



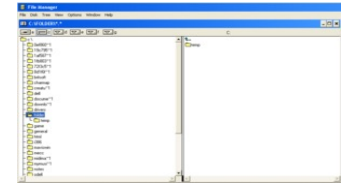
UNIVERSITY OF COPENHAGEN

# Modularity through Clients and Services, RPC Techniques for Performance

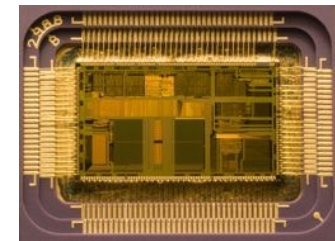
ACS, Dmitriy Traytel

## Do-it-yourself Recap: Fundamental abstractions

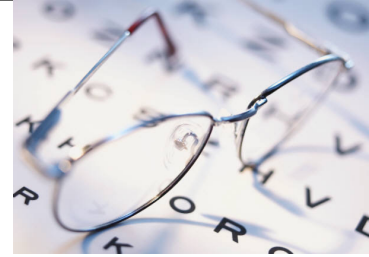
- Which were the three fundamental abstractions?
- What were their APIs?
- Must these abstractions be implemented in a single node or can they be distributed? Give an example!



`(loop (print (eval (read))))`



# What should we learn today?



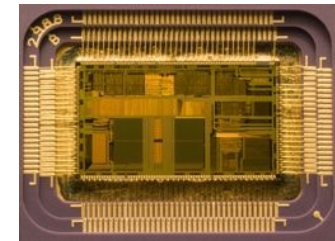
- Recognize and explain modular designs with clients and services
- Predict the functioning of service calls under different RPC semantics and failure modes
- Identify different mechanisms to achieve RPCs
- Implement RPC services with an appropriate mechanism, such as web services
- Explain performance metrics such as latency, throughput, overhead, utilization, capacity, and scalability
- List common hardware parameters that affect performance
- Apply performance improvement techniques, such as concurrency, batching, dallying, and fast-path coding

# Interpreters

- ***Interpreter***

- Instruction repertoire
- Environment
- Instruction pointer

(loop (print (eval (read))))



```
procedure INTERPRET()
```

```
  do forever
```

```
    instruction ← READ(instruction_pointer)
```

```
    perform instruction in environment context
```

```
    if interrupt_signal = TRUE then
```

```
      instruction_pointer ← entry of INTERRUPT_HANDLER
```

```
      environment ← environment of INTERRUPT_HANDLER
```

- Examples of Interpreters

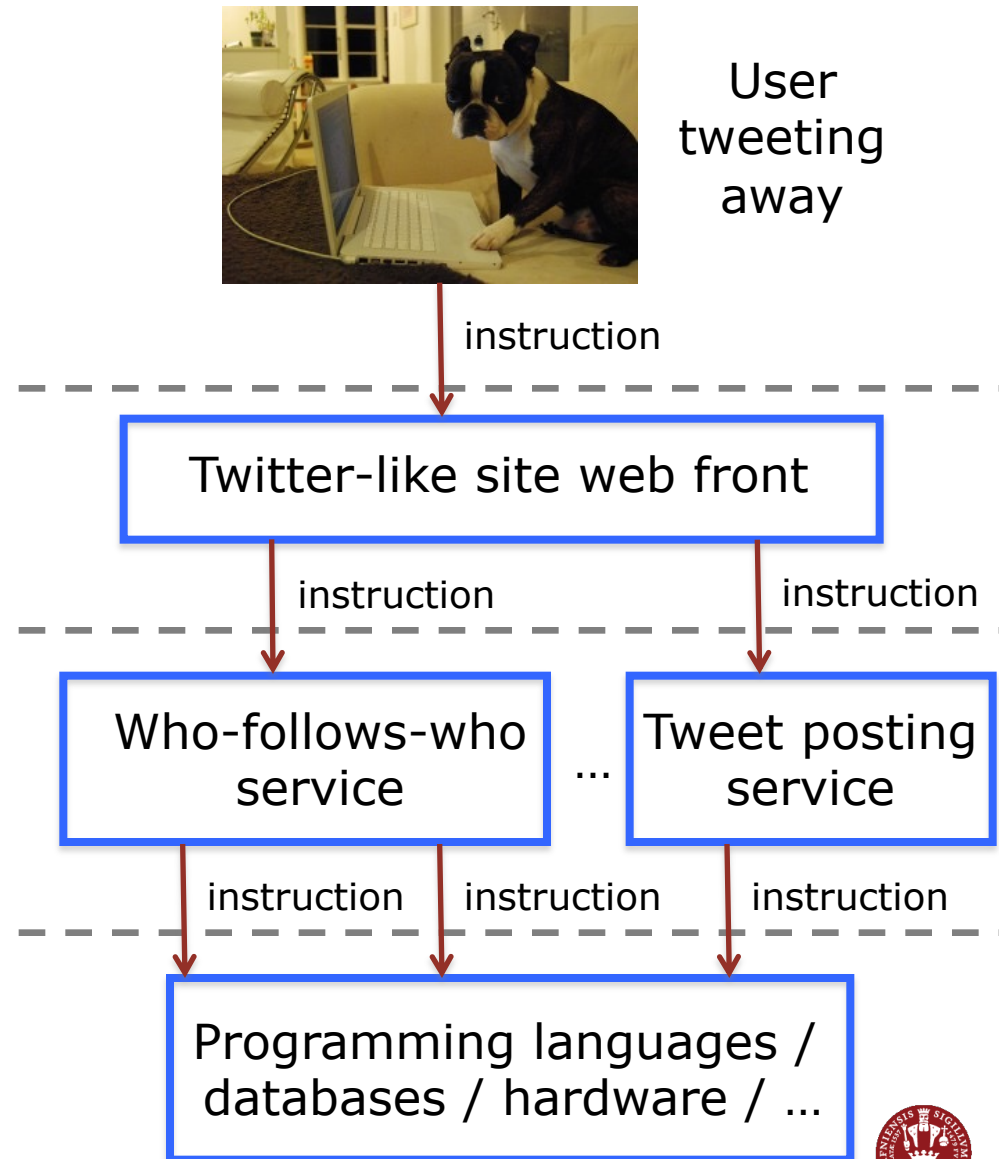
- Processors (CPU)
- Programming language interpreters
- Frameworks, e.g., MapReduce or Spark
- **Your own (layered) programs! (RPCs)**

Source:  
Saltzer &  
Kaashoek &  
Morris  
(partial)



# Layers and Modules

- Interpreters often organized in layers
- Modules
  - *Saltzer & Kaashoek glossary:*  
Components that can be separately designed / implemented / managed / replaced
  - “Instructions” of higher-level interpreters
  - Recursive: Can be whole interpreters themselves!





# Cloud-Power @ Yahoo!

My Yahoo! | May 12, 2009

Sign In | New here? Sign Up

Web Images Video Local Shopping More

Web Search

MY FAVORITES + Add

- View Yahoo! Sites
- Yahoo! Mail
- Autos
- eBay
- Finance (Down)
- Flickr
- Games
- Messenger
- Movies
- Music
- MySpace
- omg!
- Personals
- Sports
- TV
- Weather

RECOMMENDED

- Deal Of The Day
- Buzz
- Shine

TOP SEARCHES

1. ...
2. ...
3. ...
4. ...
5. ...
6. ...
7. ...
8. ...
9. ...
10. ...

Search Index

Machine Learning (e.g. Spam filters)

Stay connected  
Preview and keep up-to-date with your mail.

Attachment Storage

Ads Optimization

Image/Video Storage & Delivery

700 What's Your Credit Score? A bad credit score is 600 or below. See yours in 2 easy steps for \$0. By Experian®







# Cloud-Power @ Yahoo!

My Yahoo! | May 12, 2009 | Sign In | New here? Sign Up

Web Images Video Local Shopping More

Web Search

MY FAVORITES + Add

- View Yahoo! Sites
- Yahoo! Mail
- Autos
- eBay
- Finance (Down)
- Flickr
- Games
- Messenger
- Movies
- Music
- MySpace
- omg!
- Personals
- Sports
- TV
- Weather

TOP SEARCHES

1. ...
2. ...
3. ...
4. ...
5. ...
6. Leonard Nimoy
7. Fawcett
8. Grizzly Bear Attack
9. Stamps
10. TV Recaps

**Search Index**

**Machine Learning (e.g. Spam filters)**

**Attachment Storage**

**Image/Video Storage**

**Image/Video Storage**

What happens when modules fail with (unintended) errors?

Source: Raghu Ramakrishnan, LADIS 2009 (partial)





# Cloud-Power @ Yahoo!



What happens when modules fail with  
(unintended) errors?

Source: Raghu Ramakrishnan, LADIS 2009 (partial)



# Isolating Errors: Enforced Modularity

## Clients & Services

- Restrict communication to *messages only*
- Client request / Service response (or reply)
- Conceptually client and service in different computers

## OS Virtualization

- Create virtualized versions of fundamental abstractions
- Client and services remain isolated even on same computer
- VMs: Virtualize the virtualizer ☺



# Isolating Errors: Enforced Modularity

## Clients & Services

- Restrict communication to *messages only*
- Client request / Service response (or reply)
- Conceptually client and service in different computers

## OS Virtualization

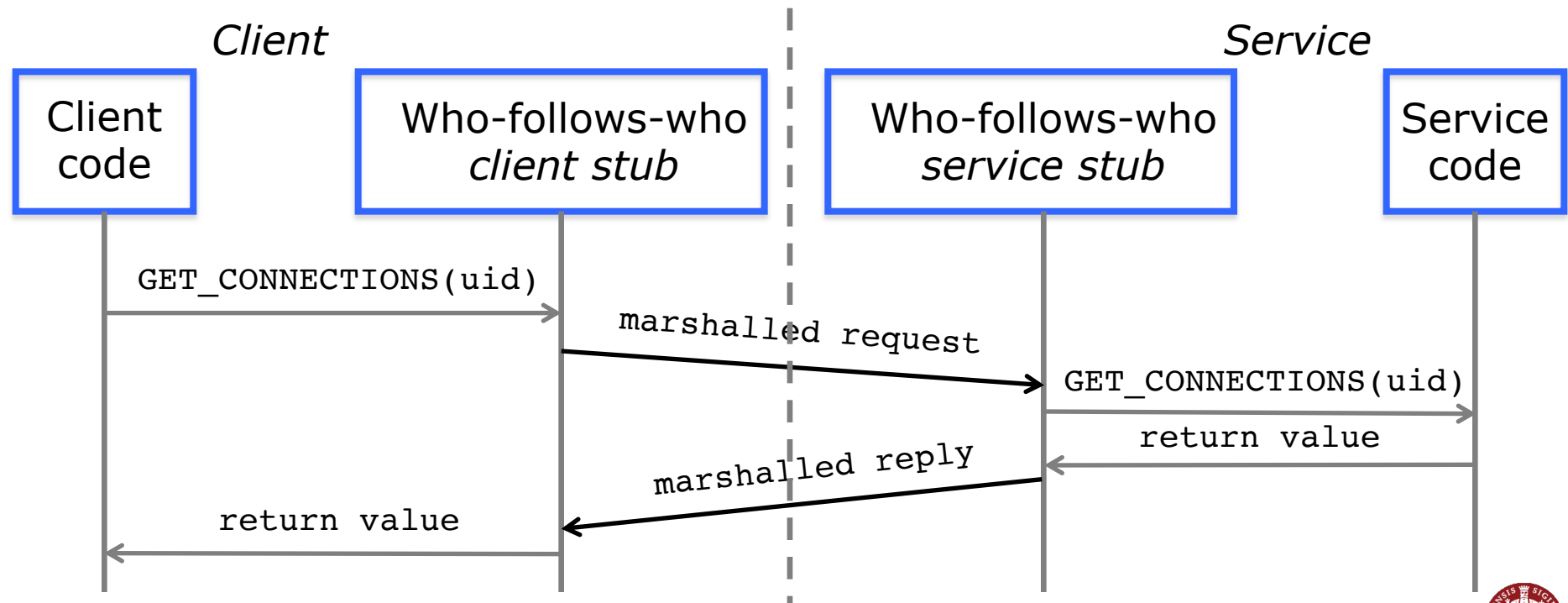
- Create virtualized versions of fundamental abstractions
- Client and services remain isolated even on same computer
- VMs: Virtualize the virtualizer ☺

Even more techniques for fault tolerance  
later in the course



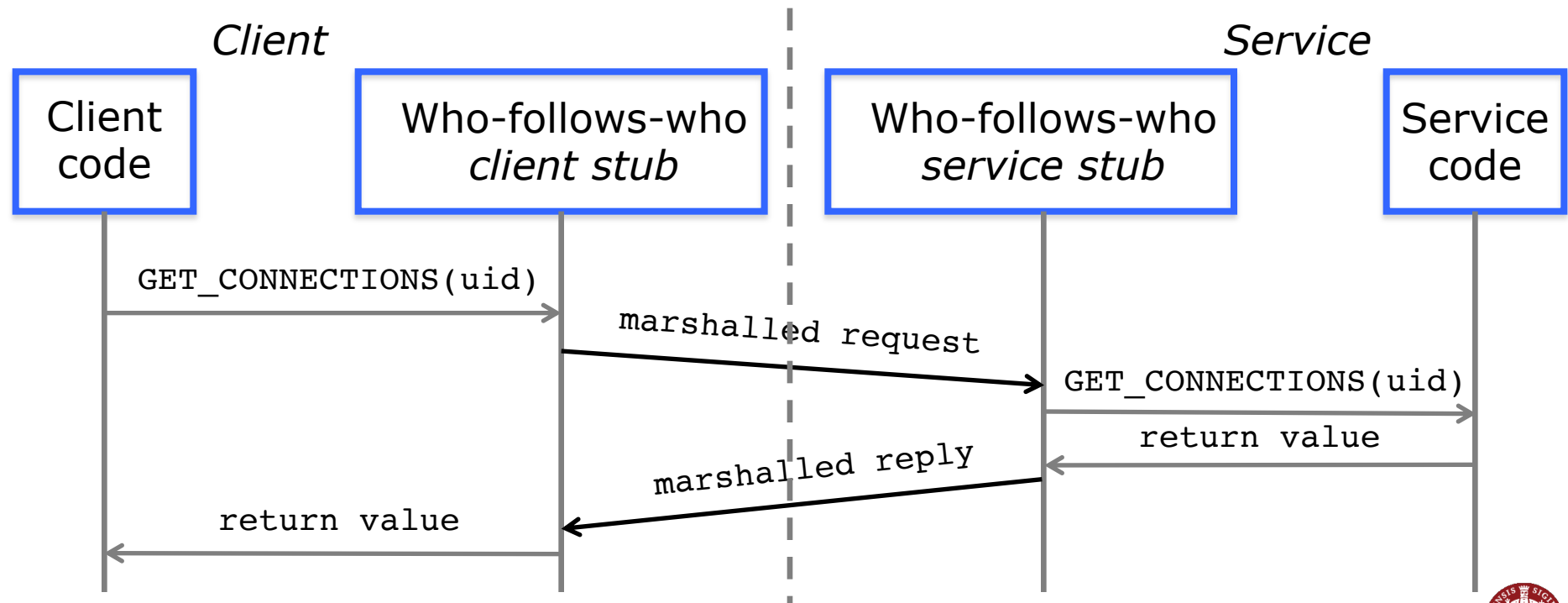
## RPC: Remote Procedure Call

- Client-service request / response interactions
- Automate *marshalling* and *communication*



## RPC: Remote Procedure Call

- Recall that RPC semantics differ from local procedure calls
- What can go wrong when a client crashes?  
What about when a server crashes?



# RPC Semantics

- **At-least-once**
  - Operation is *idempotent*
    - Naturally occurs if side-effect free
  - Stub just retries operation → failures can still occur!
  - Example: calculate SQRT
- **At-most-once**
  - Operation does have side-effects
  - Stub must ensure duplicate-free transmission
  - Example: transfer \$100 from my account to yours
- **Exactly-once**
  - Possible for certain classes of failures
  - Stub & service keep track (*durably*) of requests and responses
  - Example: bank cannot develop amnesia! 😊





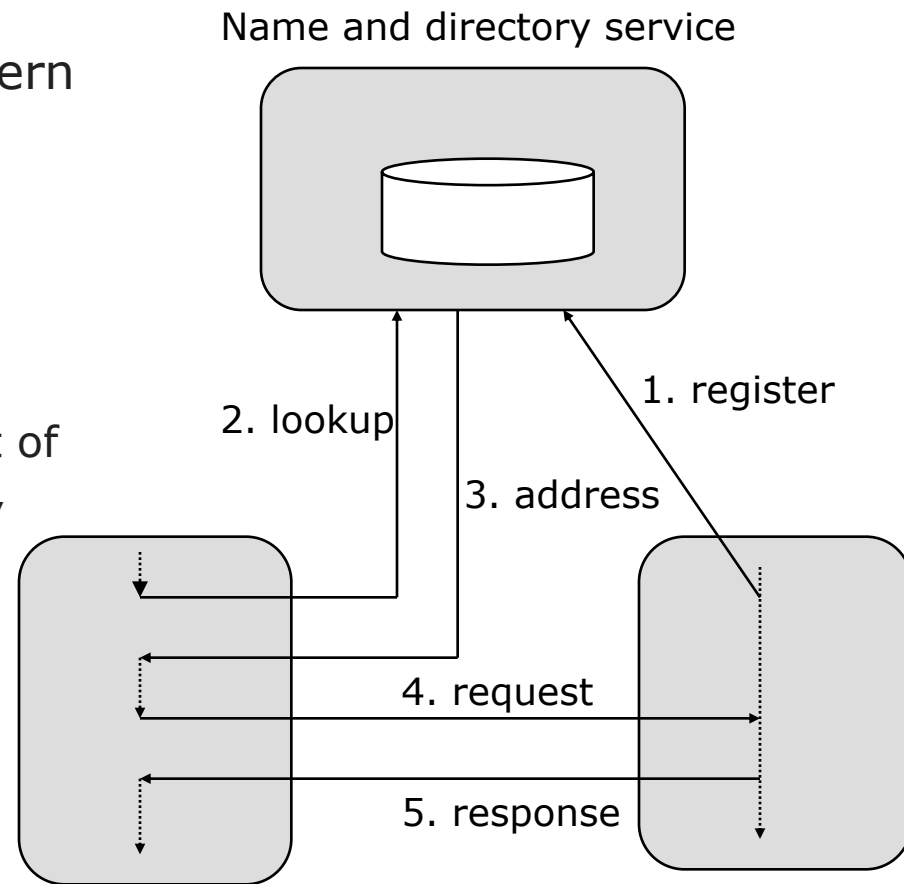
## How to achieve RPCs?

- Special-purpose ***request-reply protocol***, e.g., as in DNS
  - Developer must design protocol and marshalling scheme
- ***Classic RPC*** protocols, e.g., DCE, Sun RPC
  - Special APIs and schemes for marshalling
- ***RMI: Remote Method Invocation***
  - RPCs for methods in OO languages
  - Compiler-generated proxies
- ***Web Services***
  - Many modes of communication possible, including RPC-style communication
  - Tools available to compile proxies, e.g., JAX-WS
  - Generic marshalling (e.g., XML, JSON, Protocol Buffers) over HTTP transport
    - ***programming-language independence!***



# RPC and Naming

- Most basic extension to the synchronous interaction pattern
  - Avoid having to name the destination
  - Ask where destination is
  - Then bind to destination
- Advantages:
  - Development is independent of deployment properties (e.g., network address)
  - More flexibility:
    - Change of address
  - Can be combined with:
    - Load balancing
    - Monitoring
    - Routing
    - Advanced service search

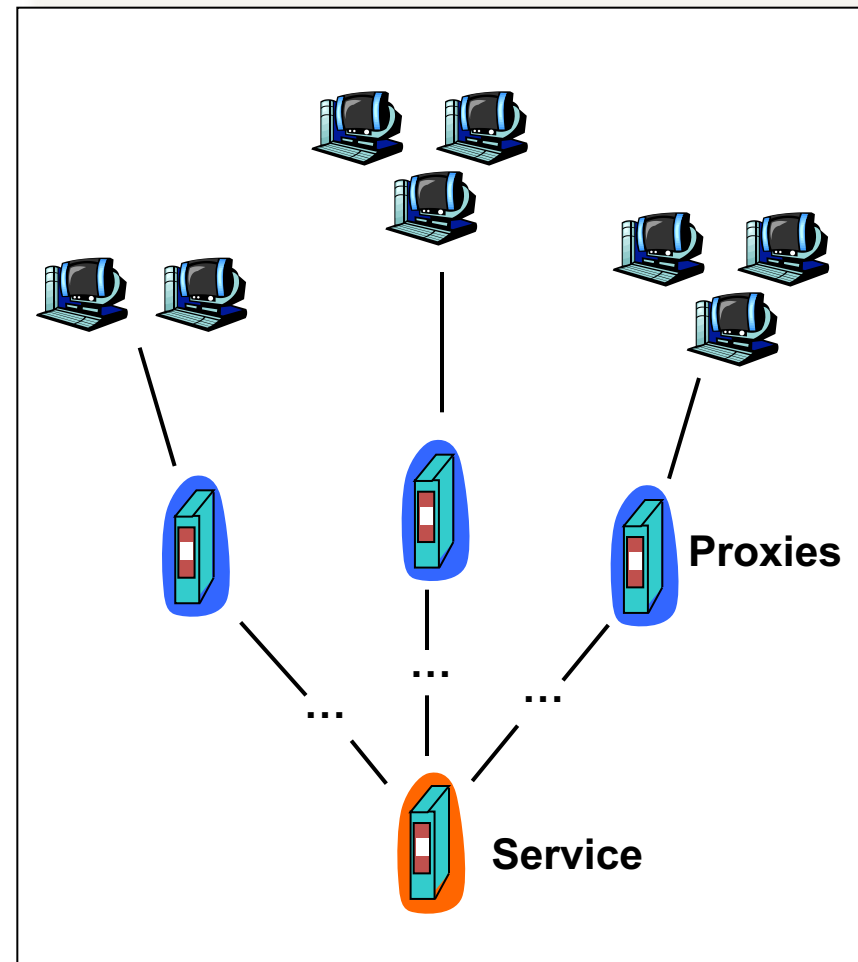


Source: Gustavo Alonso,  
ETH Zurich (partial)



## RPCs over HTTP

- Services widely exposed on the web, accessible via HTTP
- Why is this a good idea?
- Discuss how the following features of HTTP affect service interfaces, if at all:
  - Proxies
  - Persistent connections
  - Caching



# Common Issues in Designing Services

- **Consistency**
  - How to deal with *updates* from multiple clients?
- **Coherence**
  - How to refresh *caches* while respecting consistency?
- **Scalability**
  - What happens to resource usage if we increase the #clients or the #operations?
- **Fault Tolerance**
  - Under what circumstances will the service be unavailable?



## Other Examples of Services

- File systems: NFS, GFS
- Object stores: Dynamo, PNUTS
- Databases: pick your favorite relational DB ☺
- Configuration: Zookeeper
- Even whole computing clouds!
  - Infrastructure-as-a-service (IaaS), e.g., Amazon EC2, Rackspace, Windows Azure
  - Platform-as-a-service (PaaS), e.g., Windows Azure, Google AppEngine
  - Software-as-a-service (SaaS), e.g., Salesforce.com, Gmail
- And many, many others
- Differences in semantics are significant!





Questions so far?



# Abstractions, Implementation and Performance

- Let  $I_1$  and  $I_2$  be two implementations of an abstraction
- **Examples**
  - Web service with or without HTTP proxies
  - Virtual memory with or without paging
  - Transactions via concurrency or serialization



# Abstractions, Implementation and Performance

- Let  $I_1$  and  $I_2$  be two implementations of an abstraction
- **Examples**
  - Web service with or without HTTP proxies
  - Virtual memory with or without paging
  - Transactions via concurrency or serialization

How can we choose between  $I_1$  and  $I_2$ ?



## Performance Metrics

latency  
throughput  
scalability  
overhead  
utilization  
capacity

Discussion: What do these metrics mean?

# Common Issues with Performance Metrics

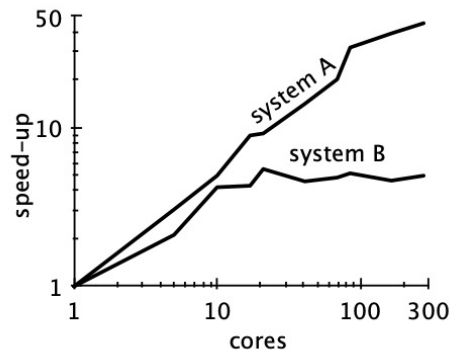
- Properties of ***resources*** and/or ***services***
  - Utilization, capacity
  - Overhead, throughput, latency
  - Scalability





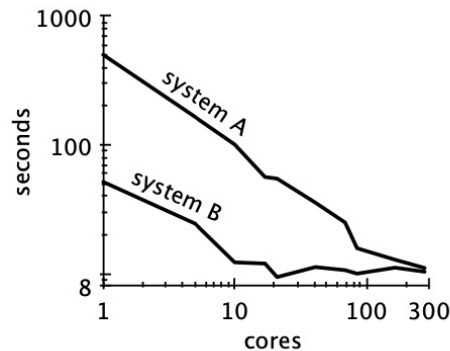
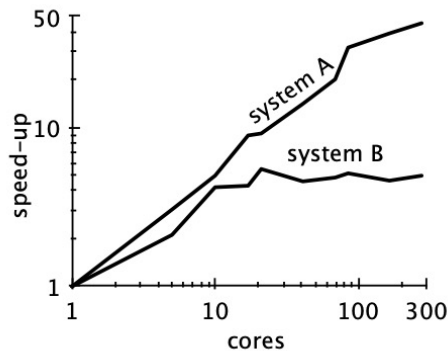
# Common Issues with Performance Metrics

- Properties of **resources** and/or **services**
  - Utilization, capacity
  - Overhead, throughput, latency
  - Scalability
- Considering scalability alone is dangerous



# Common Issues with Performance Metrics

- Properties of **resources** and/or **services**
  - Utilization, capacity
  - Overhead, throughput, latency
  - Scalability
- Considering scalability alone is dangerous

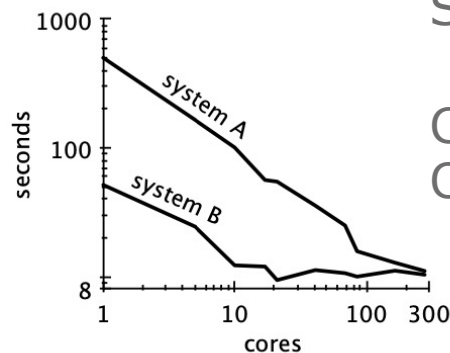
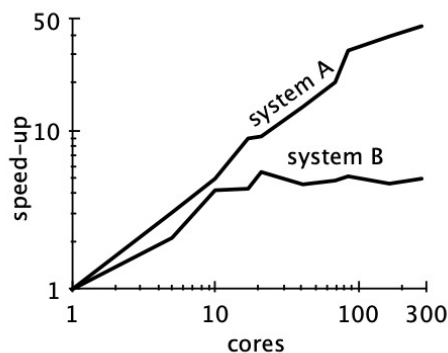


# Common Issues with Performance Metrics

- Properties of **resources** and/or **services**
  - Utilization, capacity
  - Overhead, throughput, latency
  - Scalability
- Considering scalability alone is dangerous

McSherry, Isard, Murray  
Scalability! But at what COST?

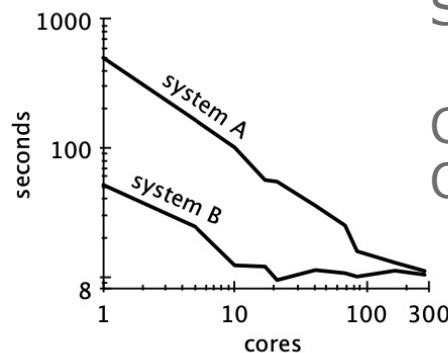
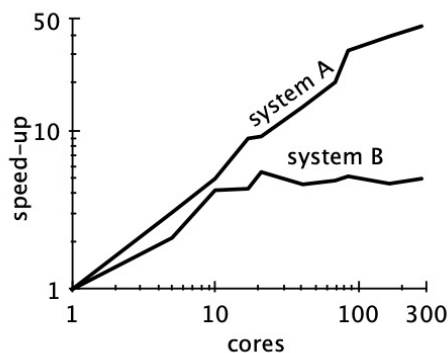
COST = Configuration that  
Outperforms a Single Thread



# Common Issues with Performance Metrics

- Properties of **resources** and/or **services**
  - Utilization, capacity
  - Overhead, throughput, latency
  - Scalability
- Considering scalability alone is dangerous

McSherry, Isard, Murray  
Scalability! But at what COST?

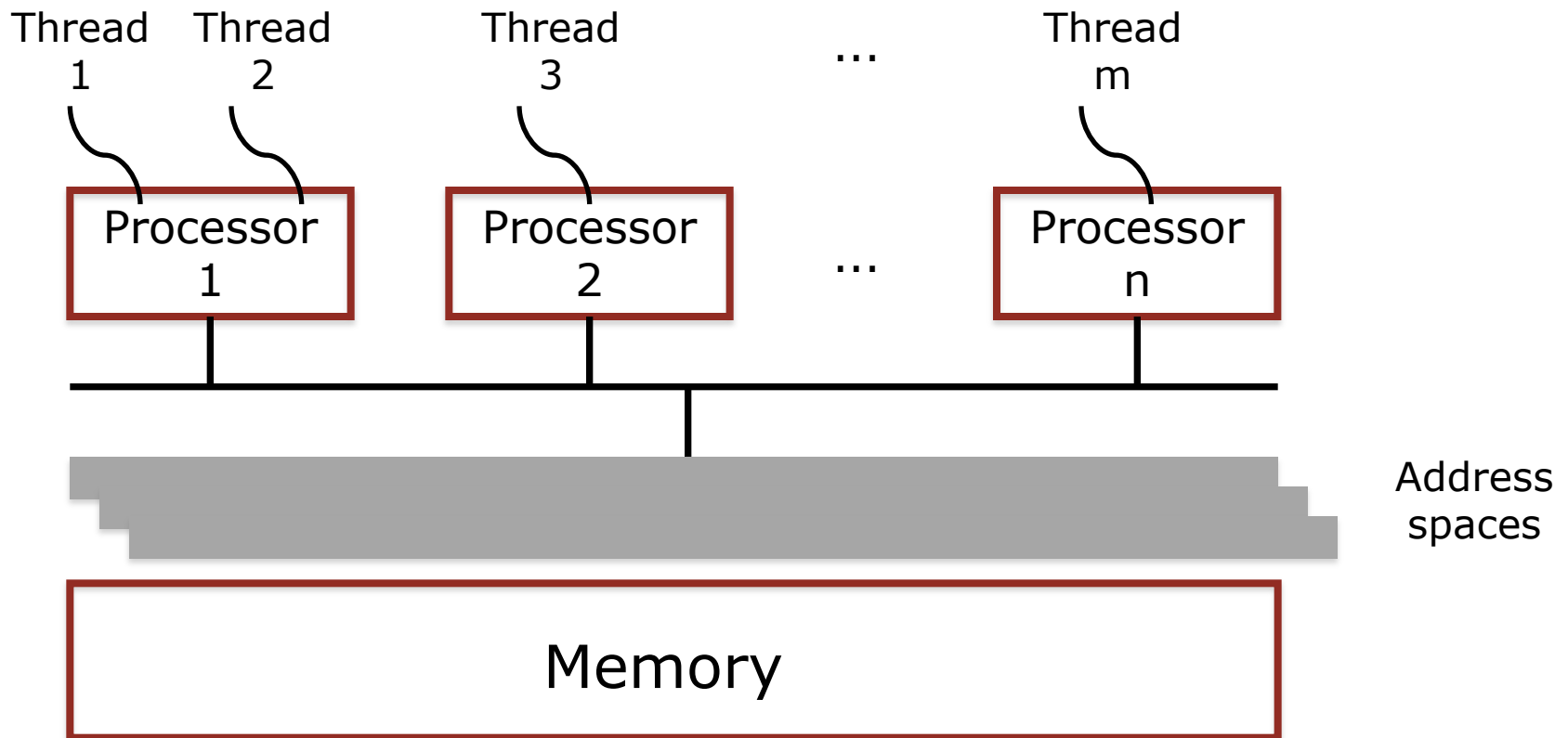


COST = Configuration that  
Outperforms a Single Thread

- Relationship between **latency** and **throughput**
  - In serial case:  $\text{latency} = 1 / \text{throughput}$
  - Not true when there is concurrency!

## Performance and Hardware Trends

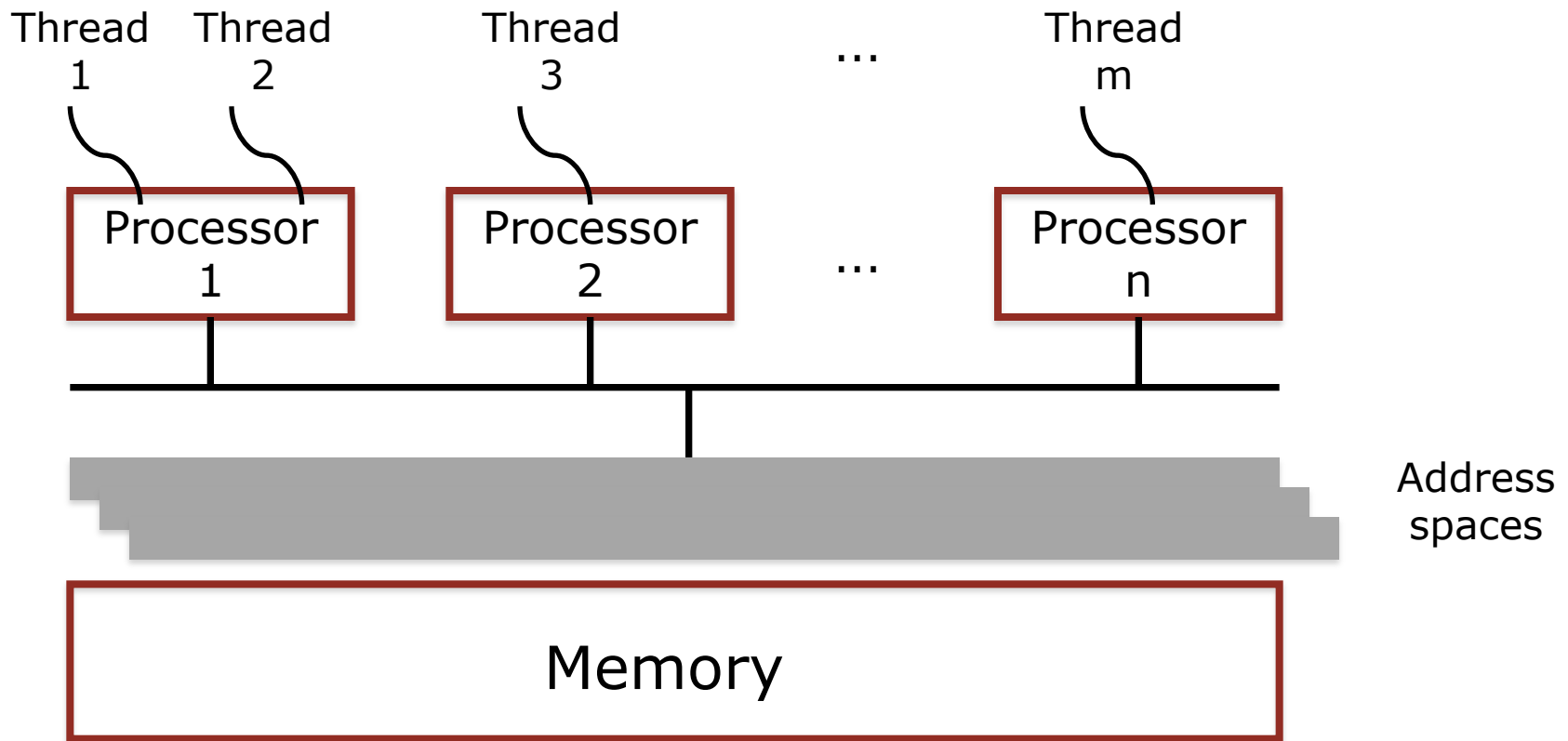
- OS-provided illusion of a computer:





## Performance and Hardware Trends

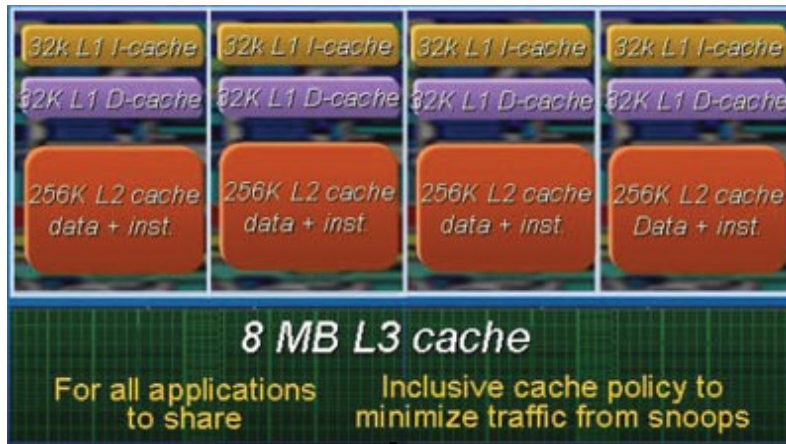
- OS-provided illusion of a computer:



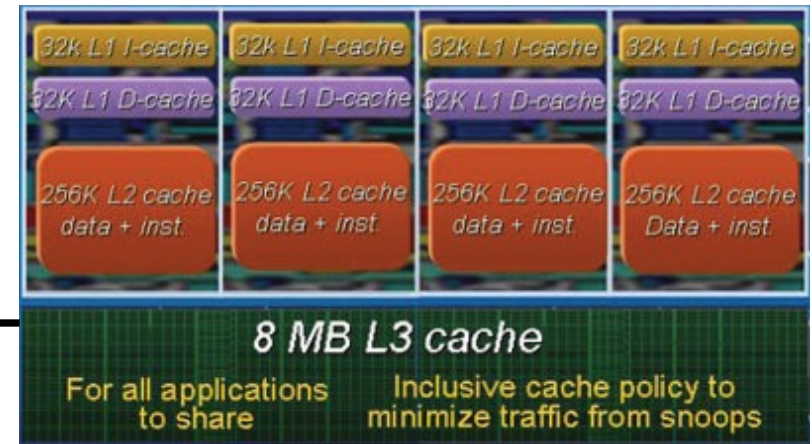
Does this picture look like a real computer?

# How about this one?

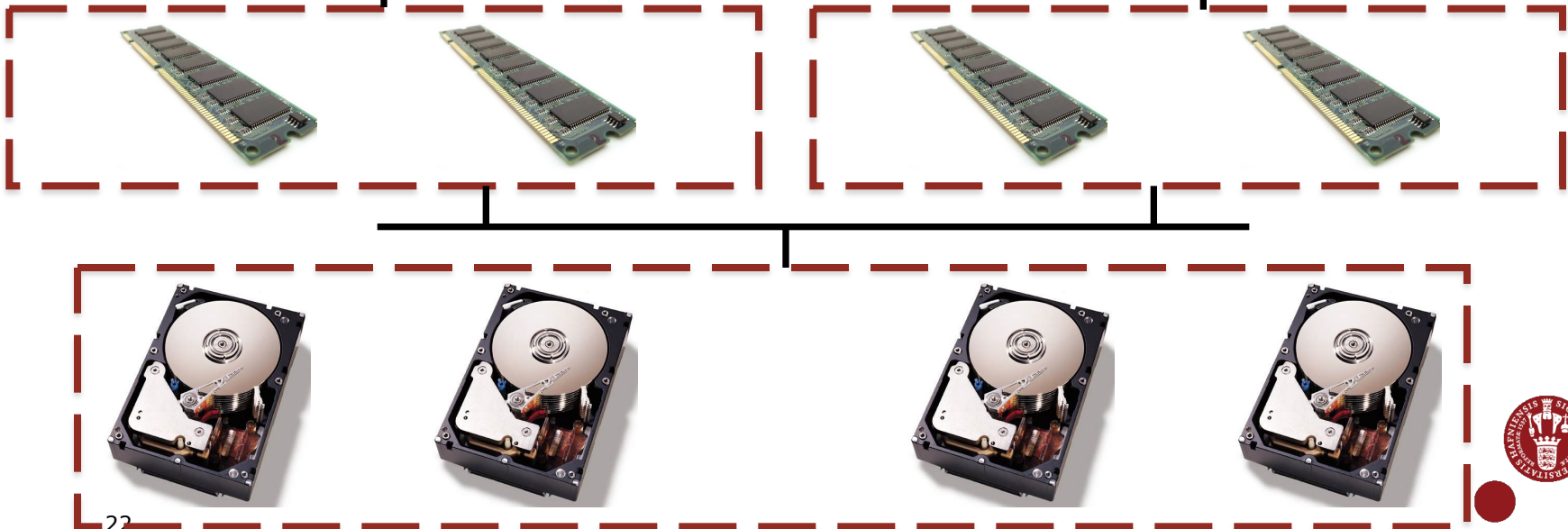
Processor 1



Processor n



...



But the picture is not to scale!

Size of  
last-level cache

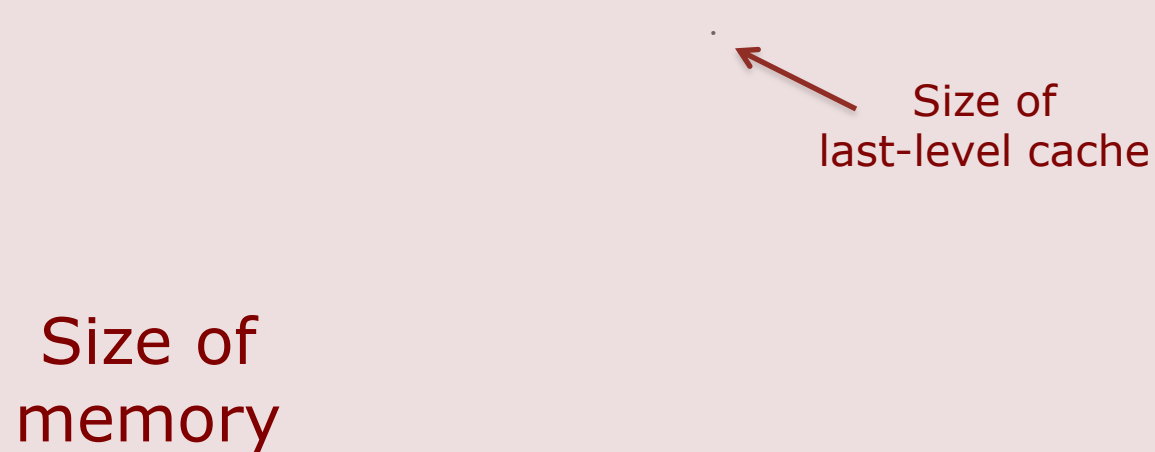


But the picture is not to scale!

Size of  
memory

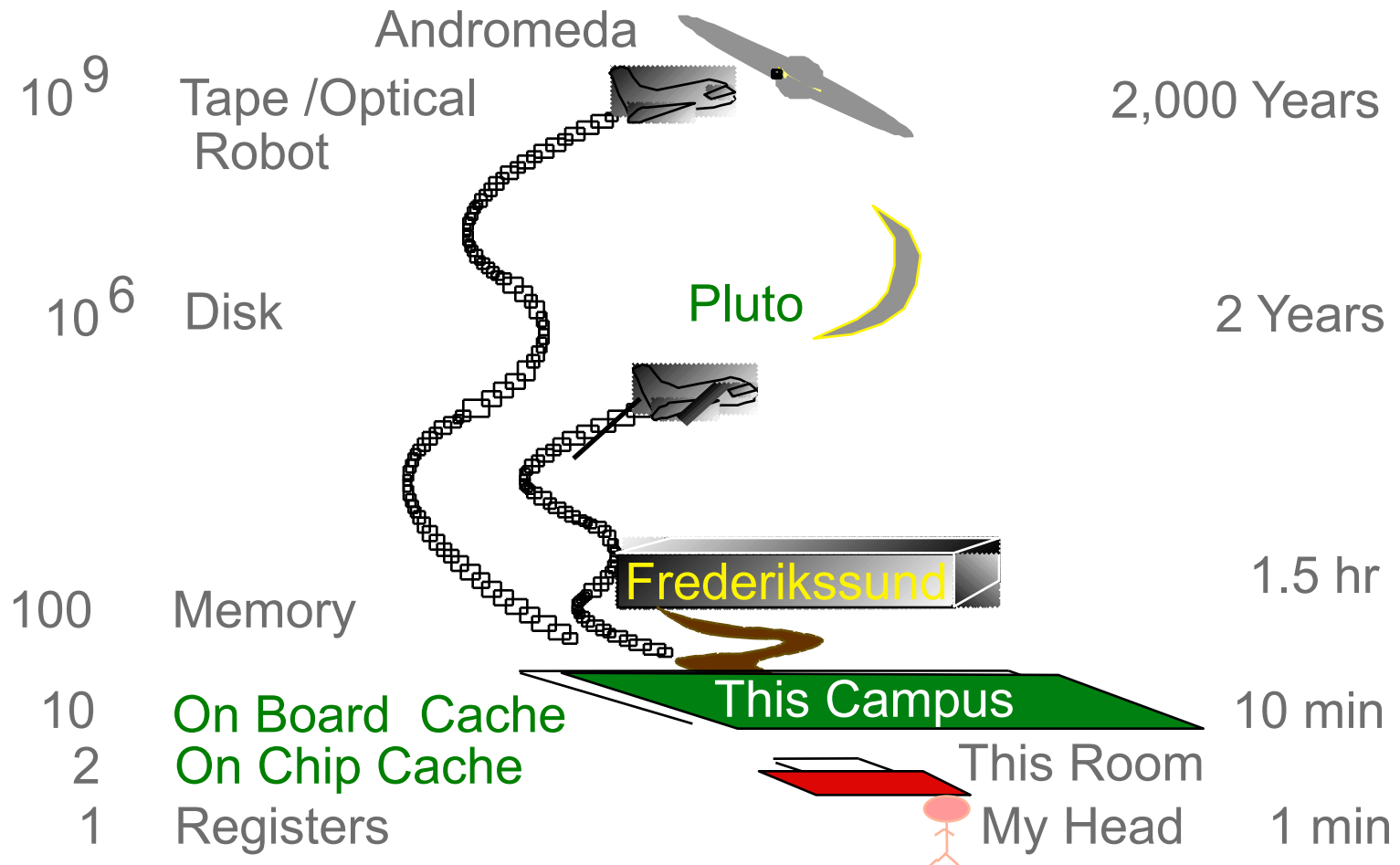
Size of  
last-level cache

But the picture is not to scale!



What about the size of disk?

# Storage Hierarchy

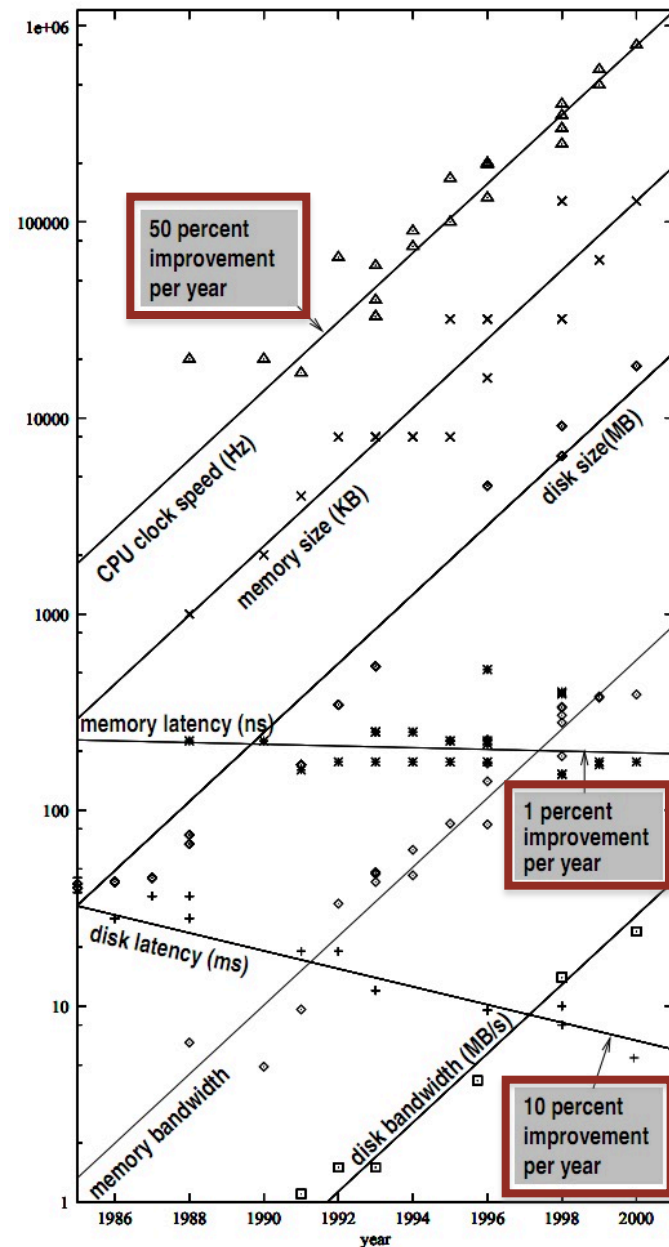


Source: Gray (partial)



## And only getting worse...

- Riding Moore's Law
  - CPU clock speed.  
(not anymore ☺)
  - Memory size
  - Memory bandwidth
  - Disk size
  - Disk bandwidth
- Going way slower
  - Memory latency
  - Disk latency
- What does that do to random accesses?

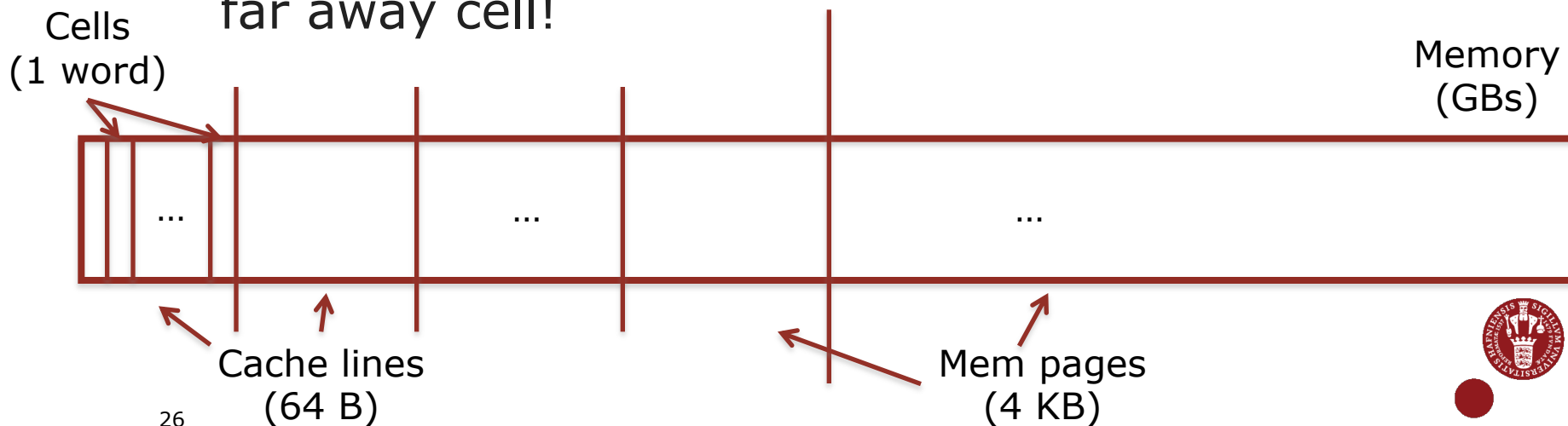


Source: Boncz (partial)



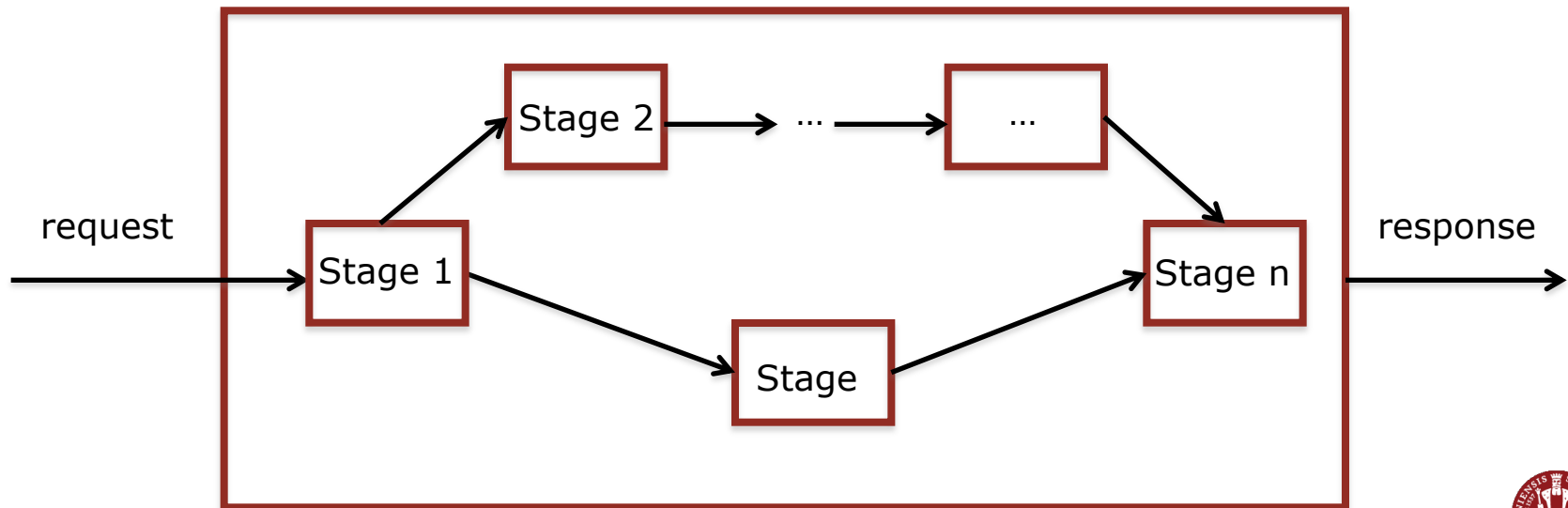
# RAM = NQSRAM?!

- What we call  
**R**andom **A**ccess **M**emory  
actually behaves as  
**N**ot-**Q**uite-**S**o-**R**andom **A**ccess **M**emory  
because of the memory hierarchy
- Access to nearby cell **much faster** than to a far away cell!



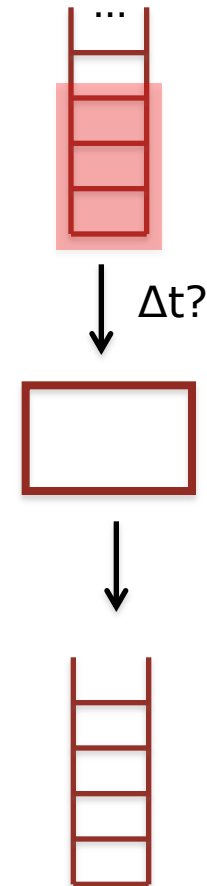
## How can we improve performance?

- Fast-path coding
  - Split processing into two code paths
  - One optimized path for common requests → fast path
  - One slow but comprehensive path for all other requests → slow path
  - Caching is an example of fast-path coding



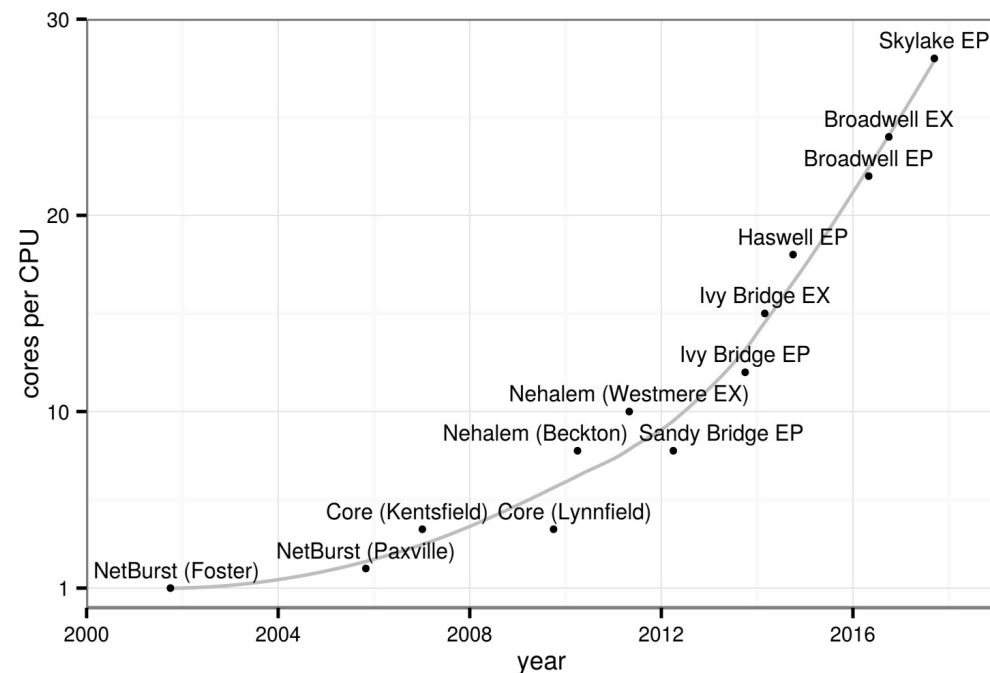
## How can we improve performance?

- Batching
  - Run multiple requests at once
  - Example: batch I/Os and use elevator algorithm
  - May improve latency **and** throughput
- Dallying
  - Wait until you accumulate some requests and then run them
  - Example: group commit
  - May improve throughput when used together with batching, but typically incurs a latency penalty



## Parallelism is only increasing

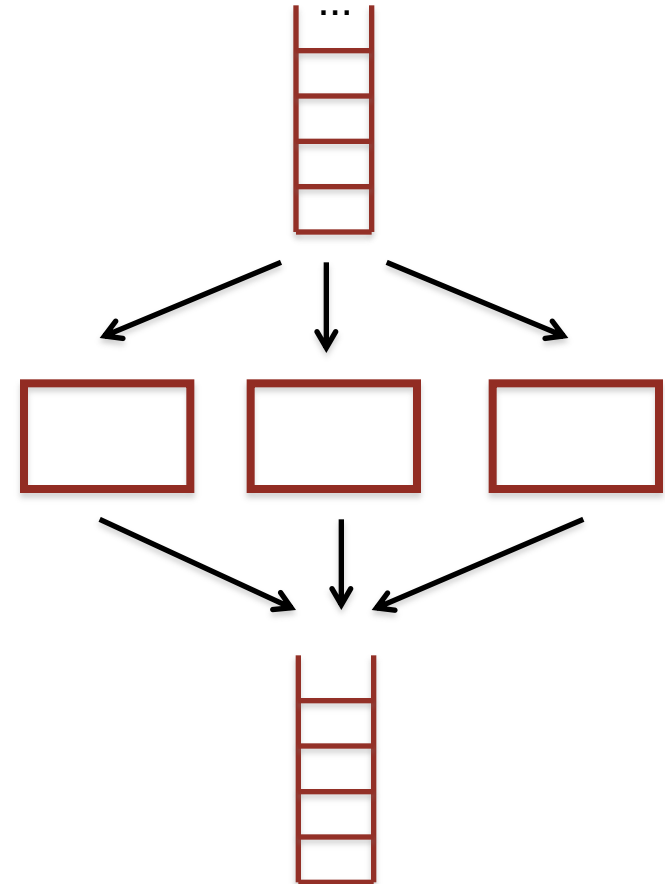
- Moore's Law demise is in the horizon
  - CPU clock speed not anymore improving
- But number of cores still increasing in general-purpose CPUs
  - Also, specialized hardware gaining traction (check out the PMPH elective)
- How can we leverage multiple cores?



Source: Leis (partial)

## How can we improve performance?

- Concurrency
  - Run multiple requests in different threads
  - Example: different web requests run in different threads or even servers
  - May improve both throughput **and** latency, but must be careful with locking, correctness
  - Can be hidden under abstractions, e.g., MapReduce and transactions



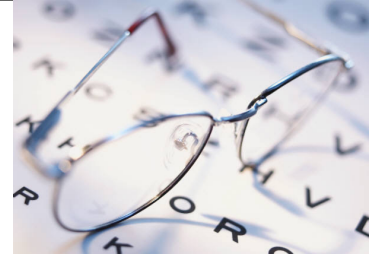
Remember RPC?

# How can we improve performance?

- Speculation, i.e., predict the future 😊
  - Guess the next requests and run them in advance
  - Example: prefetching
  - May overlap expensive operations, instead of waiting for their completion



## What should we learn today?



- Recognize and explain modular designs with clients and services
- Predict the functioning of service calls under different RPC semantics and failure modes
- Identify different mechanisms to achieve RPCs
- Implement RPC services with an appropriate mechanism, such as web services
- Explain performance metrics such as latency, throughput, overhead, utilization, capacity, and scalability
- List common hardware parameters that affect performance
- Apply performance improvement techniques, such as concurrency, batching, dallying, and fast-path coding