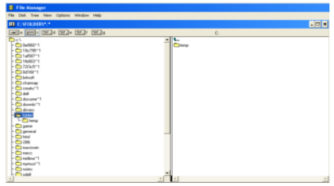
Modularity through Clients and Services, RPC
Techniques for Performance

PCSD, Marcos Vaz Salles

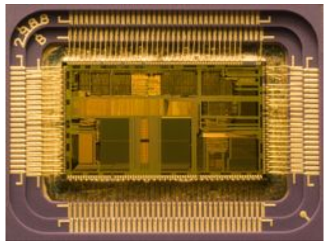
Do-it-yourself Recap: Fundamental abstractions

- Which were the three fundamental abstractions?
- What were their APIs?

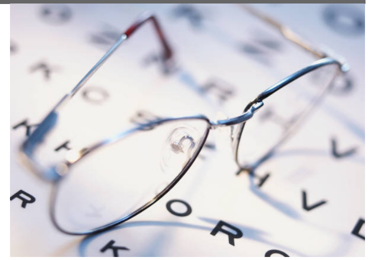
- Must these abstractions be implemented in a single node or can they be distributed? Give an example!



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



Source: Saltzer & Kaashoek & Morris (partial)



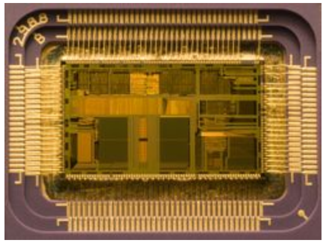
What should we learn today?

- 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

Interperters

- Interpreter
  - Instruction repertoire
  - Environment
  - Instruction pointer

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



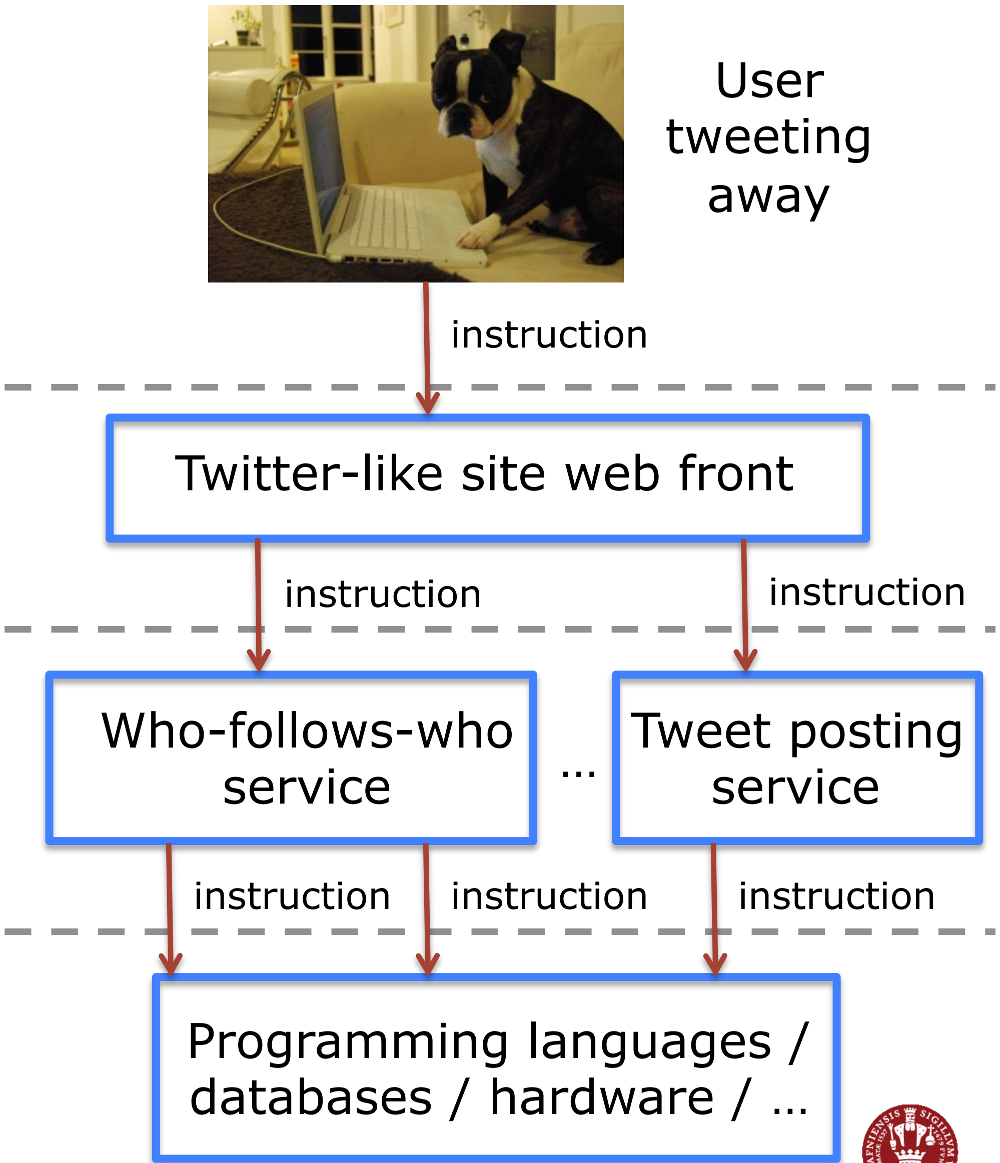
```
procedure INTERPRET()
        do forever
                instruction ← READ(instruction_pointer)
                perform instruction in environment context
                if interrupt_signal = TRUE then
                instruction_pointer ← entry of INTERRUPT_HANDLER
                environment ← environment of INTERRUPT_HANDLER
```

Your own programs often implement specialized interpreters!

Layers and Modules

- Interpreters often organized in layers

- Modules
  - *Book glossary*: Components that can be separately designed / implemented / managed / replaced
  - "Instructions" of higher-level interpreters
  - Recursive: Can be whole interpreters themselves!

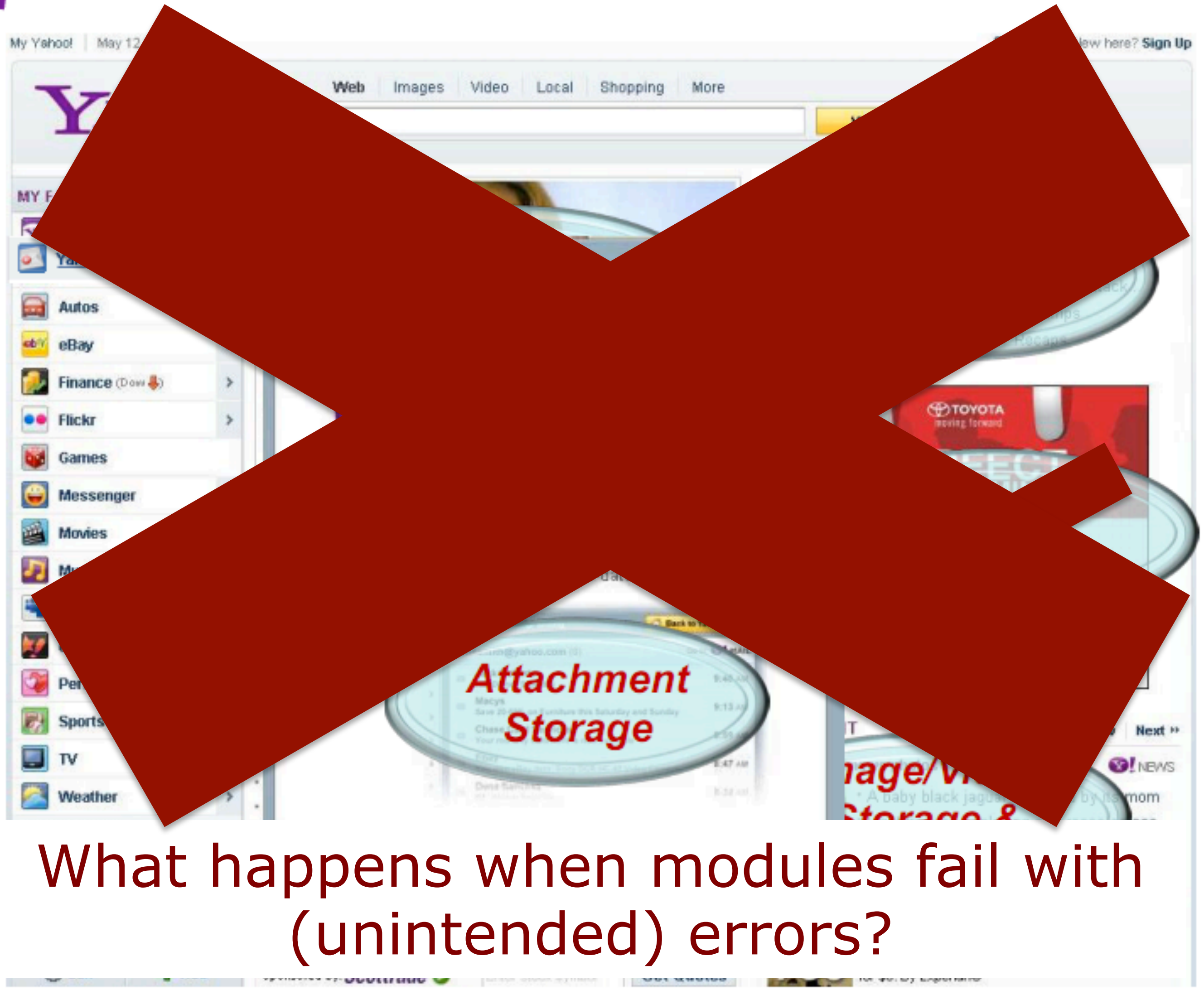


Cloud-Power @ Yahoo!

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

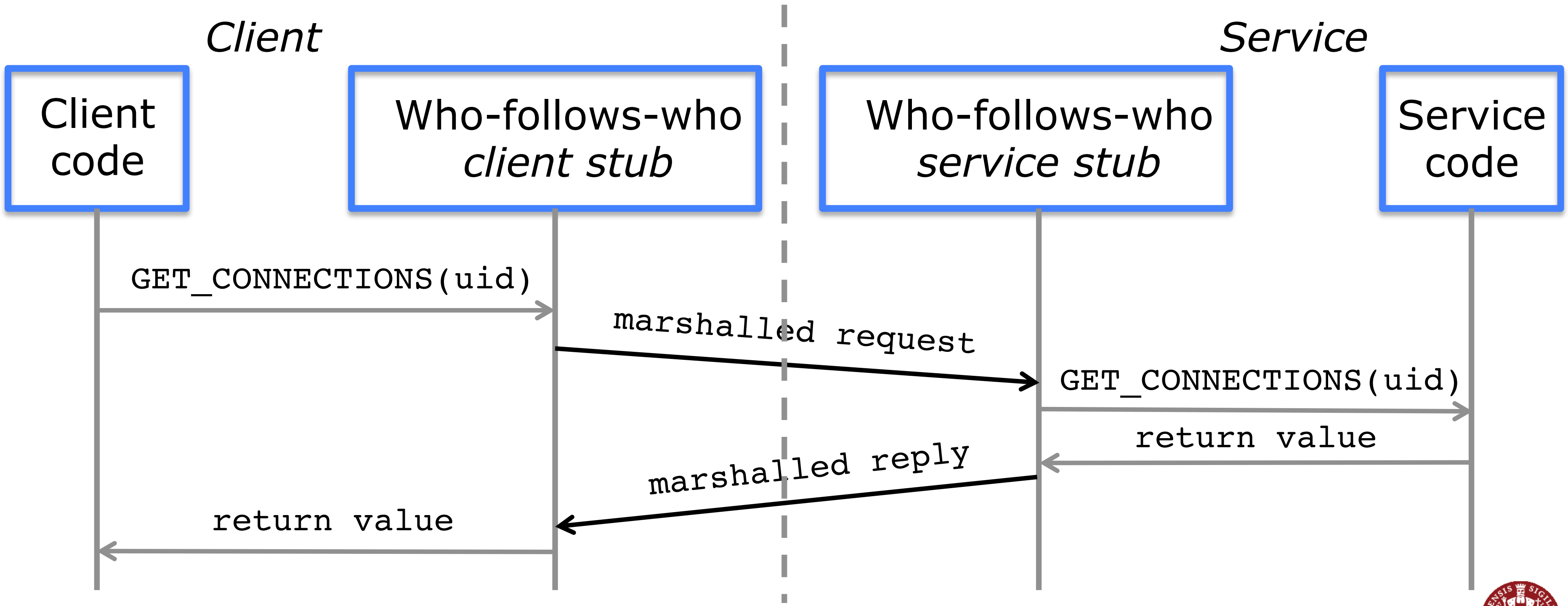
OS Virtualization

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

Even more techniques for fault tolerance later in the course

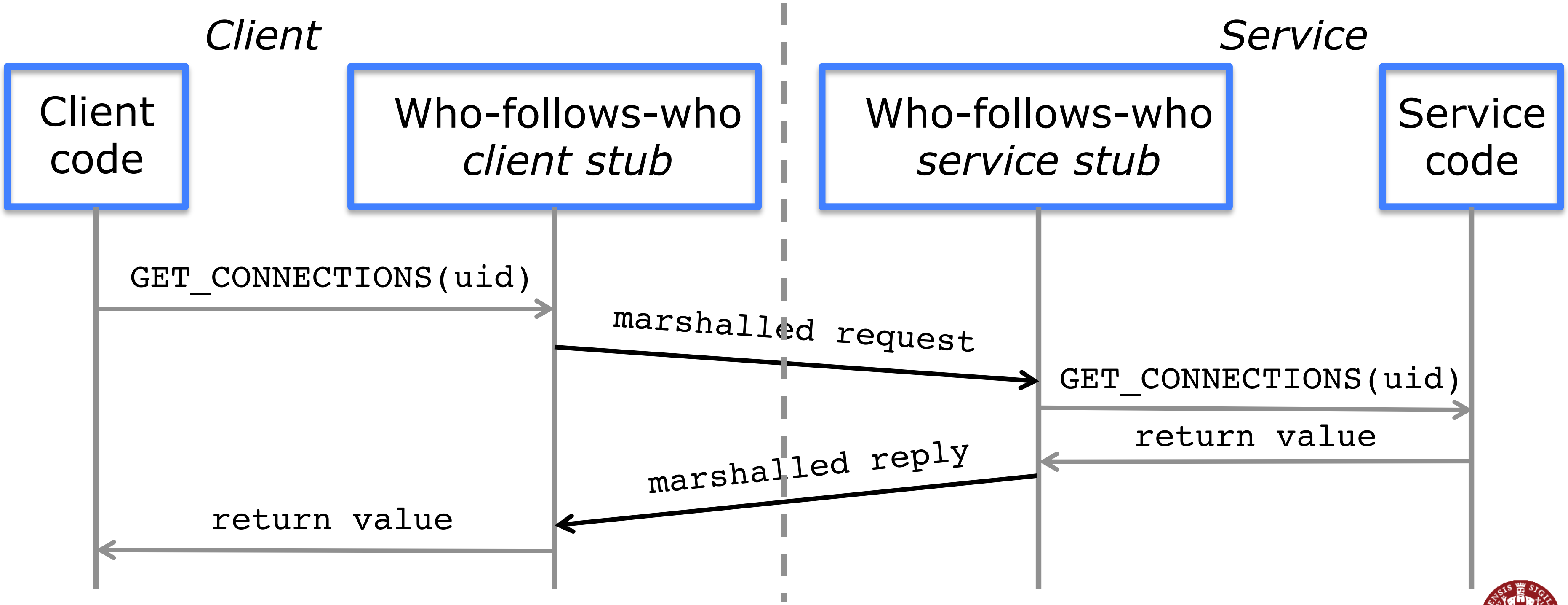
RPC: Remote Procedure Call

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



RPC: Remote Procedure Call

- How do RPC semantics differ from local procedure calls?
- What can go wrong? How can you fix it?



RPC Semantics

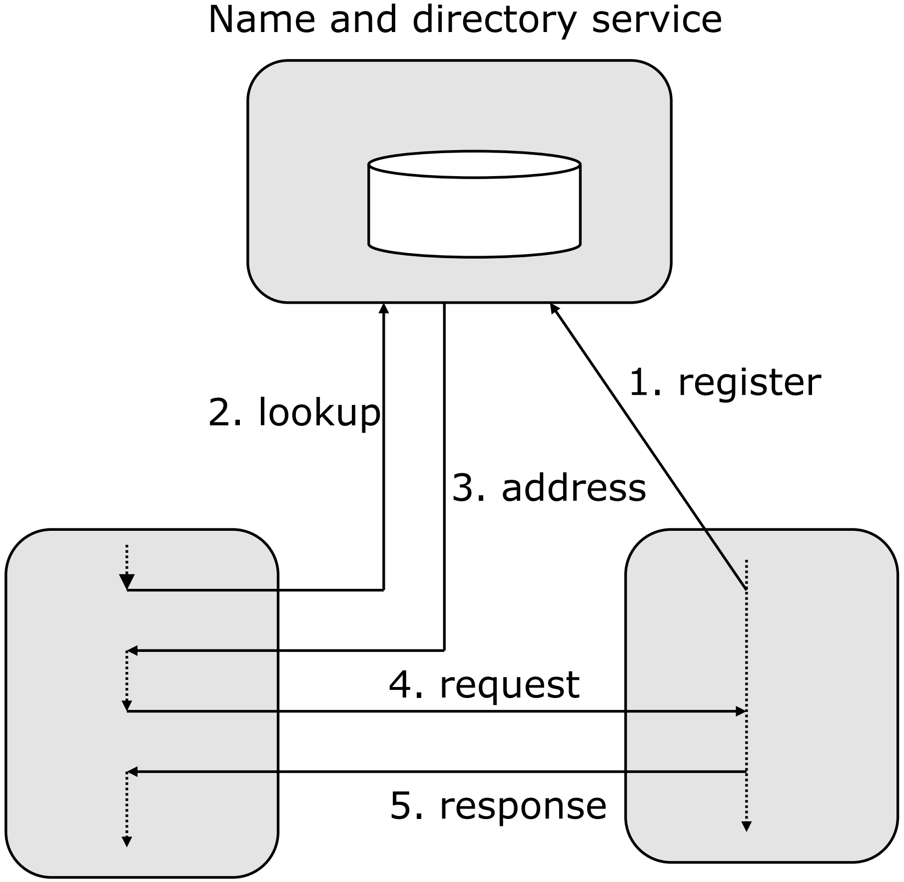
- **At-least-once**
  - Operation is *idempotent*
    - Naturally occurs if side-effect free
  - Stub just retries operation → failures can still occur!
  - Example: calculate SQRT
- **At-most-once**
  - Operation does have side-effects
  - Stub must ensure duplicate-free transmission
  - Example: transfer $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)

RPC Example: Domain Name Service (DNS)

- DNS: Translates domain names into IP addresses
  - `www.diku.dk` → `130.225.96.3` → `d.root-servers.net` → `128.8.10.90` → `p.nic.dk` → `204.61.216.36` → `dns2.diku.dk` → `130.225.96.4` → `web-aggregator.diku.dk` → `130.225.96.108`

- DNS typically implemented as a request-reply service, i.e., equivalent to RPC at high level

RPC Example: Domain Name Service (DNS)

- ***DNS interface***

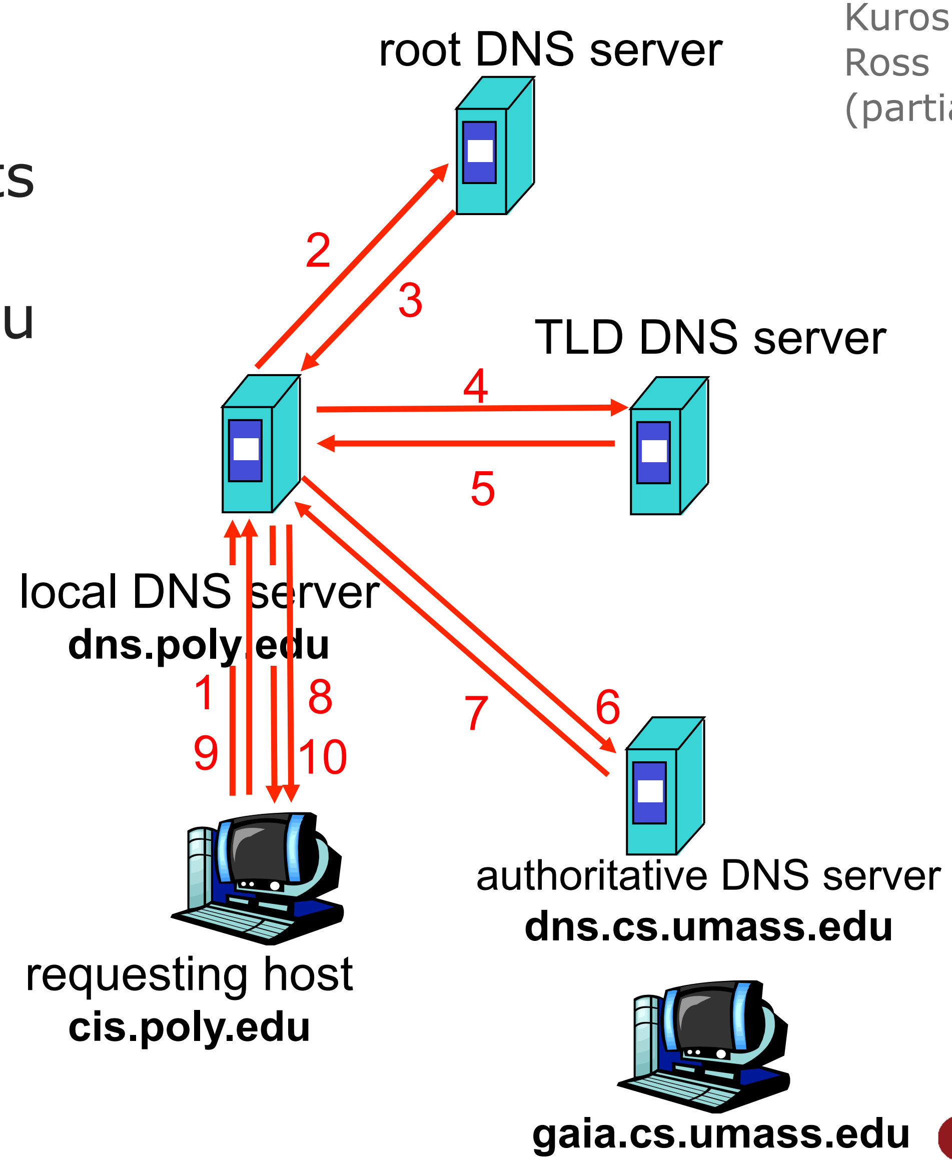
```
RESOLVE(domain_name, recursive) → value
```

- ***DNS Implementation***
  - DNS does not keep whole name database in a single server → **hierarchy of servers**
    - Not surprisingly, integrates **naming** into the service itself! ☺
  - DNS uses **delegation**
    - Repeated RPCs to different servers
    - Recursive resolution and caching on some servers

DNS Operation and Caching

Source: Kurose & Ross (partial)

- host at cis.poly.edu wants IP address for gaia.cs.umass.edu



Iterated query

- contacted server replies with name of server to contact
- ***Caching*** at local server can improve performance *significantly*
- But what about ***semantics***?

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

How can we choose between I_1 and I_2?

Performance Metrics

Discussion: What do these metrics mean?

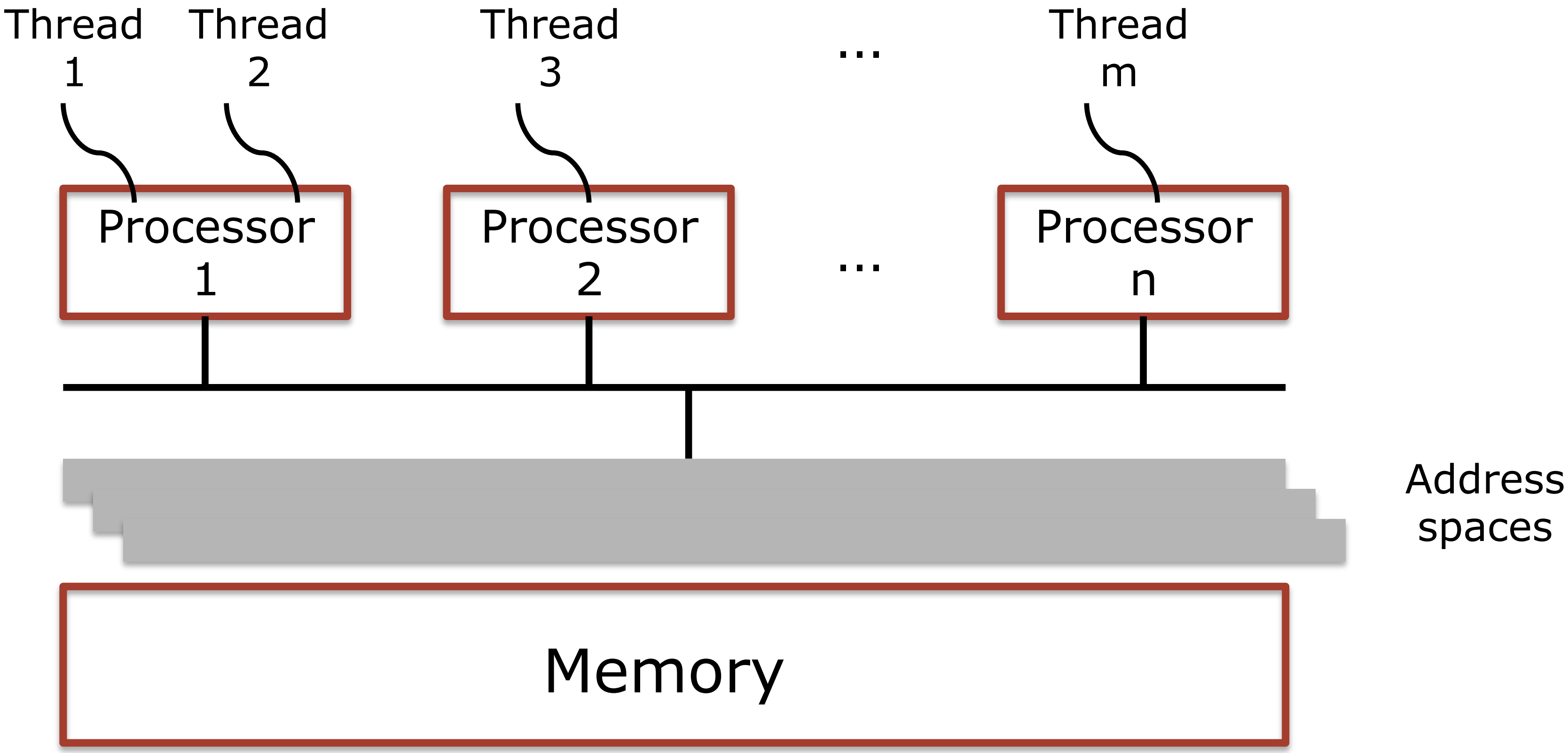
Common Issues with Performance Metrics

- Properties of ***machines*** vs. properties of ***programs***
  - Utilization, capacity
  - Overhead, throughput
  - Latency, scalability

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

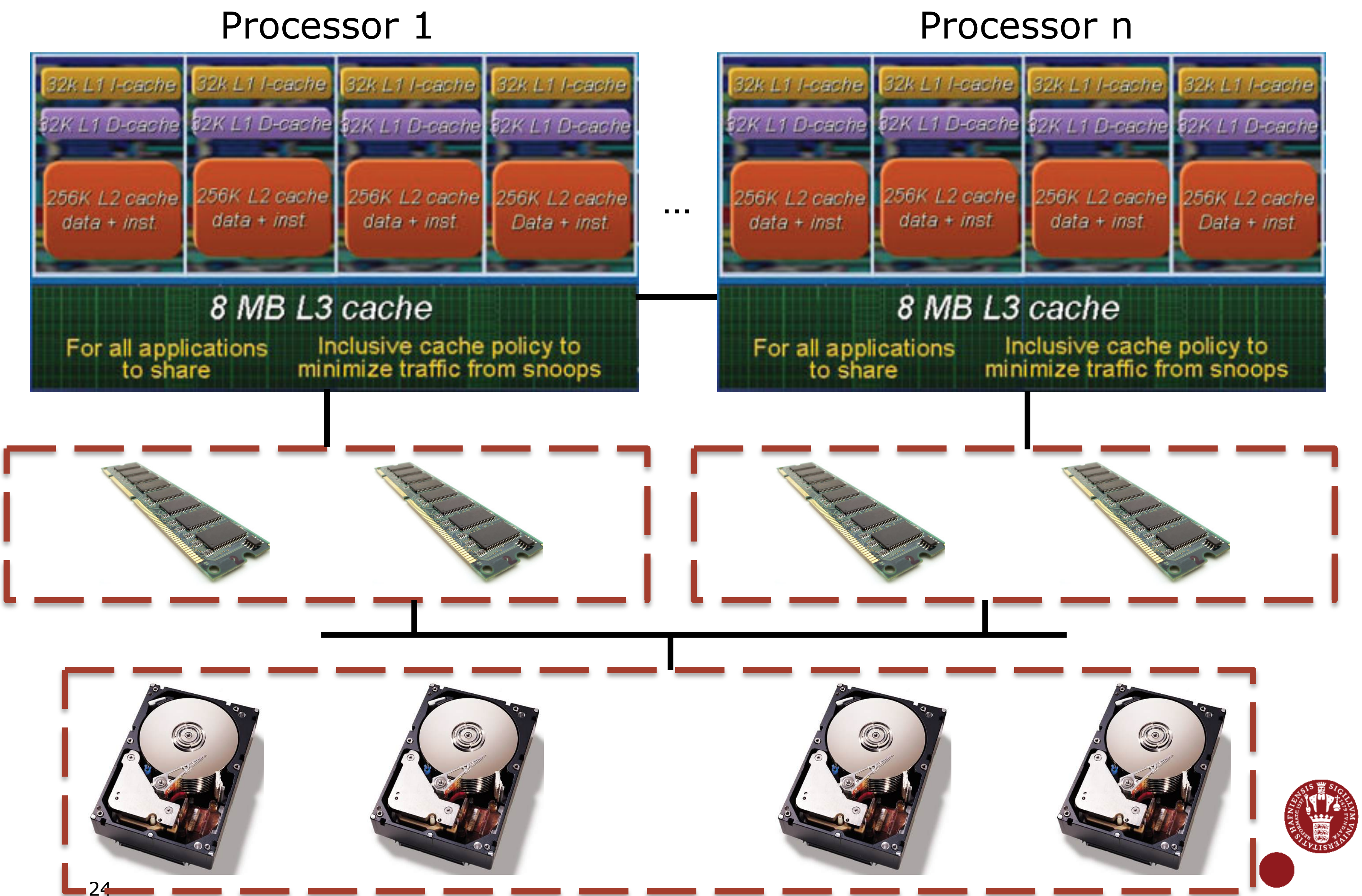
Performance and Hardware Trends

- OS-provided illusion of a computer:



Does this picture look like a real computer?

How about this one?



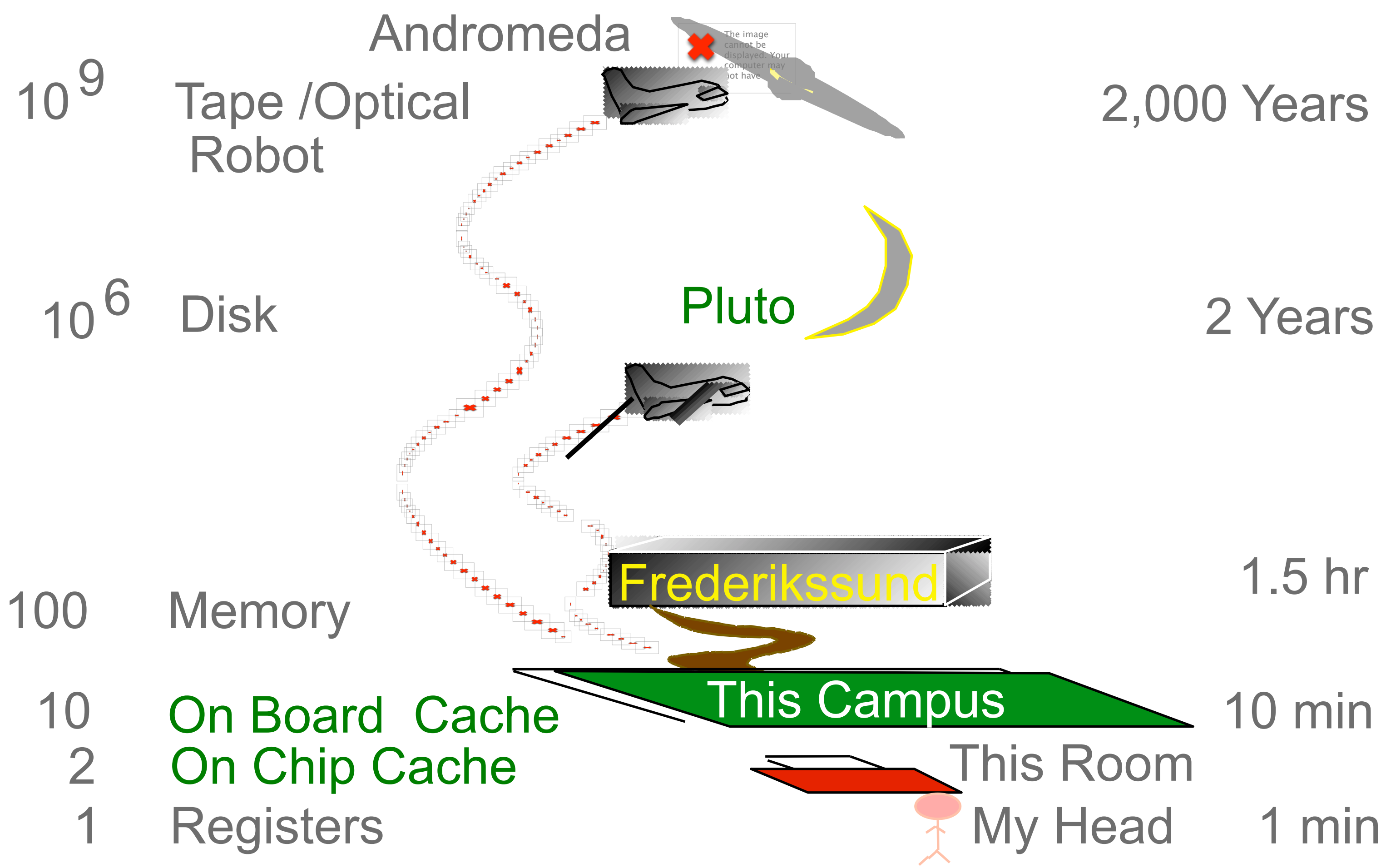
But the picture is not to scale!

Size of
last-level cache

Size of
memory

What about the size of disk?

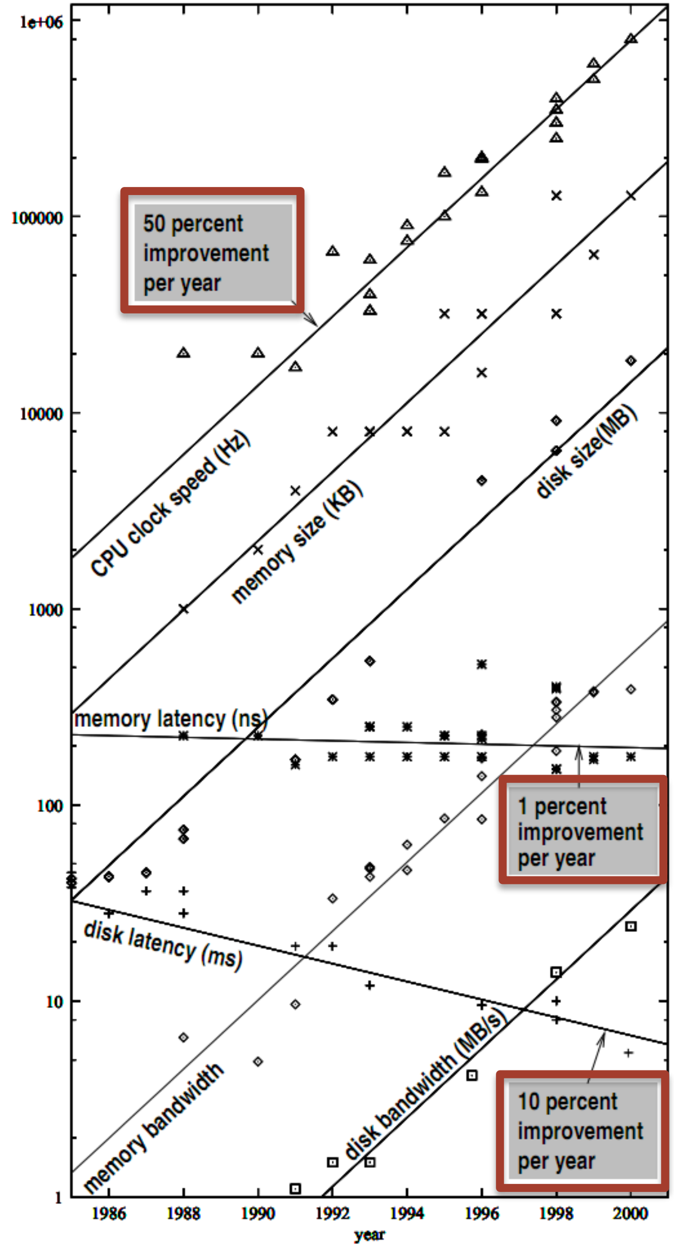
Storage Hierarchy



Source: Gray (partial)

And only getting worse…

- Riding Moore's Law
  - CPU clock speed
  - 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

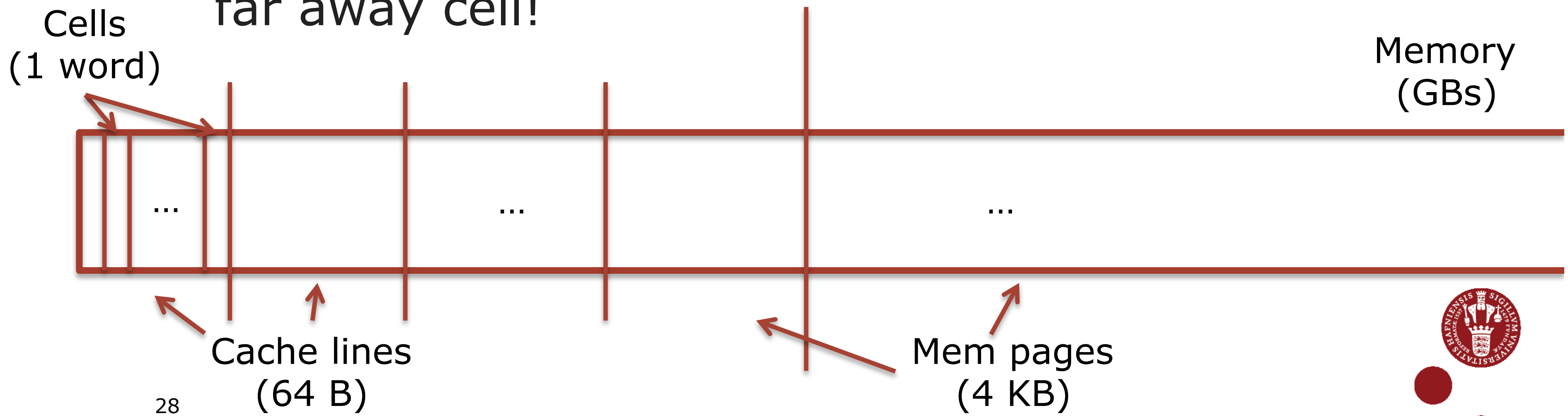
  Random **A**ccess **M**emory

  actually behaves as

  Not-**Q**uite-**S**o-**R**andom **A**ccess **M**emory

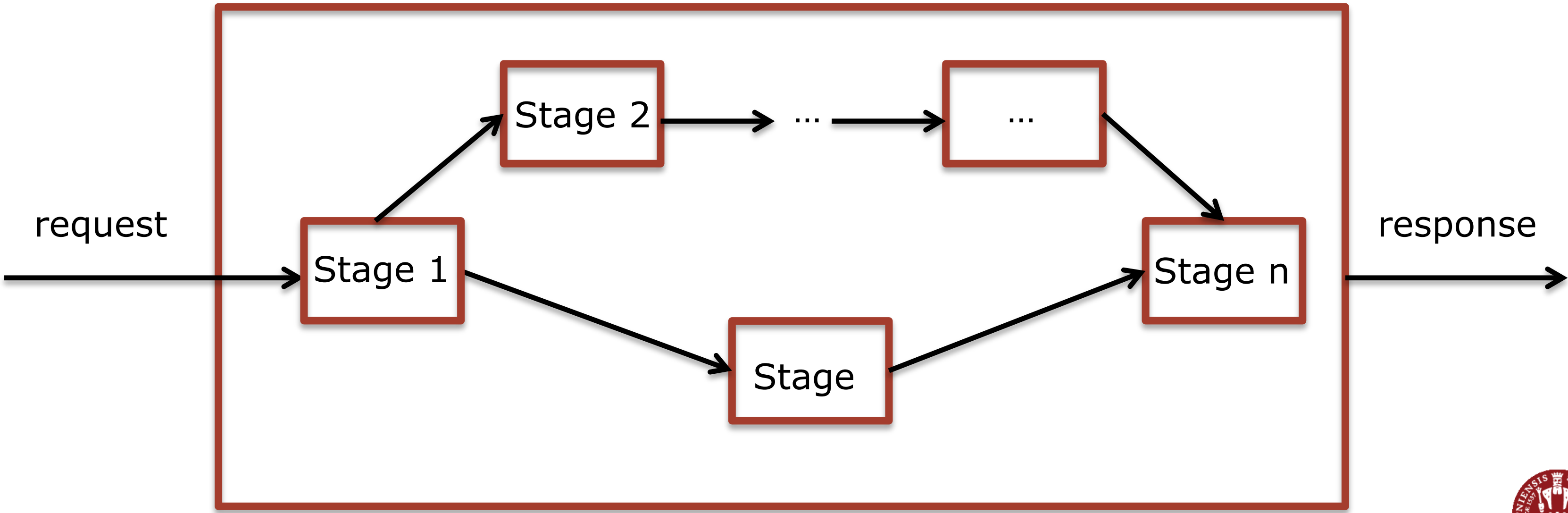
  because of the memory hierarchy

- Access to nearby cell **much faster** than to a far away cell!



How can we improve performance?

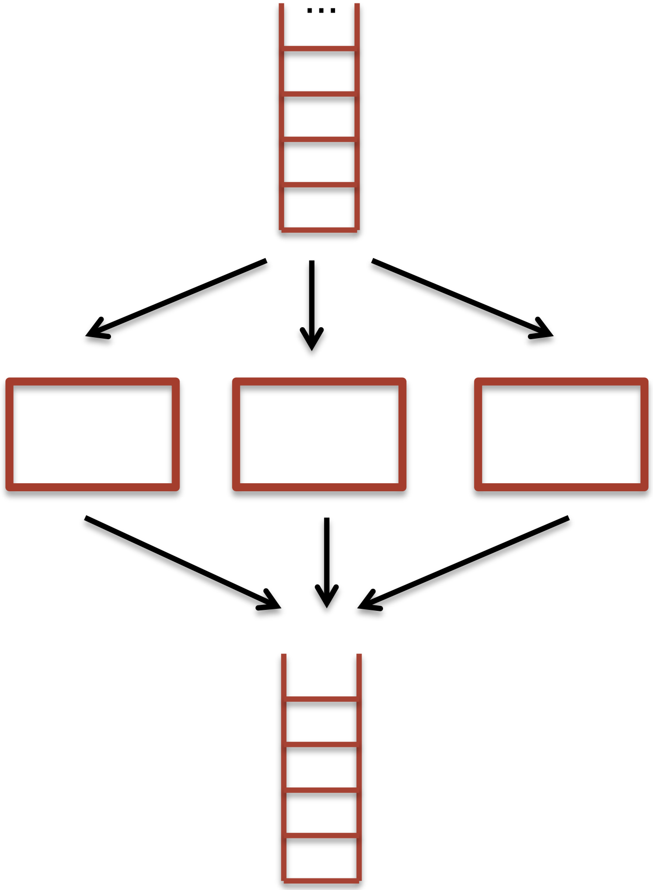
- Fast-path coding
  - Split processing into two code paths
  - One optimized path for common requests → fast path
  - One slow but comprehensive path for all other requests → slow path
  - Caching is an example of fast-path coding



Source: Saltzer & Kaashoek & Morris (partial)

How can we improve performance?

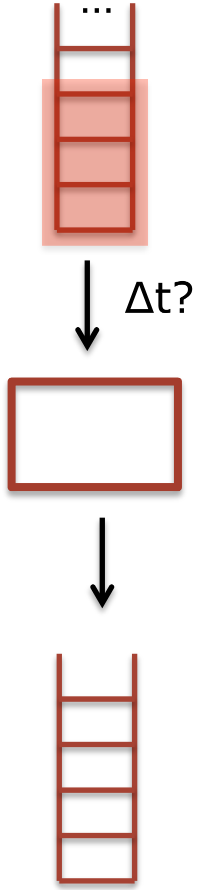
- Concurrency
  - Run multiple requests in different threads
  - Example: different web requests run in different threads or even servers
  - May improve both throughput **and** latency, but must be careful with locking, correctness
  - Can be hidden under abstractions, e.g., MapReduce and transactions



Remember RPC?

How can we improve performance?

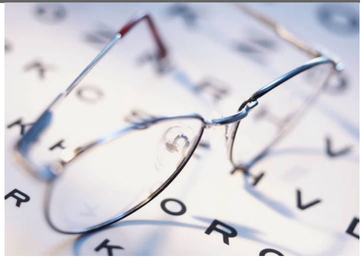
- Batching
  - Run multiple requests at once
  - Example: batch I/Os and use elevator algorithm
  - May improve latency **and** throughput
- Dallying
  - Wait until you accumulate some requests and then run them
  - Example: group commit
  - May improve throughput when used together with batching, but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



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

Sounds good with reads, but can you speculate writes?



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