What not where

Peter Alvaro

Twizzler

an Operating System for Far Out Memory Hierarchies

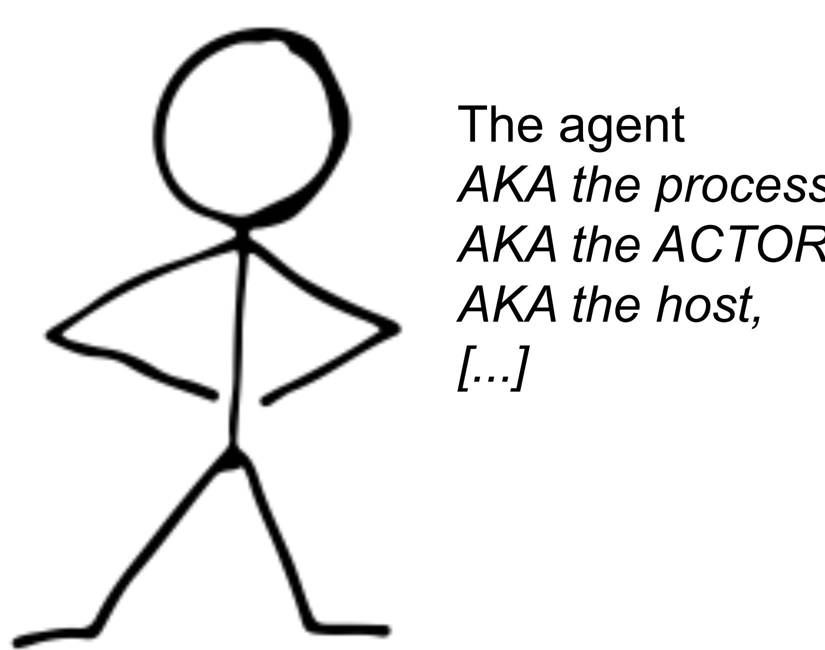
Outline

1. Twizzler
   a. The state of the art
   b. Trends: far out memories
   c. The data-centric operating system
   d. Organizing memory in Twizzler
   e. Two case studies
2. Data-centric security
3. Distribution
4. Sharing - the good and the bad
5. A new programming model for long-lived data and short-lived compute

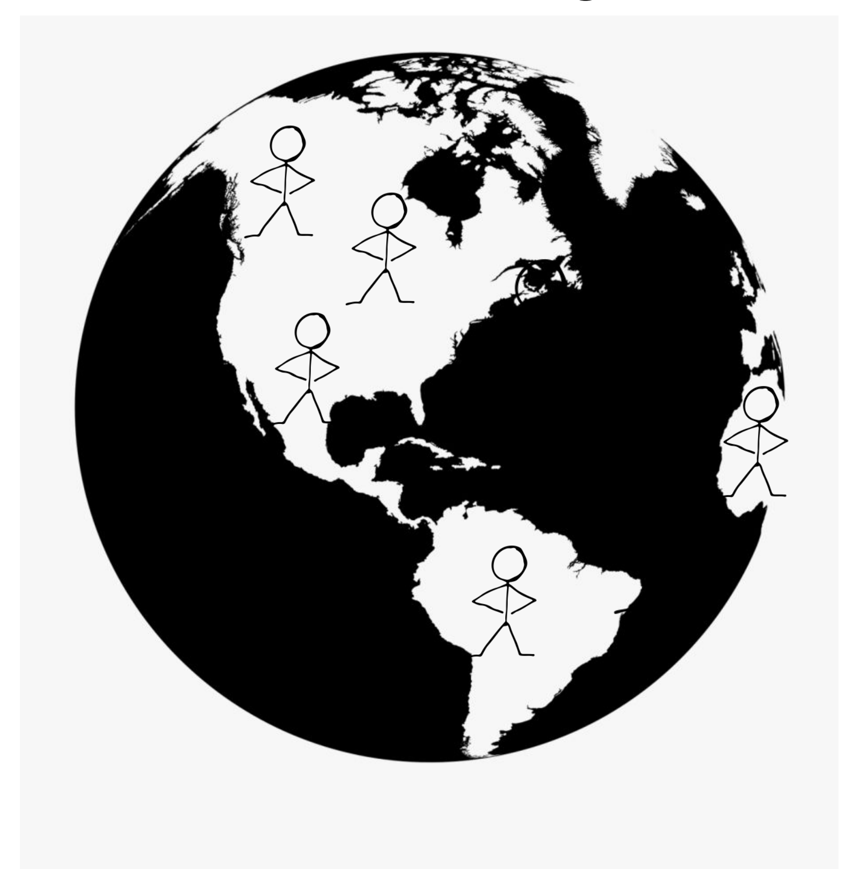
The state of the art

The state of the art

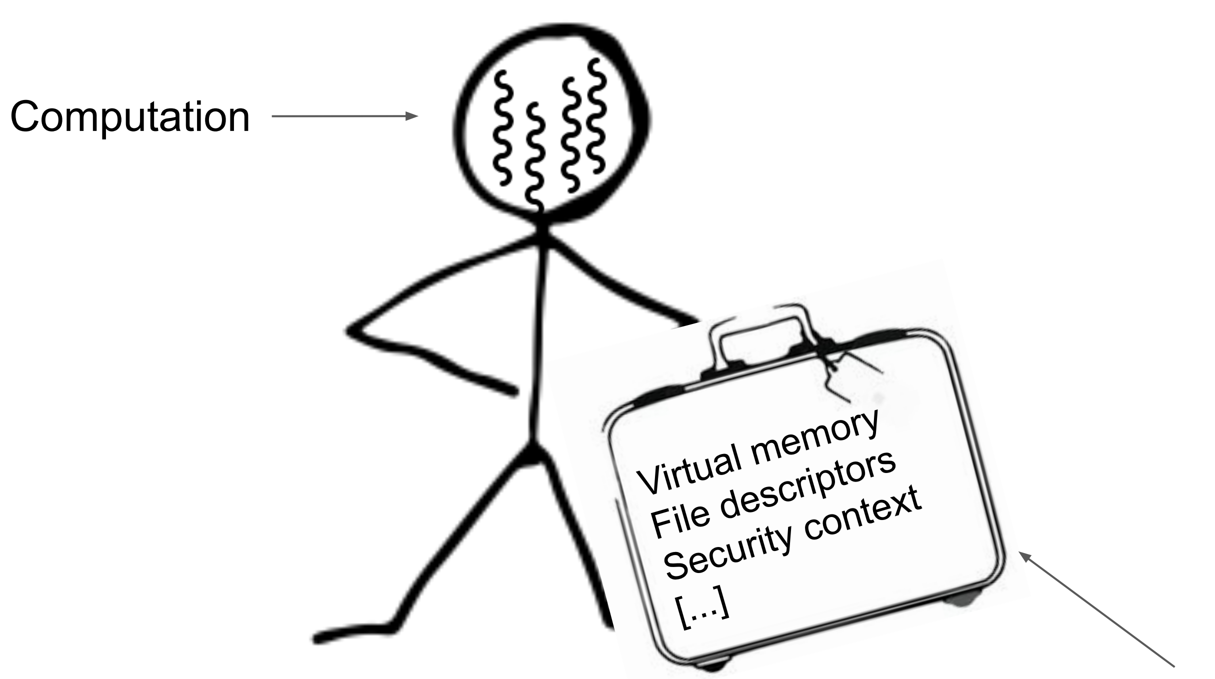
What are computers for?



It's all about agents

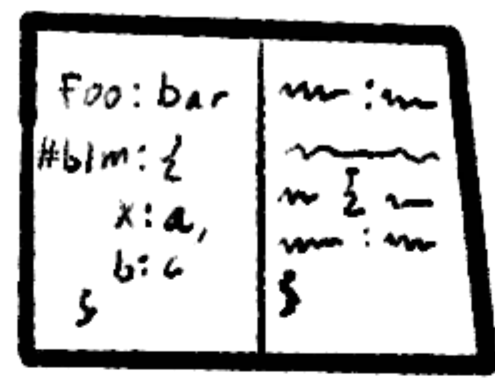


What is a process?

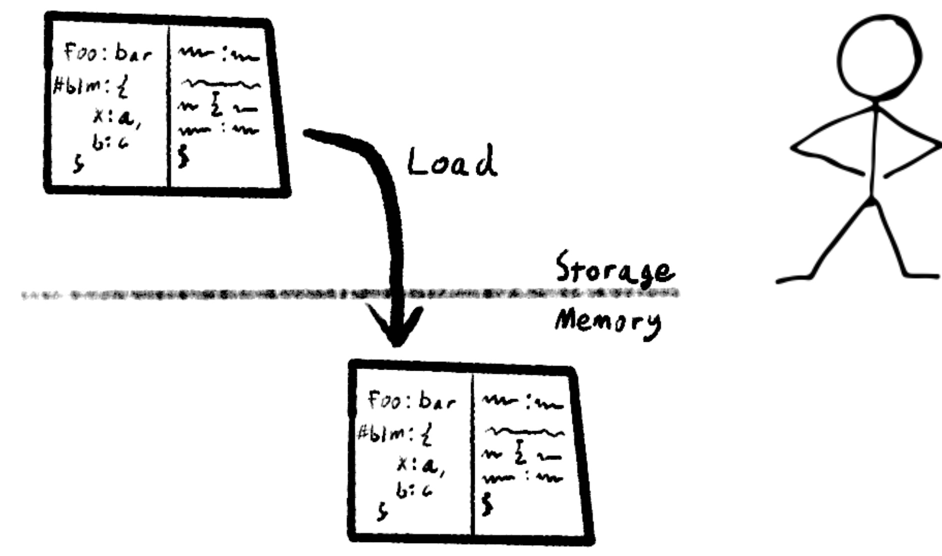


OK, but what's a process *really for?*

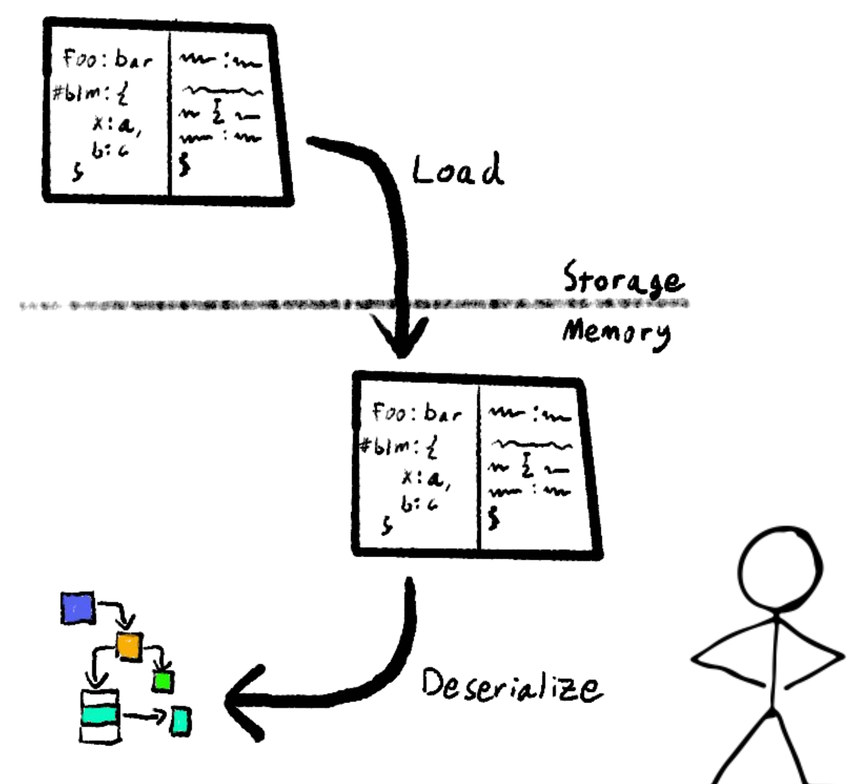
The state of the art: Systems programming



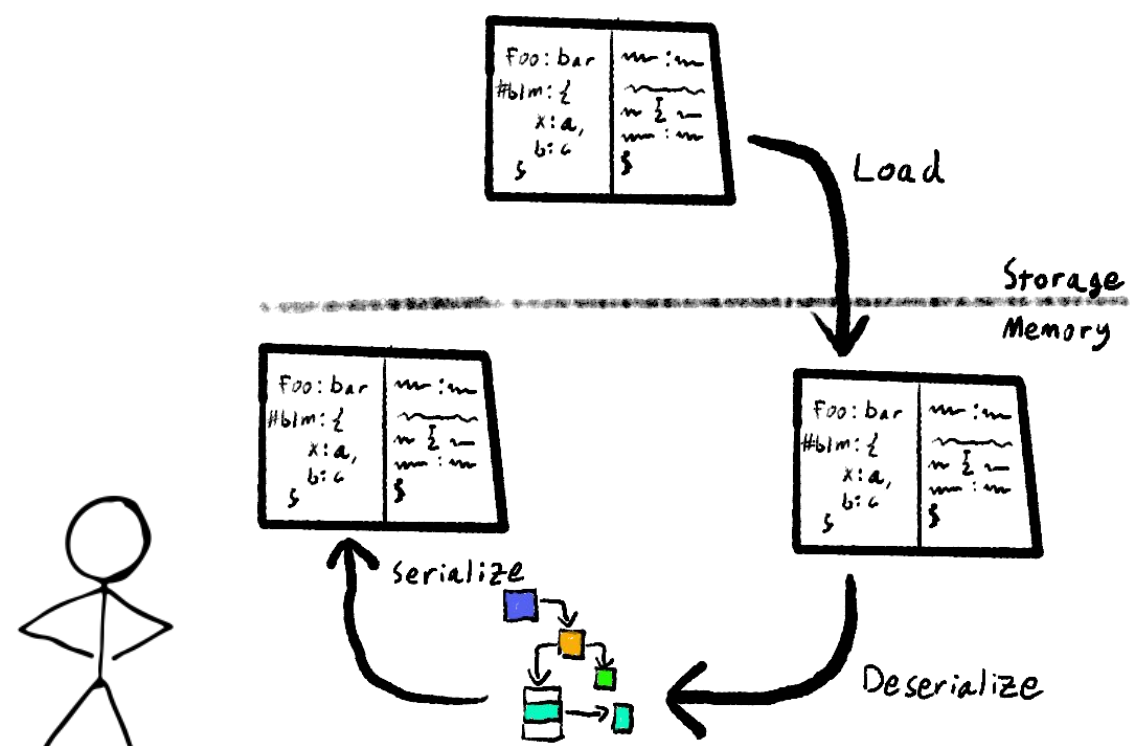
The state of the art: Systems programming



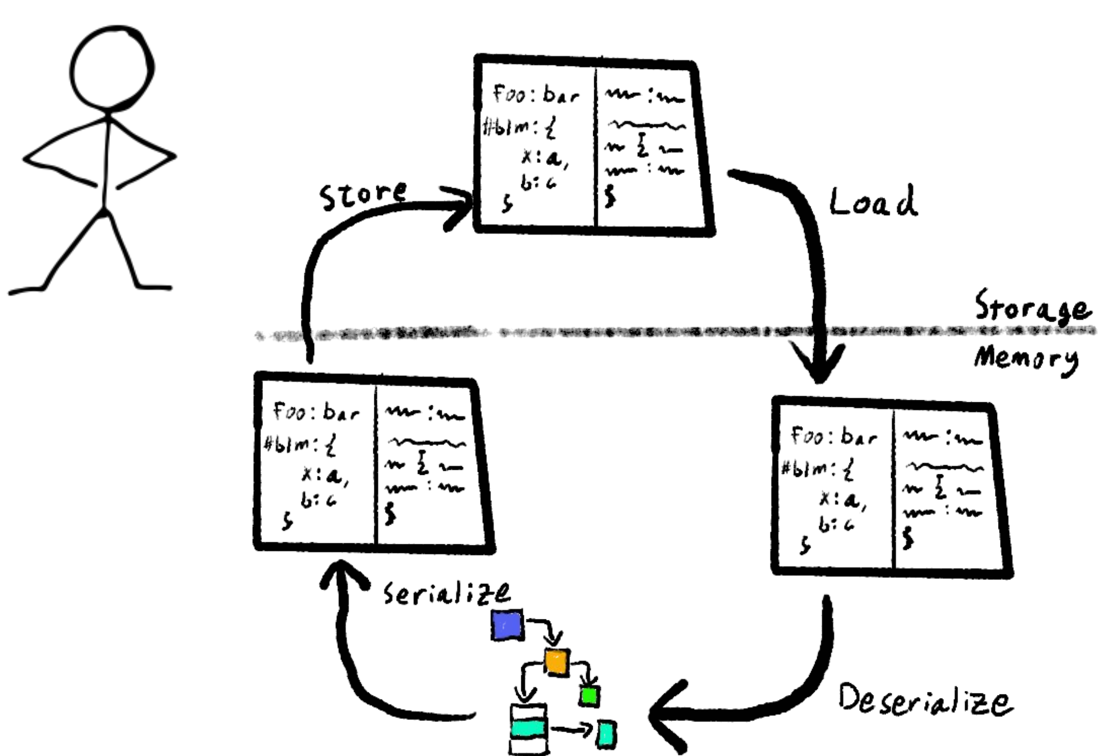
The state of the art: Systems programming



The state of the art: Systems programming



The state of the art: Systems programming



The state of the art: Systems programming

Store

Load

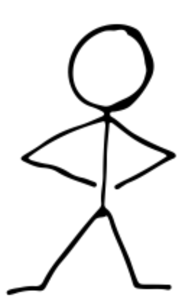
Storage

Memory

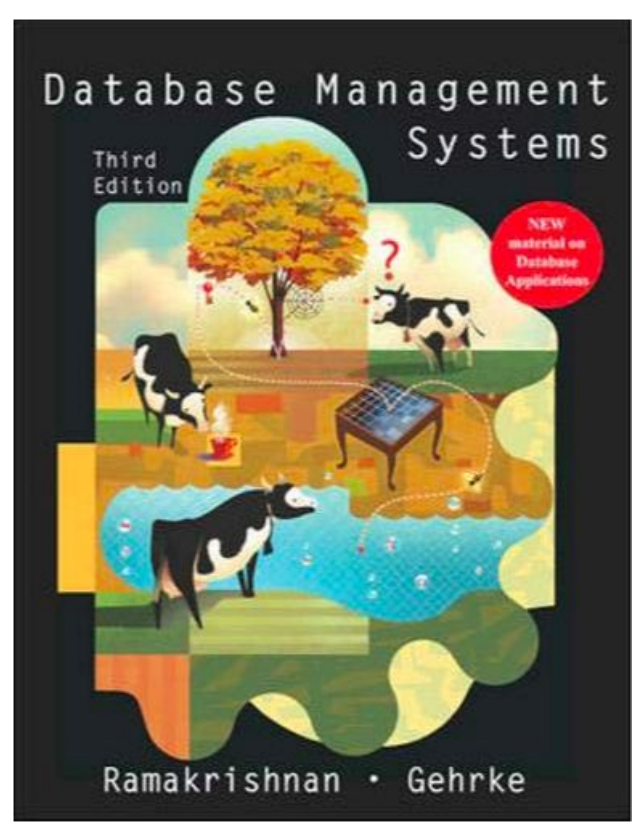
Serialize

Deserialize

Your algorithm here



"Systems programming"



"Systems programming"

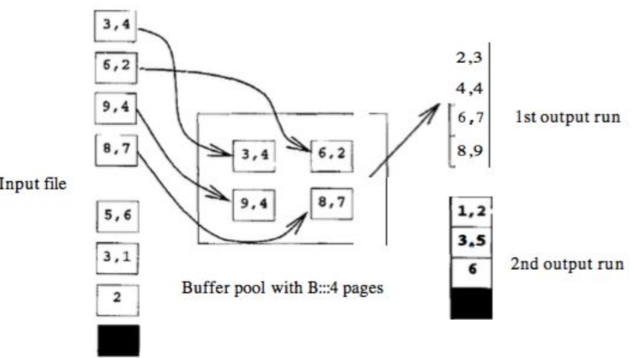


Figure 13.4 External Merge Sort with *B* Buffer Pages: Pass 0

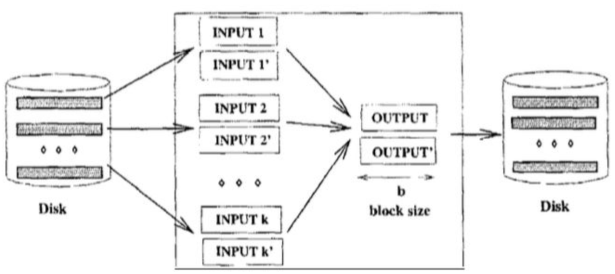


Figure 13.10 Double Buffering

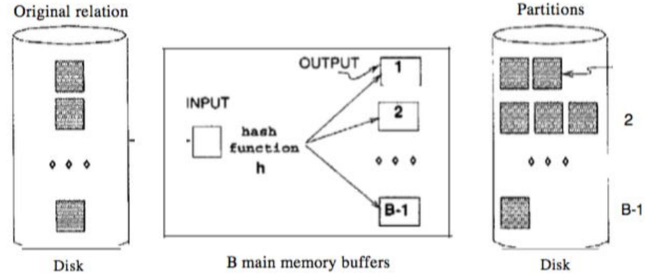


Figure 14.3 Partitioning Phase of Hash-Based Projection

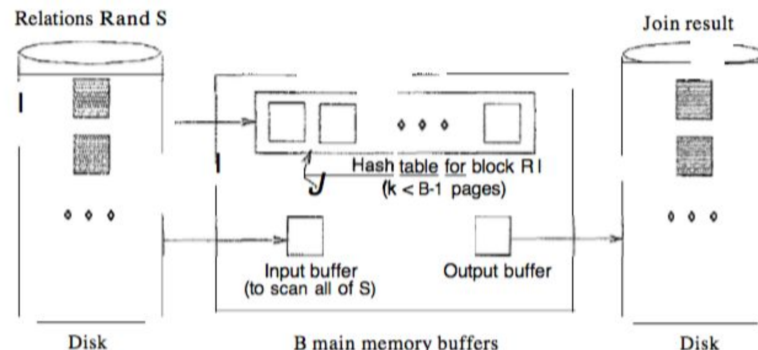
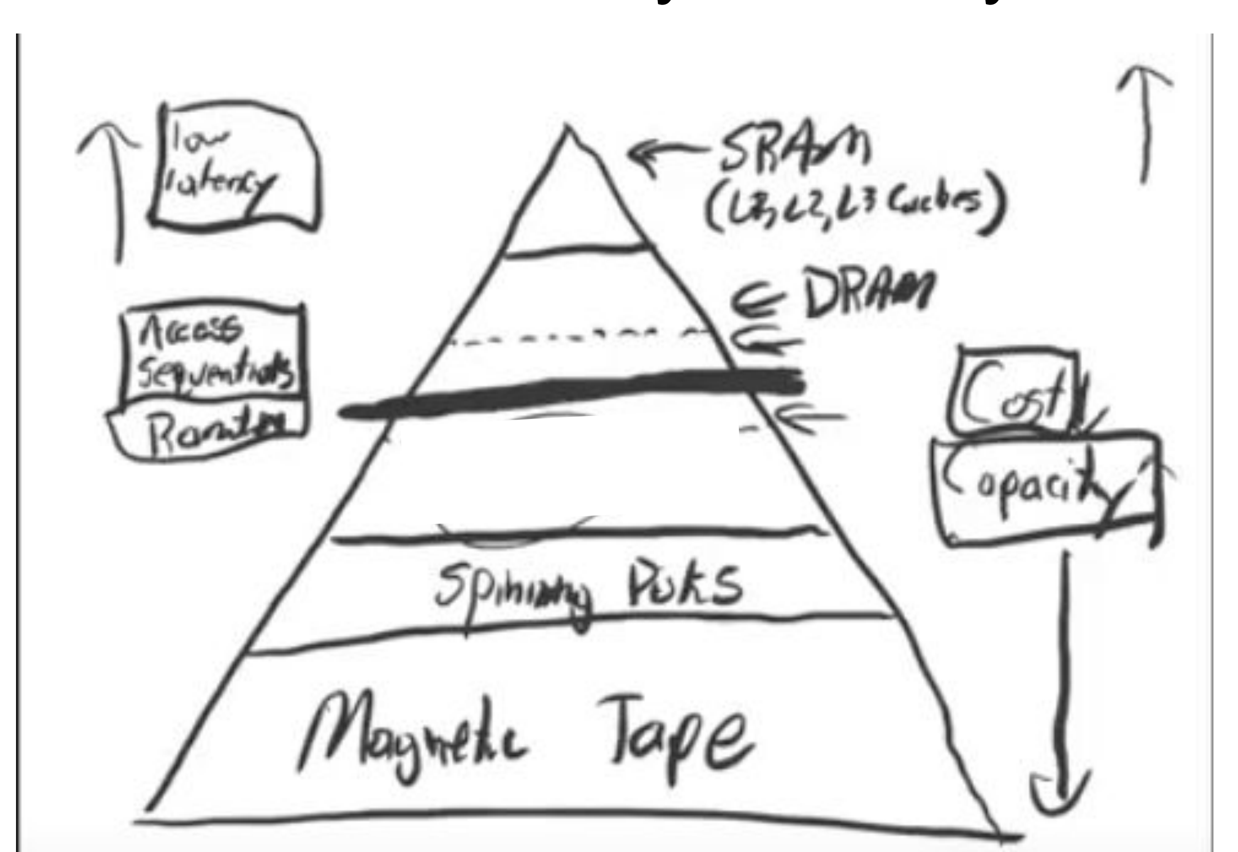


Figure 14.6 Buffer Usage in Block Nested Loops Join

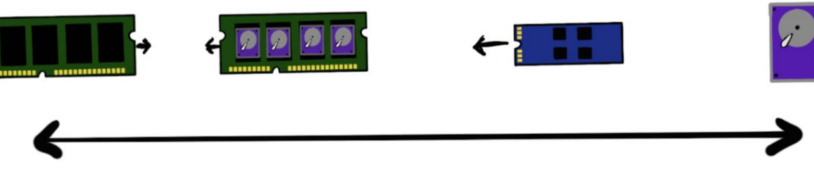
The memory hierarchy



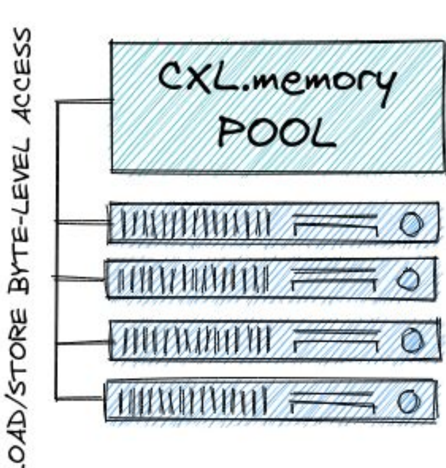
Observe these wild trends

Trends

Persistent storage is *getting ever closer*



Memory is *moving farther away*



One obvious point of inflection

sys_read

~100-300 ns

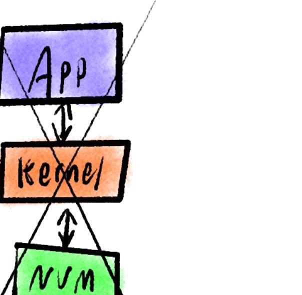
~1 us

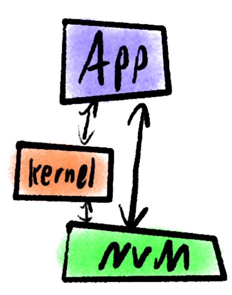
~1-10 ms

Growing, becoming persistent

Outdated interface

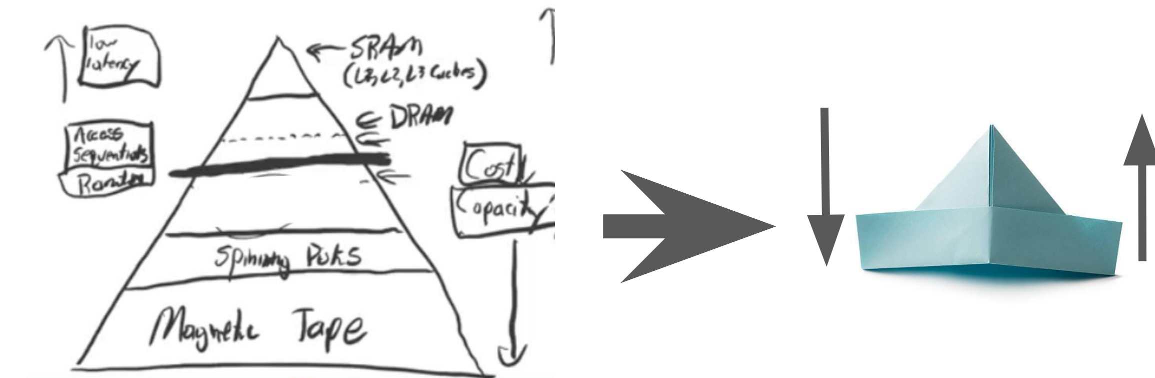
Cannot compute on directly



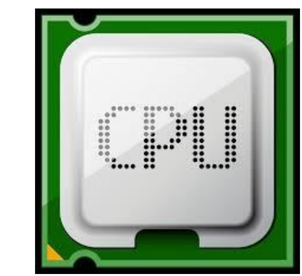


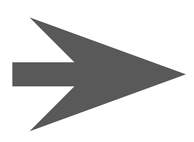
Trends

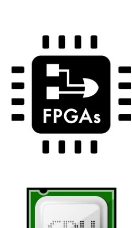
Storage is CONVERGING?

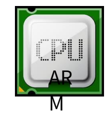


Compute is DIVERGING!









Twizzler boils this ocean

No kernel-mediated I/O

Persistent data should be operated on *directly* and *like memory*

No transient pointers

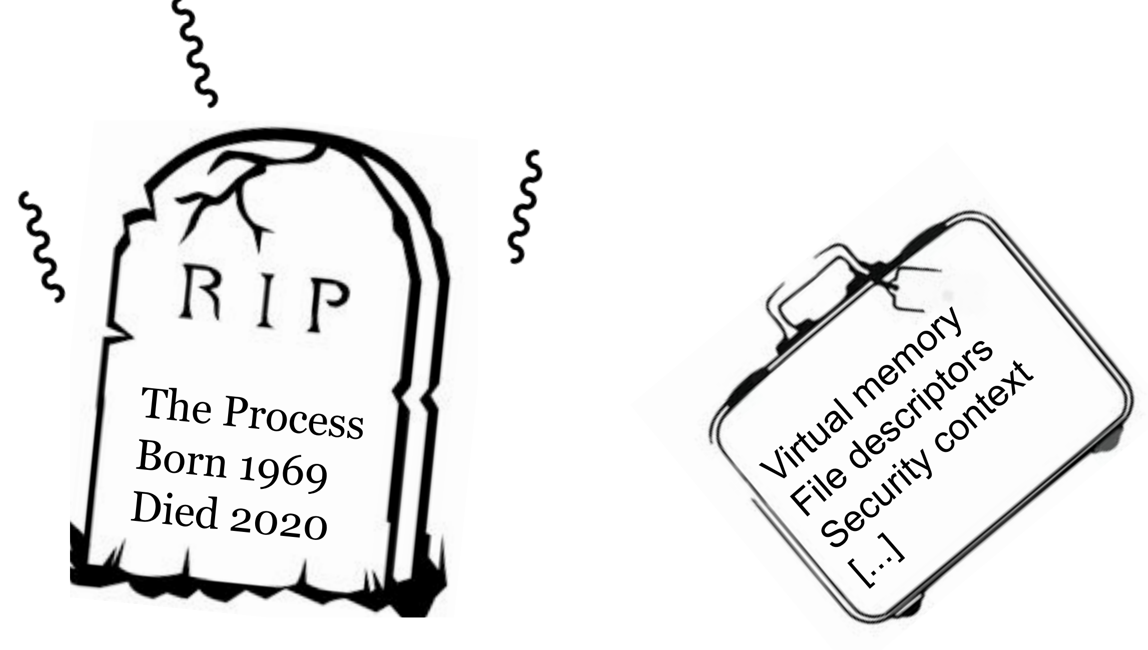
Pointers should last forever and have the same meaning anywhere

"Data-centric" operating systems

Persistent data should be operated on *directly* and *like memory*

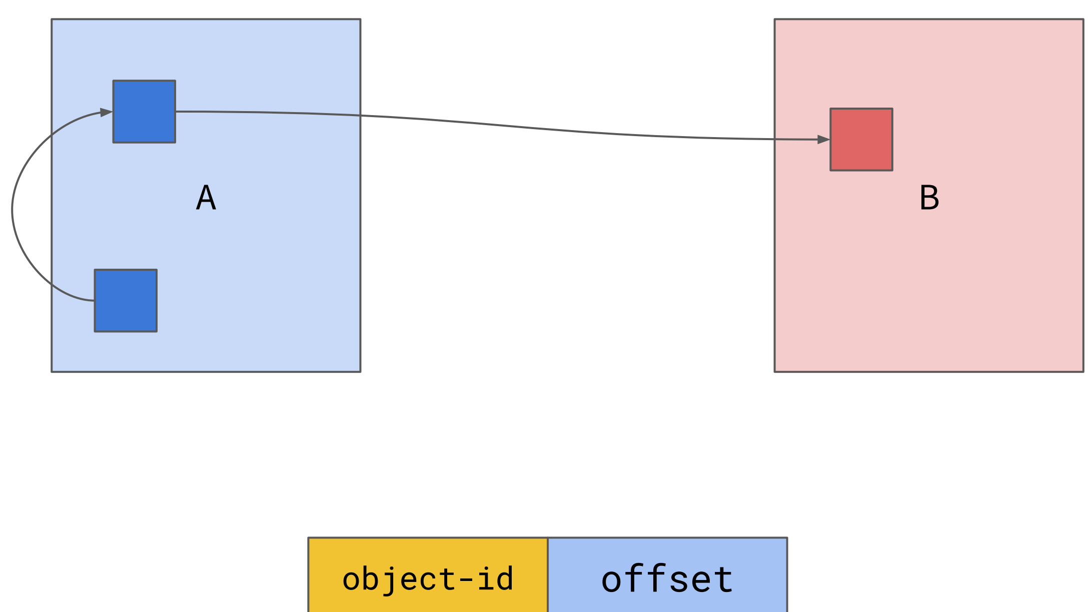
Pointers should last forever and have the same meaning anywhere

The death of the process



Organizing memory in Twizzler

Data Objects, references, context

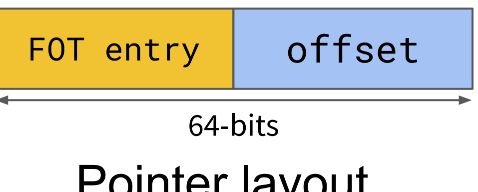


Pointers may be *cross-object*: referring to data within a different object

Data Objects and references

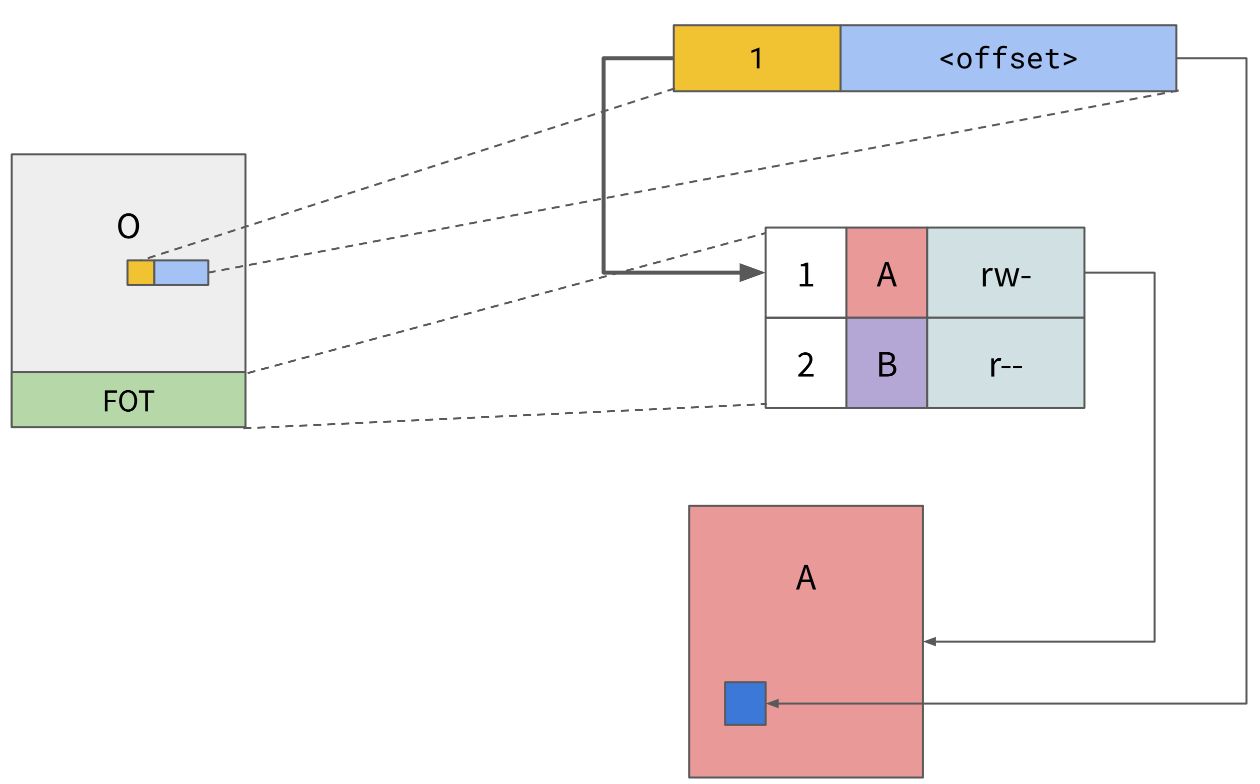
FOT | Data

Object layout

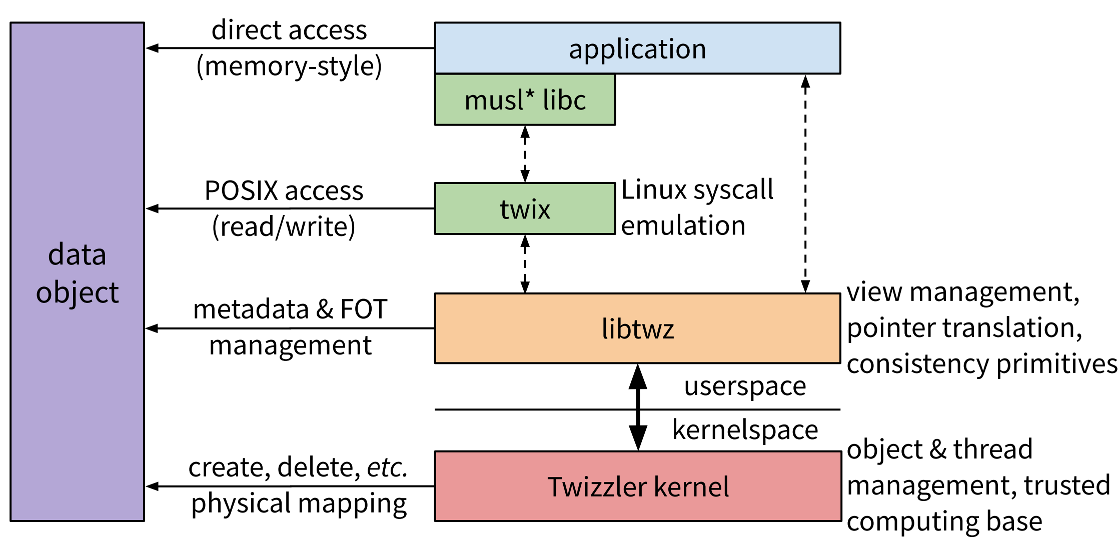


Pointer layout

Dereference



Compatibility (but second)



* modified musl to change linux syscalls into function calls

Some consequences

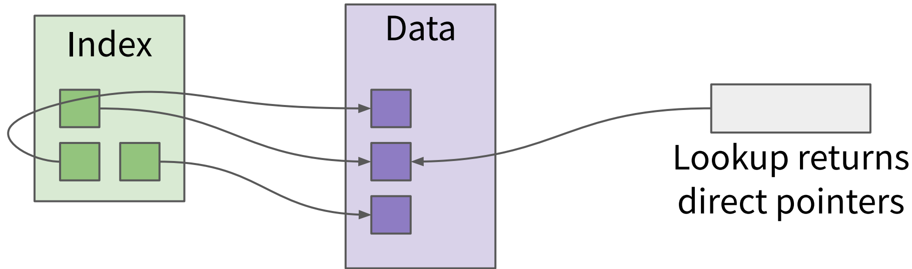
References are based on *identity* rather than *location*

Objects are self-contained and pointers never swizzle

A "galactic" 128-bit object space

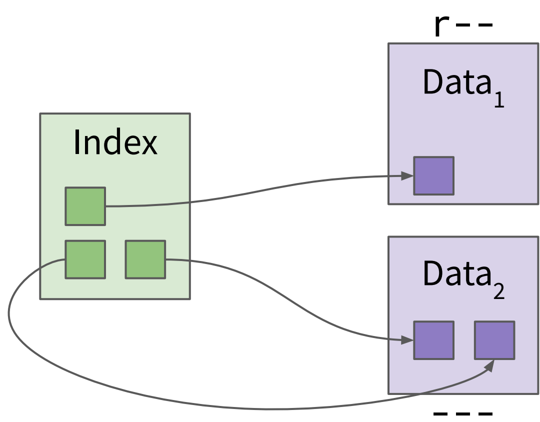
Case studies

Key / value storage done right

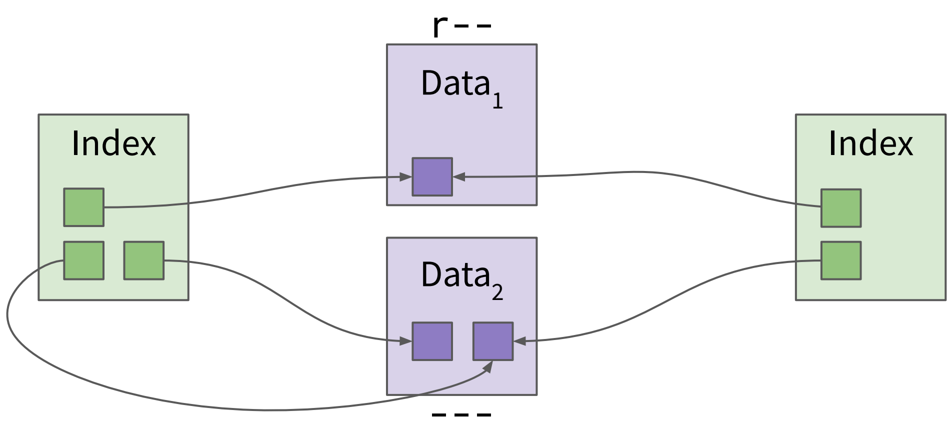


250 lines of simple C code is *all you need*

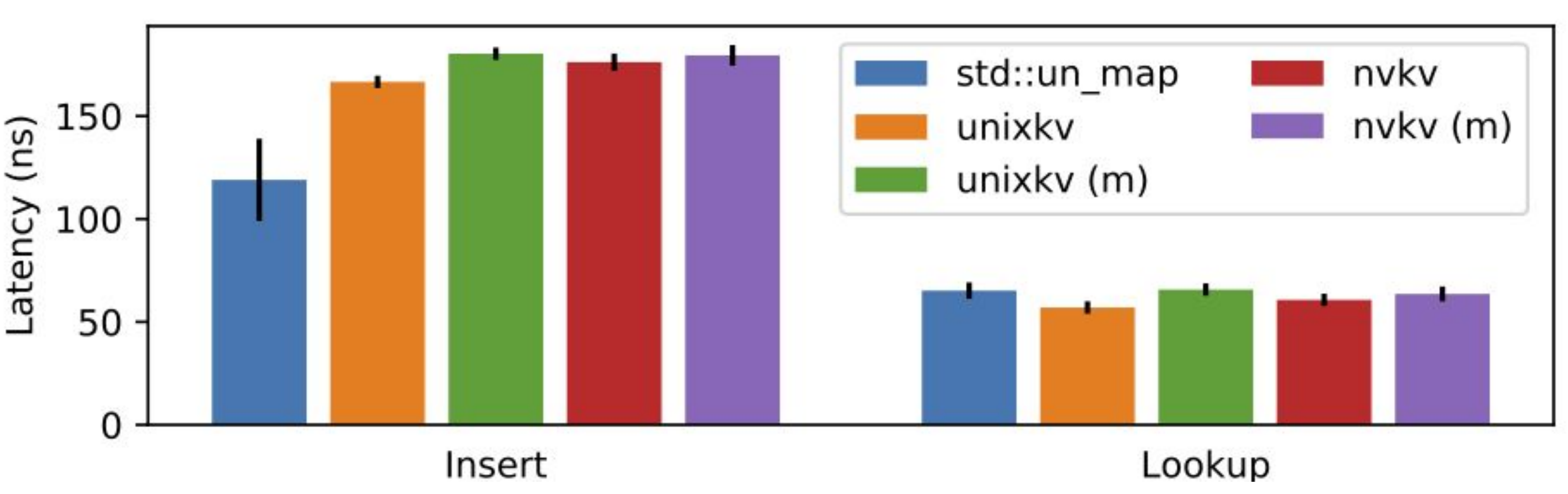
Evolution 1: access control



Evolution 2: secondary indices



Success: easier programming without a penalty



Porting SQLite

SQLite top

Native

SQLite top
LMDB
Memory-mapped files
Linux

SQLightning

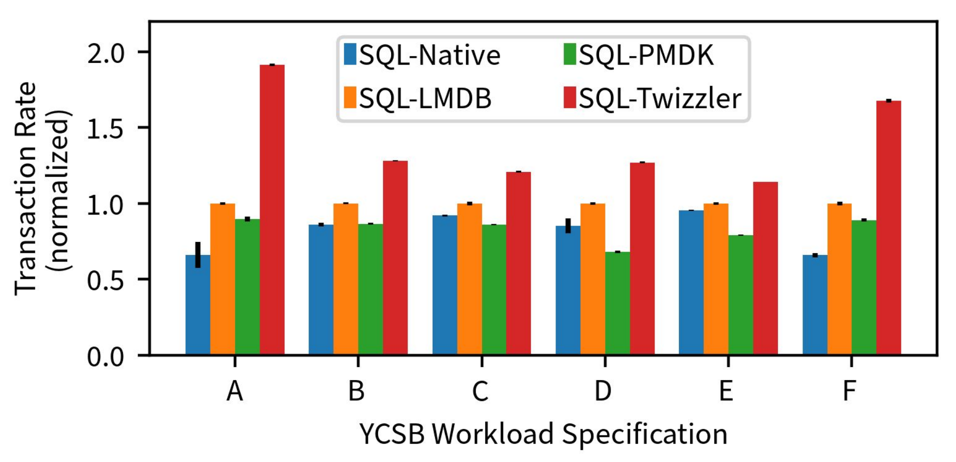
SQLite top
unixkv
unixrbt
PMDK
Linux

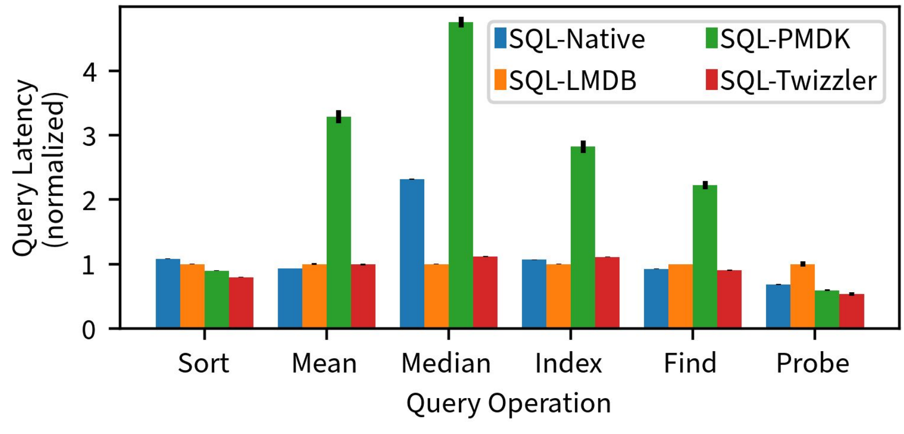
PMDK

SQLite top
LMDB
twizkv
twizrbt
Twizzler

Twiz

Not so shabby for legacy support





Security in a data-centric OS

Security Contexts

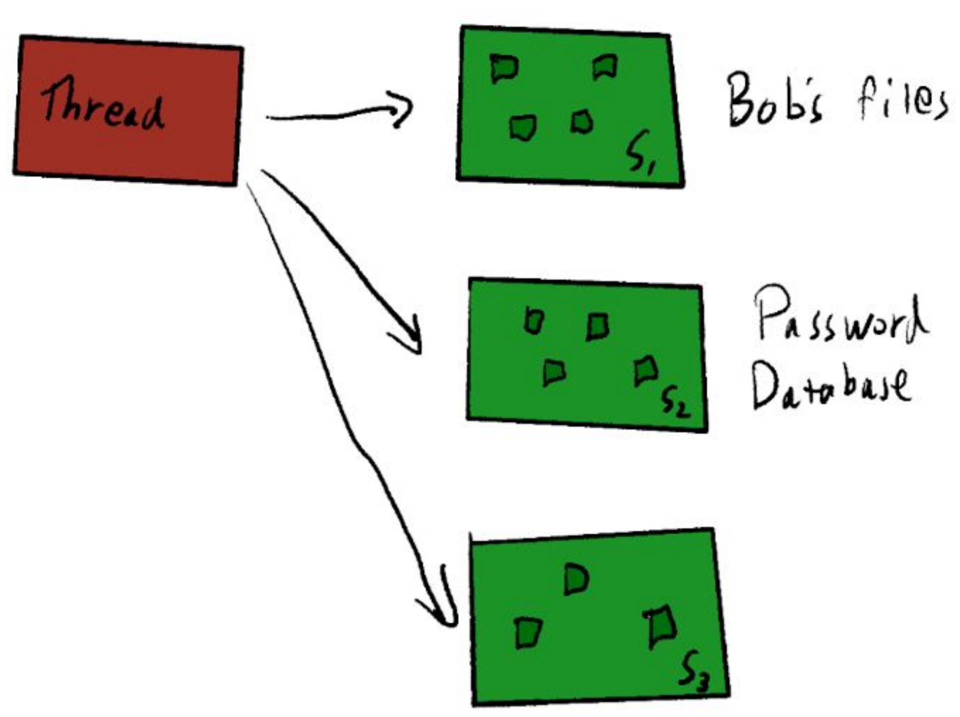
Protection information encoded as ordinary data objects

(containing cryptographically-signed capabilities)

Threads are associated with these *security contexts*.

Privilege is not accretive: a thread has *one active* security context.

Security Contexts



Security Policy

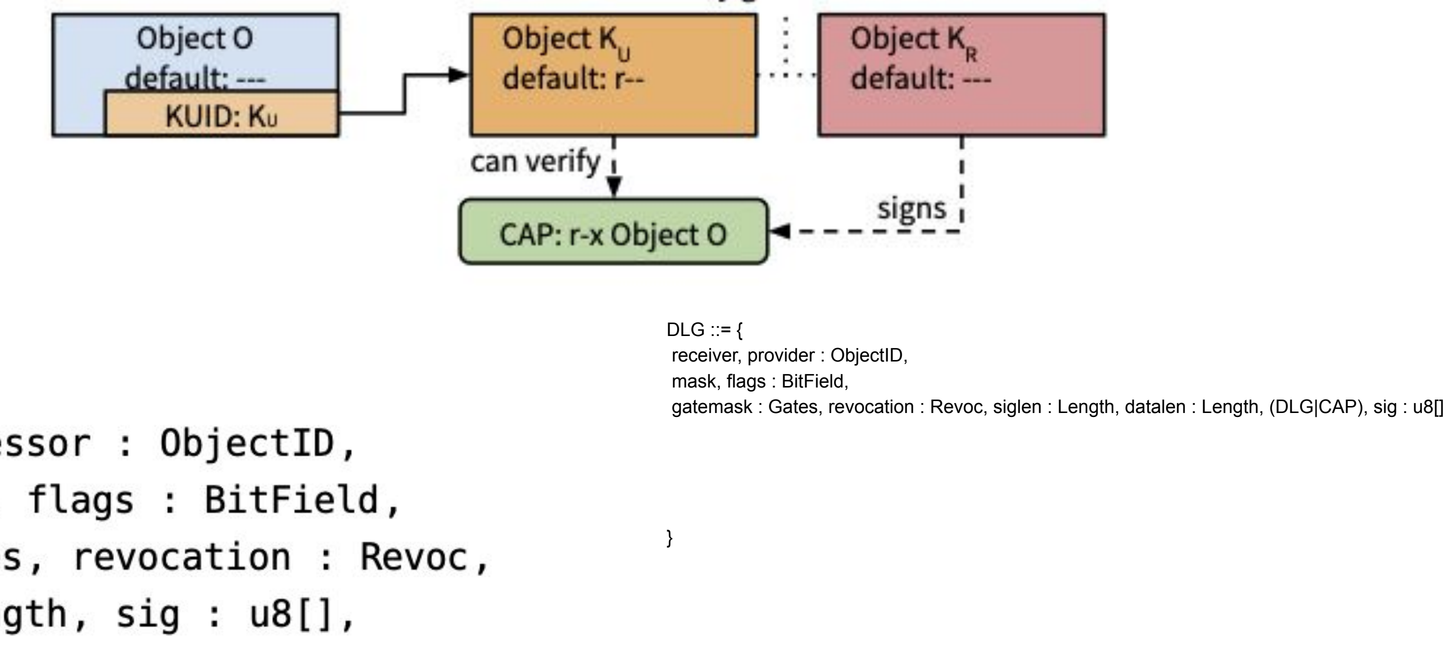
Is directly created and manipulated by *users*

Is interpreted by the *kernel*

Is enforced by *hardware*

Security Contexts reify agency as policy

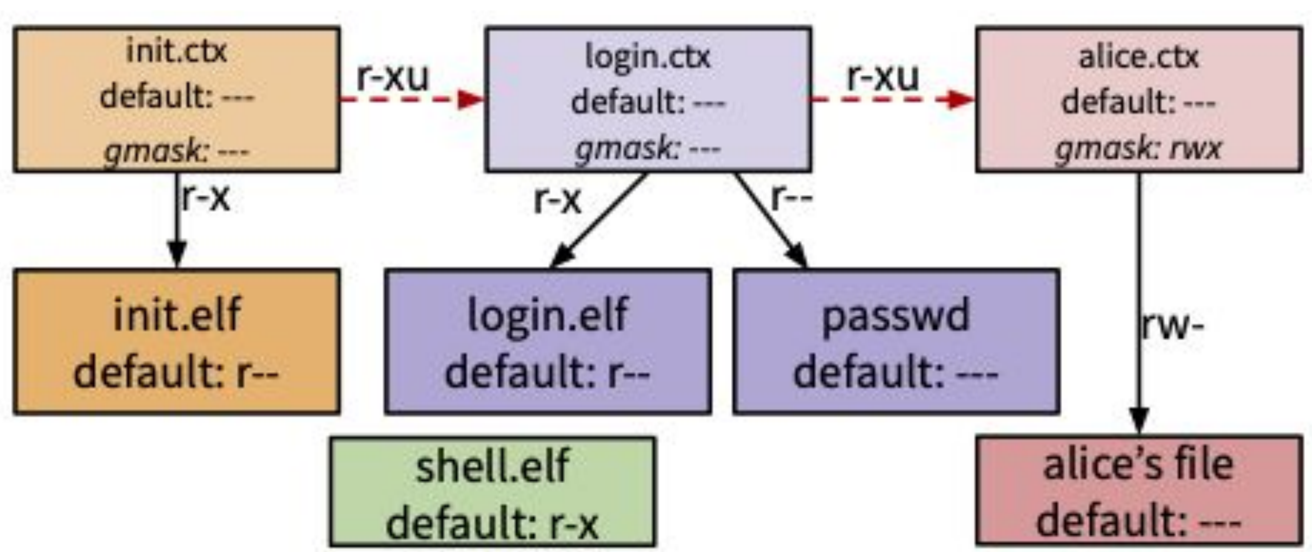
Capabilities and key management




```
CAP ::= {
    target, accessor : ObjectID,
    permissions, flags : BitField,
    gates : Gates, revocation : Revoc,
    siglen : Length, sig : u8[],
}
```

Bootstrapping

CAP login.elf r-x-	CAP passwd r--	CAP alice.ctx ---u	CAP bob.ctx ---u



The death of the superuser

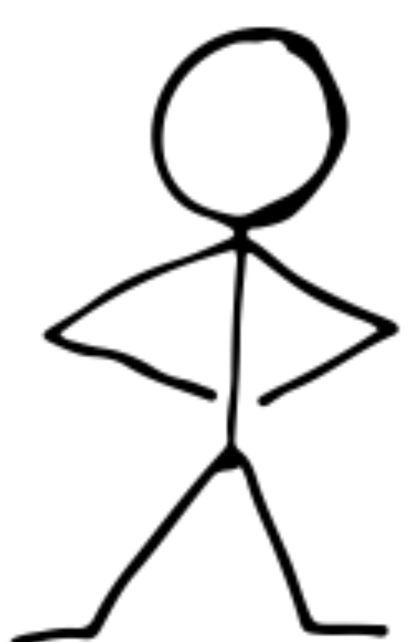
Secure Gated APIs

Secure Gated APIs

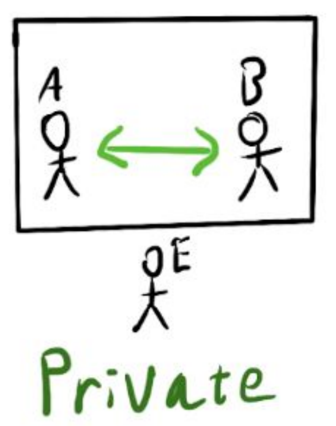
Protected library calls

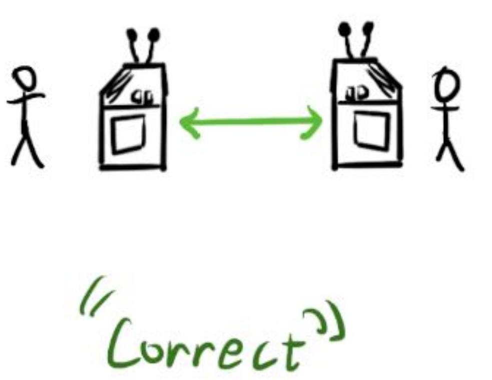
System calls (but for userspace)

Agency, without the overhead of agents



For example: IPC without the kernel



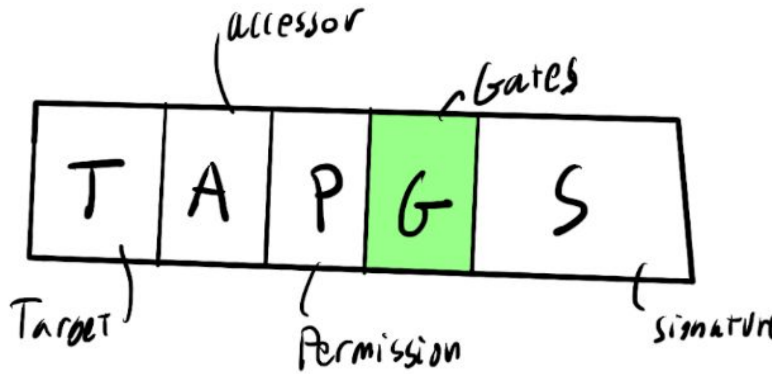


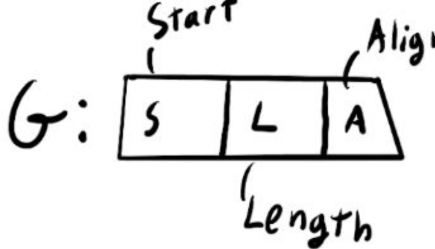
Use raw objects like shared memory?

Secure Gated APIs

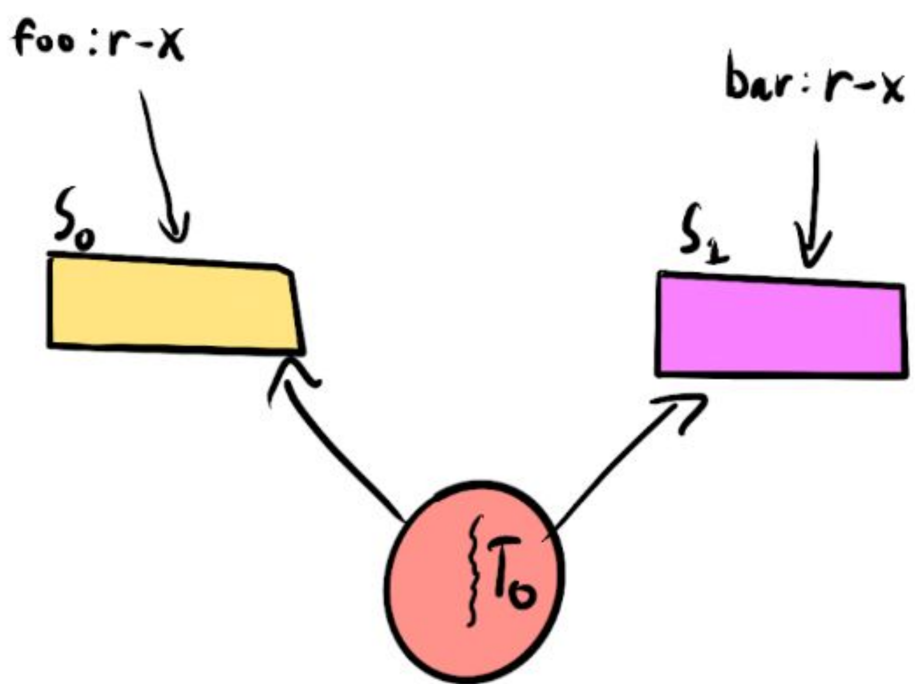
Allow Threads to Jump Programs

Secure Gated APIs

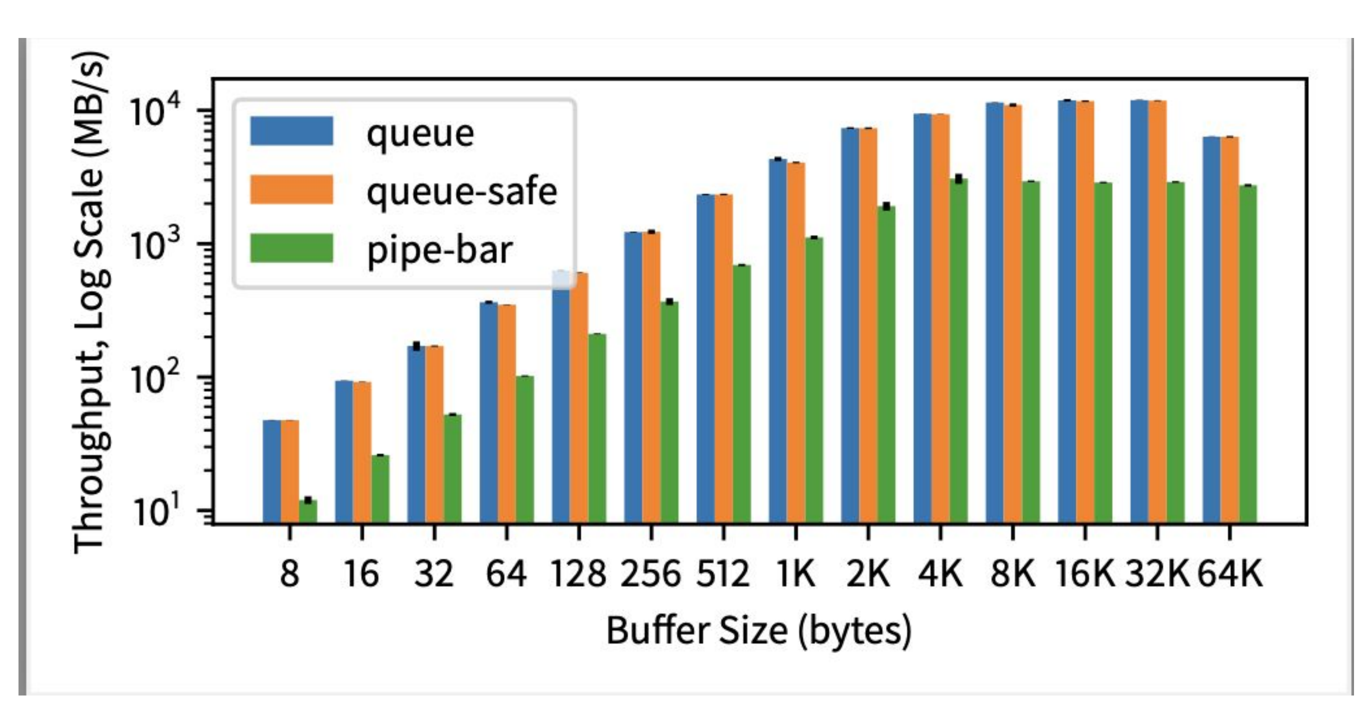




Restricting jumps

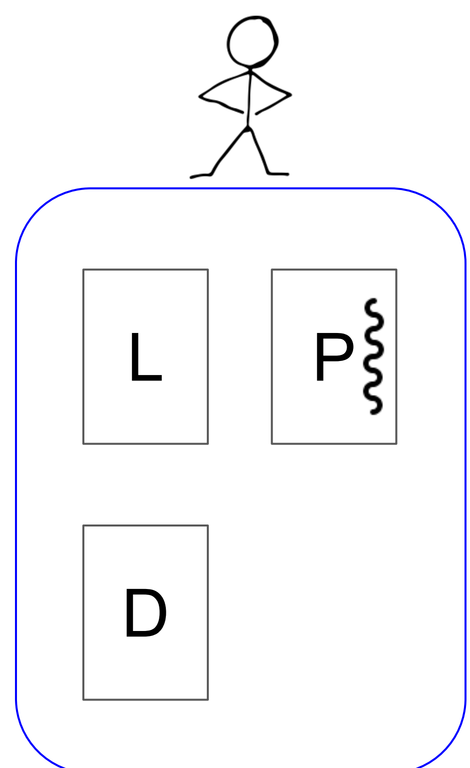


Communication links in userspace

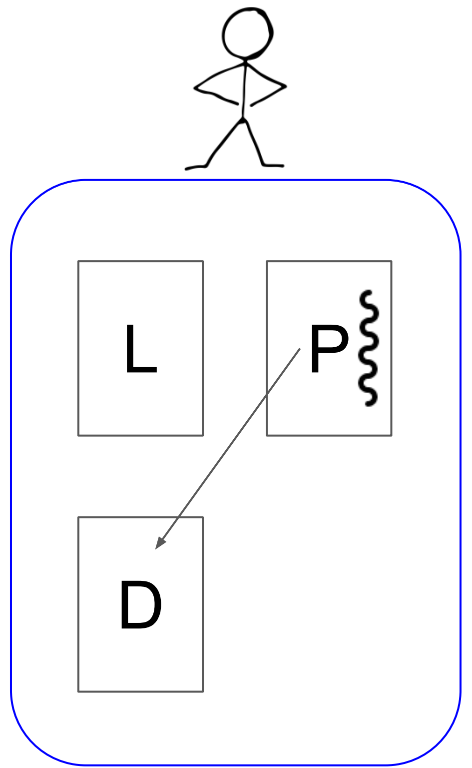


Distributing Twizzler

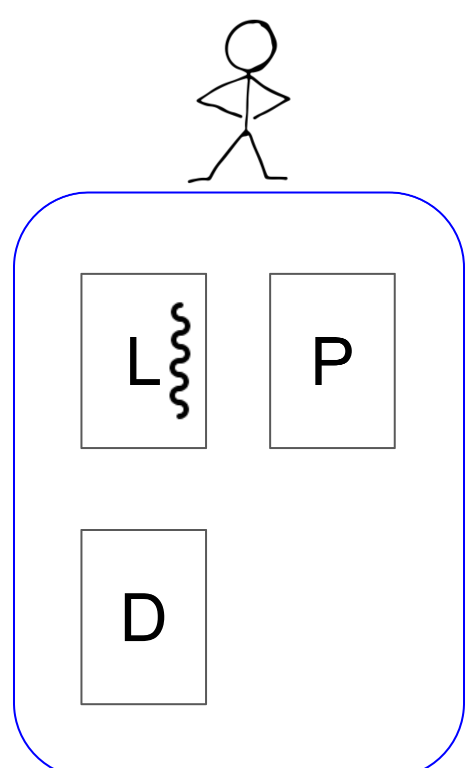
Computation



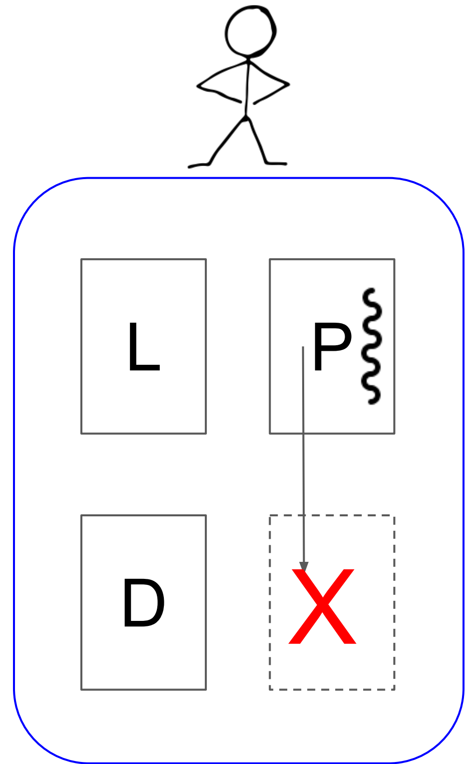
Computation



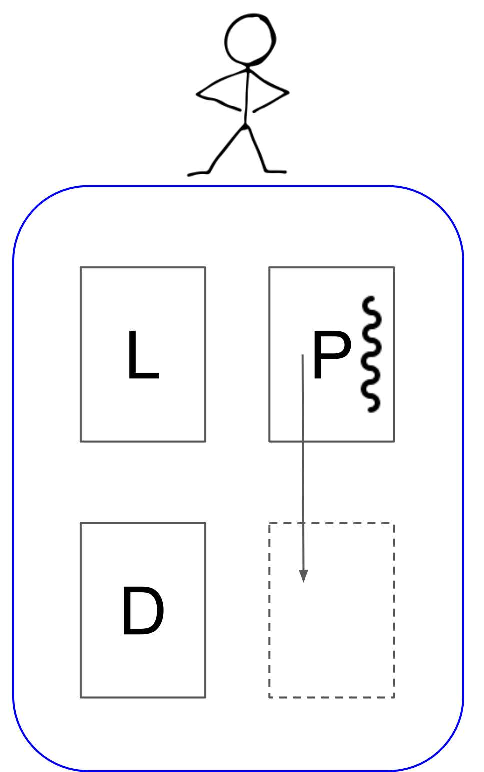
Computation

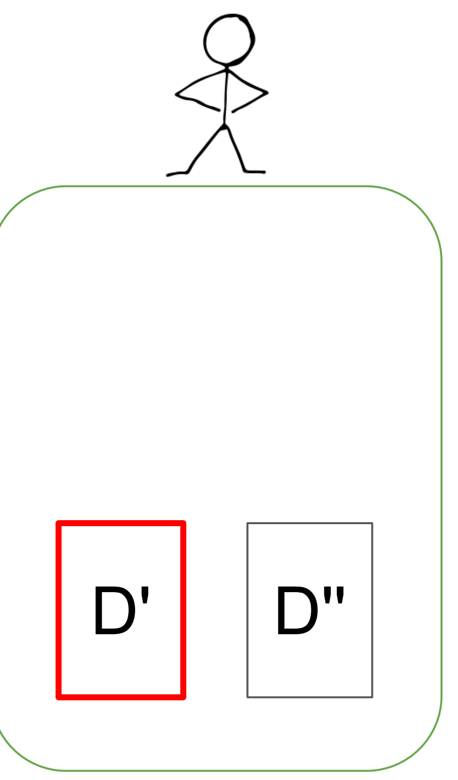


Computation

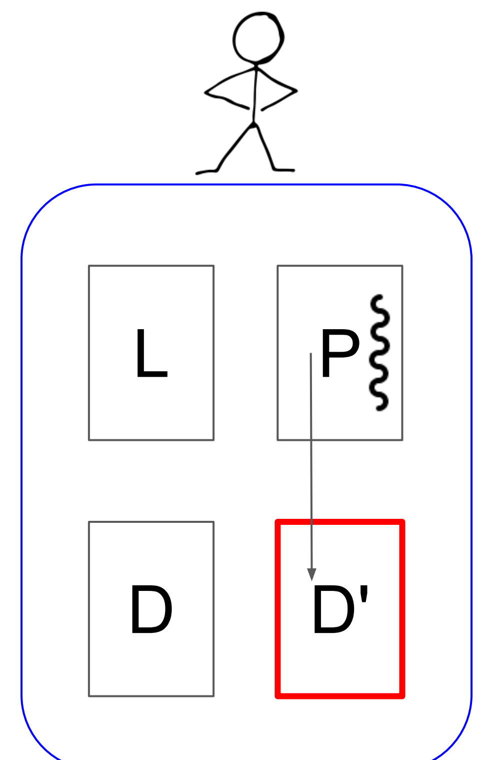


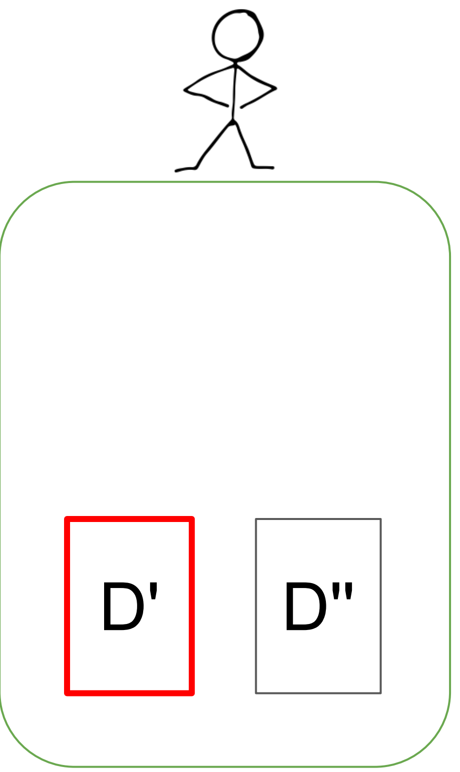
Computation over remote data



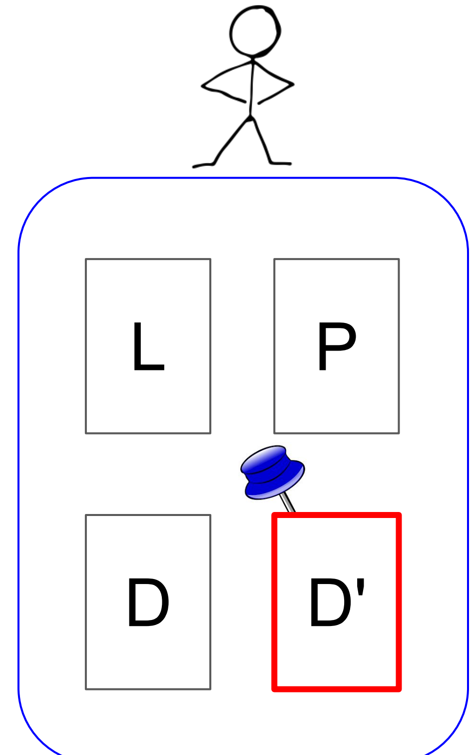


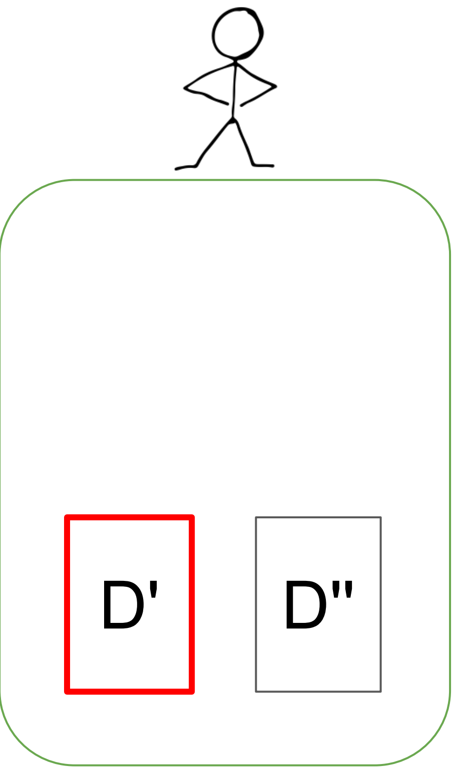
Faulting





Caching





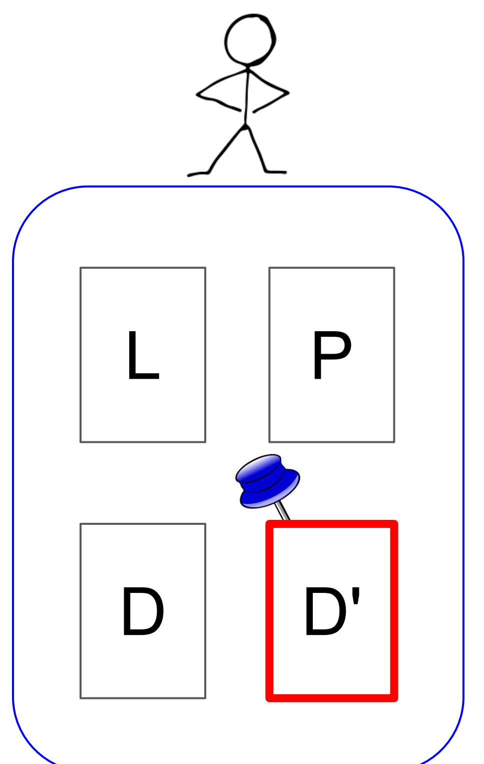
Locality of reference and identity

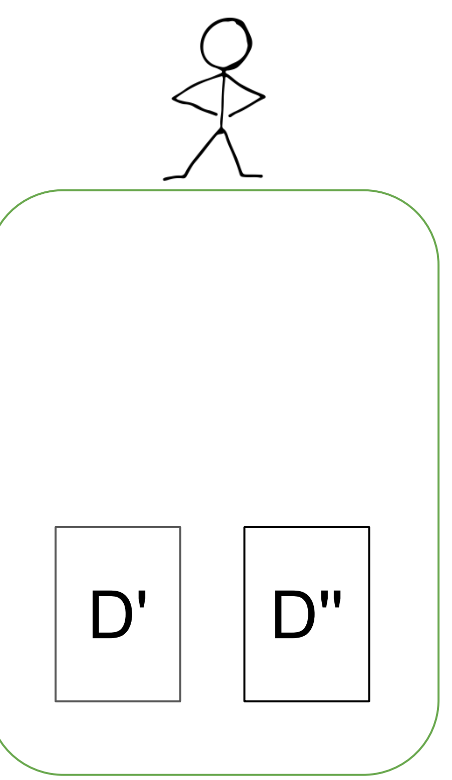
Systems "tricks" :

Caching

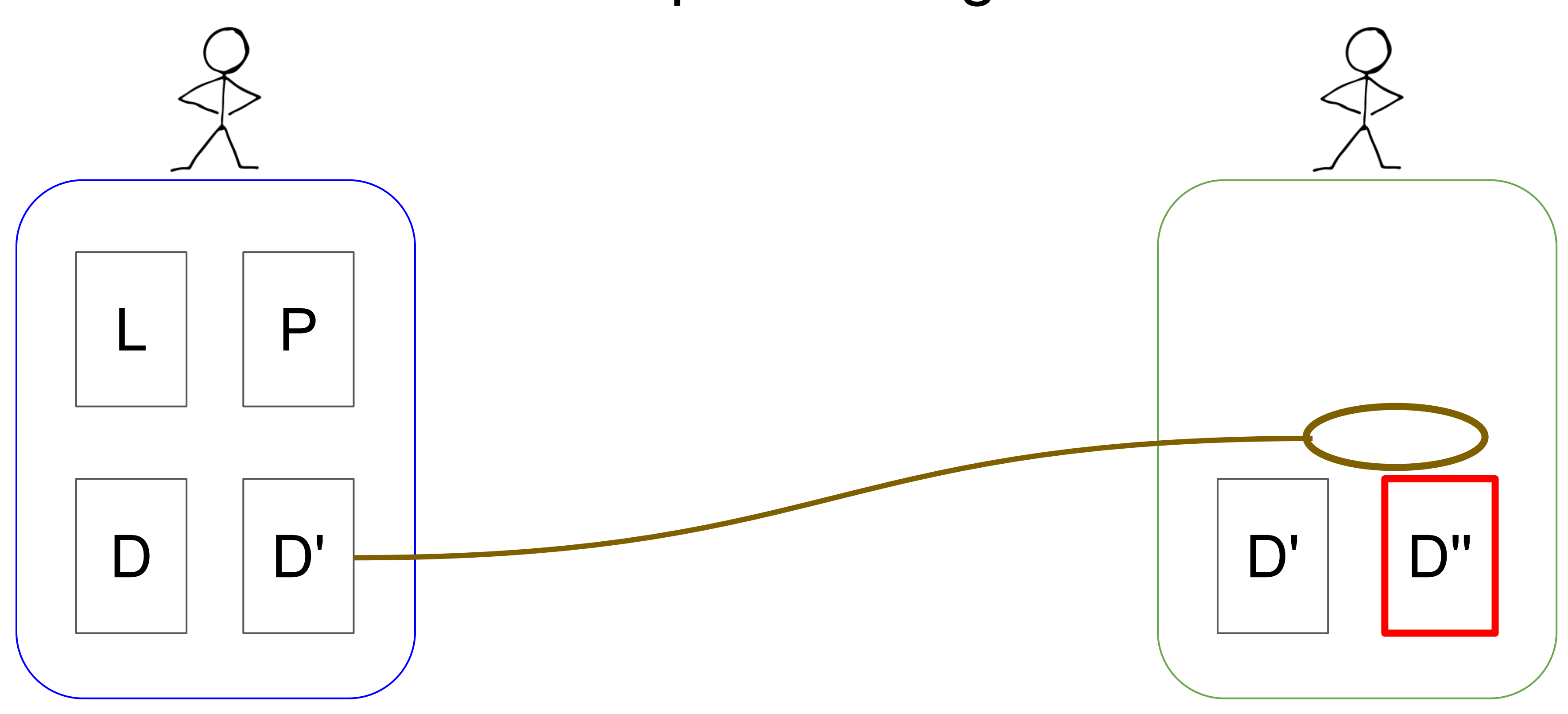
(locality in time)

Caching approximates identity using time





And prefetching?



Locality of reference and identity

Systems "tricks" :

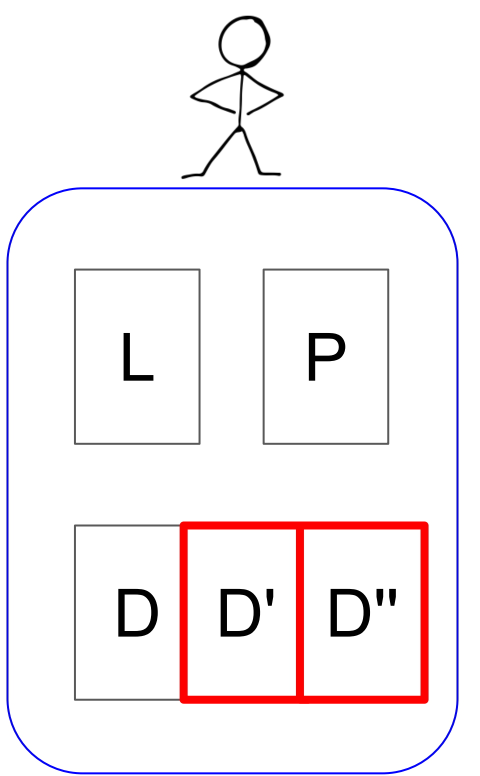
Caching

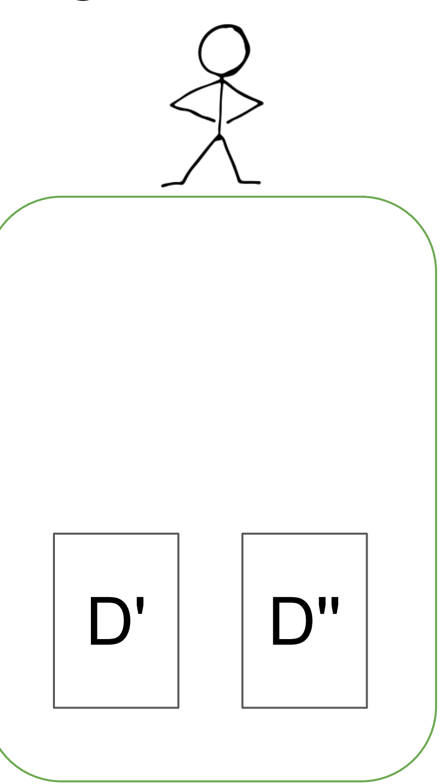
(locality in time)

Prefetching

(locality in space)

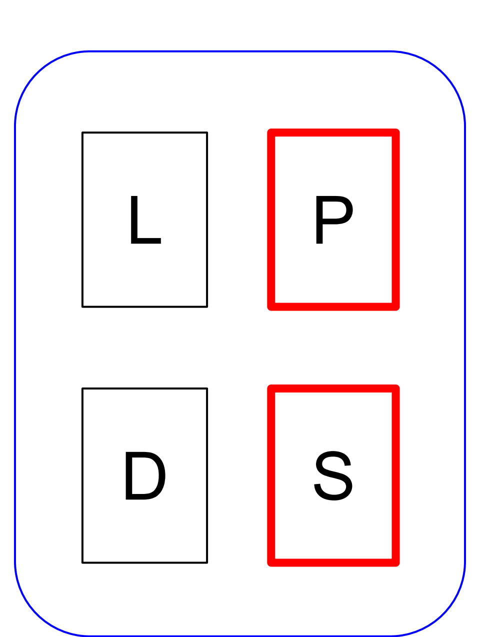
Prefetching approximates identity using space

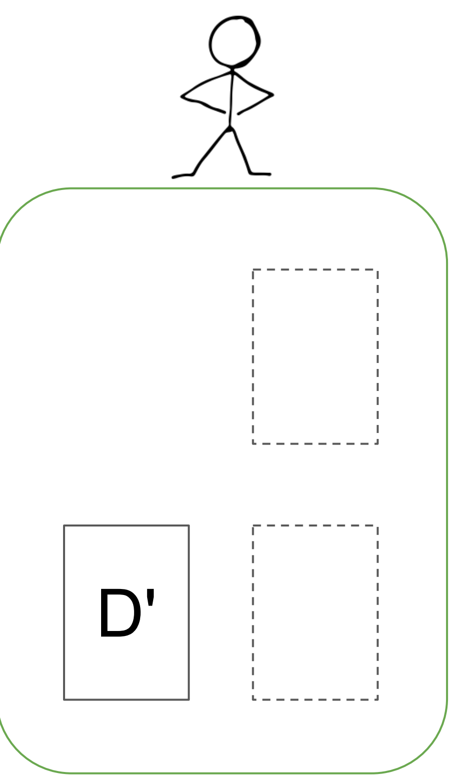




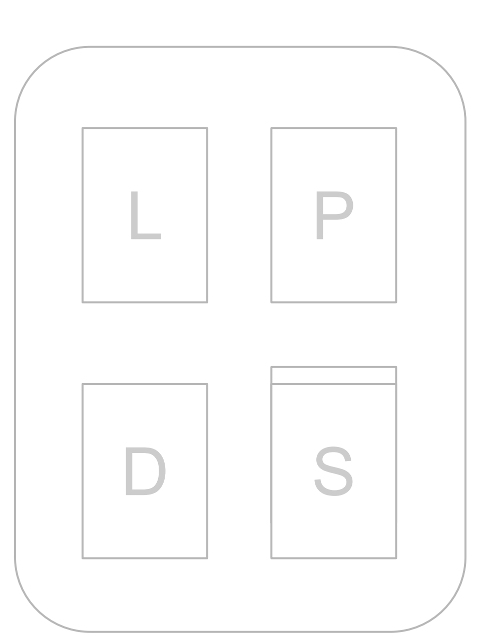
If you want identity, use identity!

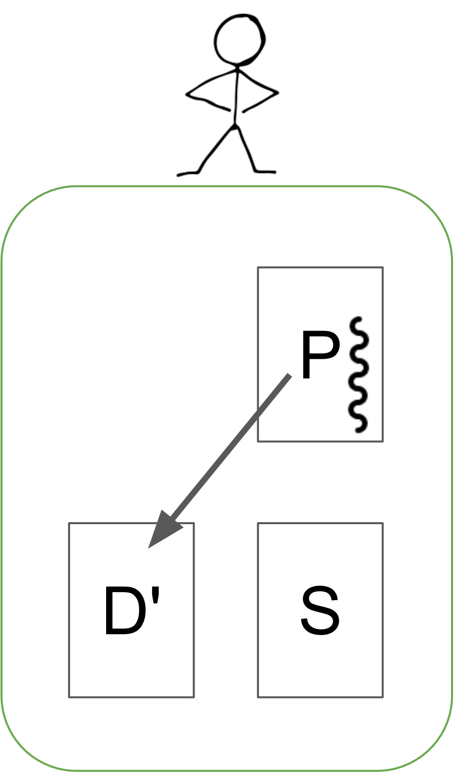
Live process migration trivializes



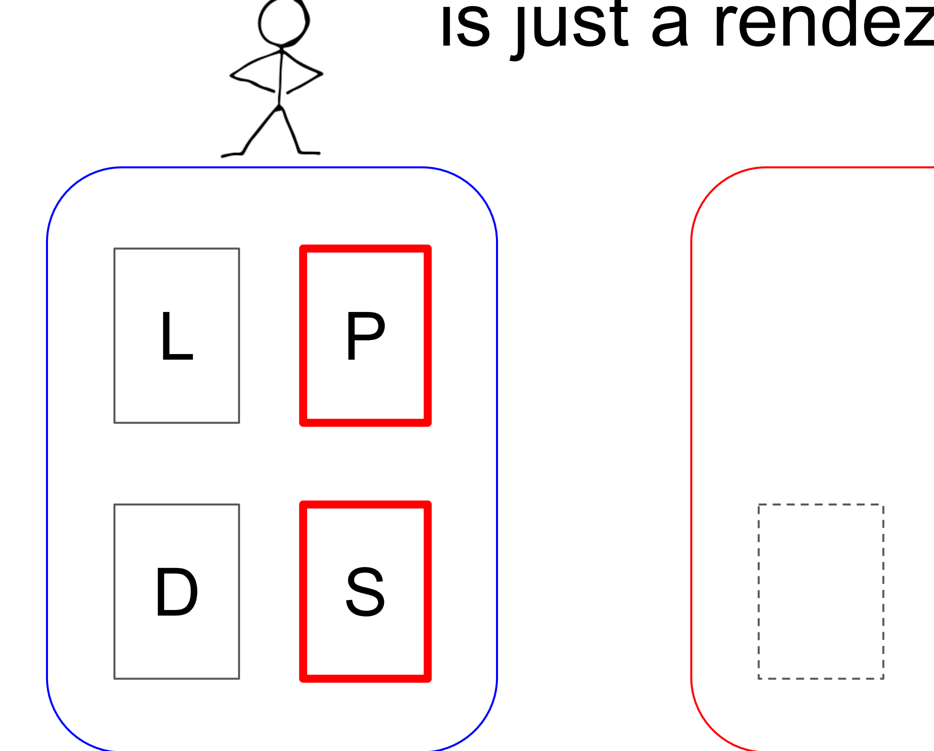


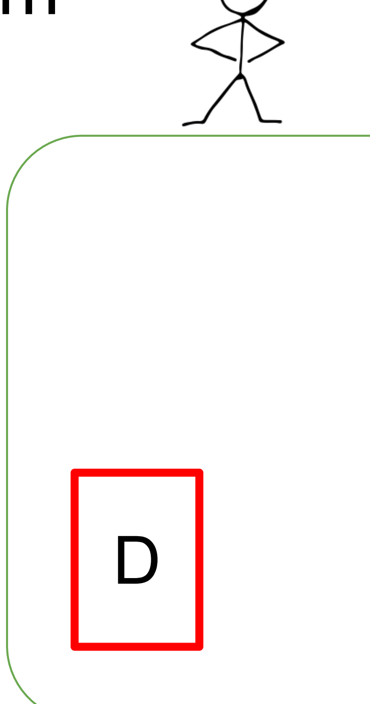
Live process migration trivializes



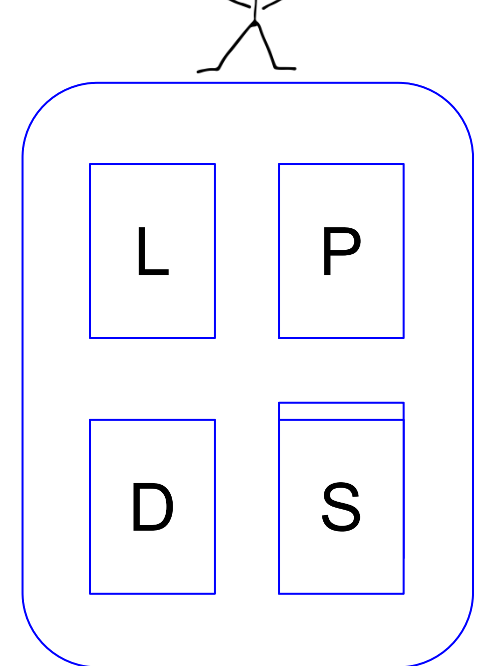


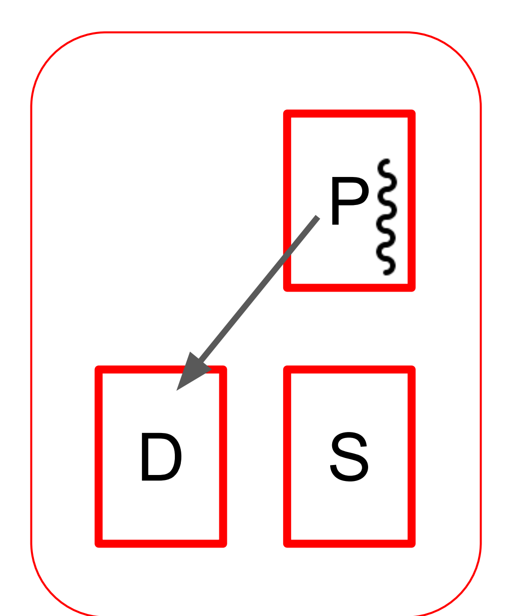
Live process migration, too,
is just a rendezvous problem

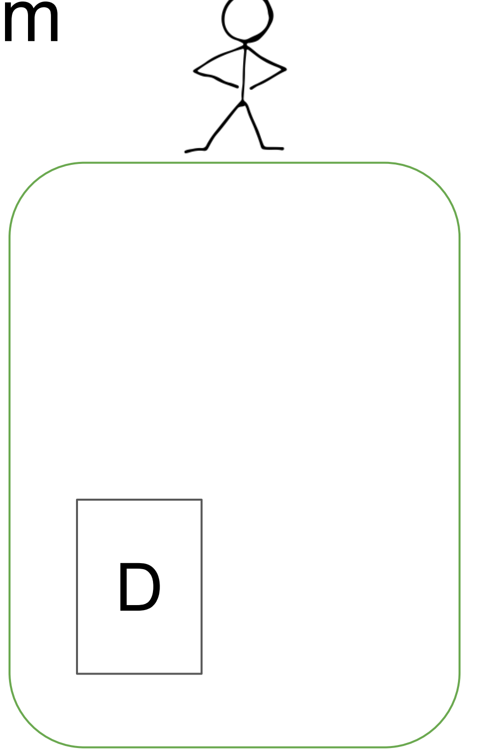

L
P
D
S


D

Live process migration, too, is just a rendezvous problem

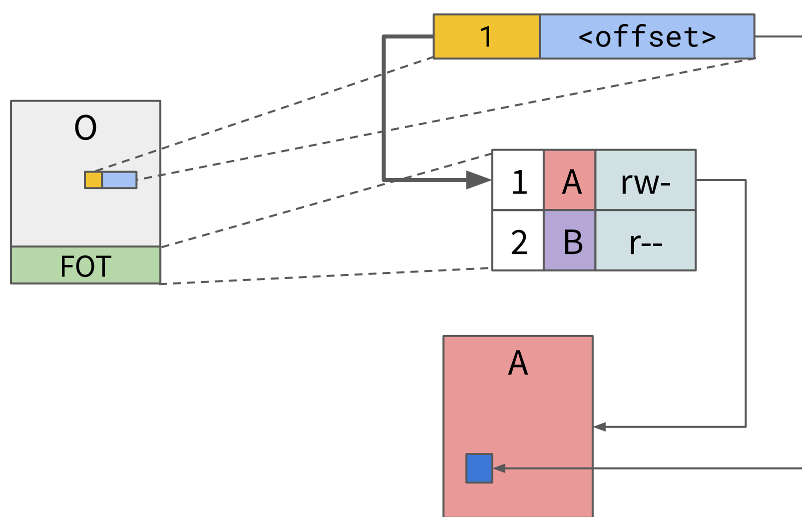




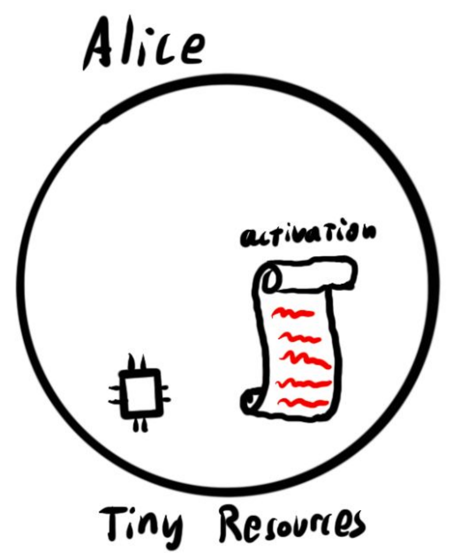


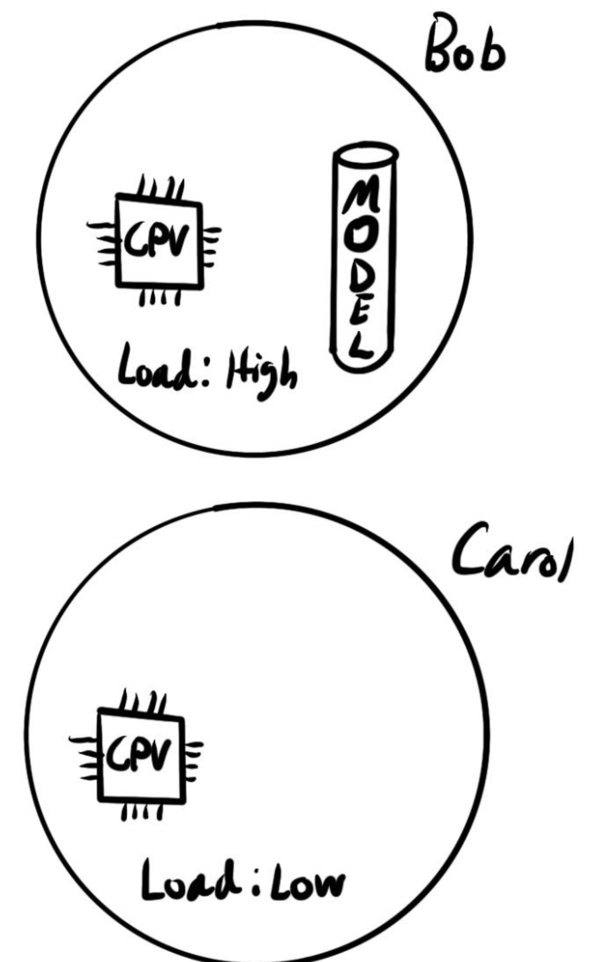
The computer is wherever we can find it

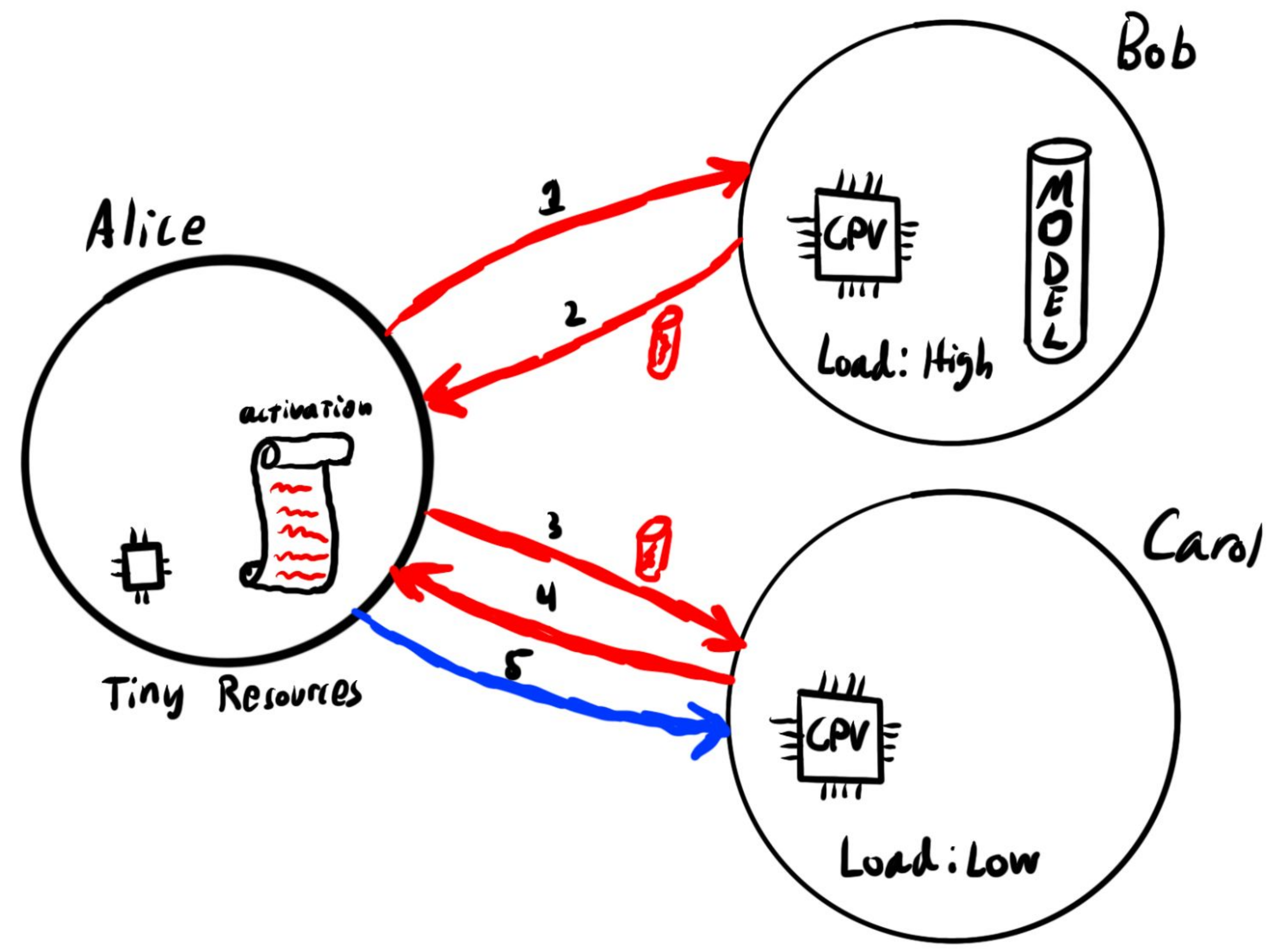
Everything is data:
Identity enables system-level program analysis

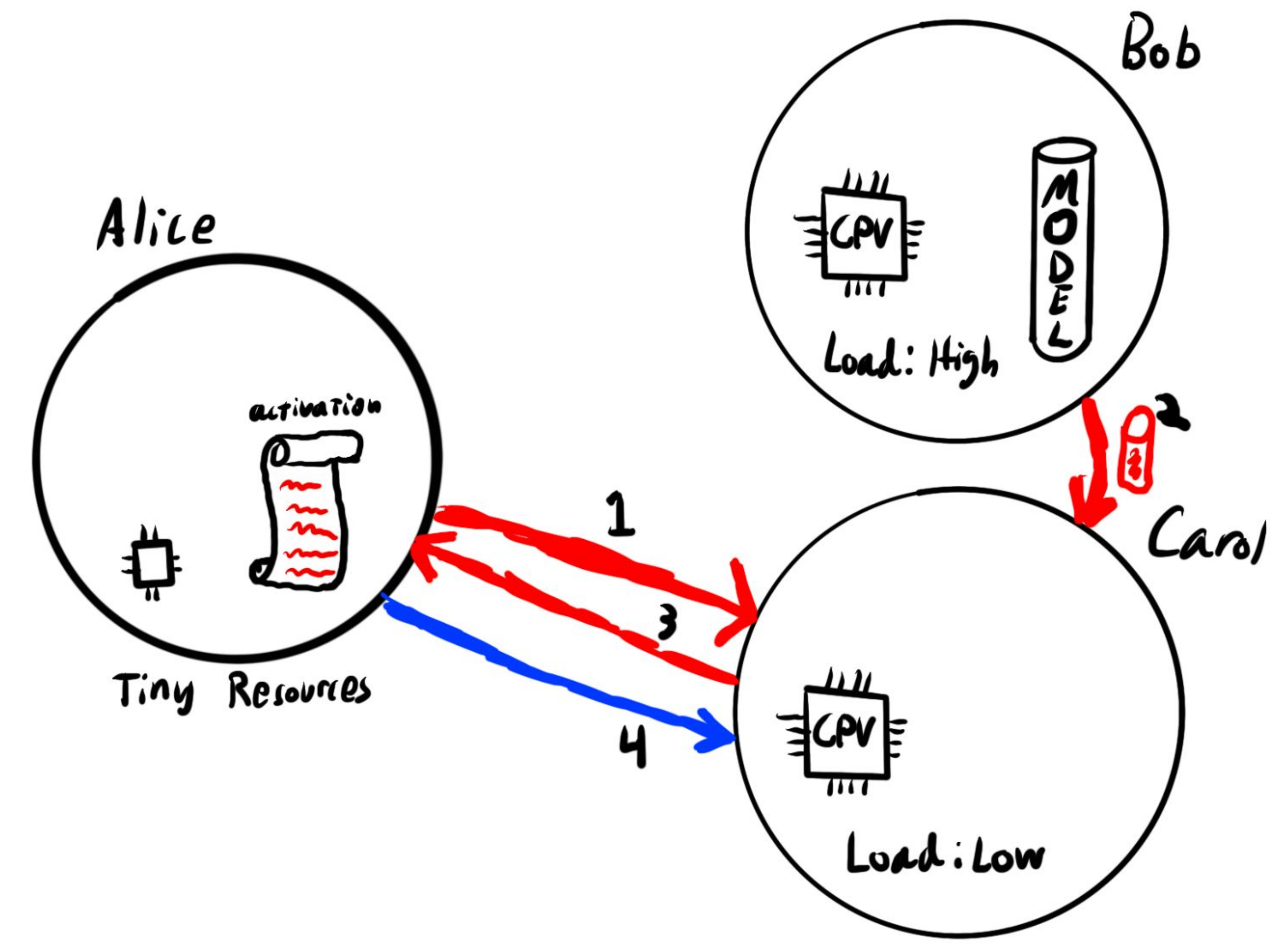


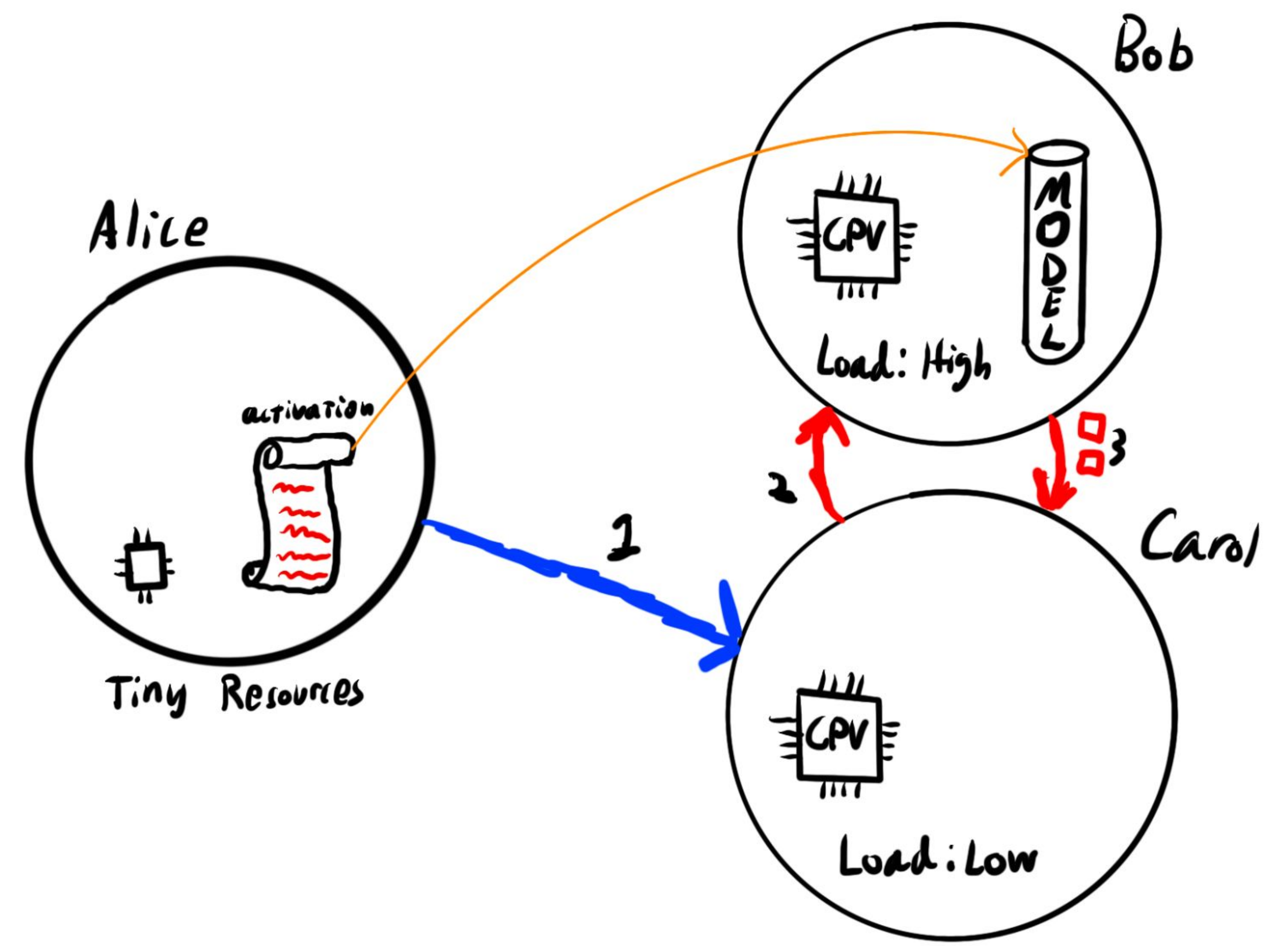
Rethinking RPC


Alice
activation
Tiny Resources


Bob
CPU
MODEL
Load: High
Carol
CPU
Load: Low


Bob
Alice
1
2
CPU
MODEL
Load: High
activation
3
4
5
Carol
Tiny Resources
CPU
Load: Low


Bob
CPU
MODEL
Load: High
2
Alice
activation
Tiny Resources
1
3
4
Carol
CPU
Load: Low


Bob
CPU
MODEL
Load: High
Alice
activation
Tiny Resources
1
2
3
Carol
CPU
Load: Low

Wins

The code mobility of RPC

The data mobility of DSM

Data movement and computation placement as infrastructure-level concerns

Don’t Let RPCs Constrain Your API

Daniel Bittman
UC Santa Cruz

Robert Soulé
Yale University

Ethan L. Miller
UC Santa Cruz
Pure Storage

Vishal Shrivastav
Purdue University

Pankaj Mehra
IEEE Member

Matthew Boisvert
UC Santa Cruz

Avi Silberschatz
Yale University

Peter Alvaro
UC Santa Cruz

ABSTRACT

As data becomes increasingly distributed, traditional RPC and data serialization limits performance, result in rigidity, and hamper expressivity. We believe that technology trends including high-density persistent memory, high-speed networks, and programmable switches make this the right time to revisit prior research on distributed shared memory, global addressing, and content-based networking. Our vision combines the code mobility of RPC with first-class data references in a global address space by co-designing the OS and the network around pervasive data identity. We have initial results showing the promise of the proposed co-design.

CCS CONCEPTS

• **Software and its engineering → Operating systems**; **Distributed systems organizing principles**; *Abstraction, modeling and modularity*; • **Hardware** → Memory and dense storage; *Networking hardware*; • **Information systems →** Storage class memory.

ACM Reference Format:
Daniel Bittman, Robert Soulé, Ethan L. Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. 2021. Don’t Let RPCs Constrain Your API. In *The Twentieth ACM Workshop on Hot Topics in Networks (HotNets ’21), November 10–12, 2021, Virtual Event, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3484266.3487389

1 INTRODUCTION

Modular design is the bedrock of modern software development [24]. It improves programmer productivity by breaking design problems into smaller, re-usable parts that hide implementation details and are more easily debugged. In distributed systems, module composition is often realized via remote procedure calls (RPC). Decoupling components with RPCs allows them to scale independently—in principle, developers need only agree on a common interface and message format to leverage the benefits of software decoupling. Yet, in reality, RPCs enforce strict interface constraints and often trade adaptability (narrow interfaces are harder to evolve) for simplicity (narrow interfaces limit cross-component interactions), ultimately hampering the goal of scalability.

The chief problem with RPCs is that they are fundamentally location- and compute-centric: RPCs force a programmer to decouple an application by explicitly separating the computational endpoint or *location* where a function is invoked from the location where the function executes. As a consequence, they are well-suited to a relatively narrow set of use cases in which function arguments (which flow from invoker to executor) and returns (which flow back) must be serialized and sent in their entirety, and hence are small, and in which reference data must be located on the executor.

Many scenarios would benefit from decoupling but are simply not feasible using existing RPC mechanisms. For example, the invoking endpoint may have abundant data but limited compute, the invoker may wish to traverse a remote data structure, or the invoker may wish to refer to data that they lack privileges to read. In Section 2, we discuss how the increasingly important problem of distributed inference for edge devices can suffer from all of these problems. Rapidly growing model sizes, privacy concerns, and the proliferation of last-mile model customizations all exacerbate the issue.

To mitigate the problem of location-centric RPCs, data center operators often deploy discovery services, load balancers, or other forms of middleware [9, 12, 20, 28, 31]. These extra indirection layers make the execution endpoint abstract, but at the cost of increased latency and added system complexity. Moreover, we argue that such systems do not address the fundamental problem, which is *we need a more general mechanism for module composition in distributed systems.*

This mechanism must be more flexible than RPCs, but not at the cost of simplicity or performance. Satisfying these conflicting goals requires a shift in our programming models, from *location-centric* abstractions such as RPC to *data-centric* abstractions more akin to distributed shared memory (DSM). These data-centric abstractions can free programmers from

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

HotNets ’21, November 10–12, 2021, Virtual Event, United Kingdom

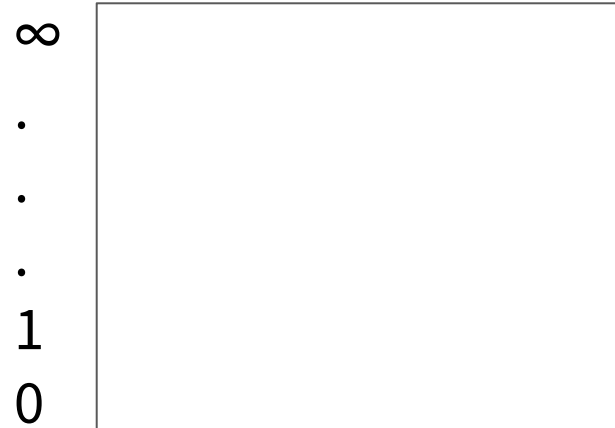
© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9087-3/21/11.
https://doi.org/10.1145/3484266.3487389

But what about sharing and synchronization?

(Immutability is the trivial case)

Multiplicity of Writers

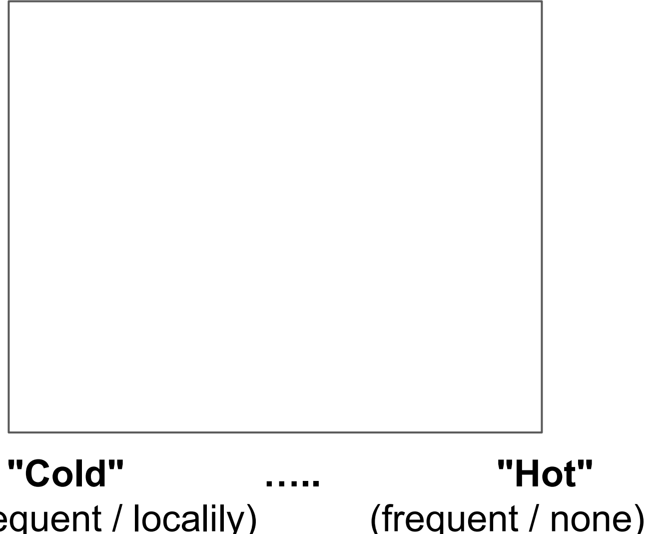


1: Multiplicity of Writers

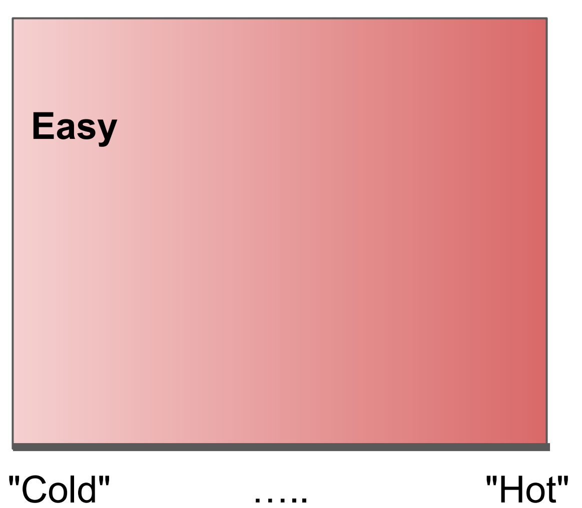
Support Mechanisms:

Gossip
MSTs
Pub/Sub

2: "Hotness" in Frequency and Locality



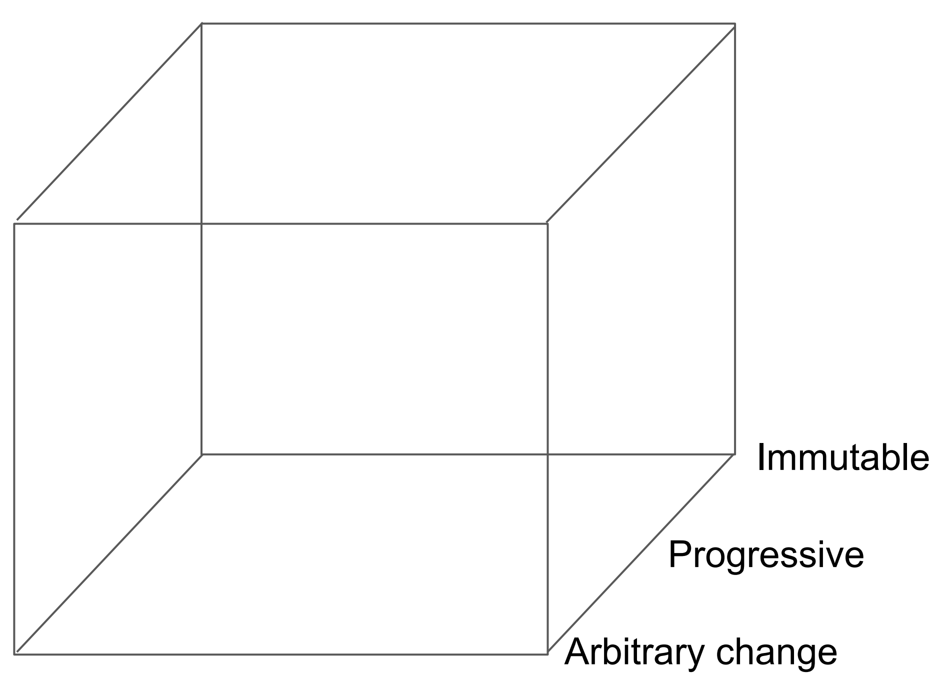
2: "Hotness" in Frequency and Locality



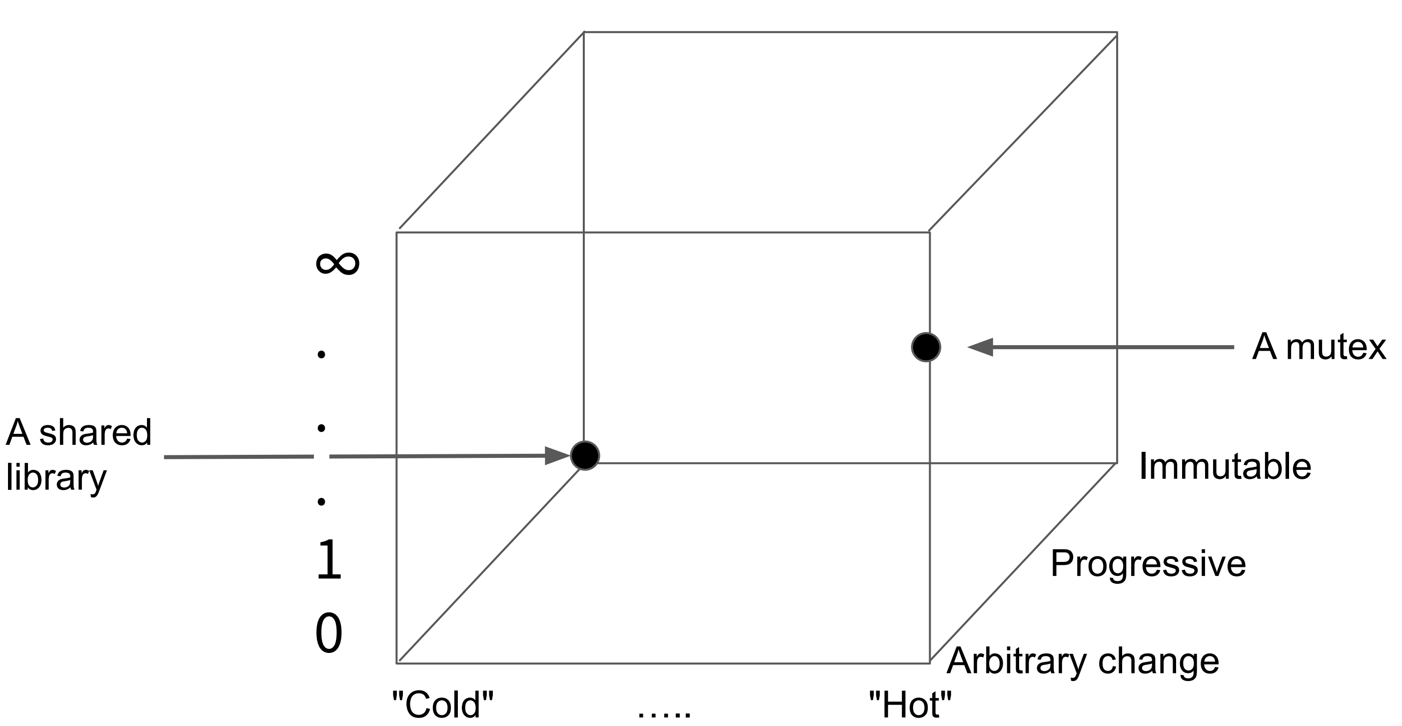
Support Mechanisms:

Partitioning
Caching

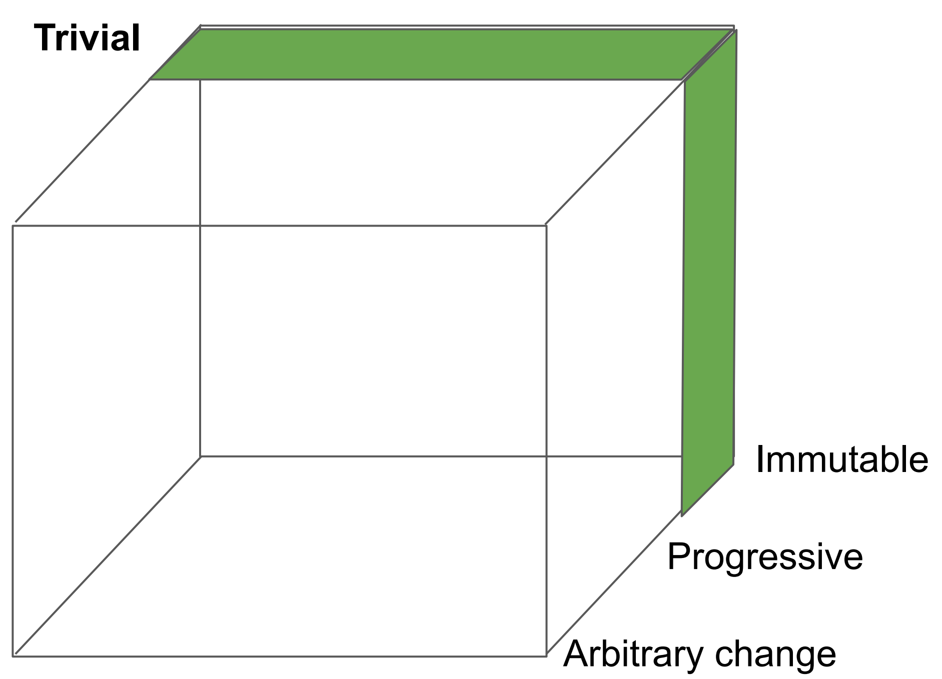
3: *How* objects are permitted to change



for example

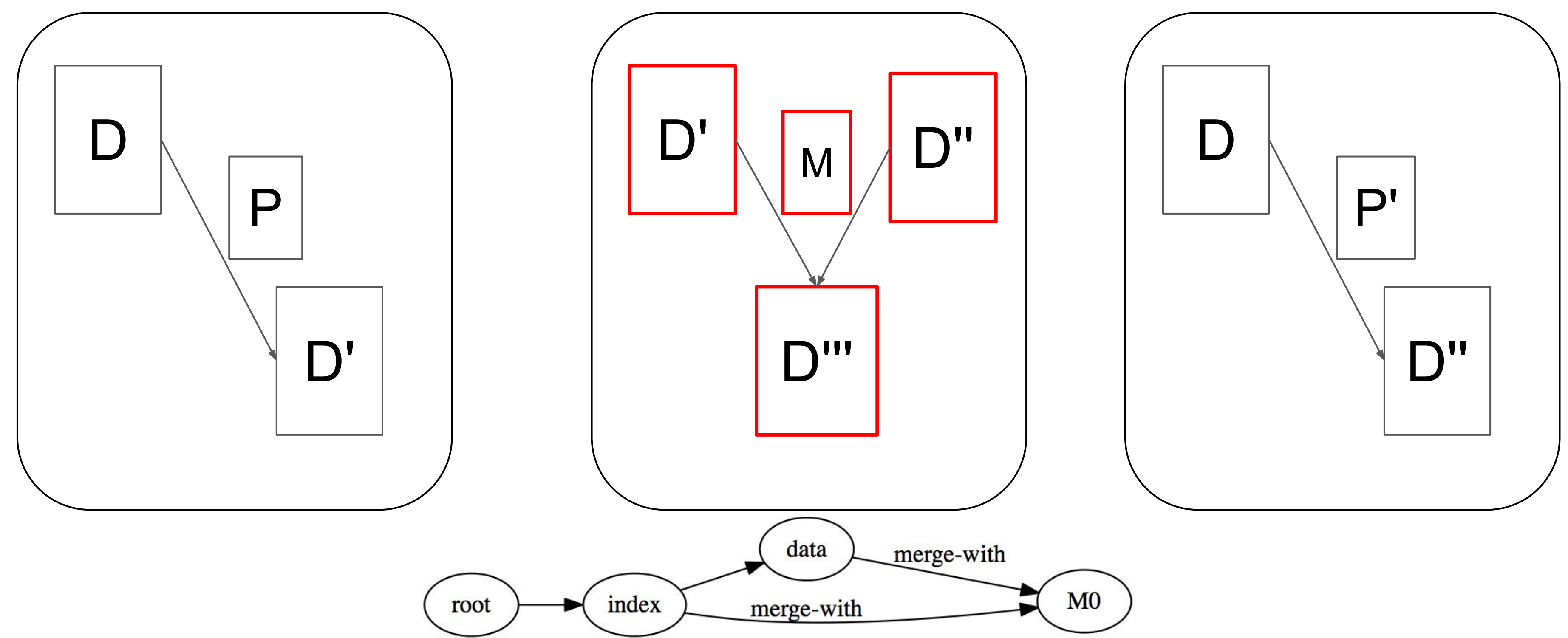


3: *How* objects are permitted to change

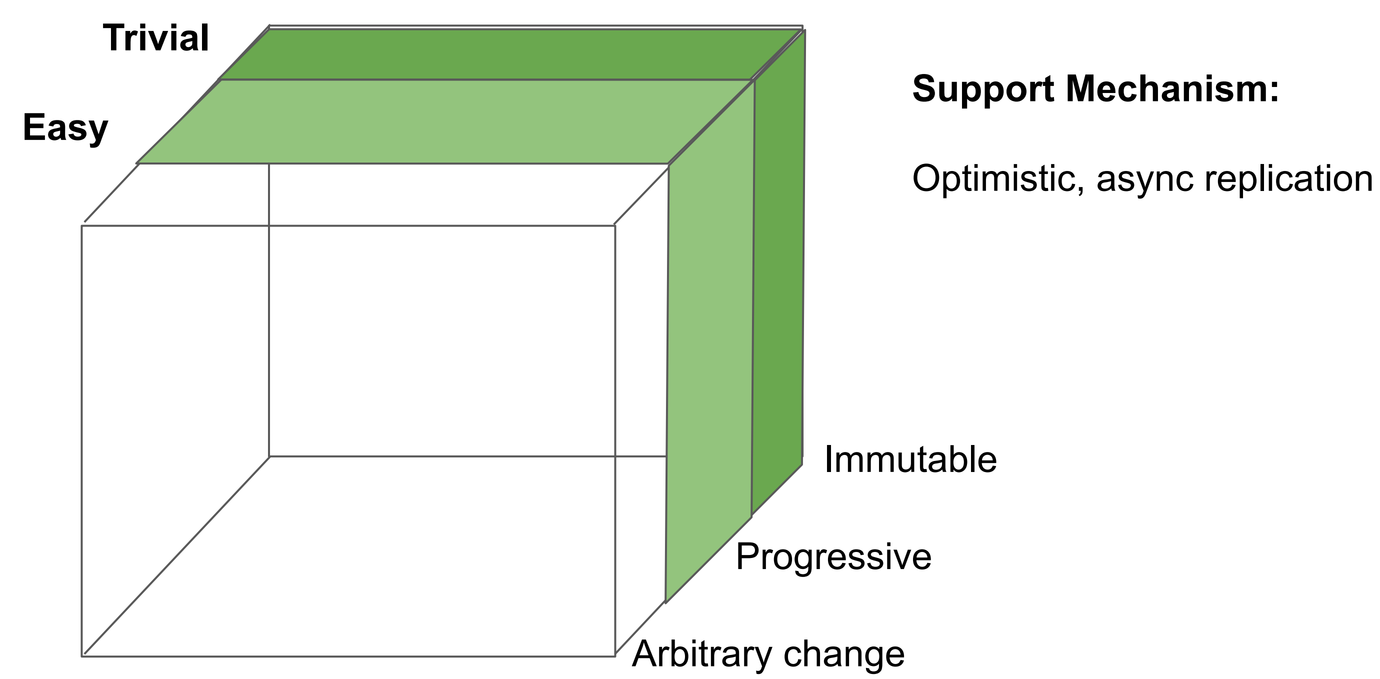


Progressive Objects

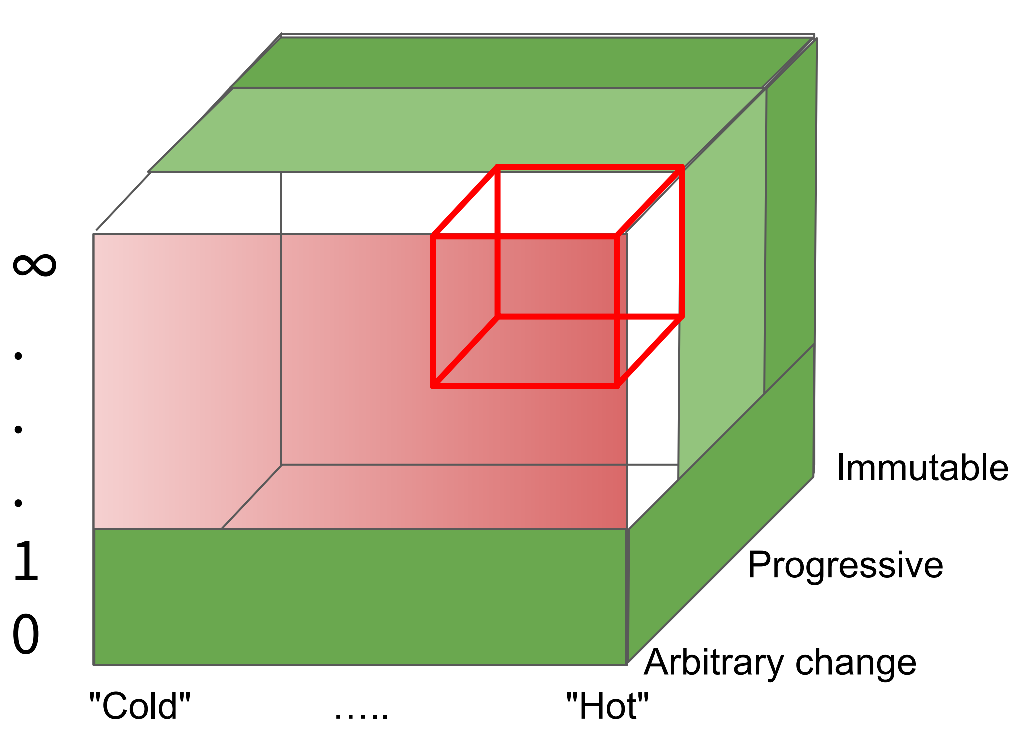
Identity enables system-level program analysis



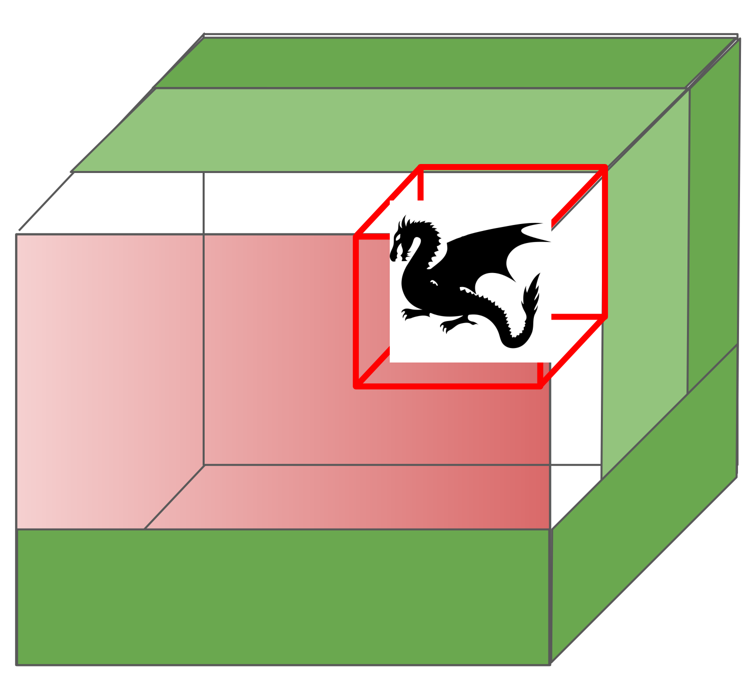
How objects are permitted to change



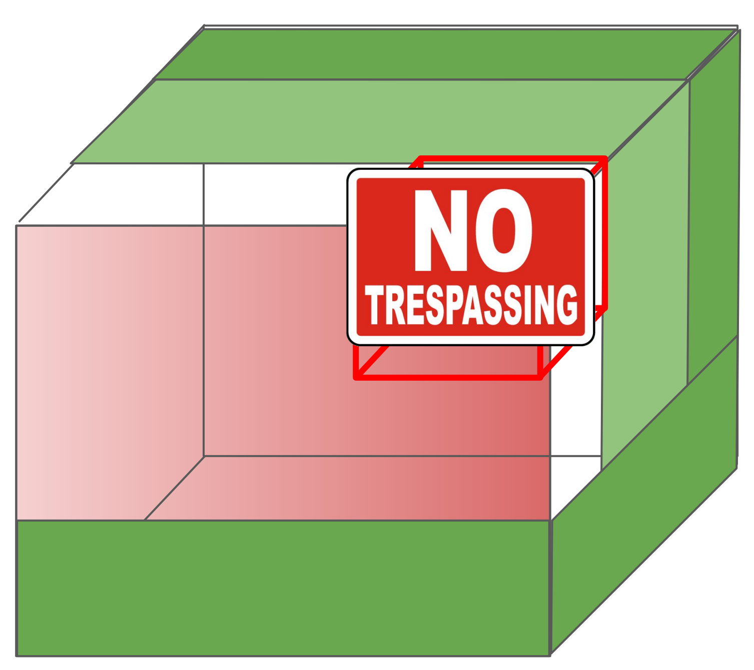
The hard stuff



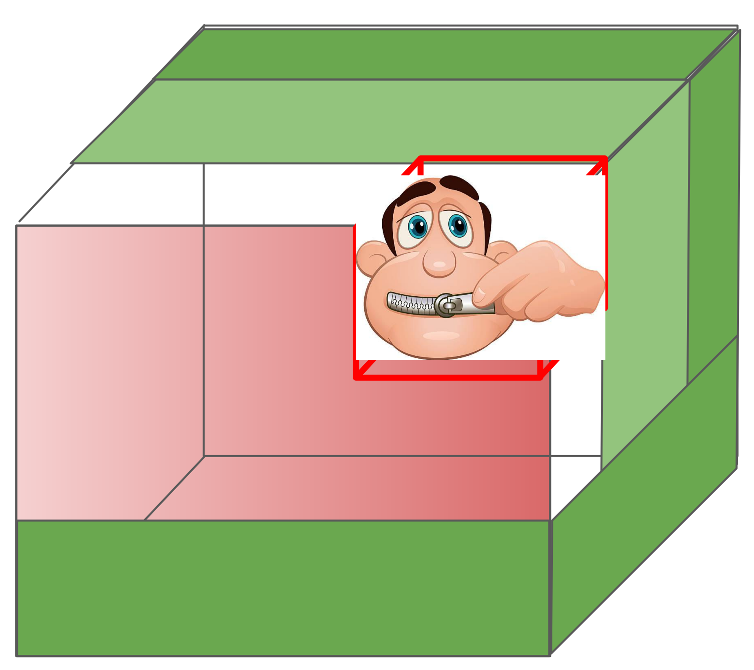
The hard stuff



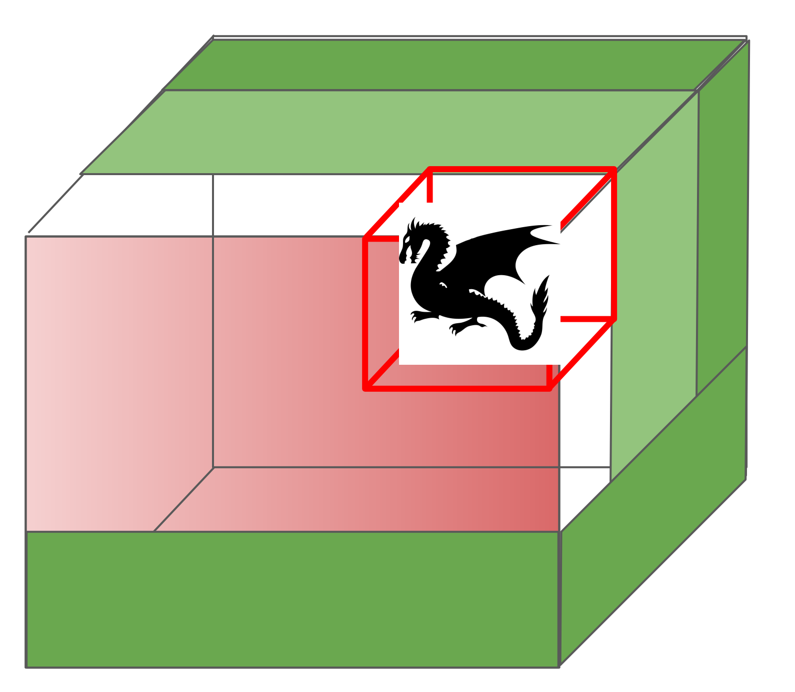
If we were building an infrastructure

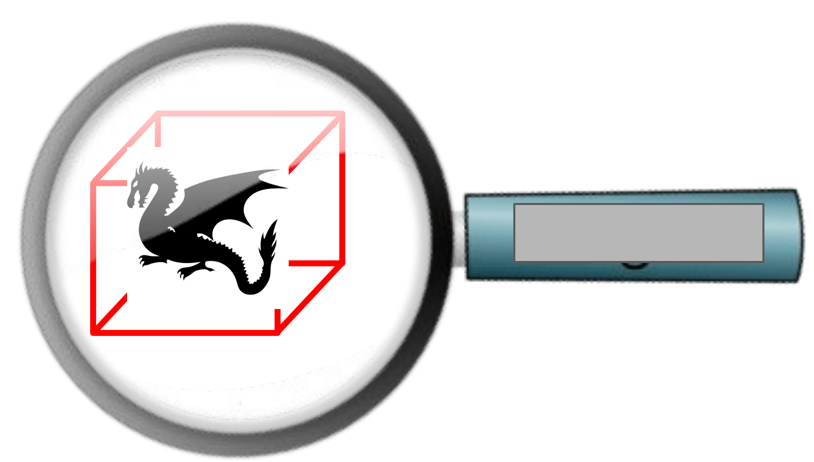


If we were designing a language

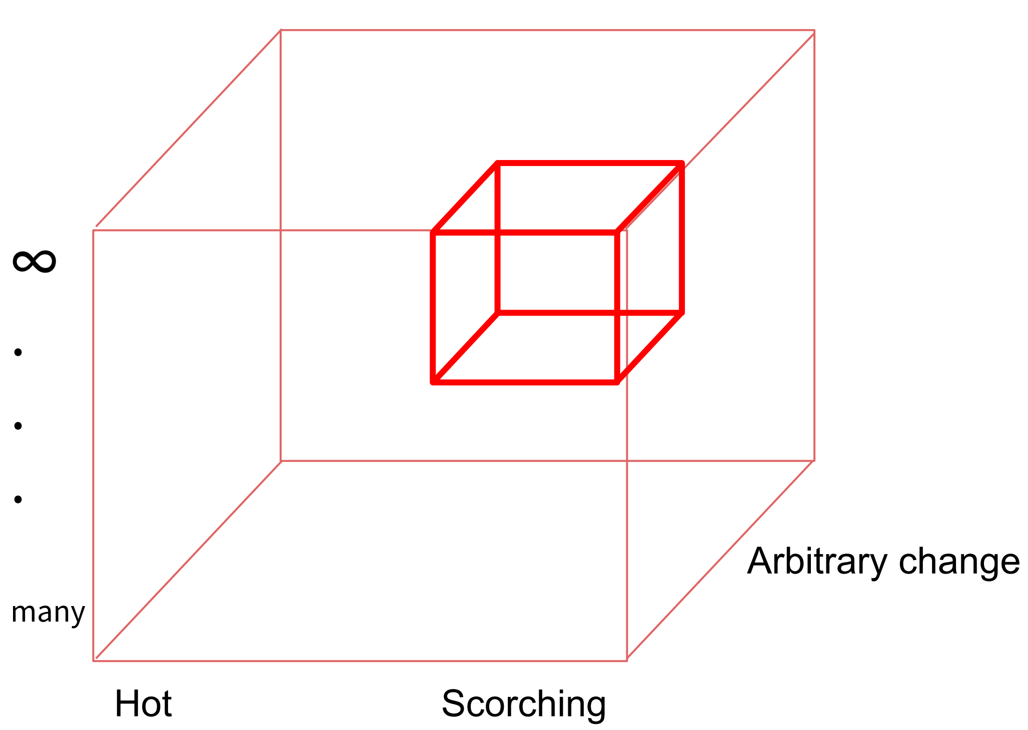


We are designing an operating system

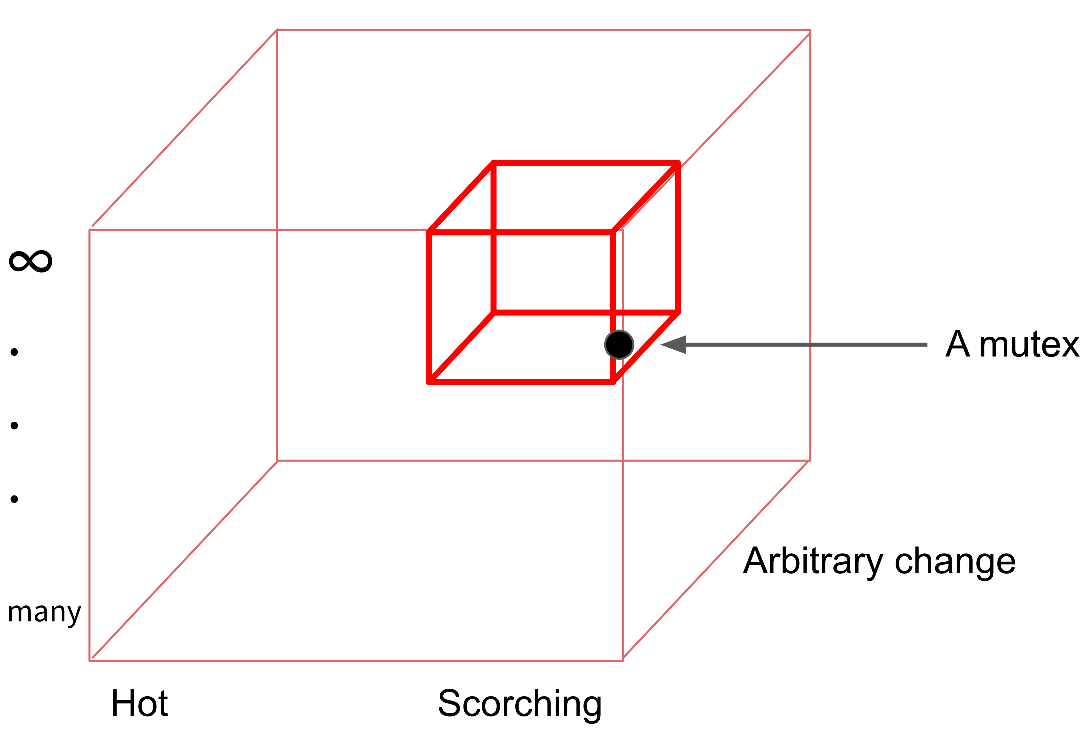




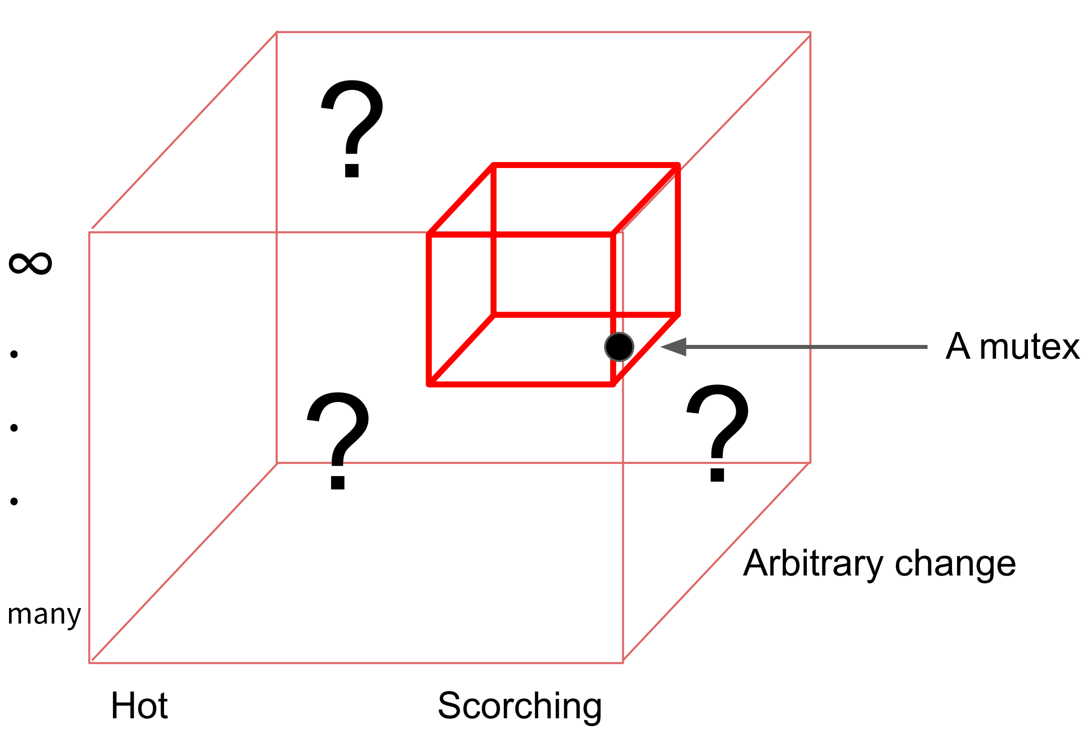
The bad place



The bad place



The bad place



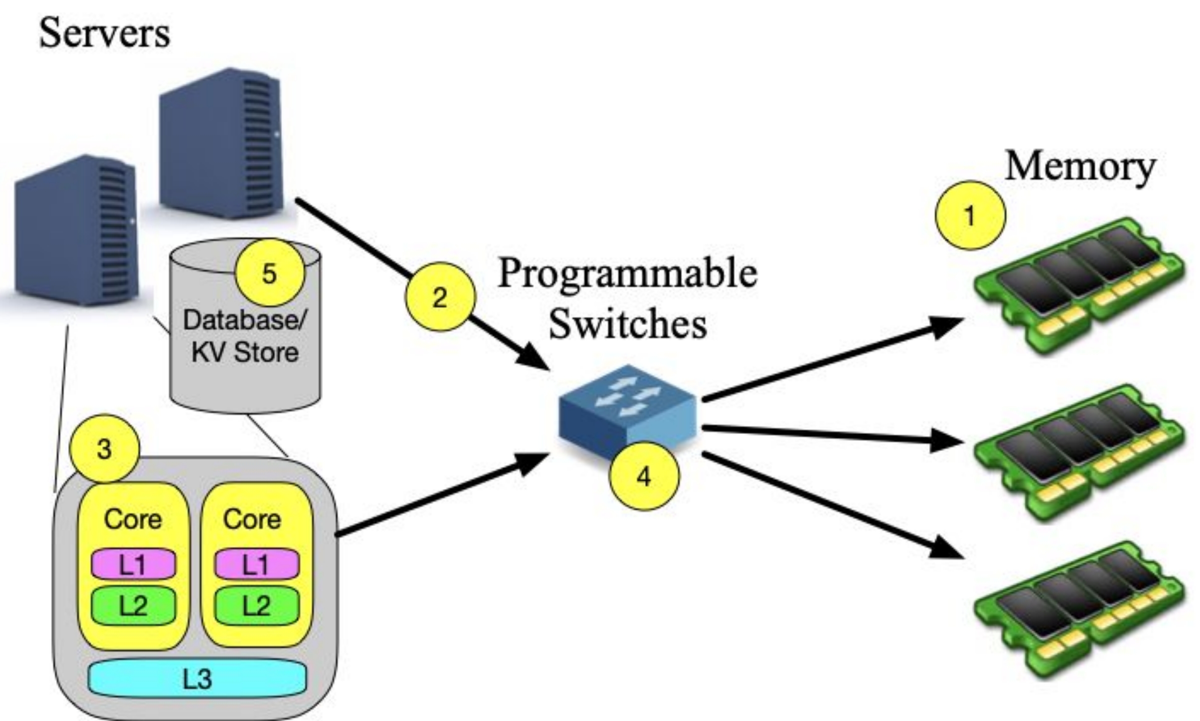
Scaleable cache coherence?

Ideas:

Network as bus: OS/Network codesigns

Glue that makes it possible: **global identity**

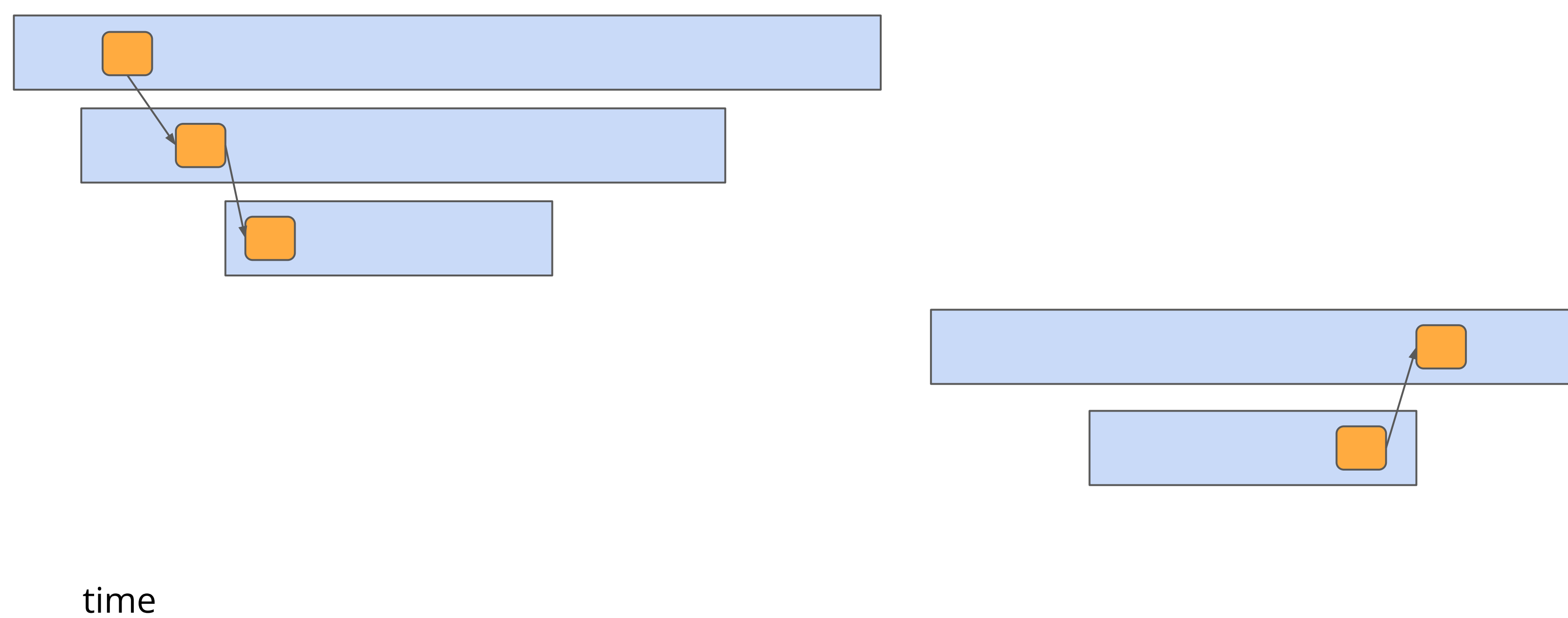
The network is the bus?

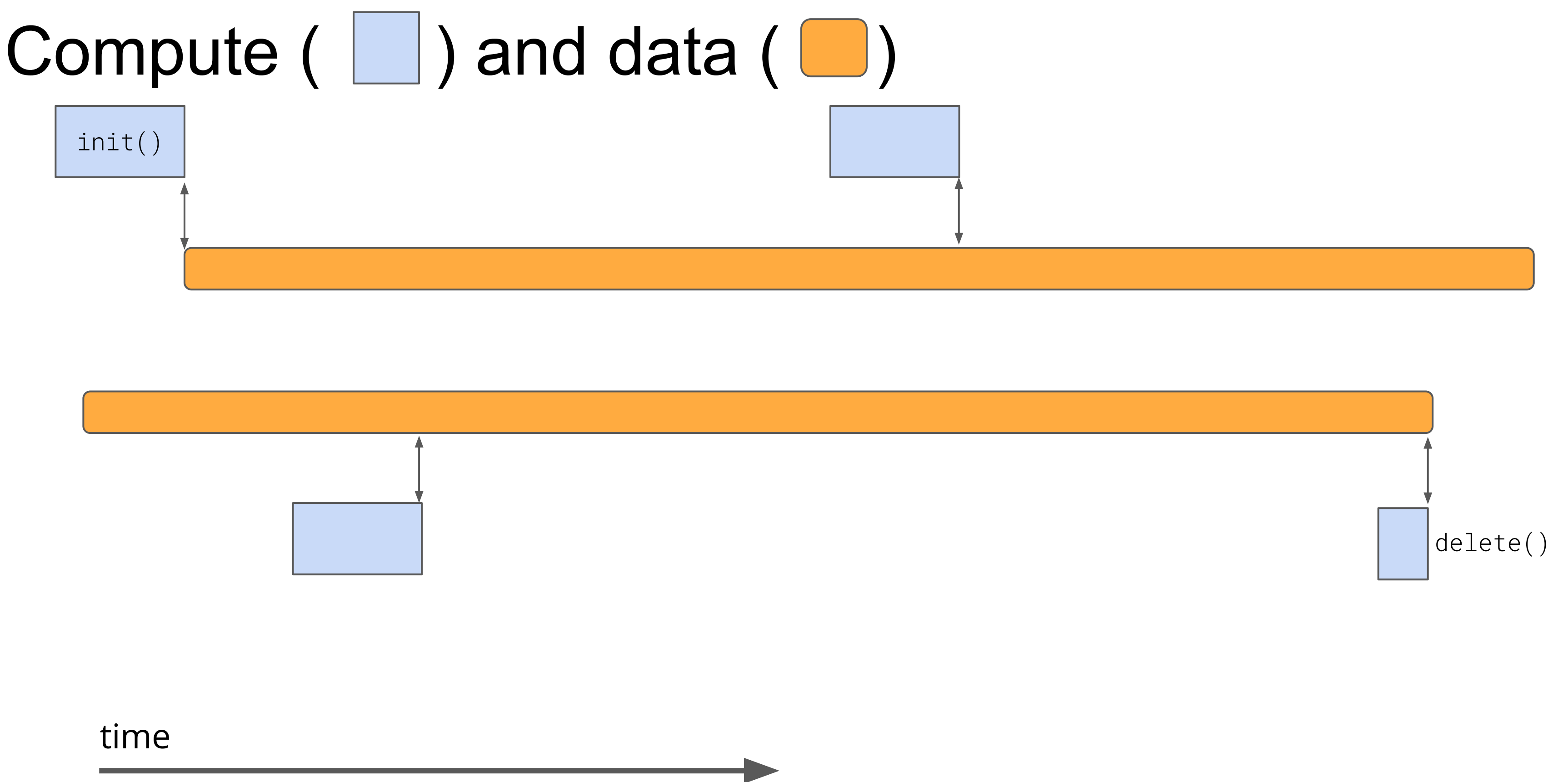


Organizing computation

Inversion of lifetimes

Compute ( ) and data ( )




Compute ( ) and data ( )
init()
delete()
time

The nanotransaction (AKA Nando)

A bounded sequence of operations on references to persistent data

May invoke a data-dependent number of other nanotransactions

A unit of locality, atomicity, and mobility

Nandos think locally

Nandos only perform local computation over local data. Hence:

They never wait (but they may speculate)

They *never* invoke coordination logic (e.g. 2-phase commit)

Nandos cut at the bones

Nandos ask little of the programmer

Only this: what are the boundaries of local computation?

Articulation

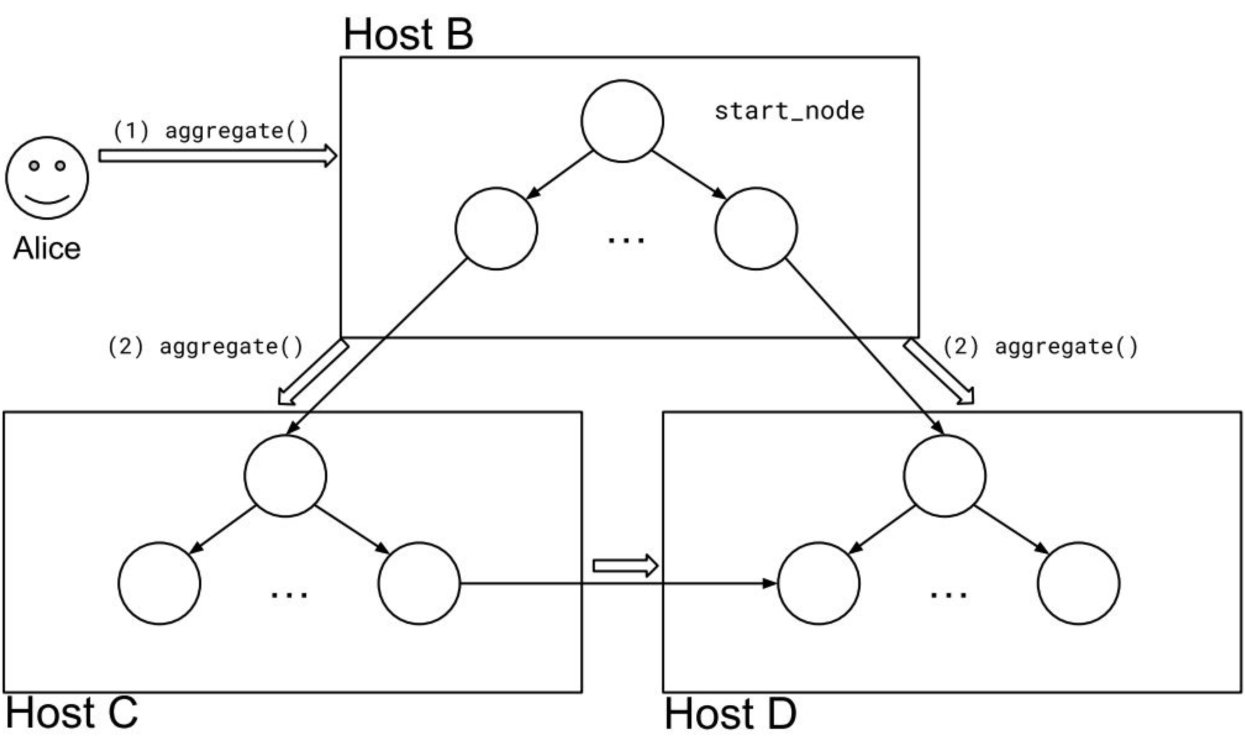
```
struct Node {
  value: u64;
  neighbors: List<Node>;
};

struct Aggregator {
  sum: Counter;
  visited: Set<Node>;
}
```

(a) Objects

```
let rec aggregate = nano(|
  node: &Node, output: &Aggregator
| {
  if node in output.visited {
    return;
  }
  output.visited.insert(node);
  output.sum += node.value;
  for neighbor in node.neighbors {
    aggregate(neighbor, output);
  }
});
```

(b) Nanotransaction

Articulation



```
struct TripDetails {
  user: &User;
  flight: &Flight;
  hotel: &Hotel;
  car: &Car;
}

struct TripBooking {
  flight_bk: &FlightBooking;
  hotel_bk: &HotelBooking;
  car_bk: &CarBooking;
  payment: &Transaction;
}
```

(a) Objects

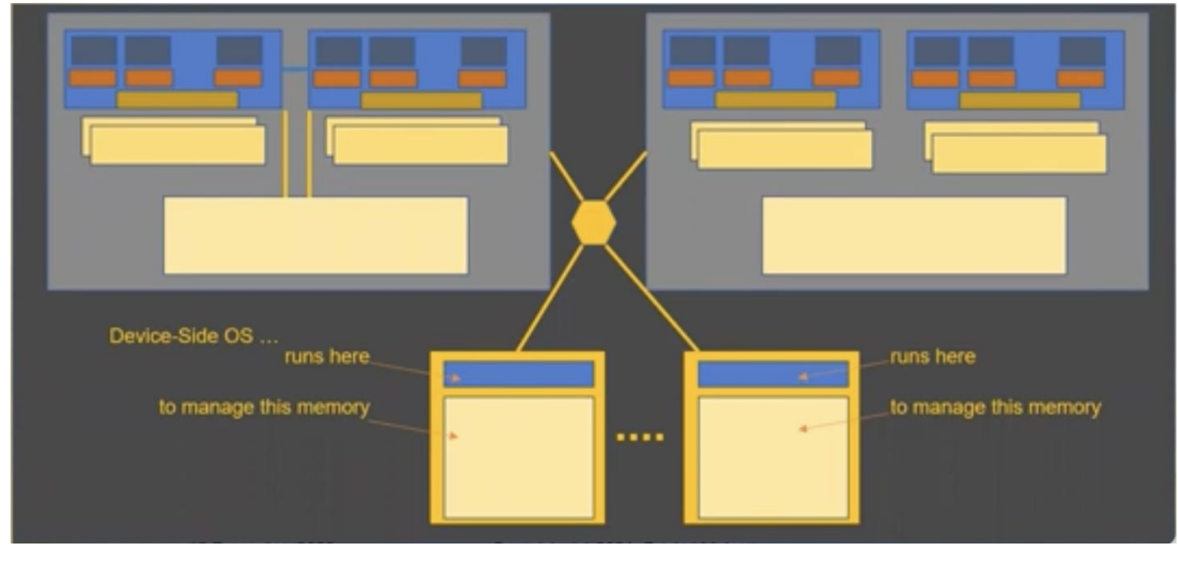
```
let book_trip = nano(|
  details: &TripDetails,
  bk: &TripBooking,
| {
  book_flight(details.flight,
    bk.flight_bk);
  book_hotel(details.hotel,
    bk.hotel_bk);
  book_car(details.car, bk.car_bk);
  let trip_amt = /* bookings sum */;
  charge_user(details.user.id,
    trip_amt, bk.payment);
});
```

(b) Nanotransaction

What else?

The "TWISTED Stick"

Elephance, the memory company



Daniel Bittman

UC Santa Cruz

Peter Alvaro

UC Santa Cruz

Achilles Benetopoulos

UC Santa Cruz

Allen Aboytes

UC Santa Cruz

Pankaj Mehra

Darrell D. E. Long

Ethan L. Miller

George Neville-Neil

Support

National Science Foundation

Defense Advanced Research Projects Agency

Intel

and gifts from **Ebay** and **Meta**.

What not where

Data lasts forever; computation comes and goes

Ready or not, far-out memory hierarchies are here

Follow the data

Twizzler

Thank you!

Twizzler

twizzler.io