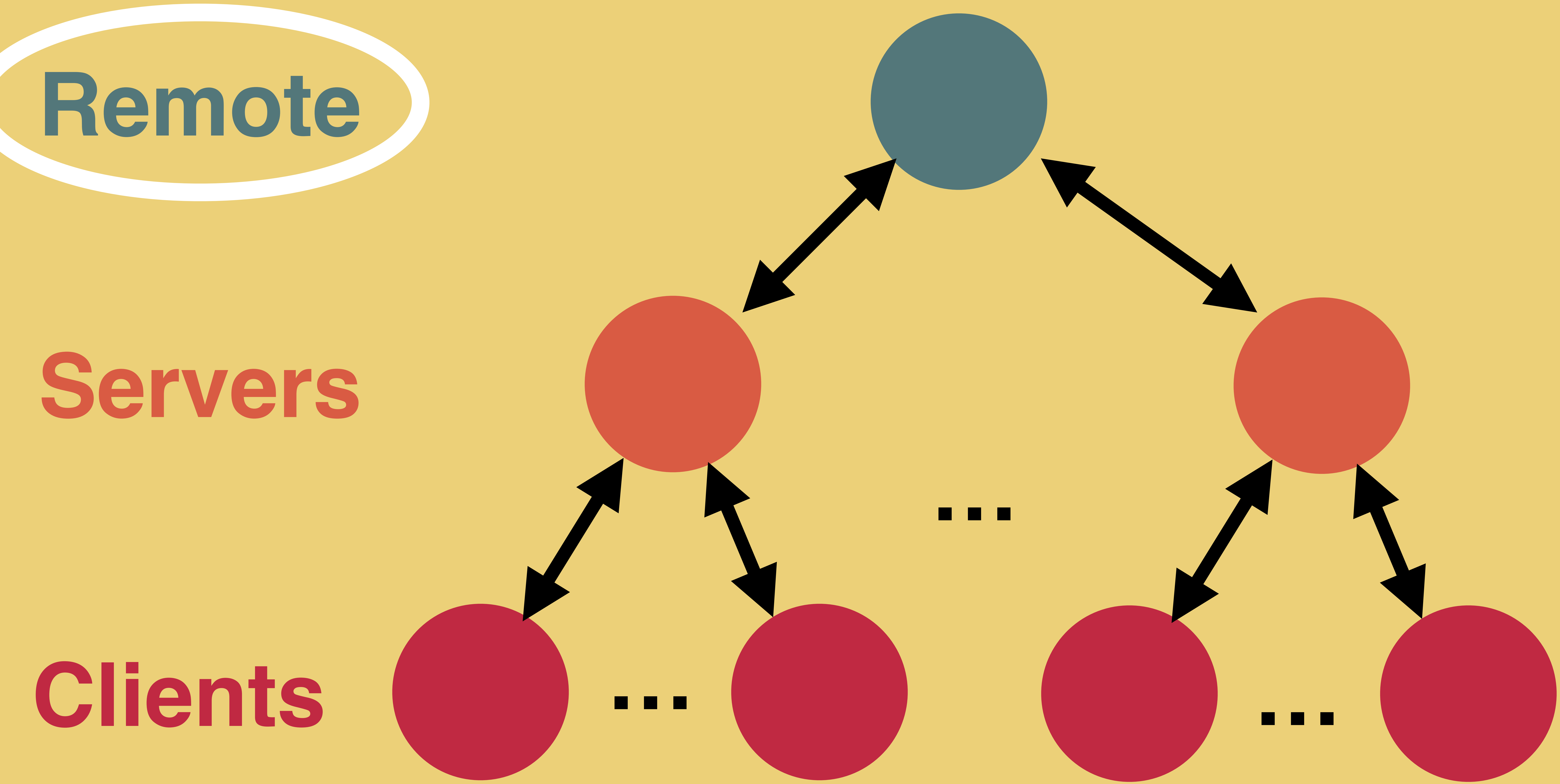
Reduce implementation costs ✓

3. Provides lightweight runtime interface

Minimise runtime overheads ✓

How does it work?

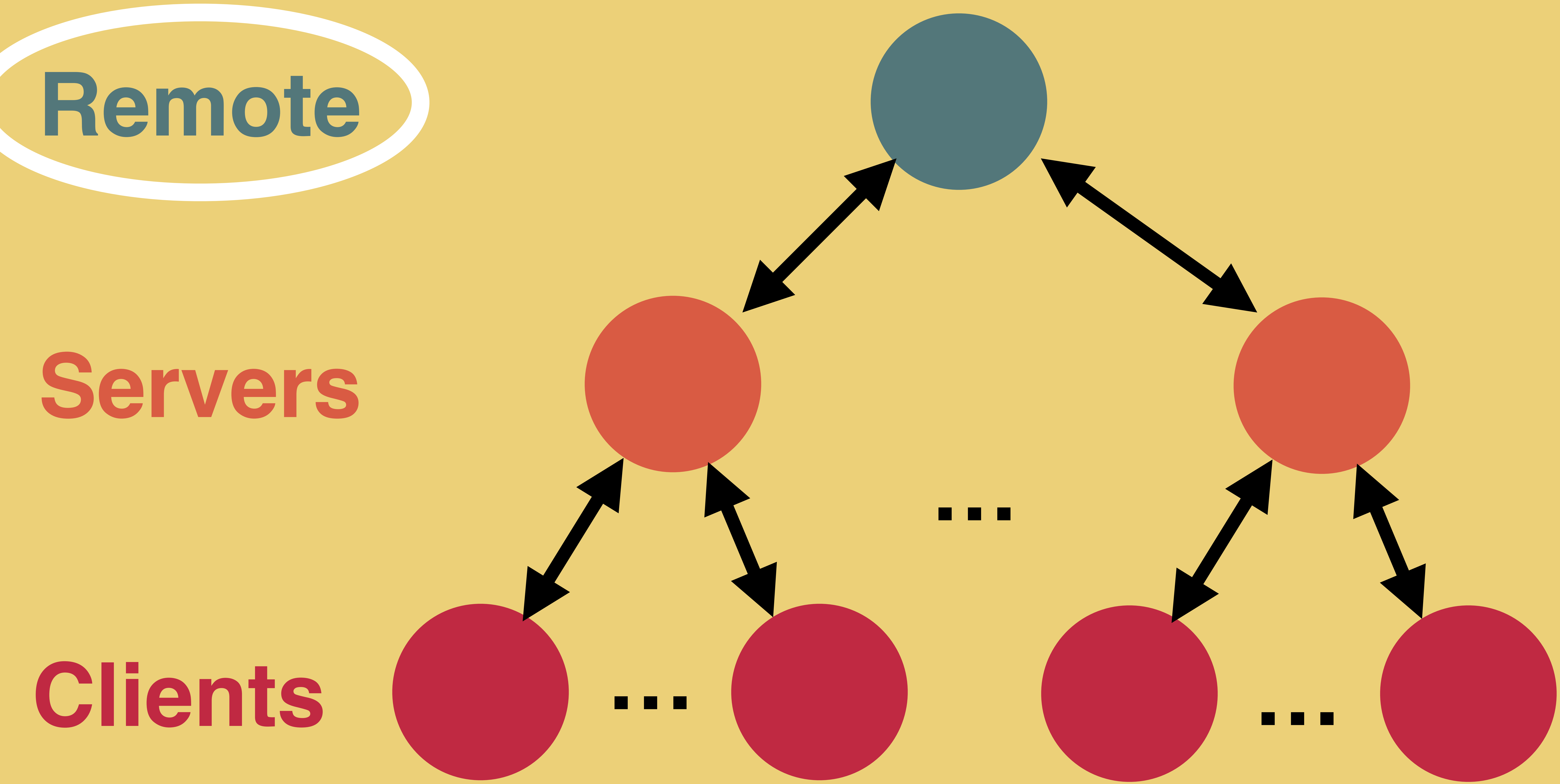

Remote
Servers
Clients
...
...
...

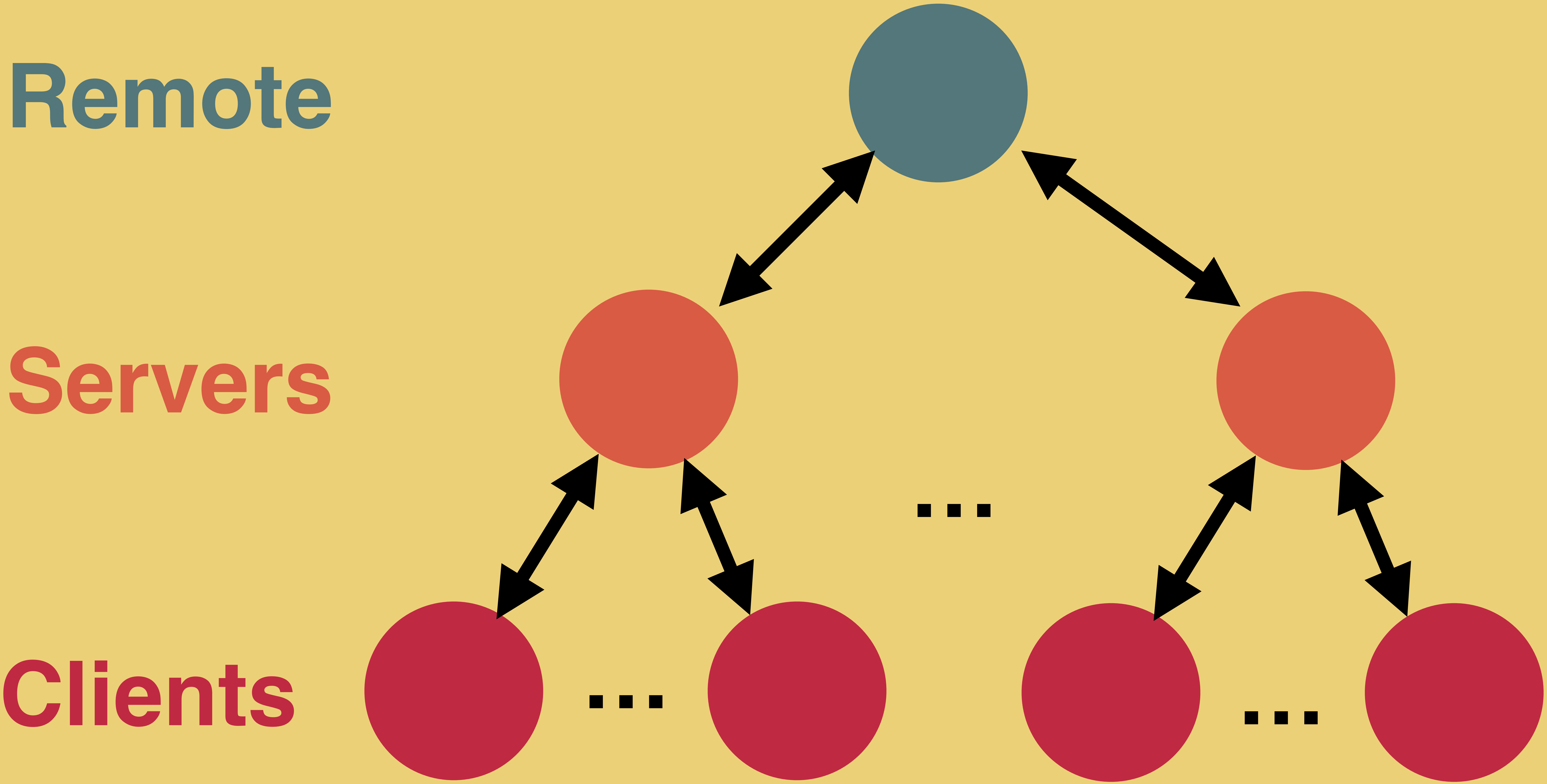

Remote
Servers
Clients
...
...
...

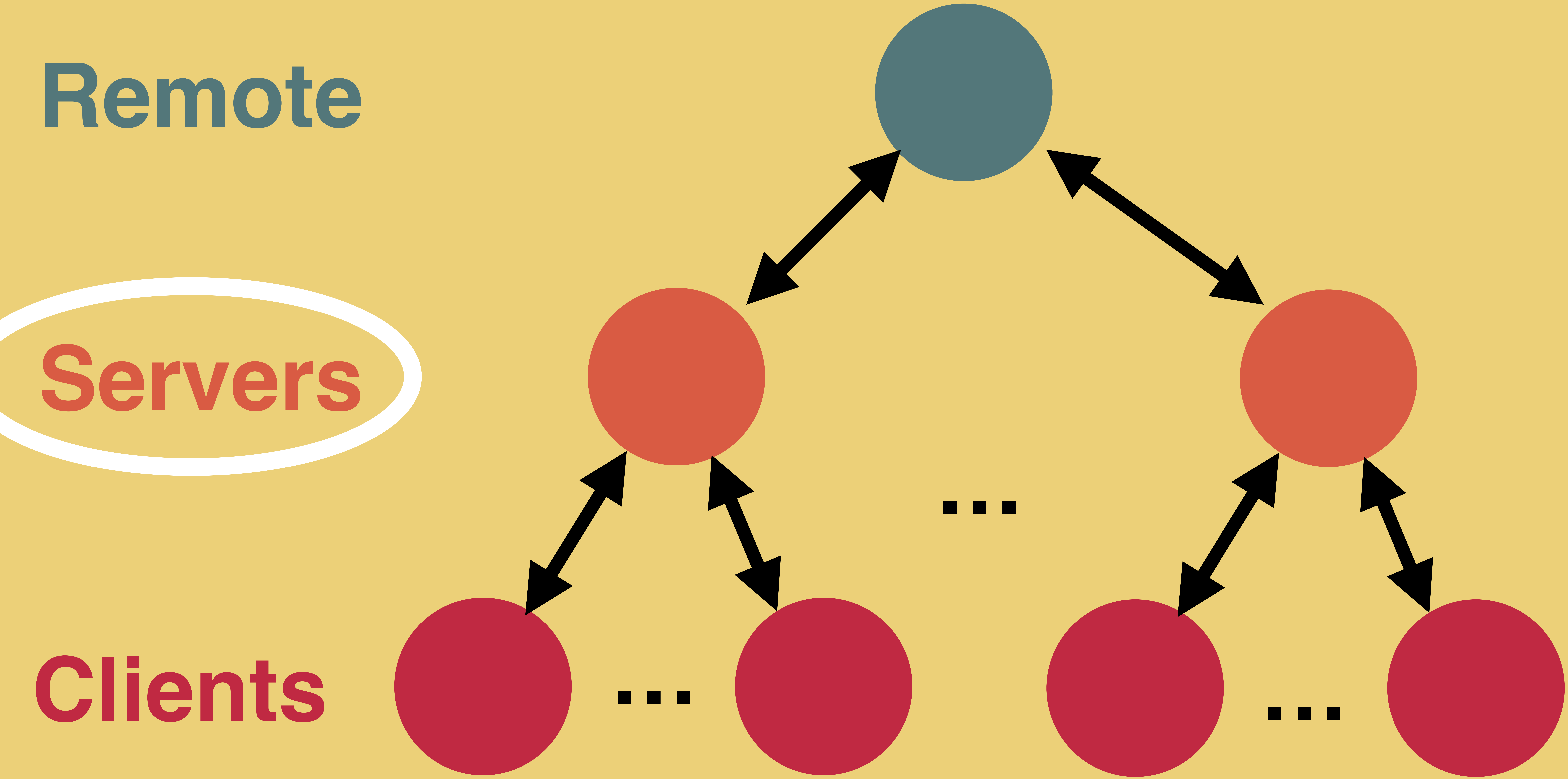
Remote

Book-keeper

Manages and stores training data


Remote
Servers
Clients
...
...
...

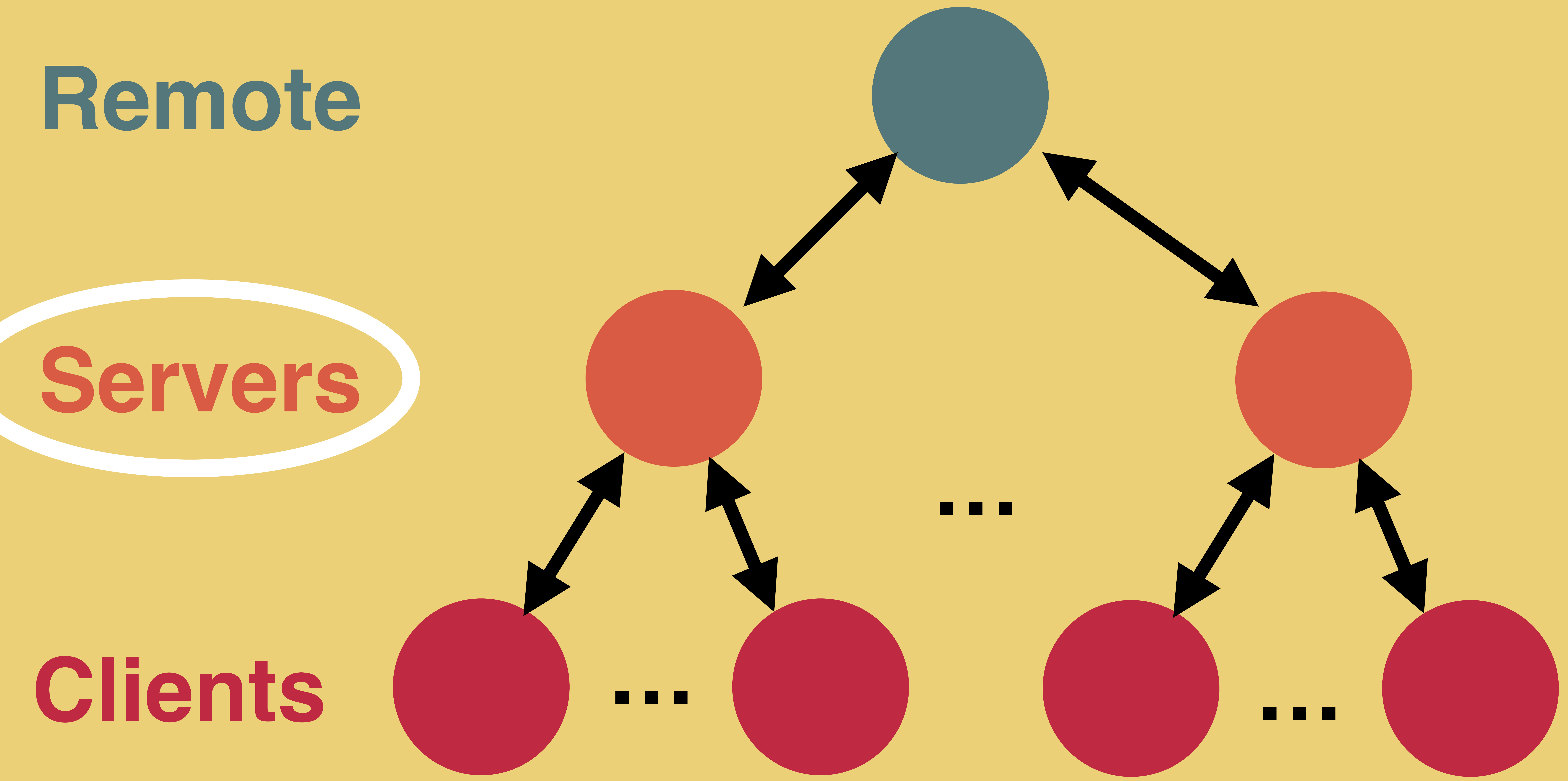

Remote
Servers
Clients
...
...
...

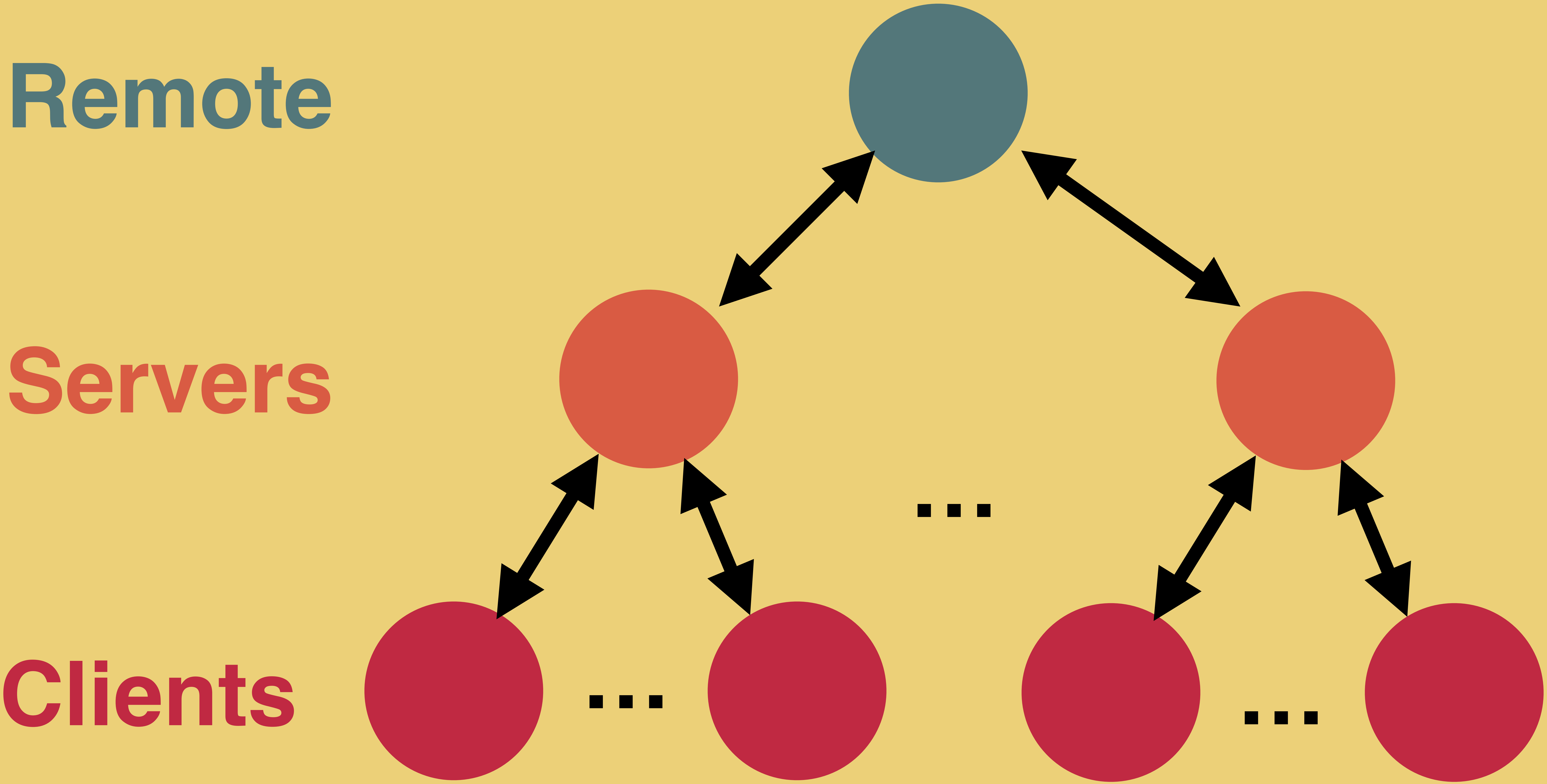

Remote
Servers
Clients
...
...
...

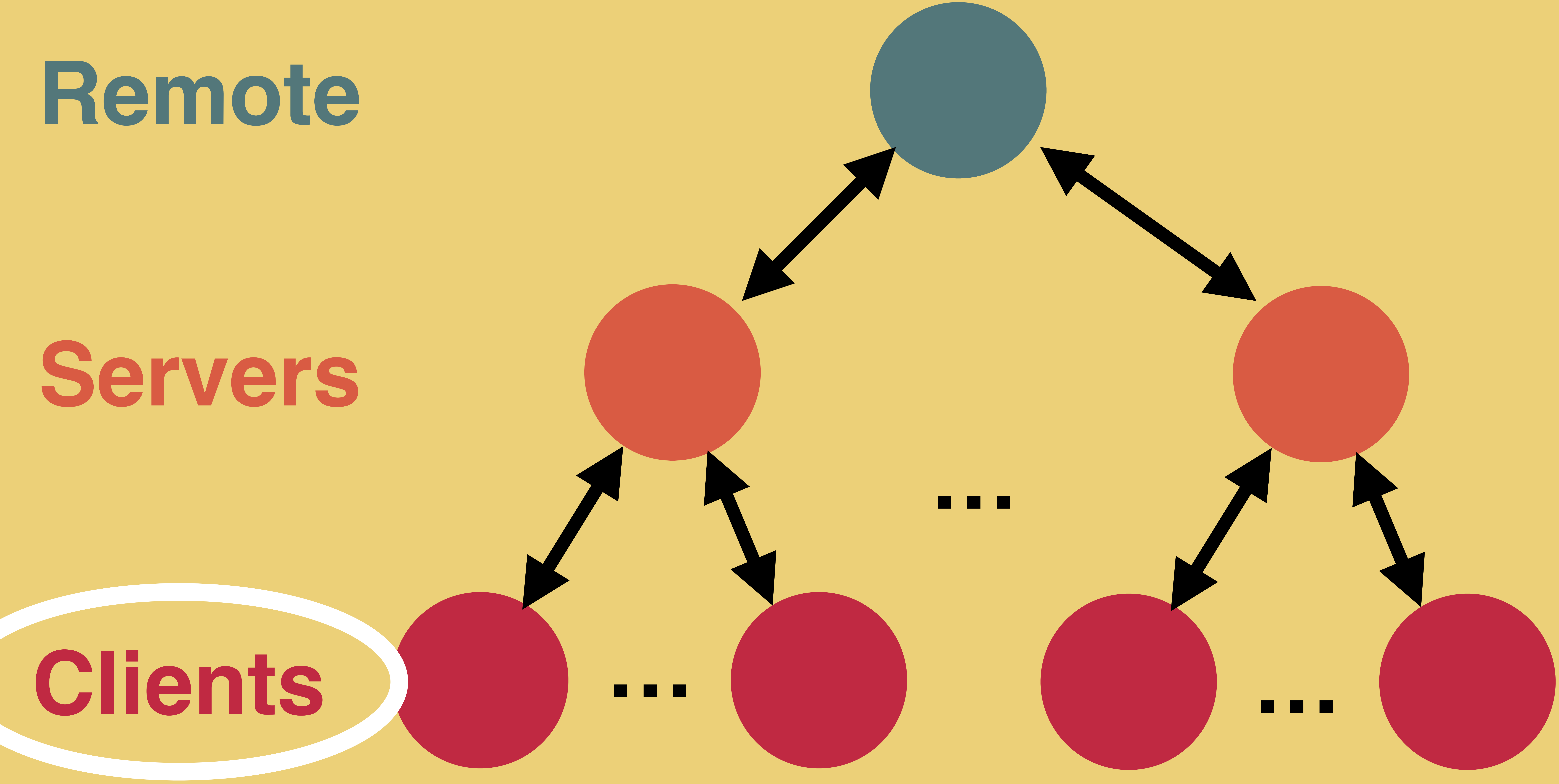
Servers

Autotuning engine

Performs machine learning


Remote
Servers
Clients
...
...
...

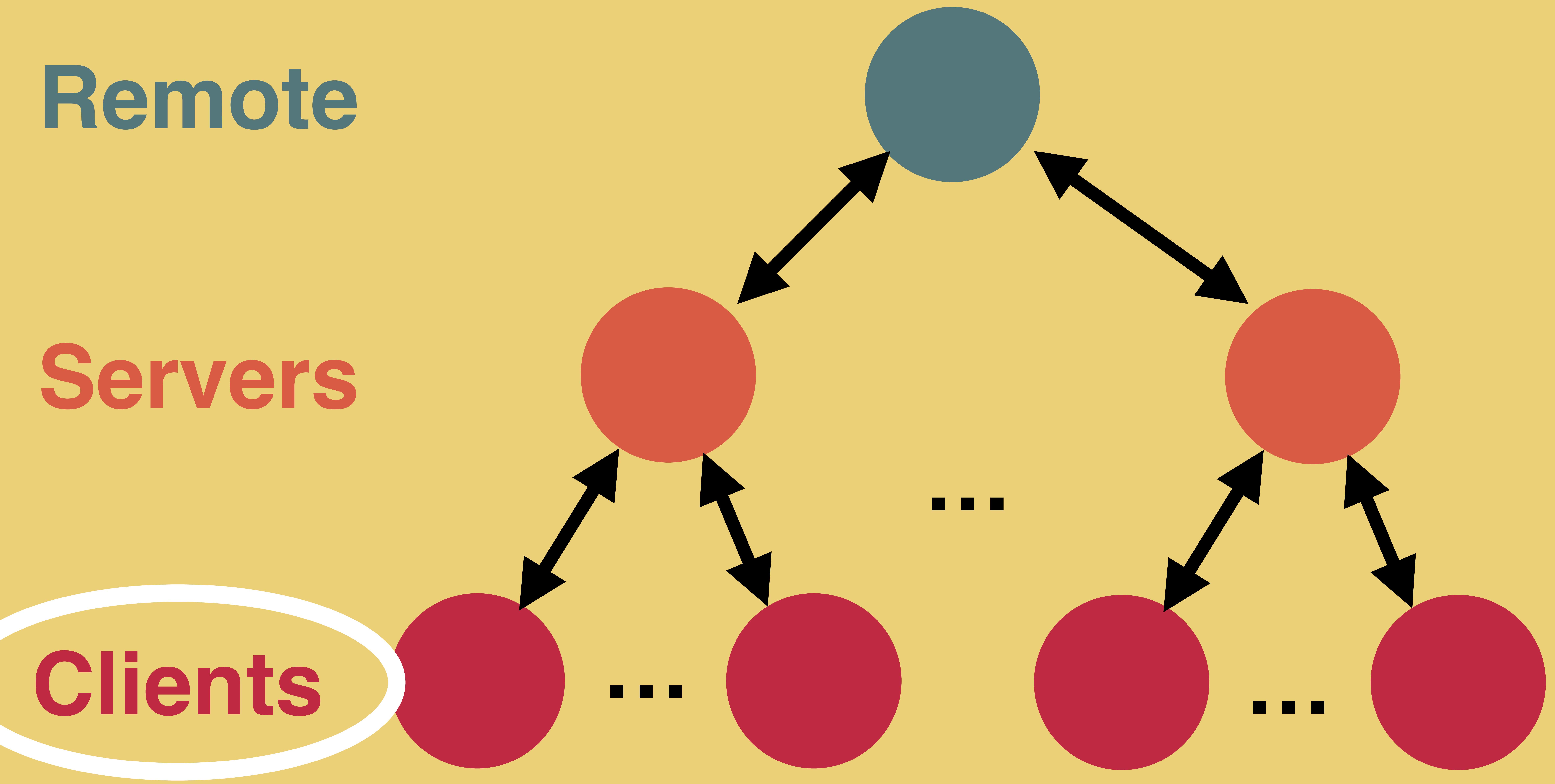

Remote
Servers
...
Clients
...
...

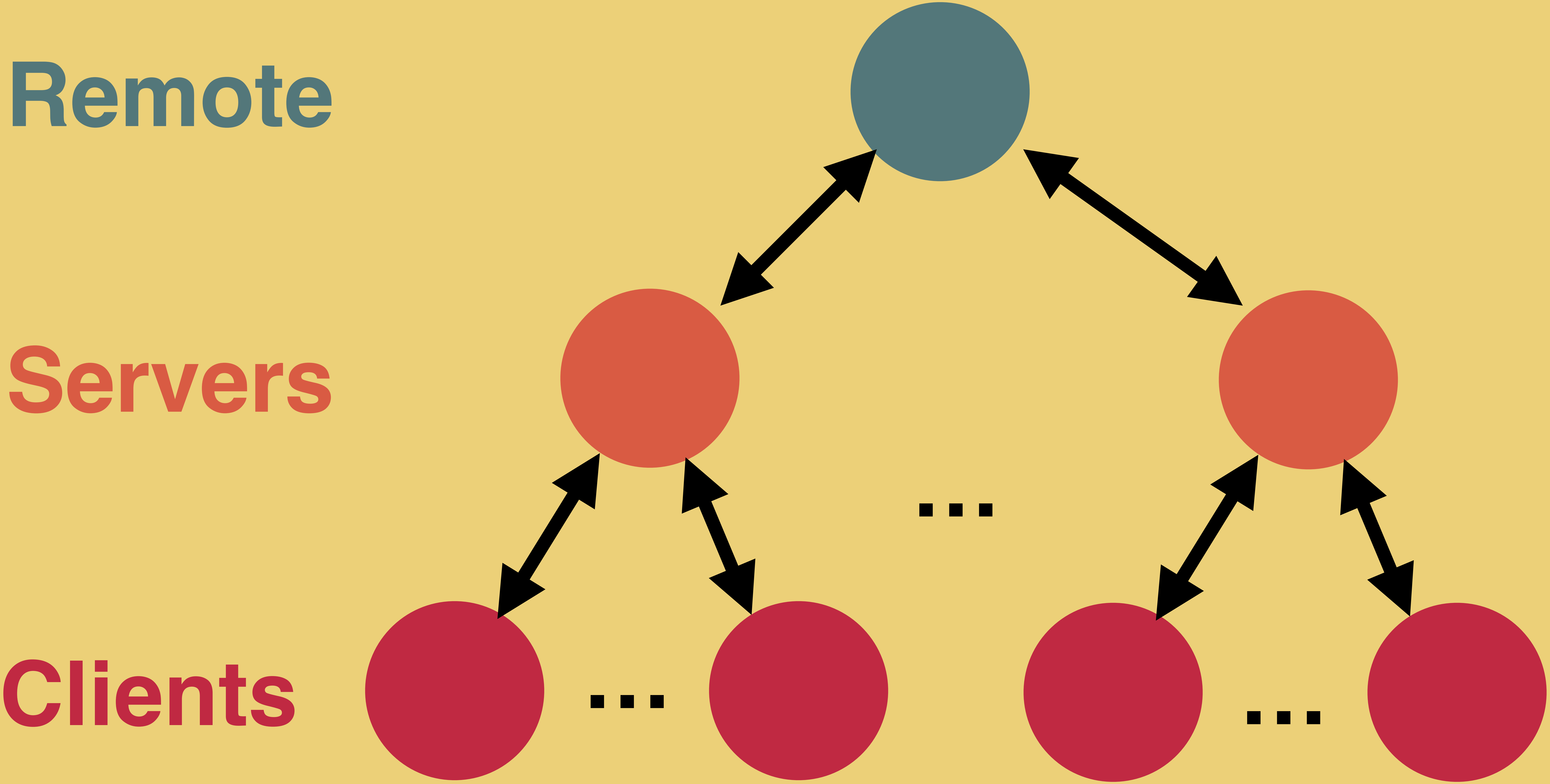

Remote
Servers
Clients
...
...
...

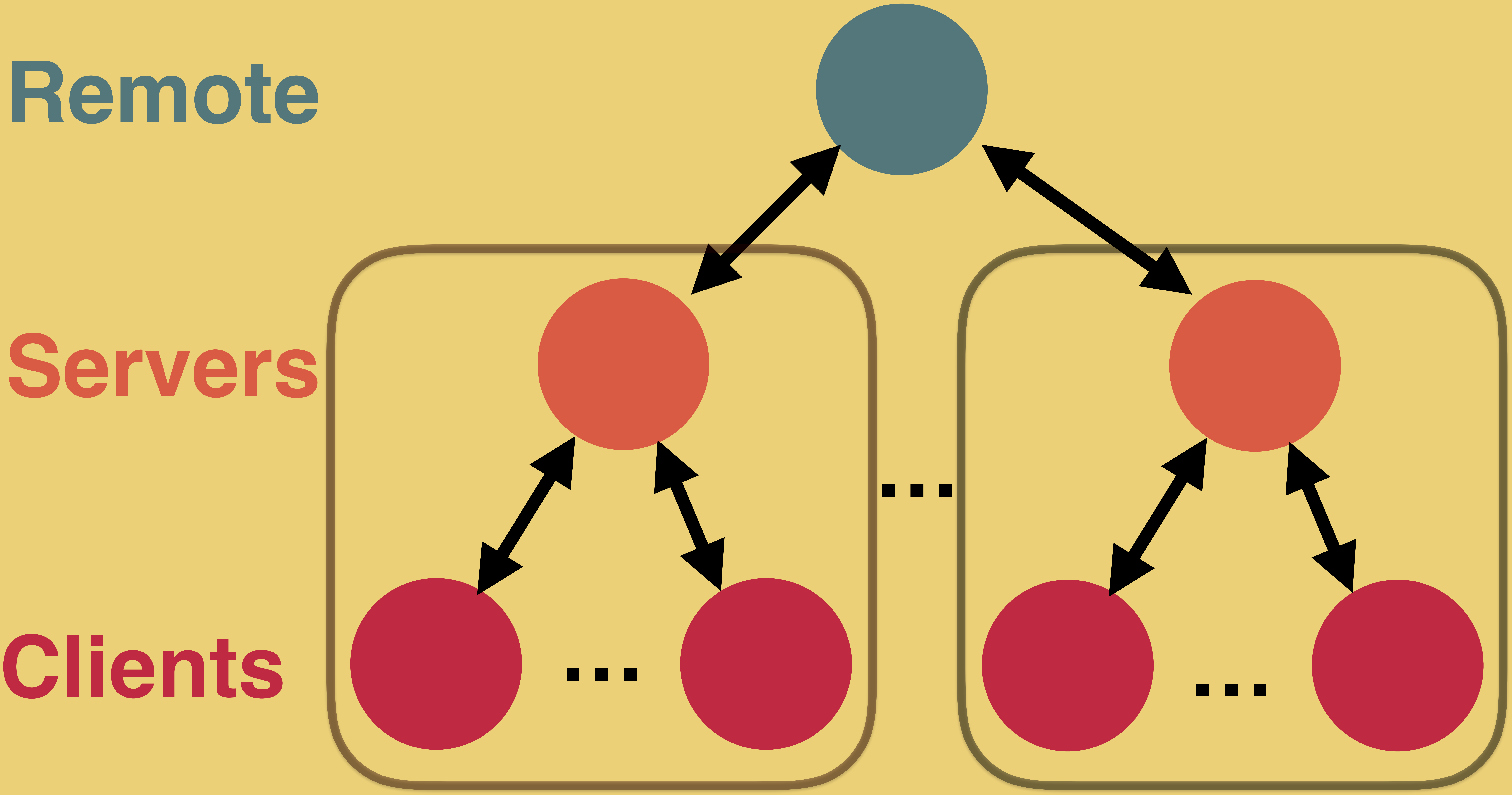
Target applications

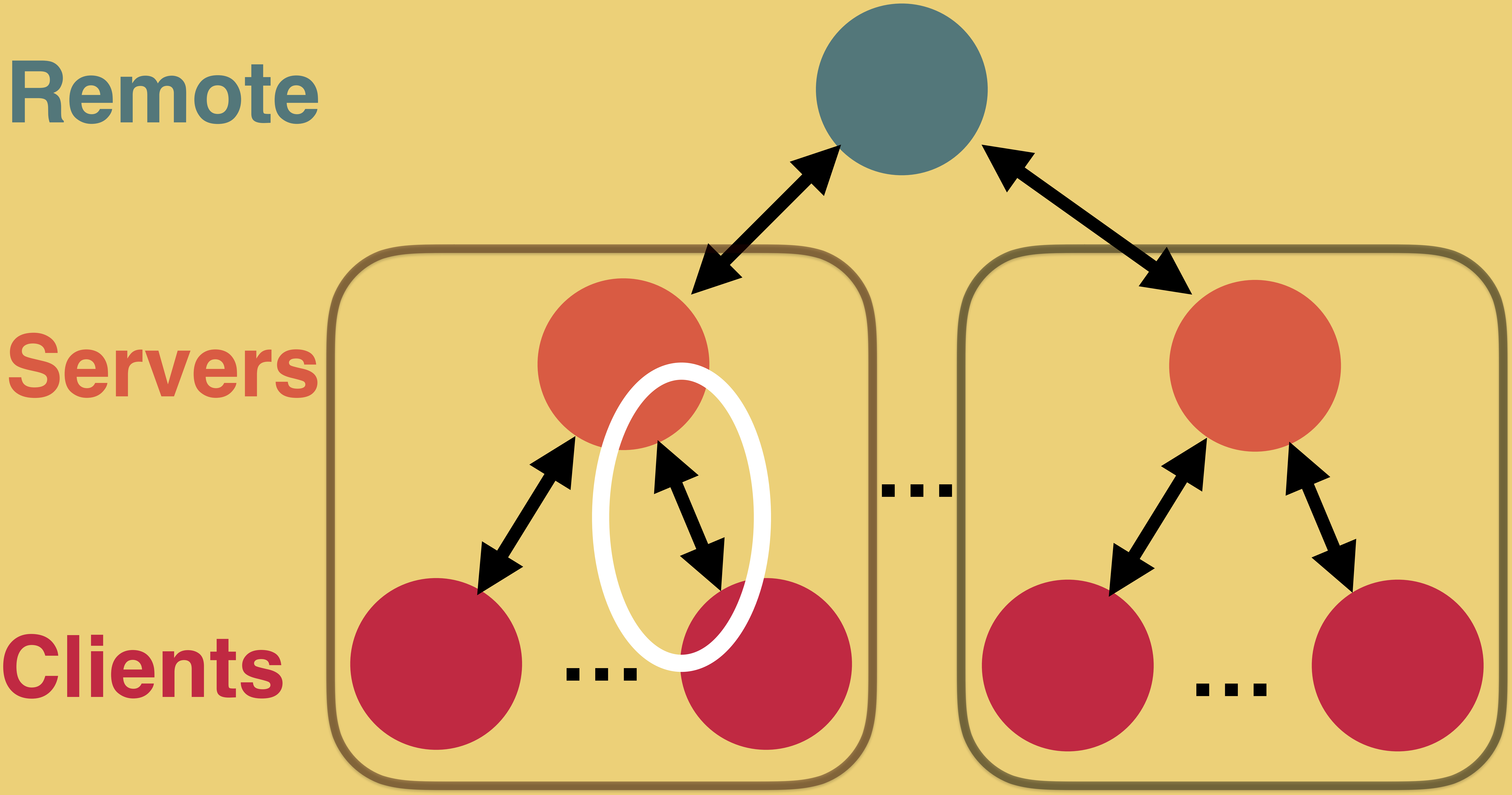
Programs we want to tune

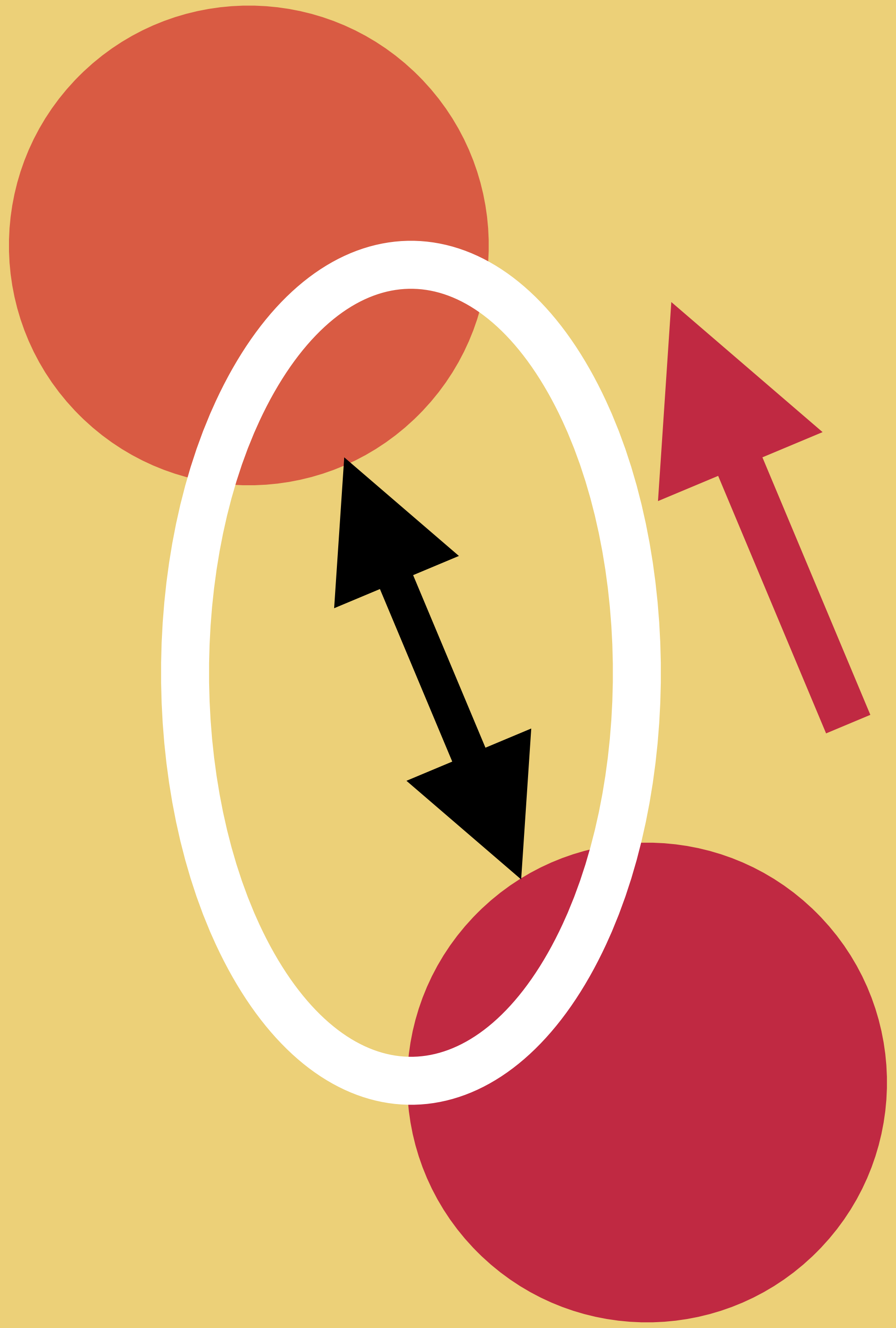
Clients


Remote
Servers
Clients
...
...
...

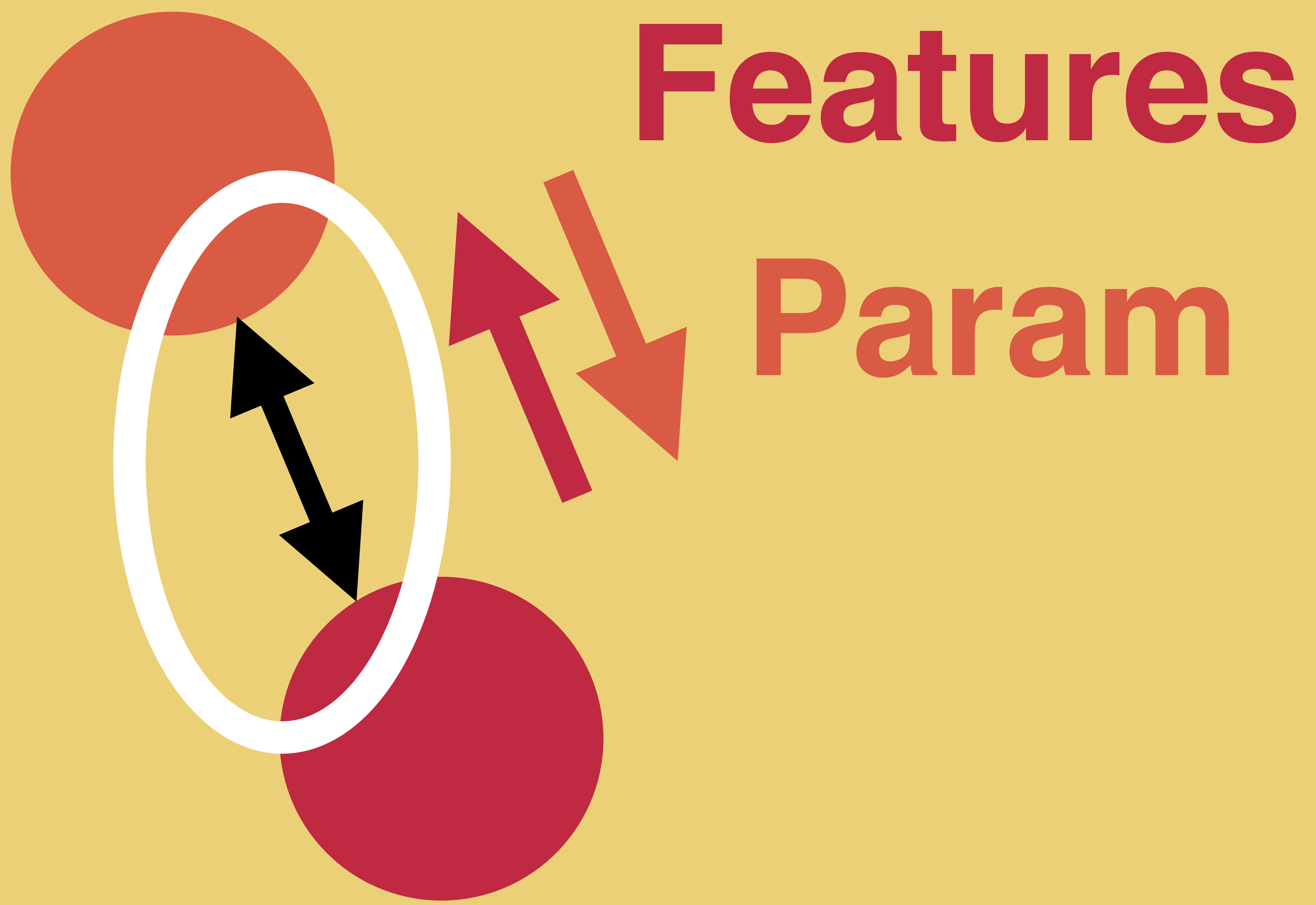

Remote
Servers
Clients
...
...
...

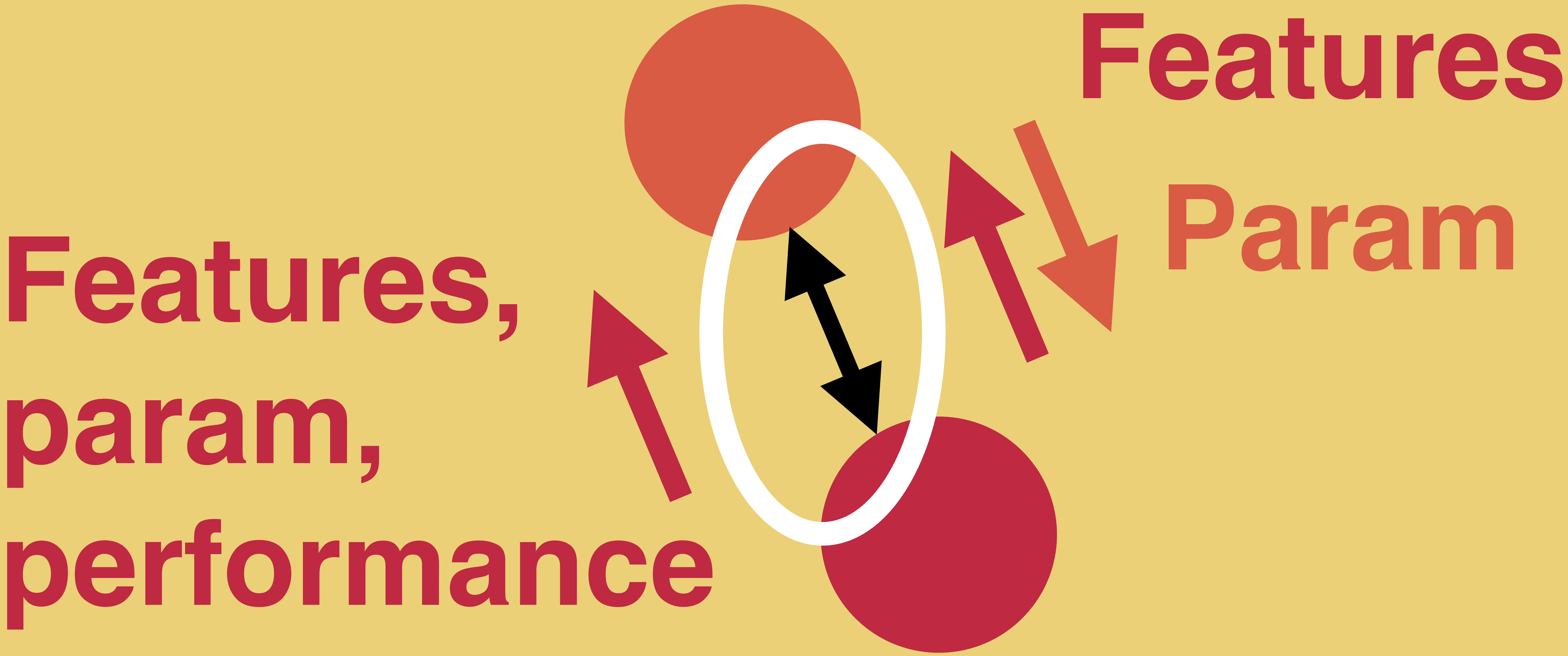

Remote
Servers
Clients
...
...
...

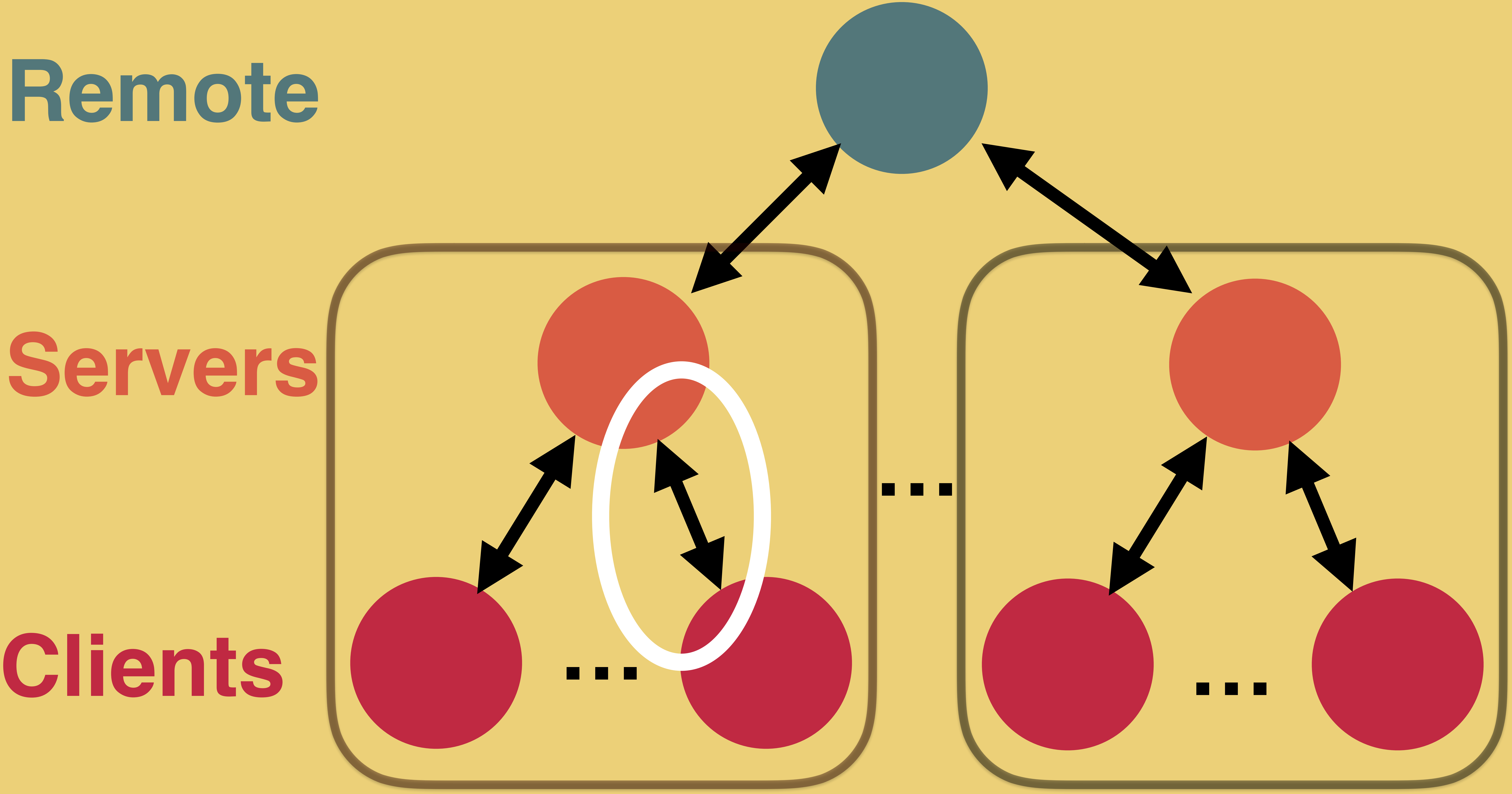

Remote
Servers
Clients

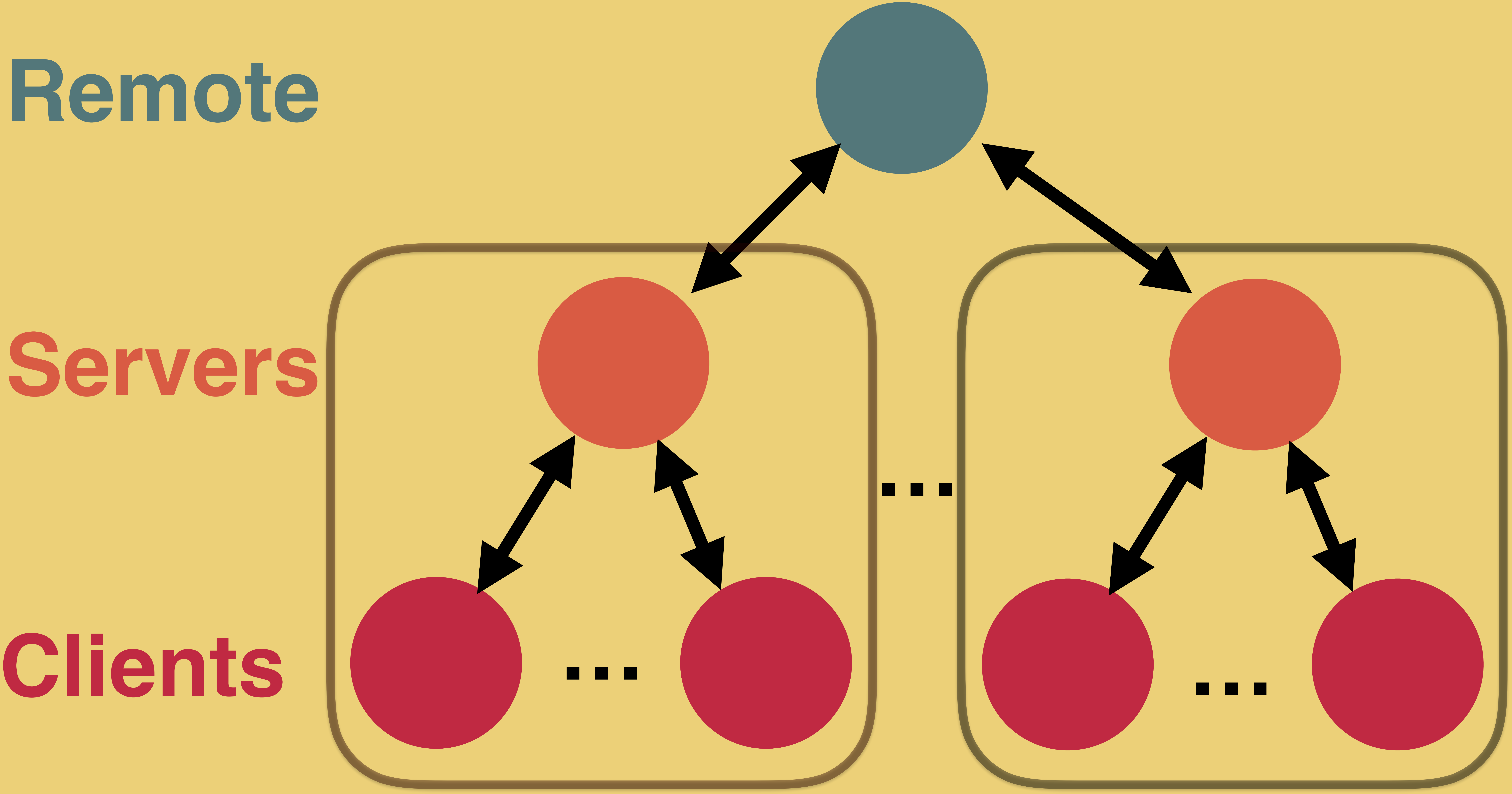


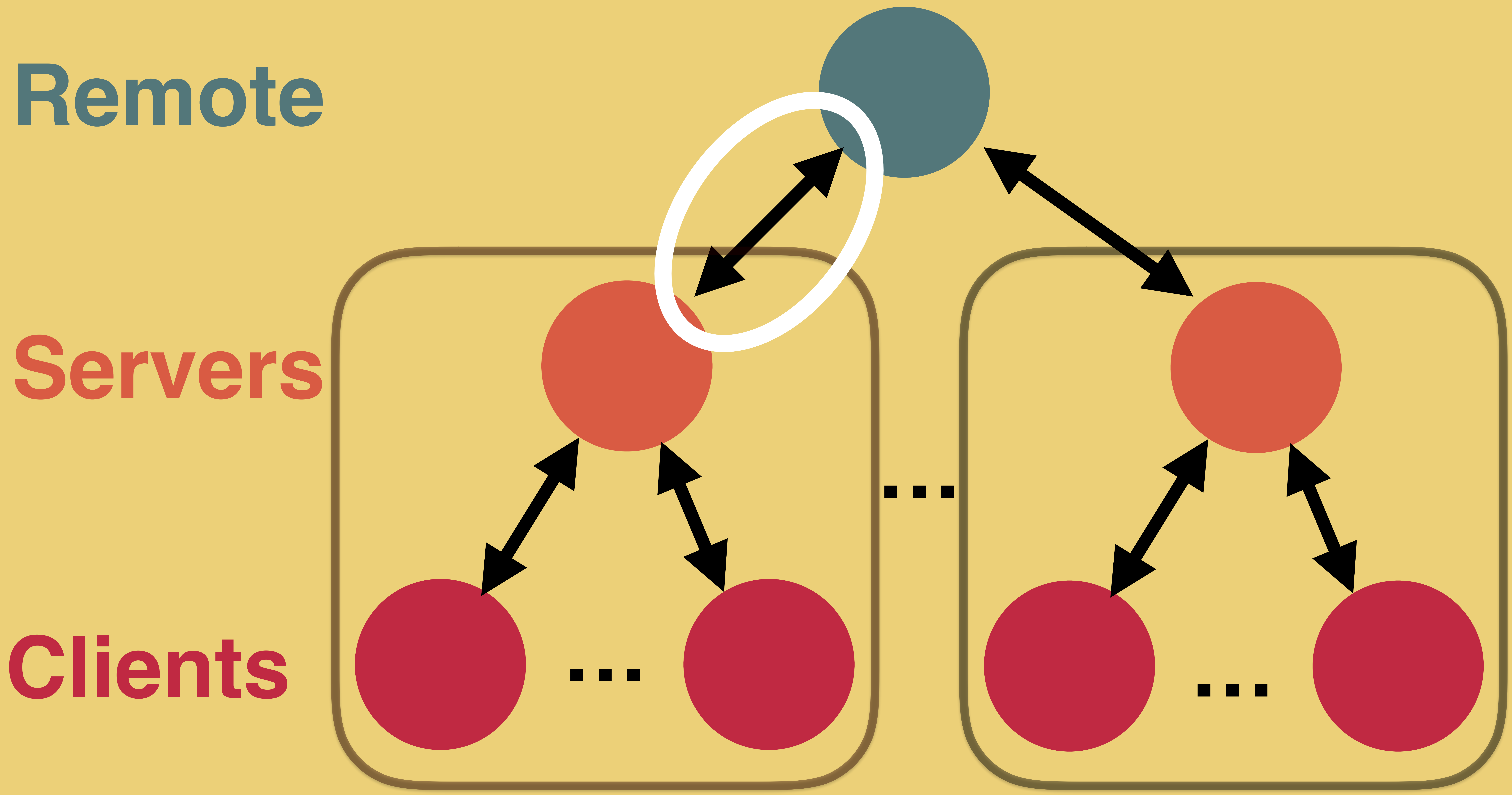
Features


Features
Param


Features
Param
Features,
param,
performance


Remote
Servers
Clients

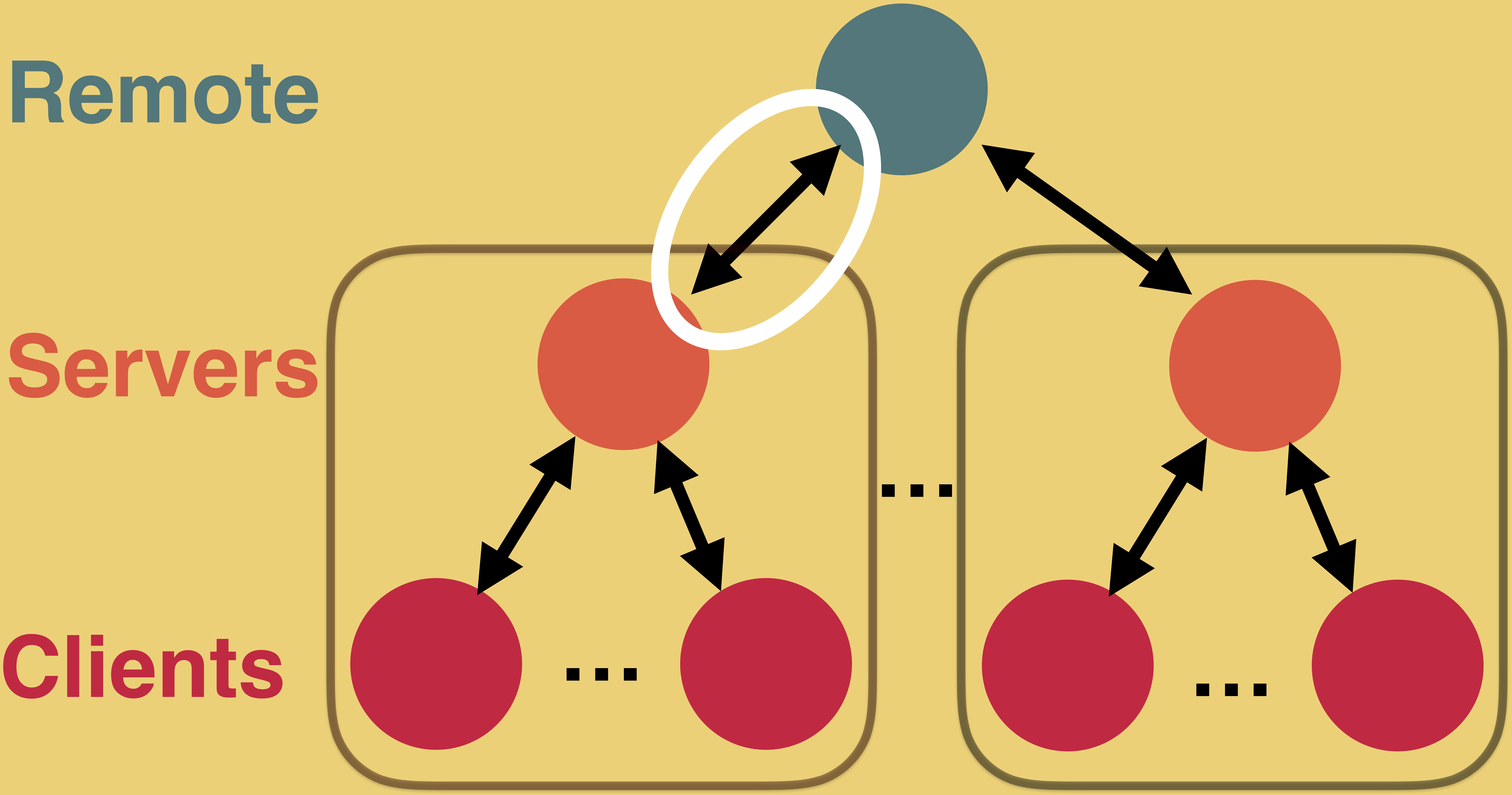

Remote
Servers
Clients
...
...
...

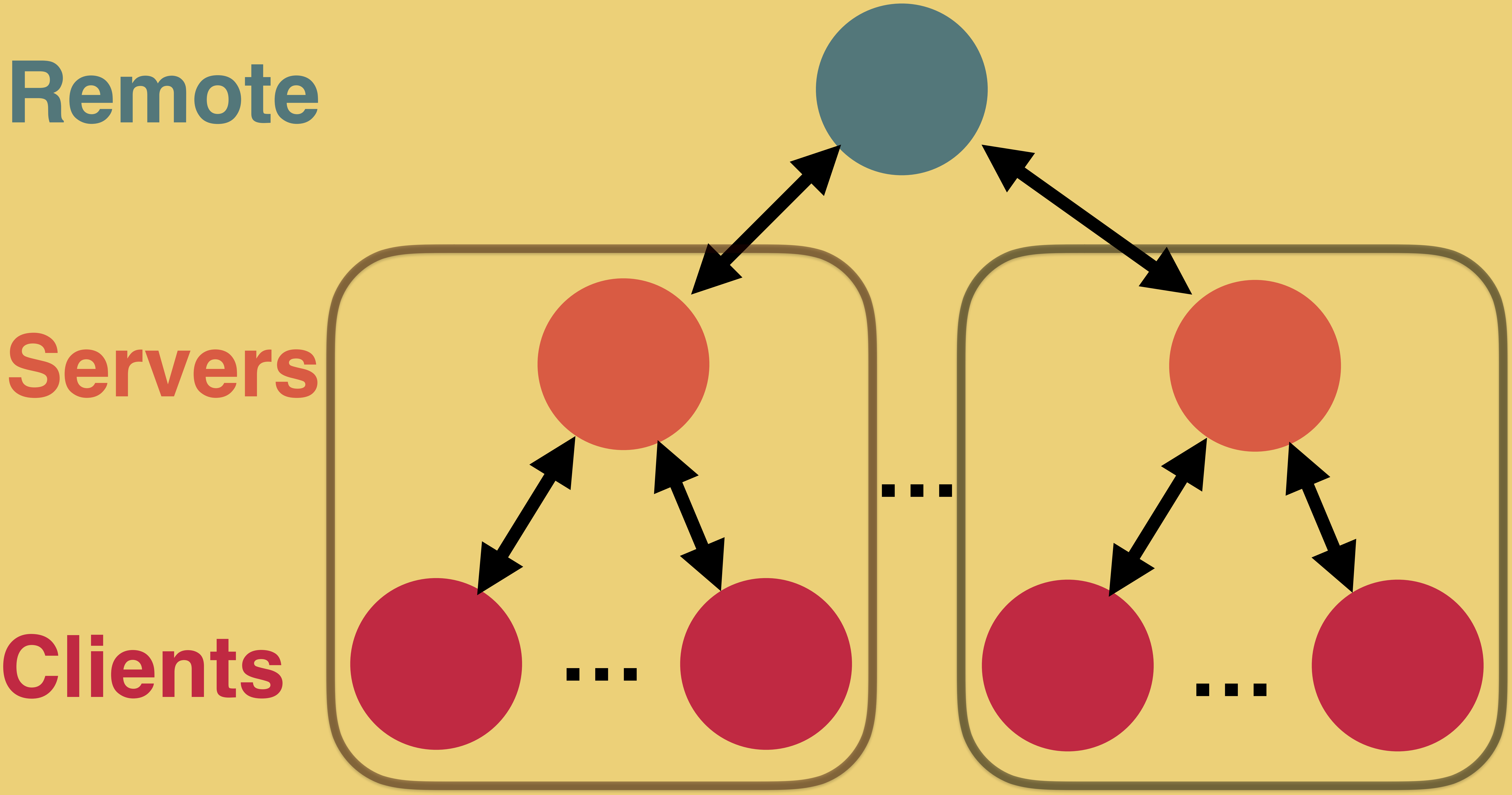


Remote
Servers
Clients


New training data


New training data
New training data


Remote
Servers
Clients


Remote
Servers
Clients

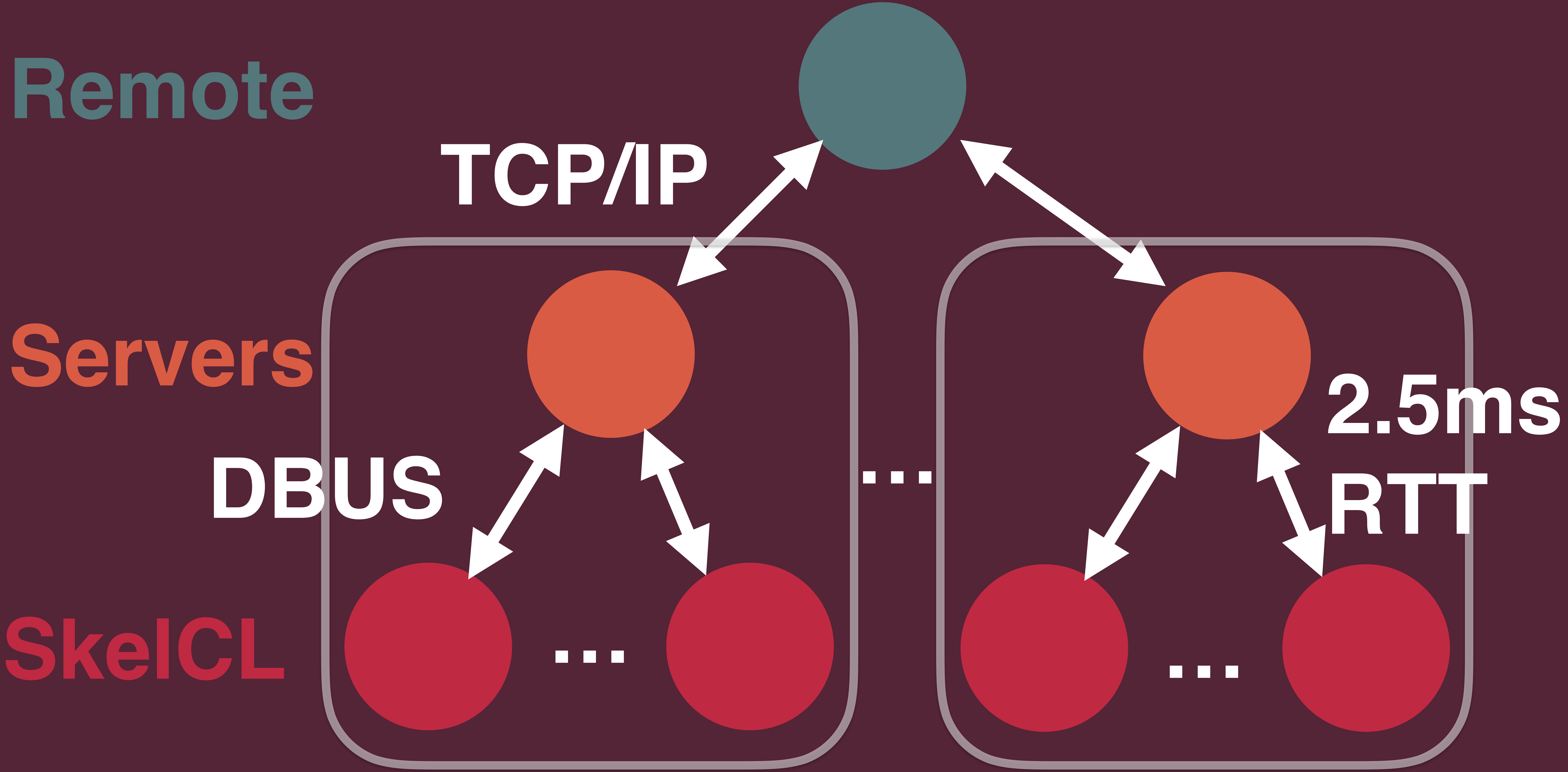
Demonstration

Implementation:

Remote: AWS instance + MySQL

Server: standalone system daemon, decision tree classifier

Client: modified SkelCL stencil pattern

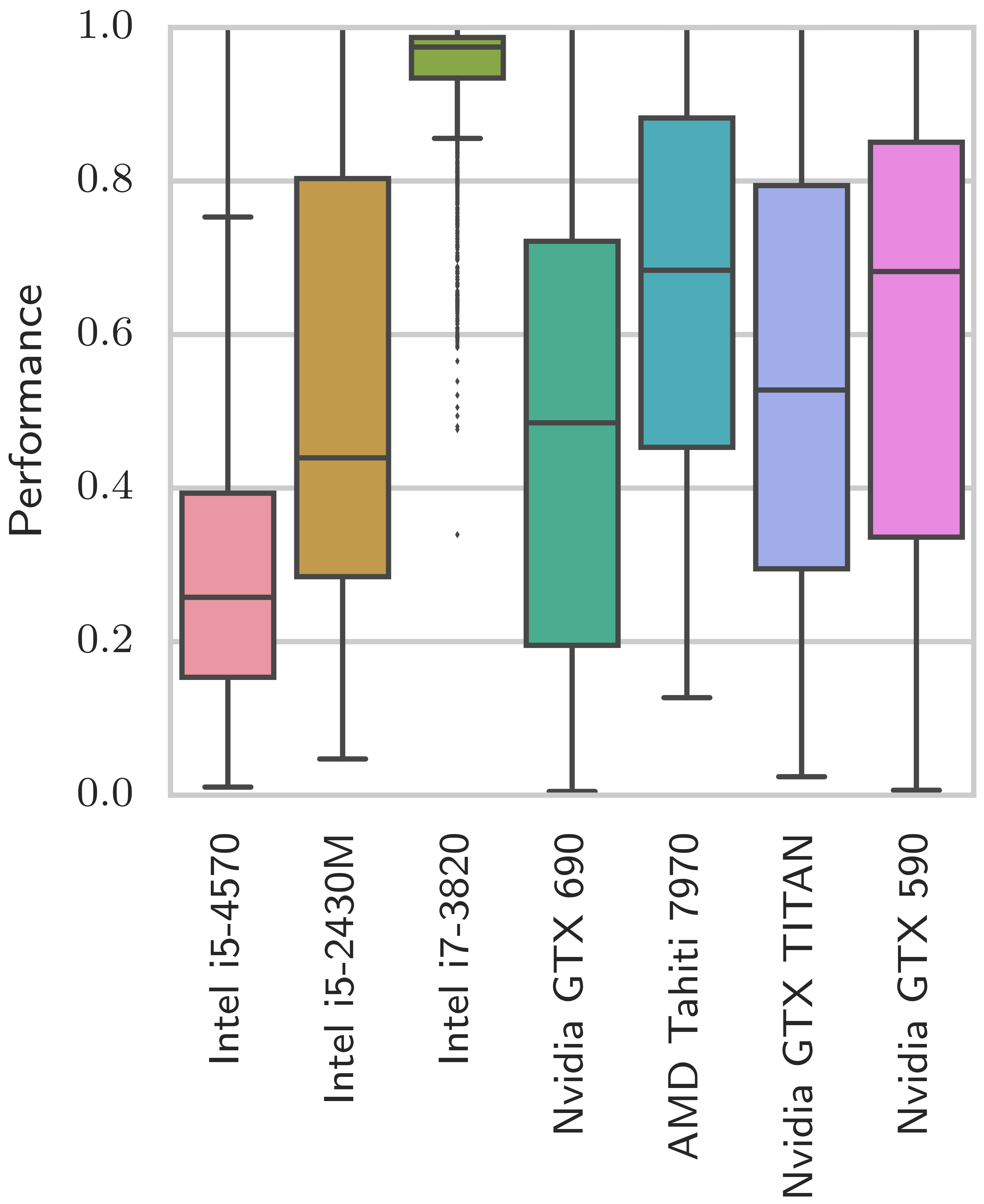

Remote
TCP/IP
Servers
DBUS
SkelCL
...
...
...
2.5ms
RTT

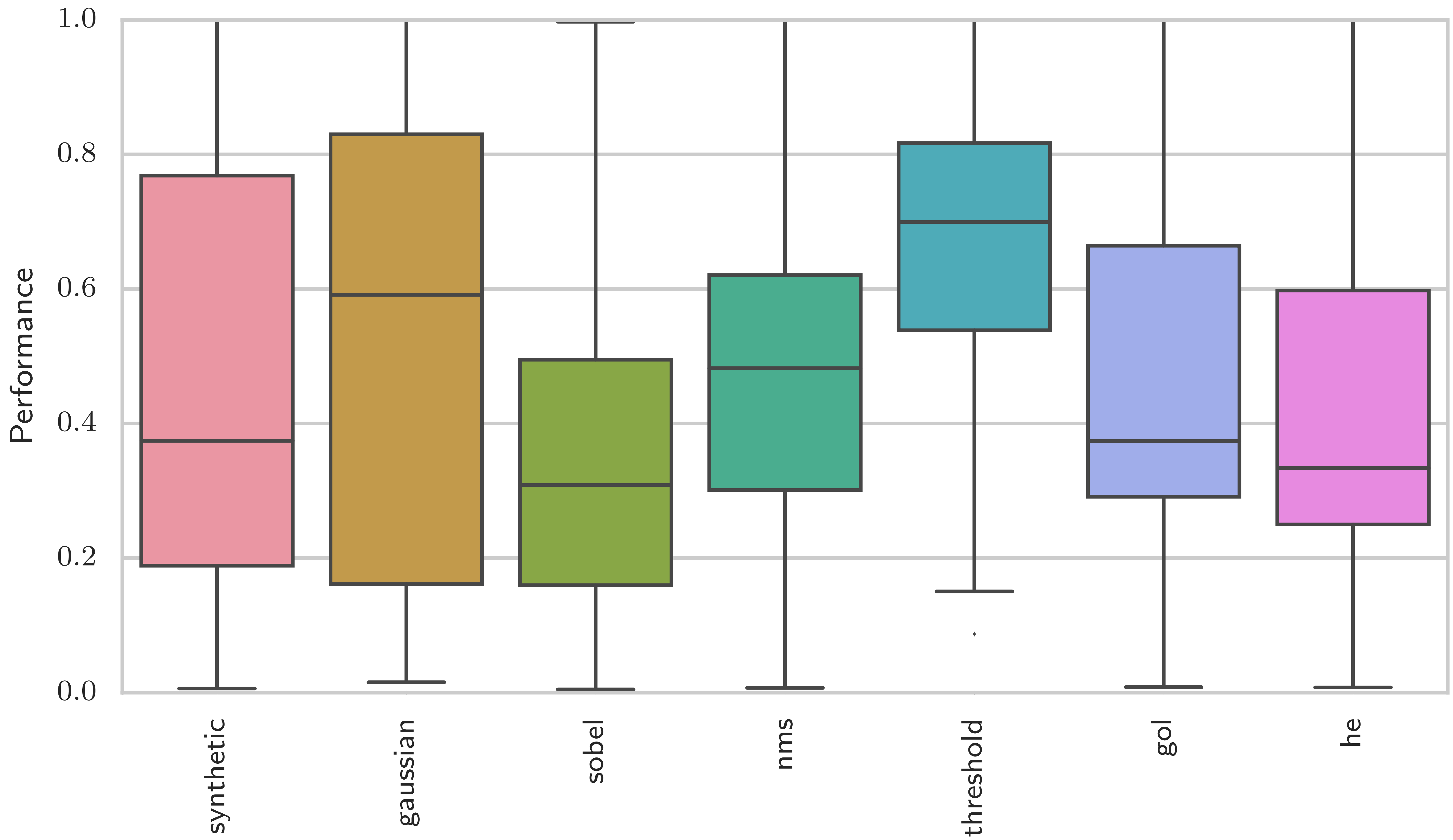
Experimental Setup:

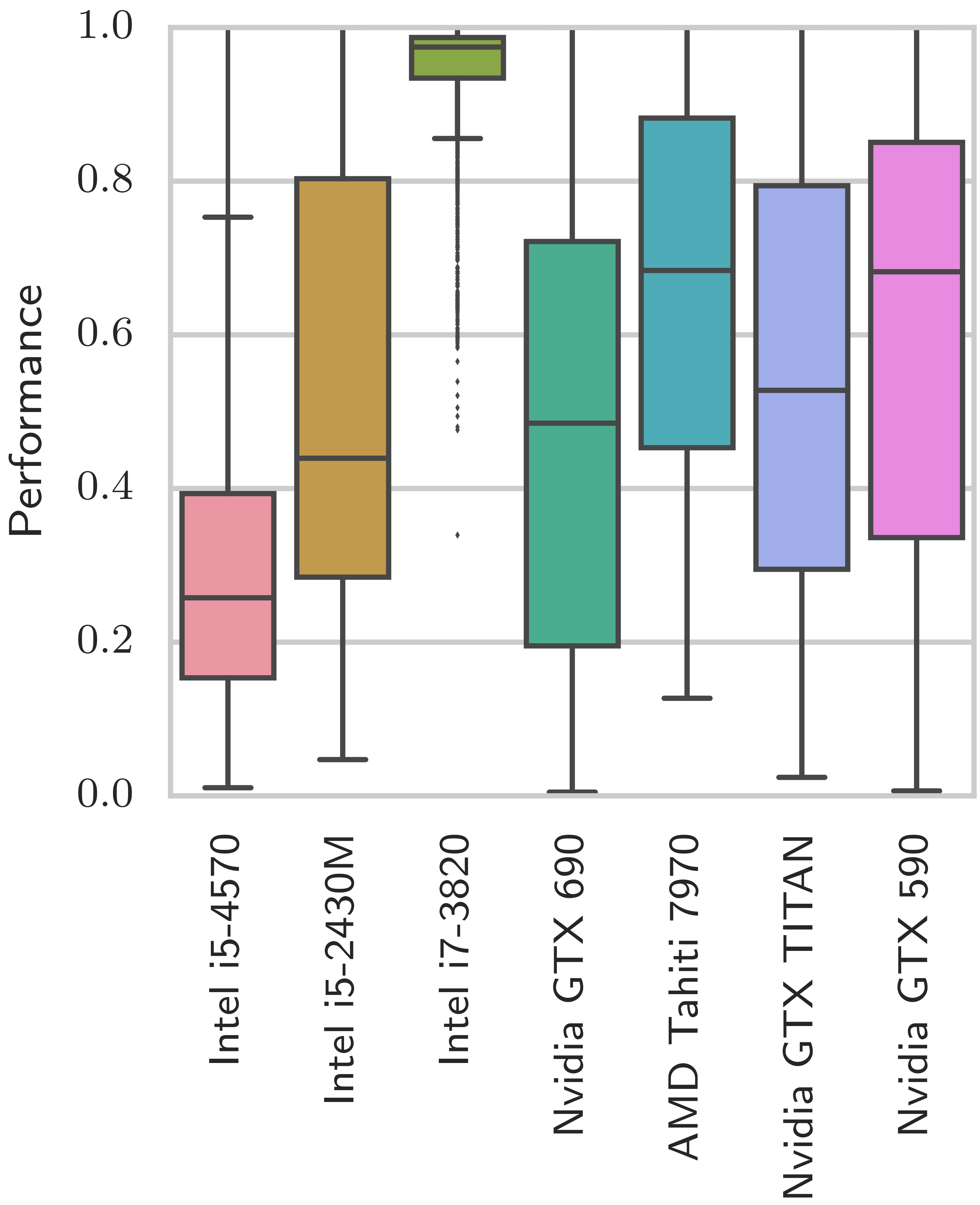
6 stencil benchmarks + synthetic.
7 different GPUs & CPUs.
4 dataset sizes.

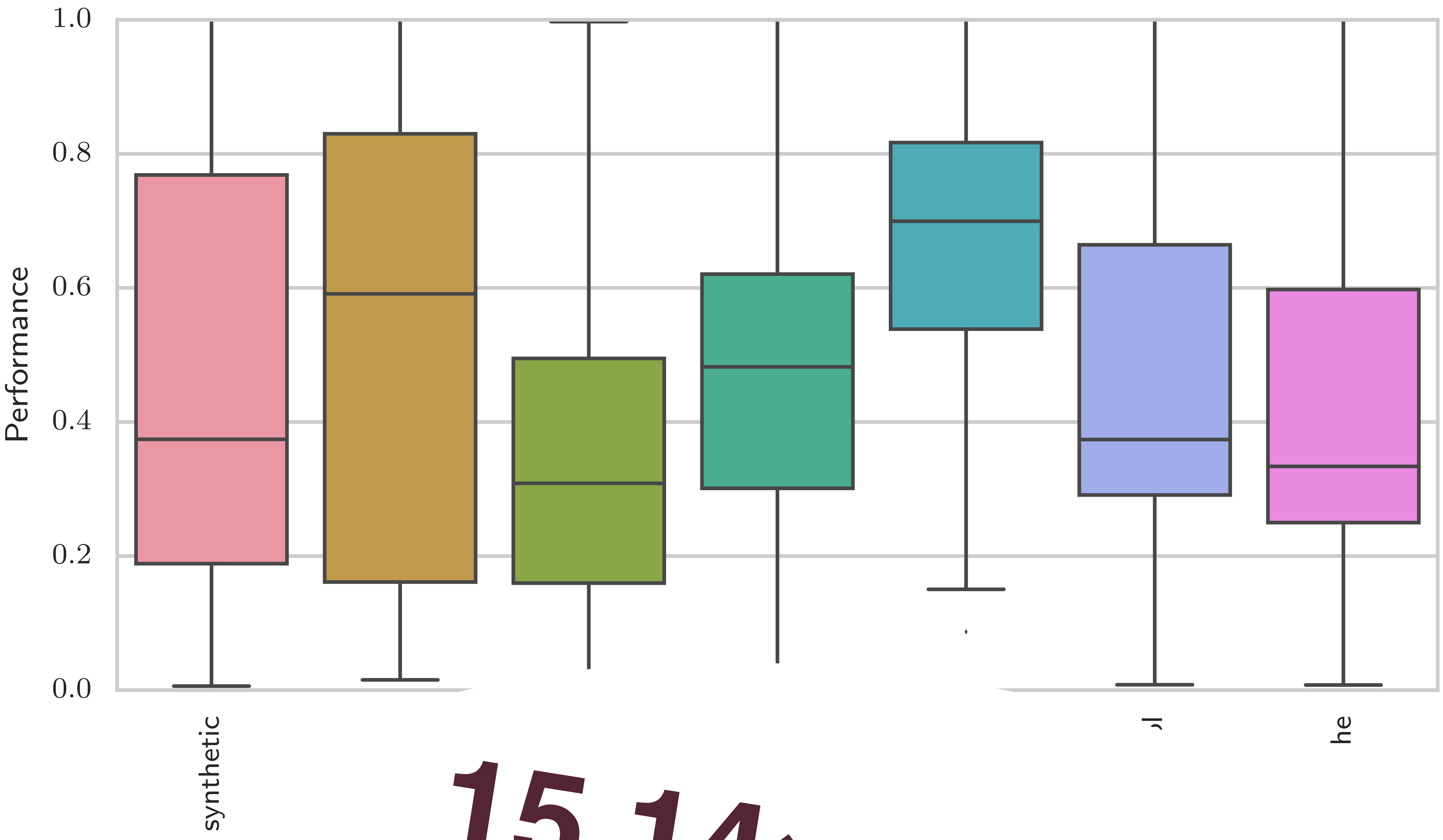
Exhaustive search of workgroup size space for each

Results

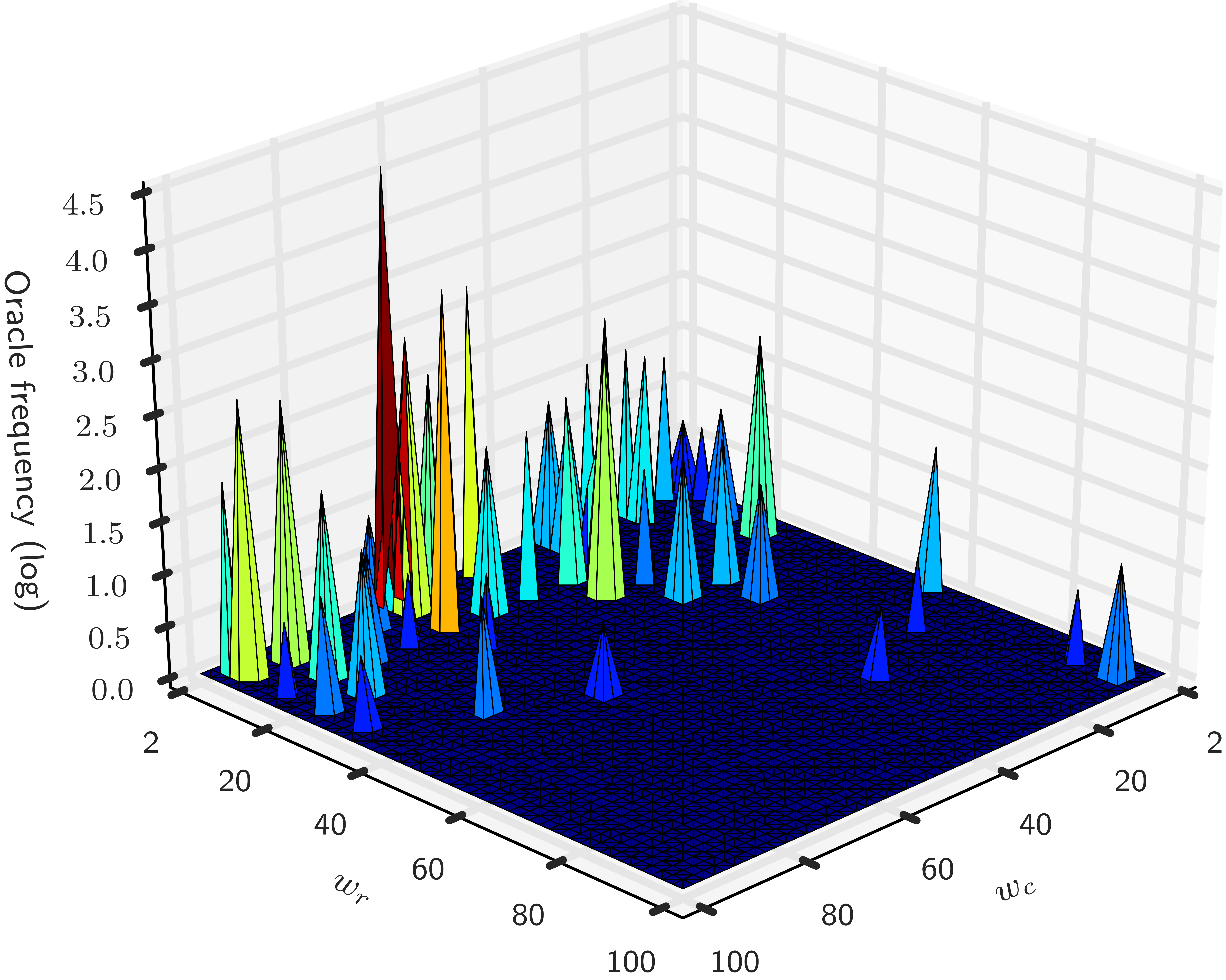

Performance
1.0
0.8
0.6
0.4
0.2
0.0
Intel i5-4570
Intel i5-2430M
Intel i7-3820
Nvidia GTX 690
AMD Tahiti 7970
Nvidia GTX TITAN
Nvidia GTX 590


Performance
1.0
0.8
0.6
0.4
0.2
0.0
synthetic
gaussian
sobel
nms
threshold
gol
he

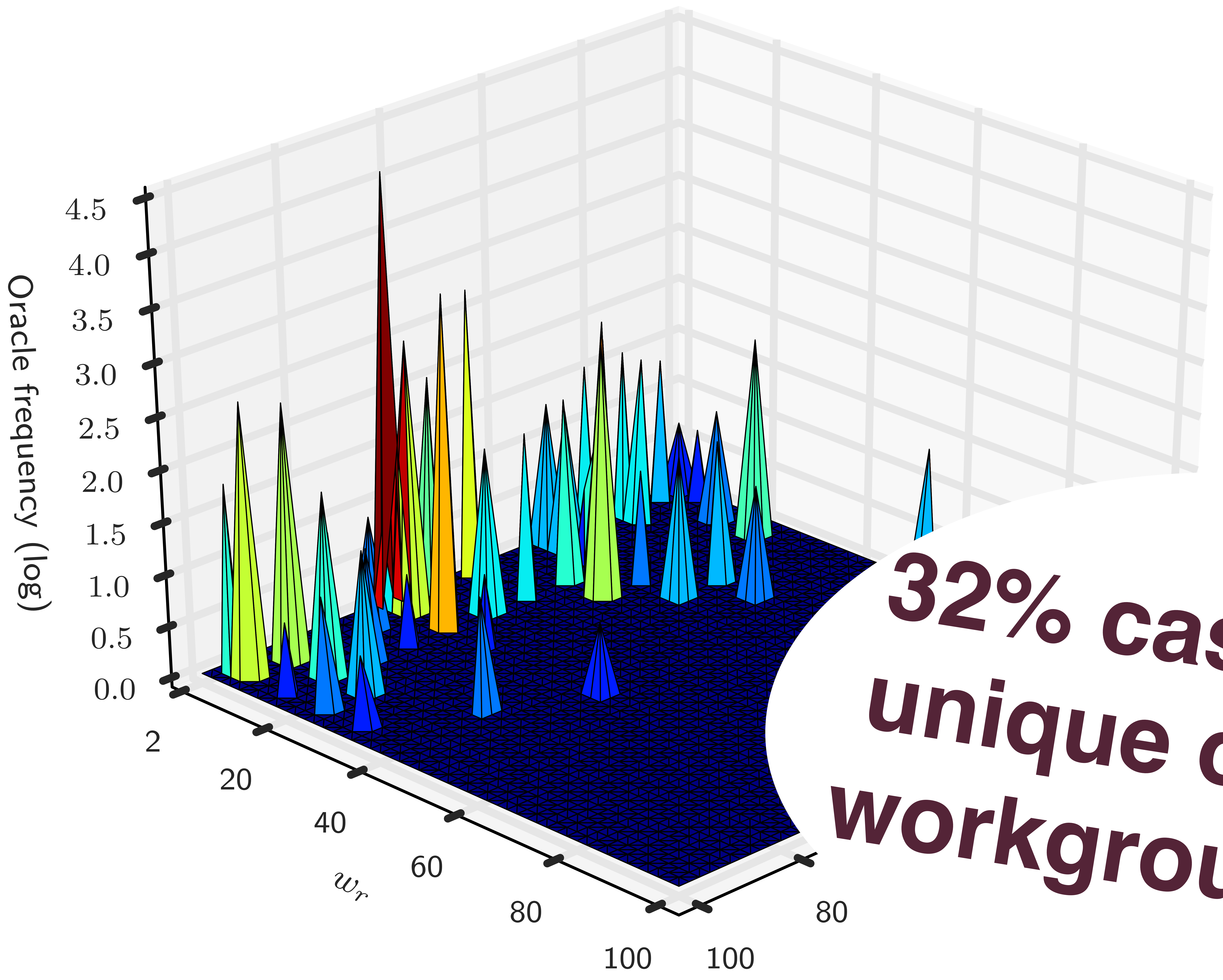




15.14× speedup best over worst

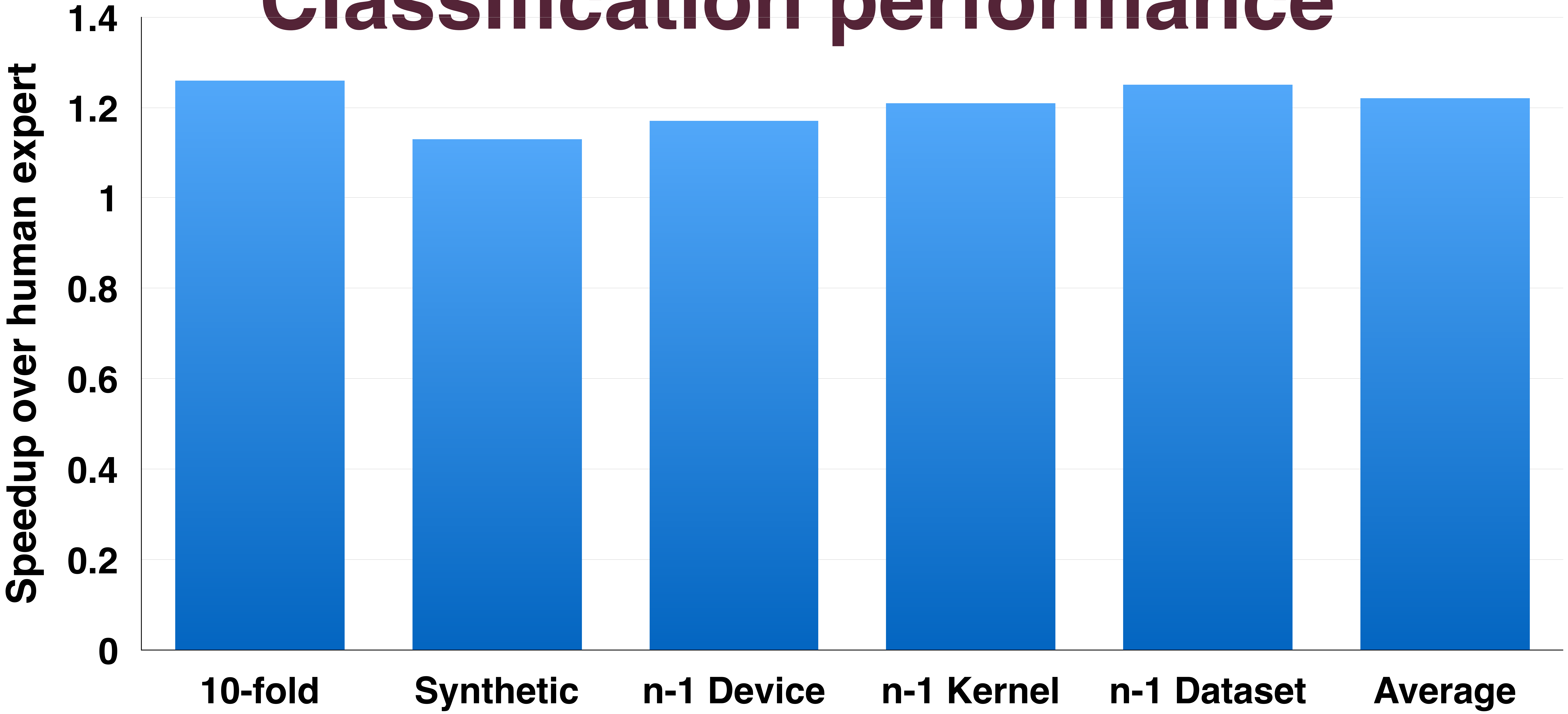


Oracle frequency (log)
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
w_r
2
20
40
60
80
100
w_c
2
20
40
60
80
100

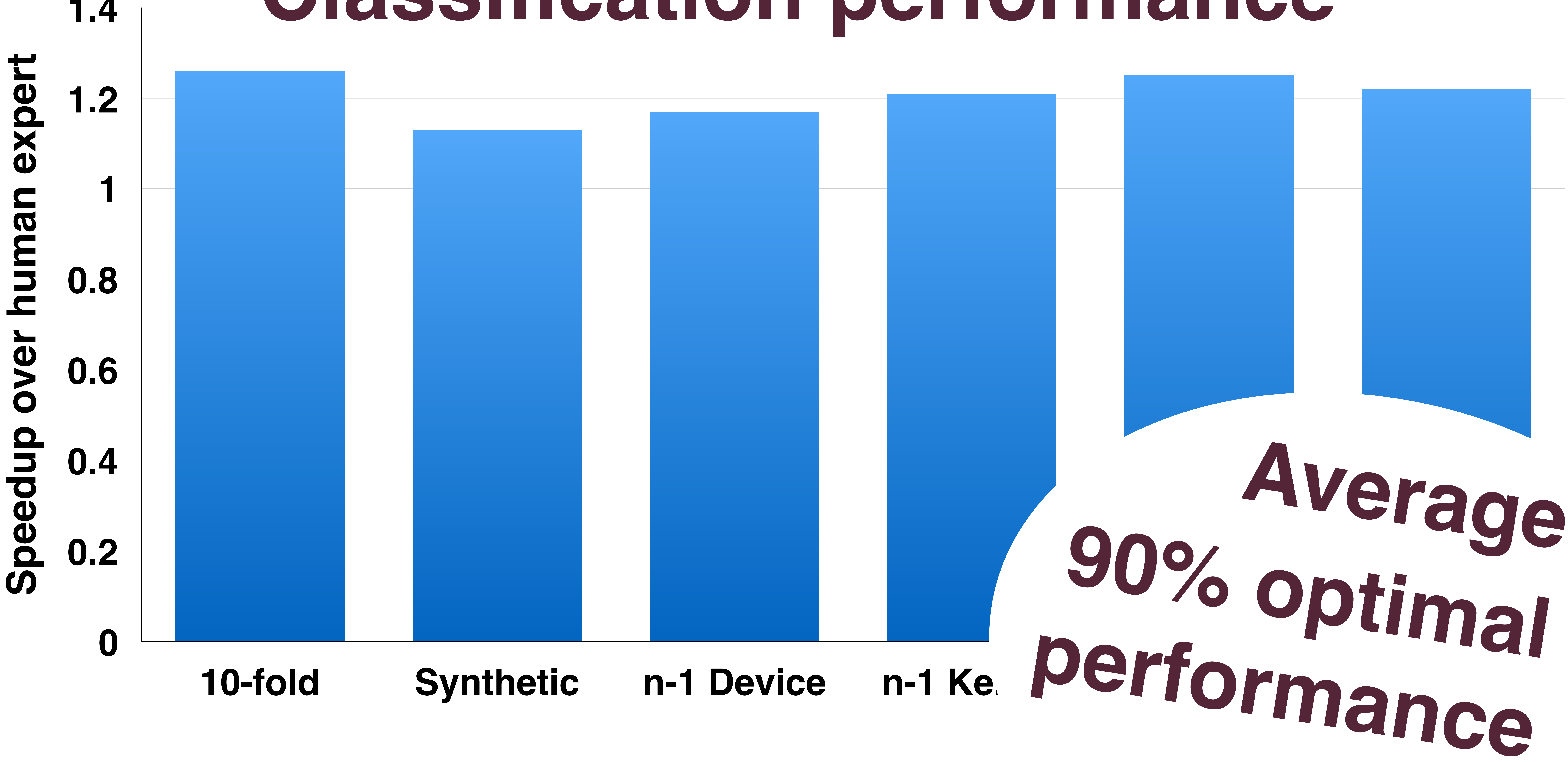


32% cases has unique optimal workgroup size

Classification performance



Classification performance



Conclusions

High level GPU code must compete with low level on *performance*

That means *automating* the kind of tuning which is typical of low level

We present a framework for doing this using machine learning

Demonstrated using SkelCL stencils

Achieves average 1.22x speedup over *human expert*

Details in the paper!

Towards Collaborative Performance Tuning of Algorithmic Skeletons

Chris Cummins Pavlos Petoumenos Michel Steuwer Hugh Leather

University of Edinburgh

c.cummins@ed.ac.uk, ppetoume@inf.ed.ac.uk, michel.steuwer@ed.ac.uk, hleather@inf.ed.ac.uk

Abstract

The physical limitations of microprocessor design have forced the industry towards increasingly heterogeneous designs to extract performance. This trend has not been matched with adequate software tools, leading to a growing disparity between the availability of parallelism and the ability for application developers to exploit it.

Algorithmic skeletons simplify parallel programming by providing high-level, reusable patterns of computation. Achieving performant skeleton implementations is a difficult task; skeleton authors must attempt to anticipate and tune for a wide range of architectures and use cases. This results in implementations that target the general case and cannot provide the performance advantages that are gained from tuning low level optimization parameters. Autotuning combined with machine learning offers promising performance benefits in these situations, but the high cost of training and lack of available tools limits the practicality of autotuning for real world programming. We believe that performing autotuning at the level of the skeleton library can overcome these issues.

In this work, we present OmniTune — an extensible and distributed framework for dynamic autotuning of optimization parameters at runtime. OmniTune uses a client-server model with a flexible API to support machine learning enabled autotuning. Training data is shared across a network of cooperating systems, using a collective approach to performance tuning.

We demonstrate the practicality of OmniTune in a case study using the algorithmic skeleton library SkelCL. By automatically tuning the workgroup size of OpenCL Stencil skeleton kernels, we show that that static tuning across a range of GPUs and programs can achieve only 26% of the optimal performance, while OmniTune achieves 92% of this maximum, equating to an average 5.65× speedup. OmniTune achieves this without introducing a significant runtime overhead, and enables portable, cross-device and cross-program tuning.

1. Introduction

General purpose programming with GPUs has been shown to provide huge parallel throughput, but poses a significant programming challenge, requiring application developers to master an unfamiliar programming model (such as provided by CUDA or OpenCL) and architecture (SIMD with a multi-level memory hierarchy). As a result, GPGPU programming is often considered beyond the realm of everyday development. If steps are not taken to increase the accessibility of such parallelism, the gap between potential and utilized performance will continue to widen as hardware core counts increases.

Algorithmic skeletons offer a solution to this this *programmability challenge* by raising the level of abstraction. This simplifies parallel programming, allowing developers to focus on solving problems rather than coordinating parallel resources. Skeleton frameworks provide robust parallel implementations of common patterns of computation which developers parameterise with their application-specific code. This greatly reduces the challenge of parallel programming, allowing users to structure their problem-solving logic sequentially, while offloading the cognitive cost of parallel coordination to the skeleton library author. The rising number of skeleton frameworks supporting graphics hardware illustrates the demand for high level abstractions for GPGPU programming [1, 2]. The challenge is in maintaining portable performance across the breadth of devices in the rapidly developing GPU and heterogeneous architecture landscape.

1.1 The Performance Portability Challenge

There are many factors — or *parameters* — which influence the behavior of parallel programs. For example, setting the number of threads to launch for a particular algorithm. The performance of parallel programs is sensitive to the values of these parameters, and when tuning to maximize performance, one size does not fit all. The suitability of parameter values depends on the program implementation, the target hardware, and the dataset that is operated upon. Iterative compilation and autotuning have been shown to help in these cases by automating the process of tuning parameter values to match individual execution environments [3]. However, there have been few attempts to develop general mechanisms for these techniques, and the time taken to develop ad-hoc autotuning solutions and gather performance data is often prohibitively expensive.

We believe that by embedding autotuning at the skeletal level, it is possible to achieve performance with algorithmic skeletons that is competitive with — and in some cases, exceeds — that of hand tuned parallel implementations which traditionally came at the cost of many man hours of work from expert programmers to develop.

Incorporating autotuning into algorithmic skeleton libraries has two key benefits: first, it minimizes development effort by requiring only a modification to the skeleton implementation rather than to every user program; and second, by targeting a library, it enables a broader and more substantive range of performance data to be gathered than with ad-hoc tuning of individual programs.

2016/1/6

Towards *Collaborative Performance Tuning* of Algorithmic Skeletons

http://chriscummins.cc