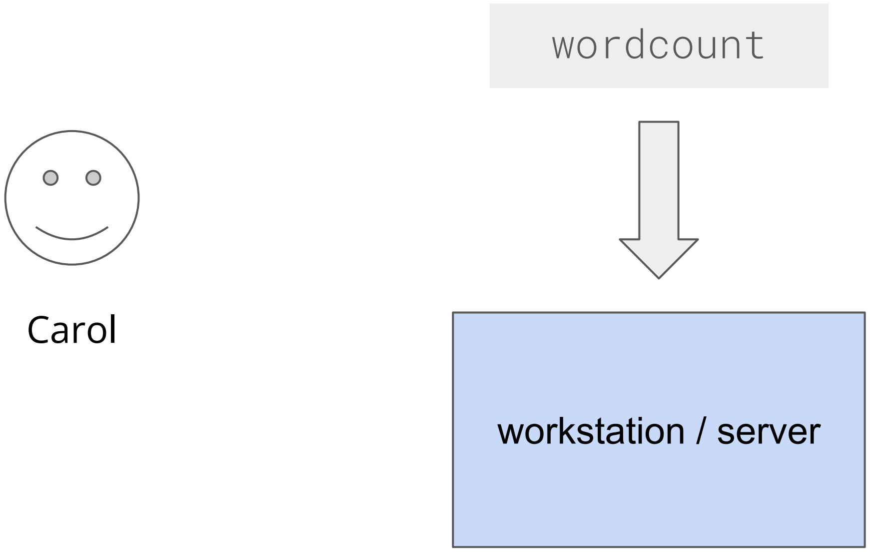
A data-first, hands-free, distributed programming model

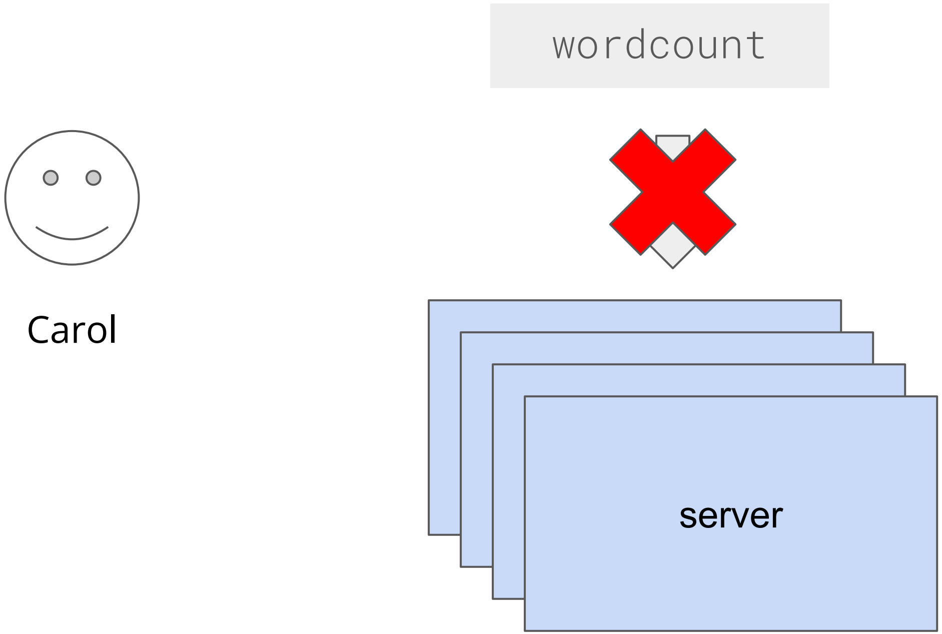
Achilles Benetopoulos (abenetop@ucsc.edu)

Motivation

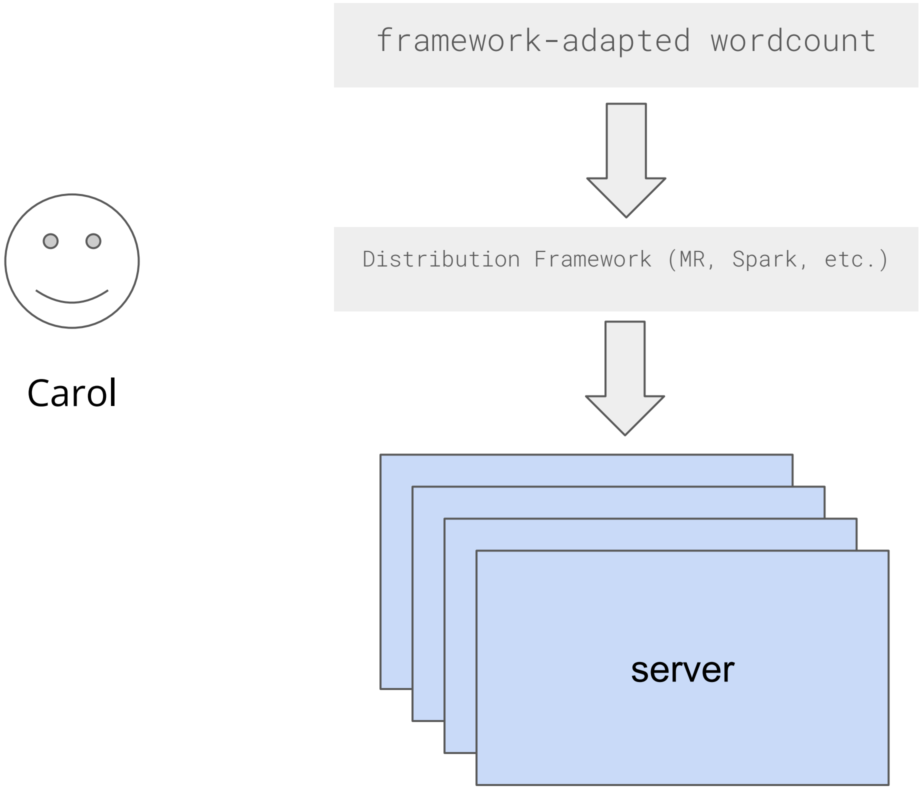
Use Case – Word Frequency



Use Case – Word Frequency



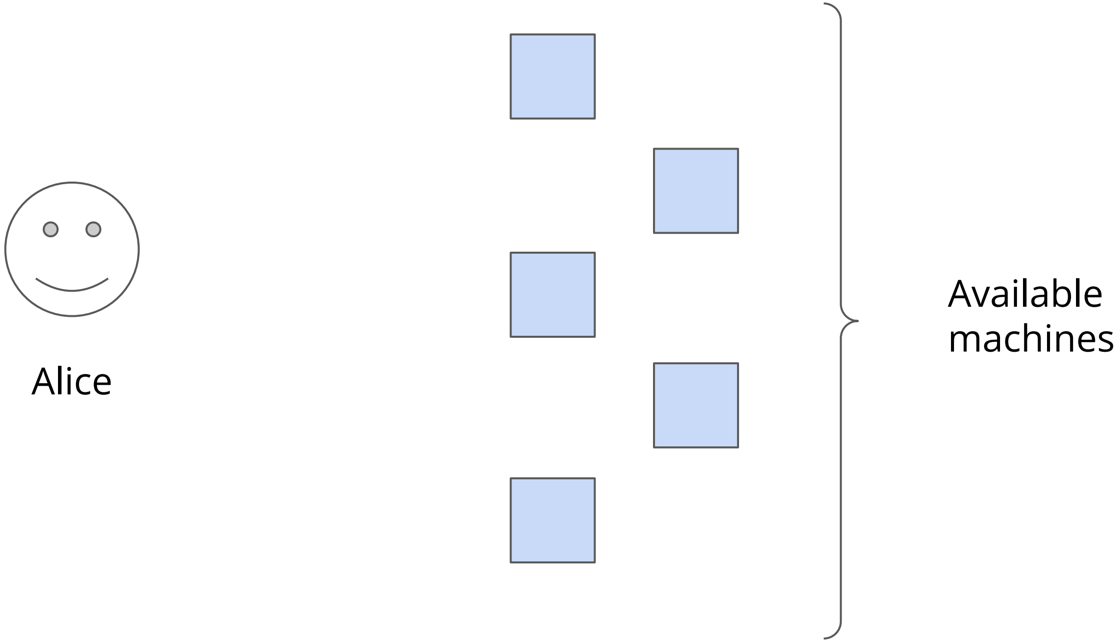
Use Case – Word Frequency



Shortcomings of Existing Approaches

- Users have to think about their problem through the underlying system's mechanisms

Use Case – Distributed Graph Processing



Use Case – Distributed Graph Processing

Alice

Reference	Ds?	Data location	Arch.	F?	Con?	B?	sB?	T?	acid?	P?	L?	S?	D?	Edge updates	Vertex updates	Remarks
STINGER [79]	✖	M-mem.	CPU	S	✖	■	■	✖	✖	◧	■	✖	✖	■ (A/R)	◧* (A/R)	*Removal is unclear
UNICORN [222]	■	M-mem.	CPU	C	✖	■	✖	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	**Extends IBM InfoSphere Streams [45]**
DISTINGER [85]	■	M-mem.	CPU	S	✖	■	✖	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	**Extends STINGER [79]**
cuSTINGER [103]	✖	GPU mem.	GPU*	S	✖	■	✖	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	**Extends STINGER [79]**. *Single GPU.
EvoGraph [205]	✖	M-mem.	GPU*	C	✖	■	✖	✖	✖	◧	■	✖	✖	■ (A/R)	■ (A/R)	Supports multi-tenancy to share GPU resources. *Single GPU.
Hornet [49]	✖	GPU, M-mem.	GPU†	S	✖*	■	■	✖	✖	✖	■	✖	✖	■ (A/R/U)	■ (A/R/U)	*Not mentioned. †Single GPU
GraPU [210], [211]	■	M-mem., disk	CPU	C	✖	■	✖*	✖	✖	✖	■	✖	✖	■ (A/R)	✖	*Batches are processed with non-straightforward schemes
Grace [193]	✖	M-mem.	CPU	S+C	■ (s:C)	■	?	■	■	◧†	■	✖	✖	■ (A/R/U)	■ (A/R)	†To implement transactions
Kineograph [56]	■	M-mem.	CPU	C+S	■ (s:P)	■	✖	■	✖	■	■	■	■	■ (A/U*)	■ (A/U*)	*Custom update functions are possible
LLAMA [162]	✖	M-mem., disk	CPU	S	■ (s:C)	■	■	✖	✖	■	■	✖	✖	■ (A/R)	■ (A/R)	—
CellIQ [120]	■	Disk (HDFS)	CPU	C	■ (s)	?	✖	✖	✖	■	■	■	✖	■ (A/R)	■ (A/R)	**Extends GraphX [101] and Spark [244]**. *No details.
GraphTau [121]	■	M-mem., disk	CPU	C	■ (s)*	■	✖	✖	✖	■	■	■	✖	■ (A/R)	■ (A/R)	**Extends Spark**. *Offers **more** than simple snapshots.
DeltaGraph [69]	✖	M-mem.	CPU	C	■ (s:C)*	✖	✖	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	*Relies on Haskell's features to create snapshots
GraphIn [206]	✖*	M-mem.	CPU	C+S	◧ (s)	■	✖	✖	✖	✖†	■	✖	✖	◧* (A/R)	◧* (A/R)	*Details are unclear. †Only mentioned
Aspen [71]	✖	M-mem., disk	CPU	S+C	■ (s:C)*	?	?	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	*Focus on lightweight snapshots; enables **serializability**
Tegra [122]	■	M-mem., disk	CPU	C+S	■ (s)	?	?	✖	✖	■	◧*	■	?	■ (A/R)	■ (A/R)	**Extends Spark**. *Live updates are considered but outside core focus.
GraphInc [51]	■	M-mem., disk	CPU	C	■ (s)*	?	?	✖	✖	✖	■	✖	✖	■ (A/R/U)	■ (A/R/U)	**Extends Apache Giraph [1]**. *Keeps separate storage for the graph structure and for Pregel computations, but no details are provided.
ZipG [139]	■	M-mem.	CPU	S+C	■ (s)	?	?	✖	✖	◧	■	✖	✖	■ (A/R/U)	■ (A/R/U)	**Extends Spark & Succinct [5]**
GraphOne [148]	✖	M-mem.	CPU	S+C	■ (s:T)	■	■	✖	✖	◧	■	✖	✖	■ (A/R)	■ (A/R)	Updates of weights are possible
LiveGraph [250]	✖	M-mem., disk	CPU	S+C	■ (s:C)	✖	na	■	■	✖	■	✖	✖	■ (A/R/U)	■ (A/R/U)	—
Concerto [152]	■	M-mem.	CPU	S+C	■ (f)*	■	✖	■	■	✖	■	✖	✖	◧ (A/U)	◧ (A/U)	*A two-phase commit protocol based on fine-grained atomics
aimGraph [236]	✖	GPU mem.	GPU*	S+C	◧ (f)†	■	?	✖	✖	✖	■	✖	✖	■ (A/R)	✖	*Single GPU. †Only **fine** reads/updates are considered.
faimGraph [237]	✖	GPU, M-mem.	GPU*	S+C	◧ (f)†	■	■	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	*Single GPU. †Only **fine** reads/updates, using locks/atomics.
GraphBolt [166]	✖	M-mem.	CPU	C+S	◧ (f)*	■	■	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	**Uses Ligra [215]**. *Fine edge updates are supported.
DZiG [165]	✖	M-mem.	CPU	C+S	■ (f)	■	?	✖	✖	✖	■	✖	✖	■ (A/R)	■ (A/R)	
RisGraph [86]	✖	M-mem.	CPU	C/S	■ (sc)*	■†	?	✖	✖	■	■	✖	✖	■ (A/R)	◧ (A/R)	*Details in § 5.1.
GPMA (Sha [207])	◧*	GPU mem.	GPU*	S	◧ (o)†	■	?	✖	✖	✖	■	■	✖	■ (A/R)	✖	*Multiple GPUs within one server. †Details in § 5.1.
KickStarter [233]*	■	M-mem.	CPU	C	na*	■	na*	na*	na*	na*	■	na*	na*	■ (A/R)	?	**Uses ASPIRE [232]**. *It is a *runtime technique*.
Mondal et al. [178]	■	M-mem.*	CPU	C+S	◧†	?†	?†	■	■	✖	■	?†	?†	◧† (A)	◧† (A)	***Uses CouchDB as backend [15], †Unclear (relies on CouchDB)**
iGraph [126]	■	M-mem.	CPU	C	?	■	✖	✖	✖	✖	■	✖	✖	◧ (A/U)	◧ (A/U)	**Extends Spark**
Sprouter [2]	■	M-mem., disk	CPU	C	?	?	✖	✖	✖	✖	■	✖	✖	◧ (A)	?	**Extends Spark**

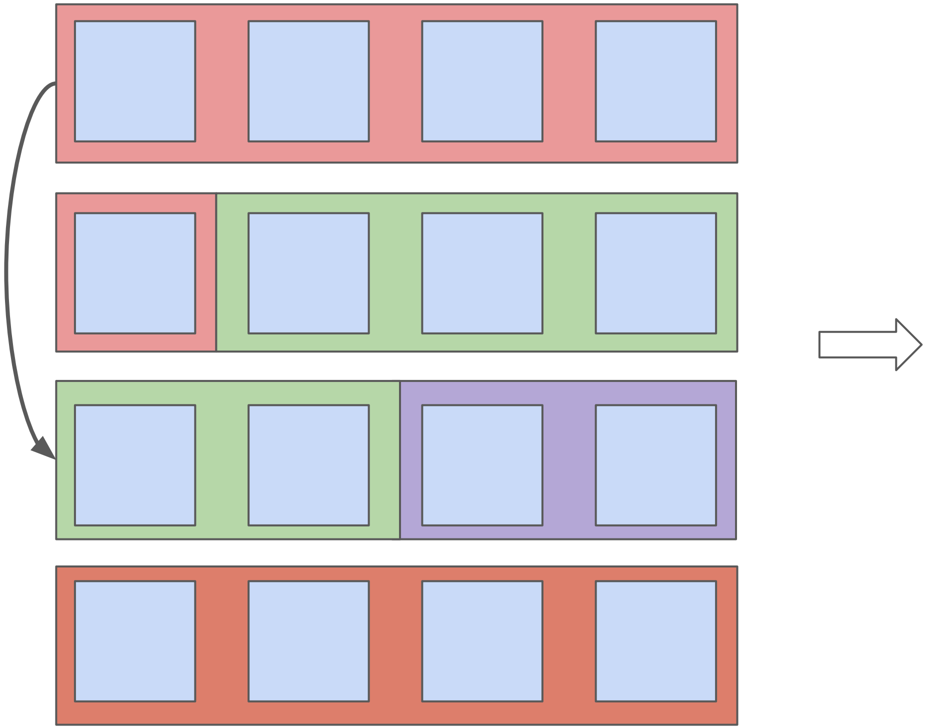
Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems (IEEE TPDS)

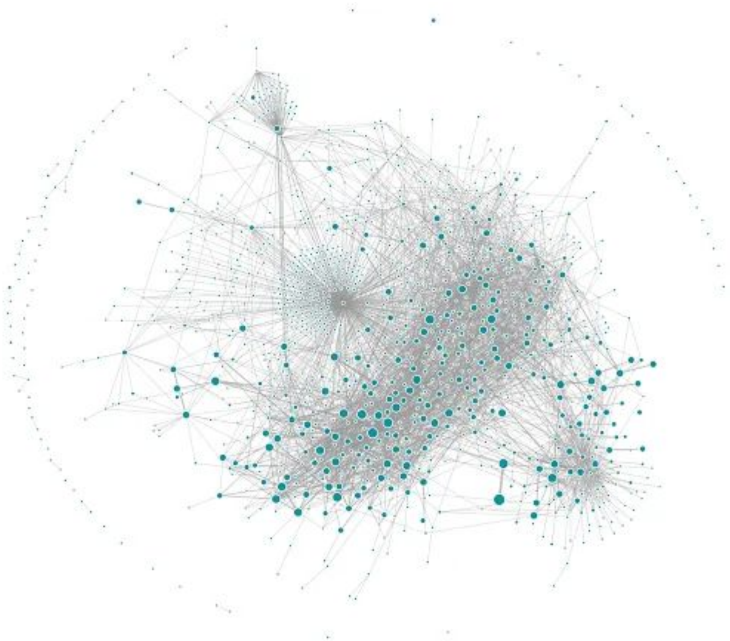
Shortcomings of Existing Approaches

- Users have to think about their problem through the underlying system’s mechanisms
- Users are limited in what they can express because of the underlying system’s distribution details

Short-lived computations over structured data

Use Case - Microservice Meshes





Uber's microservice architecture from Jaeger (2018)

Shortcomings of Existing Approaches

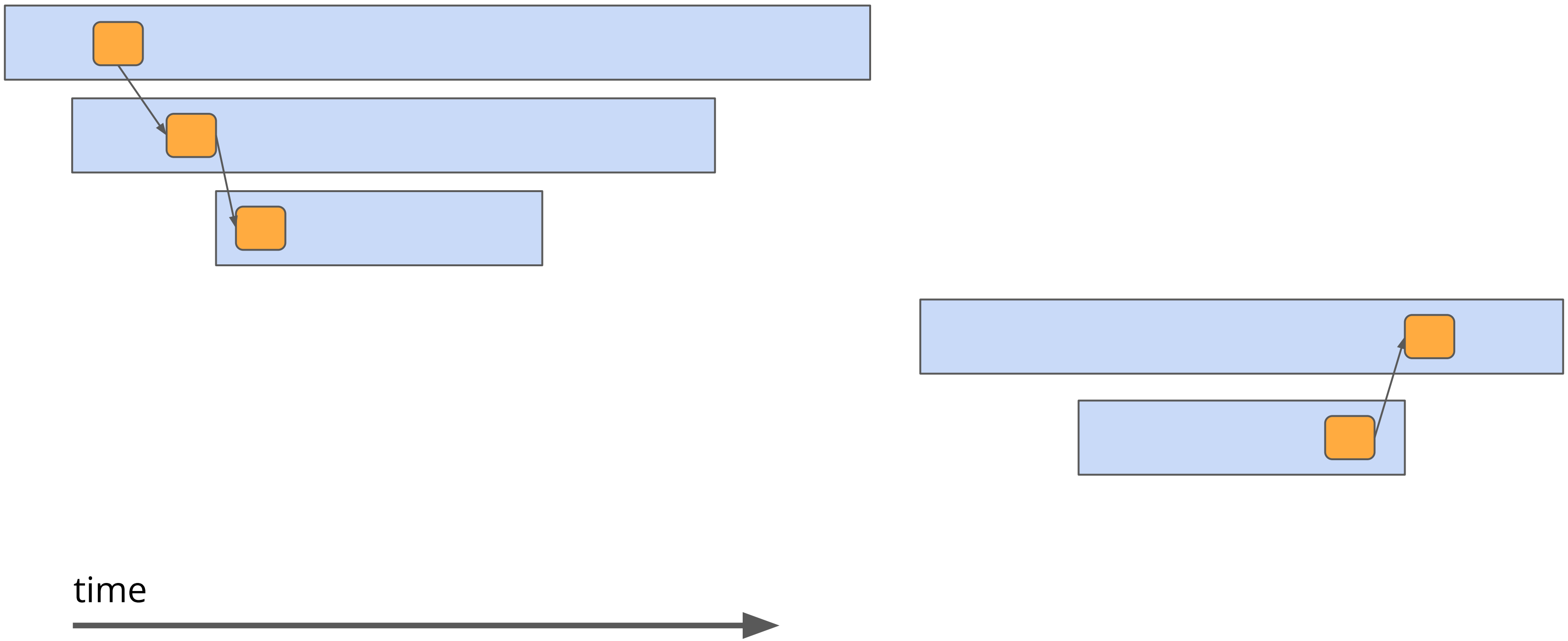
- Users have to think about their problem through the underlying system’s mechanisms
- Users are limited in what they can express because of the underlying system’s distribution details
- Systems have a hard time adapting end-to-end dynamically

Could We Do Better?

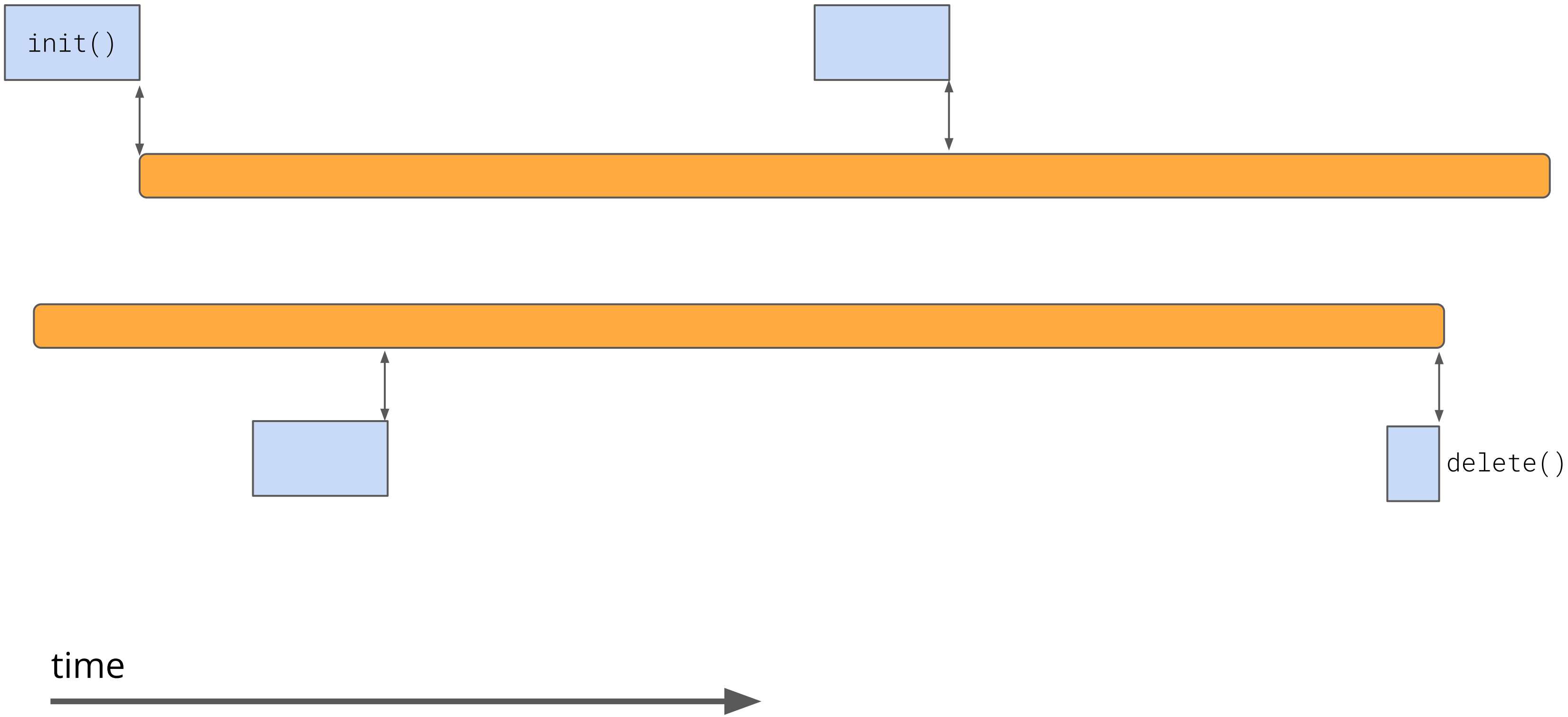
- Users have to think about their problem through the underlying system's mechanisms
  - Could we fulfill the promise of transparent distribution?
- Users are limited in what they can express because of the underlying system's distribution details
  - Could we do so while exposing a truly general-purpose programming model?
- Systems have a hard time adapting end-to-end dynamically
  - Could we use this model to construct more flexible systems?

Foundations

Compute ( ) and data ( )



Compute ( ) and data ( )

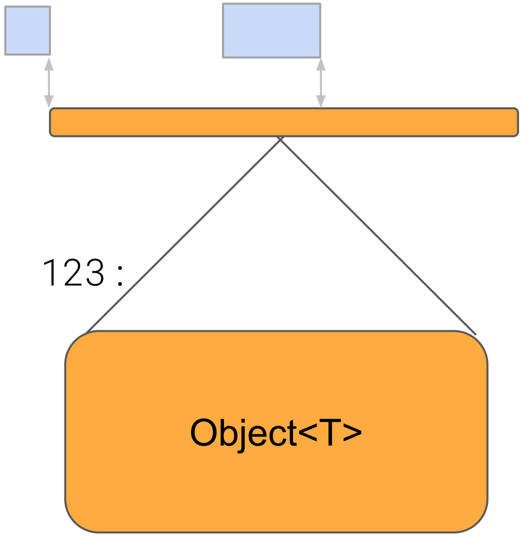


Objects: Organizing Memory

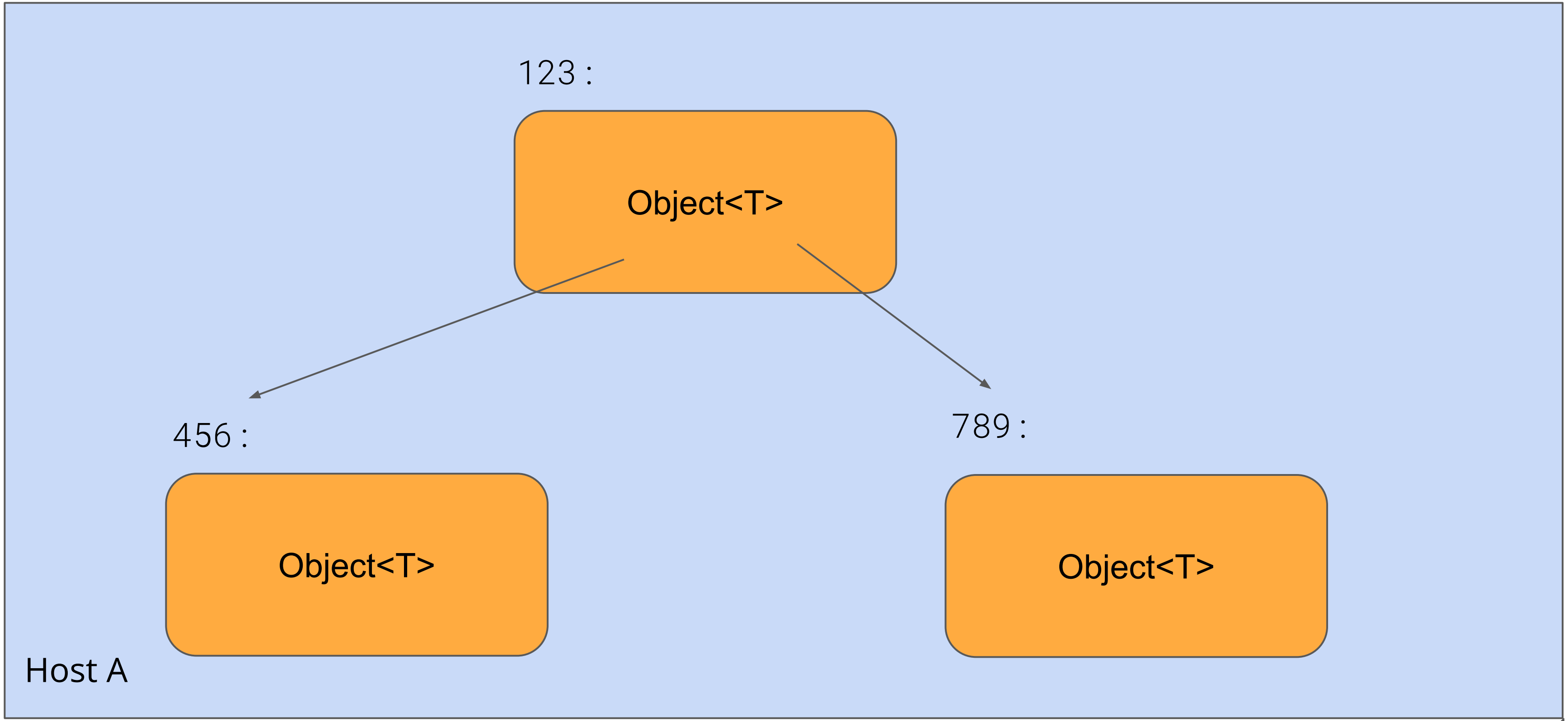
A typed region of semantically-related data items.

Unique, *invariant* identity in a global address space.

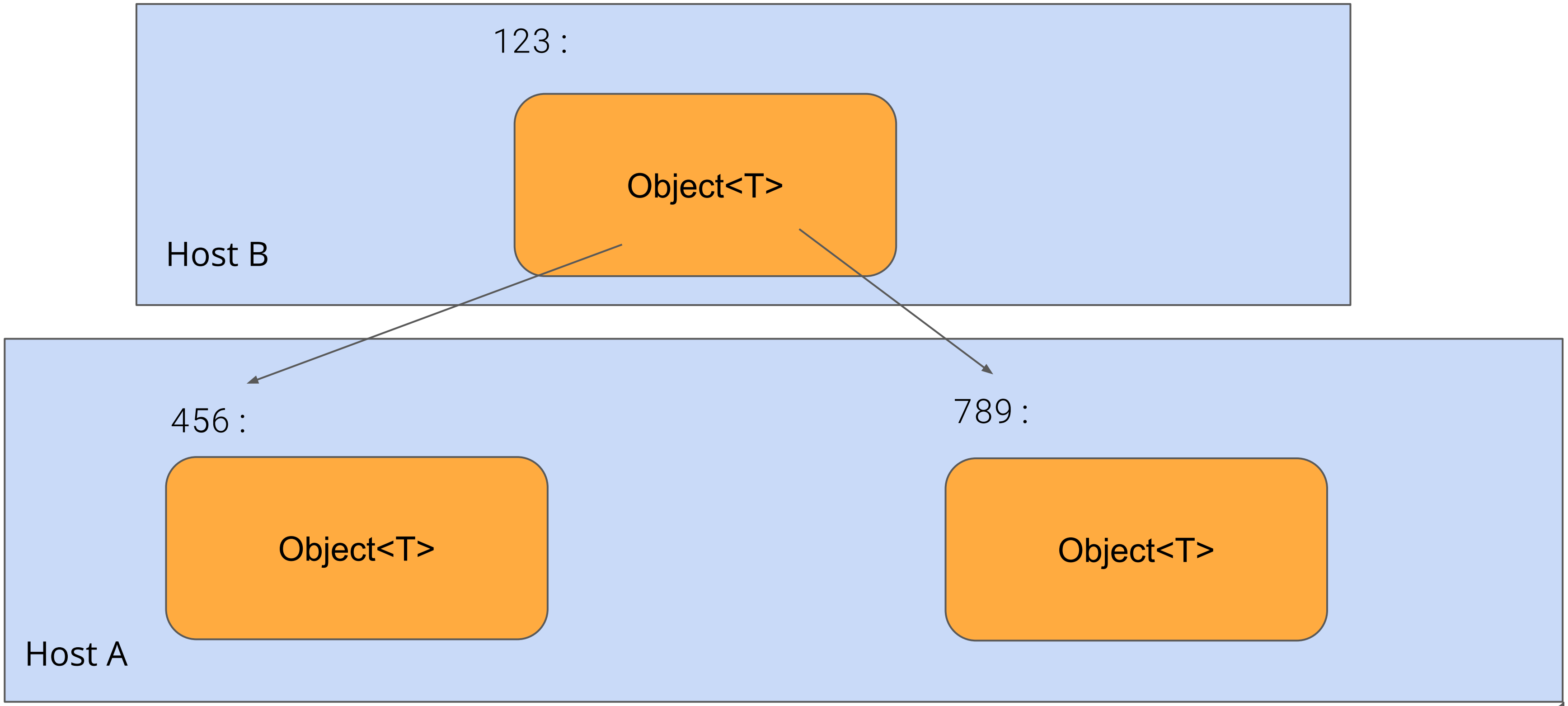
Object are mobile.



Objects: Organizing Memory

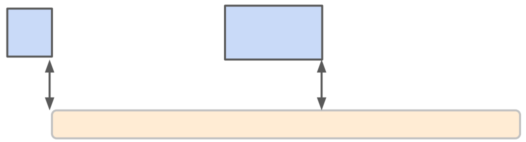


Objects: Organizing Memory



Nanotransactions: Organizing Computation

A constrained data access mechanism.



All accesses to objects happen only through nanotransactions.

- Unrestricted access to (shared) data makes it harder for the runtime to assist in distribution
- Transactional semantics ease the burden of consistency

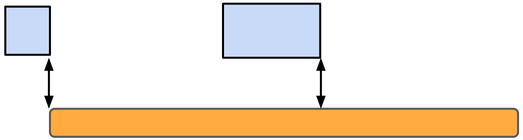
Nanotransactions: Organizing Computation

Nanotransactions are also mobile.

From the perspective of the nanotransaction, all data is local.

Local computation is much easier to express correctly.

Objects + Nanotransactions: Organizing Distribution



Our ask: factor your program into composable operations over local data.

Our promise: the runtime will do *the right thing**.

Possible because of:

- the visibility into application semantics
- the freedom around protocol

* maintain the integrity of their data, while optimally orchestrating execution

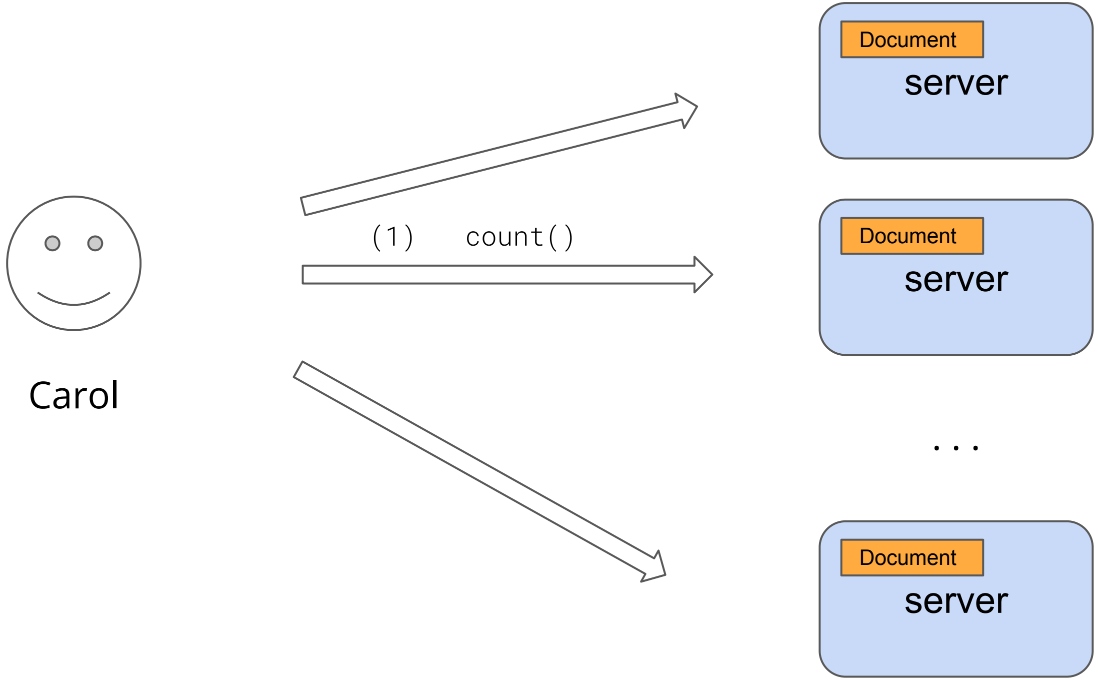
Use cases, through the data lens

Use case – Word Frequency

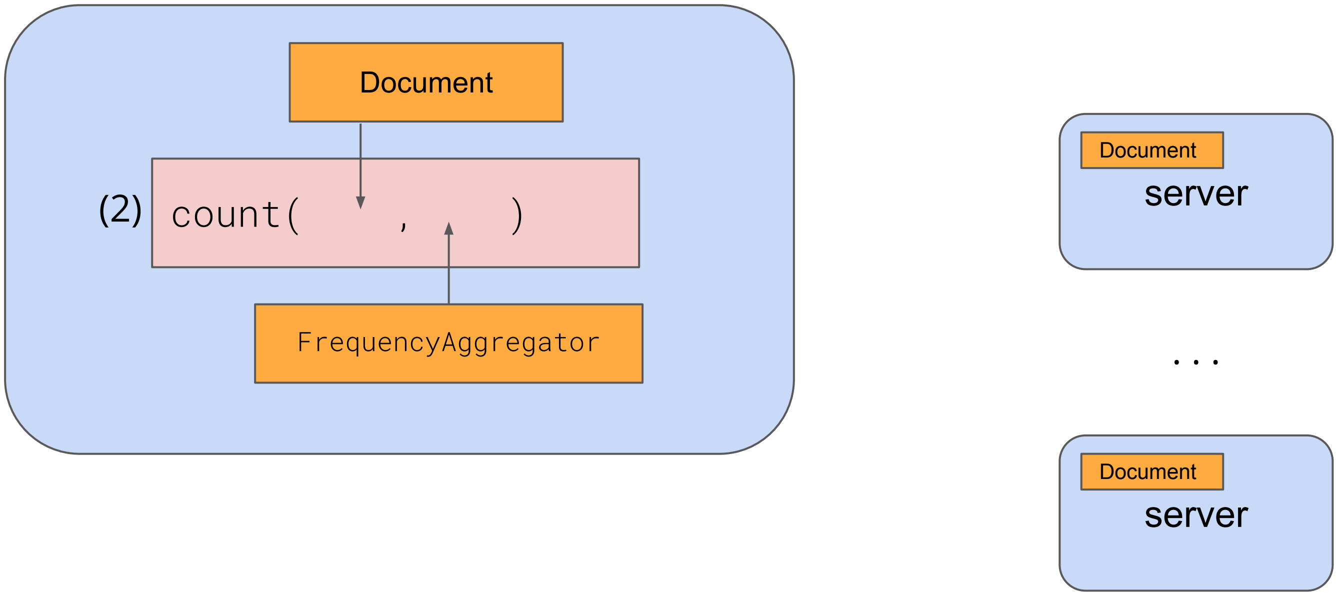
```
struct Document {
  lines : List<String>;
}

struct FrequencyAggregator {
  frequencies: Map<String, Counter>;
}
```

```
let count = nando(|
  body: &Document,
  output: &FrequencyAggregator,
| {
  for line in body.lines {
    for word in line.split(‘ ‘) {
      output[word] += 1;
    }
  }
});
```

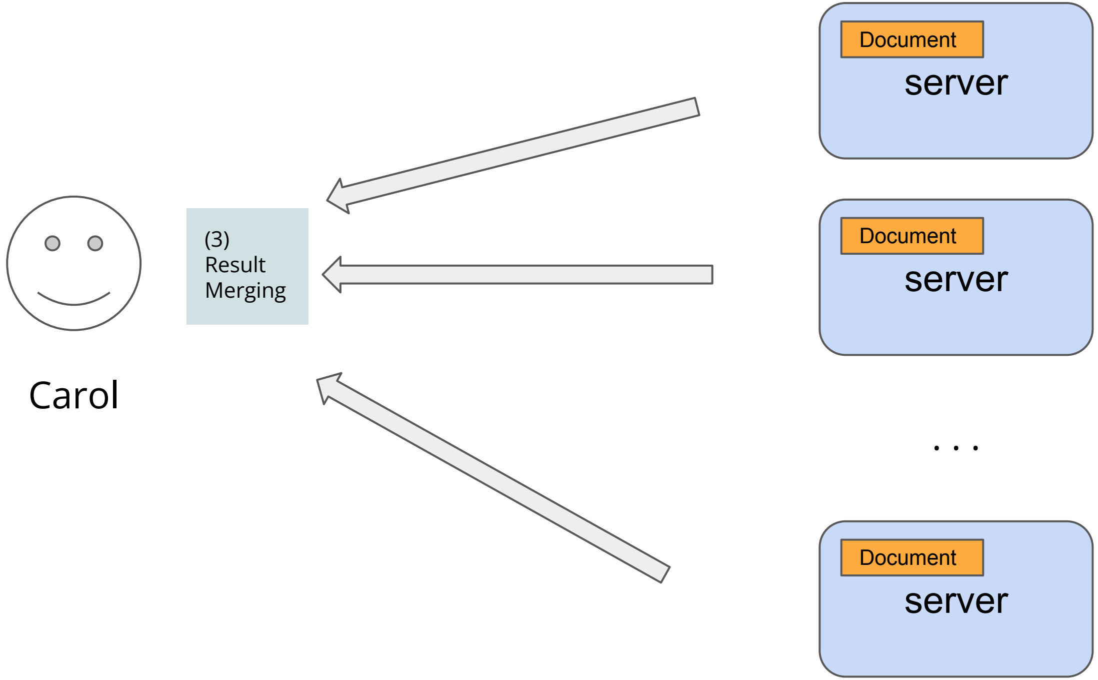
Use case – Word Frequency



Use case – Word Frequency



Use case – Word Frequency

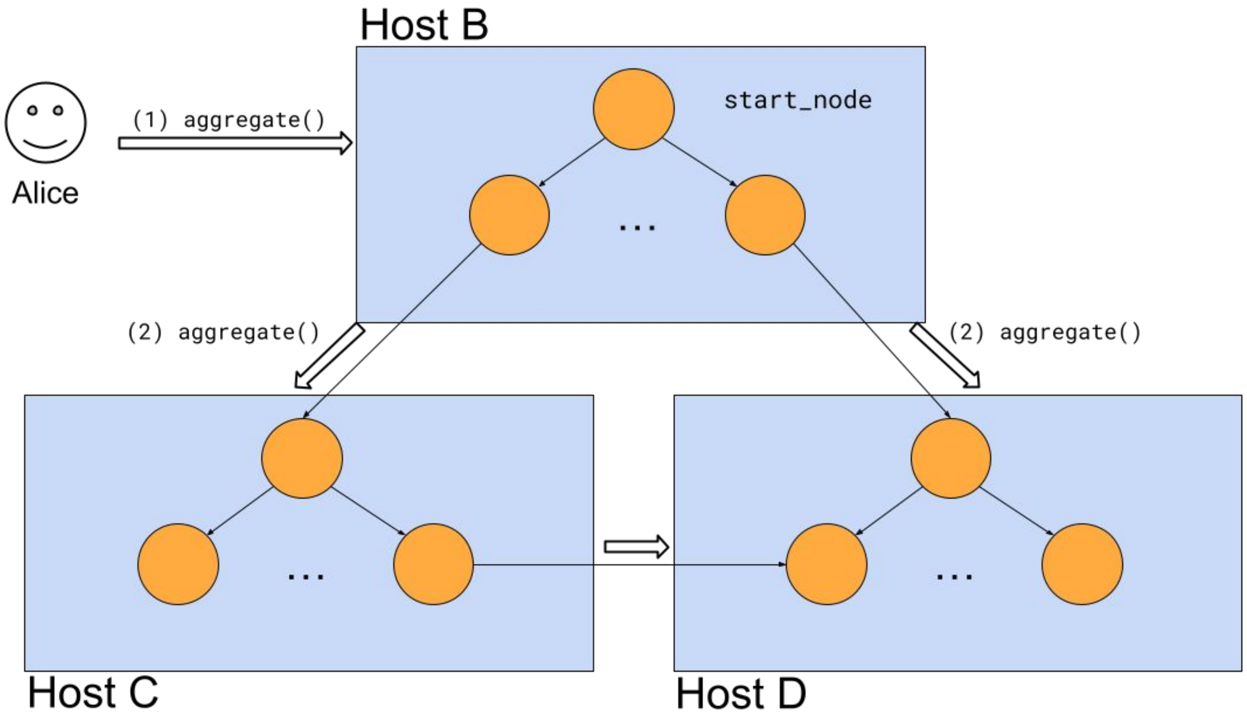


Use case – Distributed Graph Processing

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

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

```
let rec aggregate = nando(|
  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);
  }
});
```

Use case – Distributed Graph Processing



Local code, but actually distributed

```
let count = nando(|
  body: &Document,
  output: &FrequencyAggregator,
| {
  for line in body.lines {
    for word in line.split(‘ ‘) {
      output[word] += 1;
    }
  }
});
```

```
let rec aggregate = nando(|
  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);
  }
});
```

Use Case - Microservice Meshes

Teams now maintain models of their data, and a set of nanotransactions.

Any computation is free to happen anywhere in the cluster, since data is free to move to any machine.

Takeaways

There is an opportunity to reconsider how we distribute.

- Objects
  - Invariant references
  - Global Address Space
  - Mobility
- Nanotransactions
  - Shippable, local computation
  - Transactional semantics
- Objects + nanotransactions
  - The runtime can peek into the application’s semantics
  - Can effectively orchestrate execution

If we take a **data-first** approach...

... we can distribute computation in a **hands-free** way for users...

... while also enabling more flexible systems.

Just follow the data!

Achilles Benetopoulos (abenetop@ucsc.edu, @singingcircuits)
twizzler.io