Mergeable Types

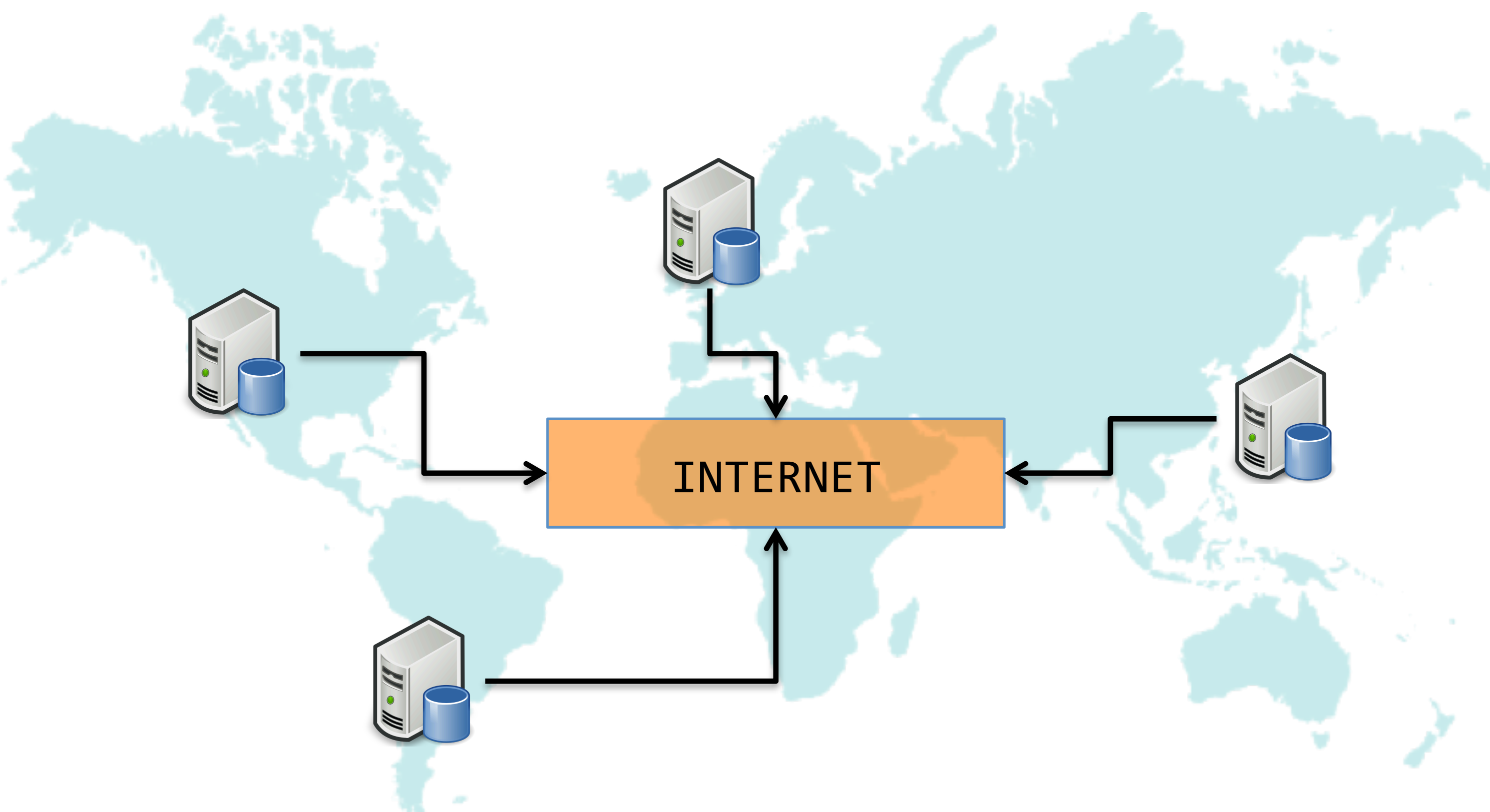
Gowtham Kaki, KC Sivaramakrishnan, Samodya Abeysiriwardane & Suresh Jagannathan

PURDUE UNIVERSITY PURPL

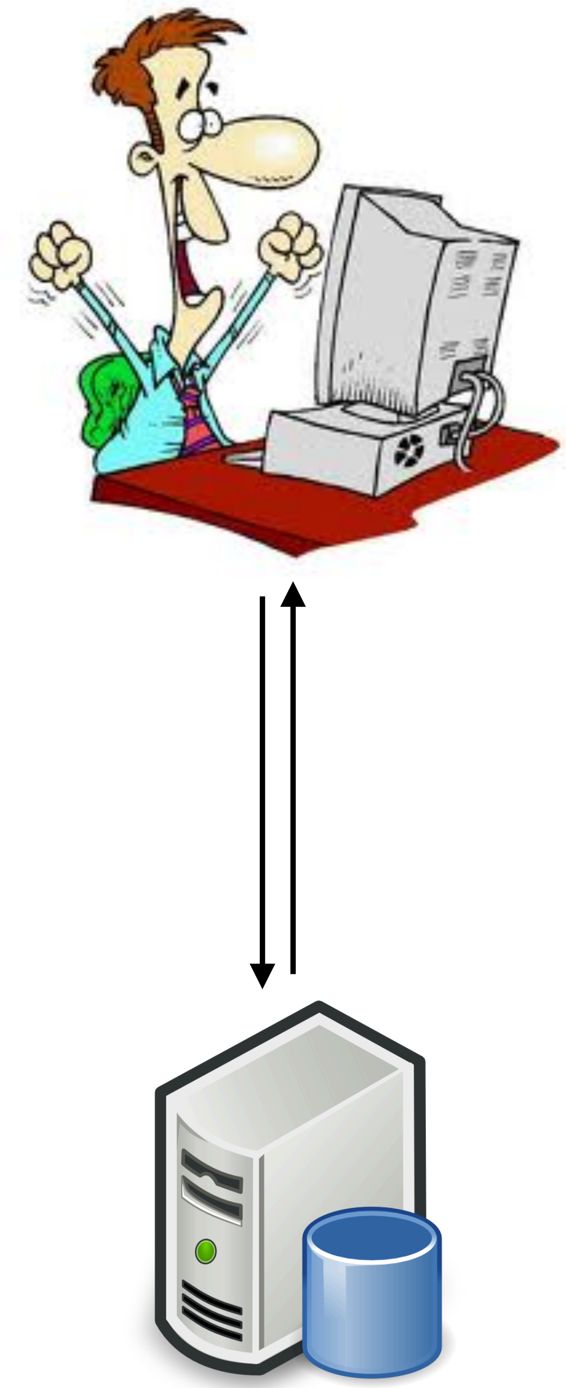
https://purduepl.github.io

UNIVERSITY OF CAMBRIDGE

http://ocamllabs.io



≠



- Serializability
- Linearizability

- Replication
- Eventual Consistency

When system-level concerns like replication, consistency etc., affect application-level design decisions, programming becomes complicated.

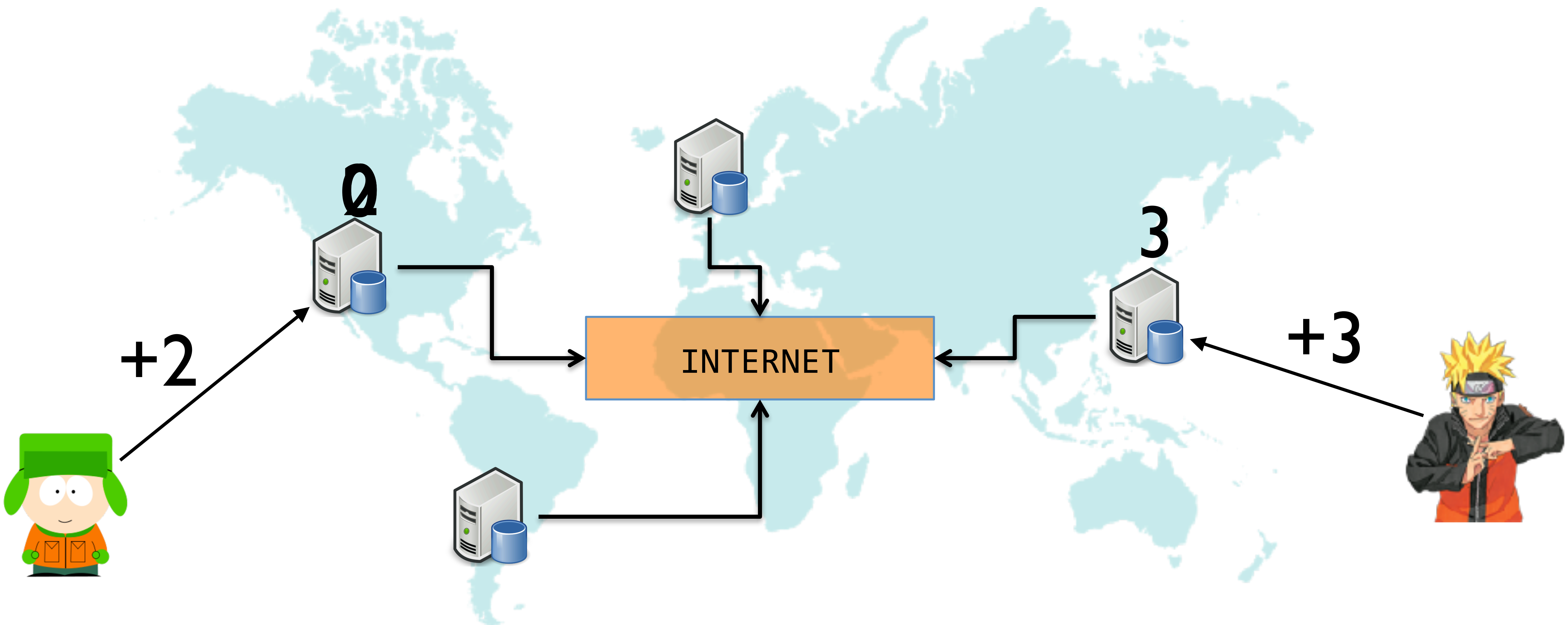
An example…

Monotonic Counter Datatype in OCaml

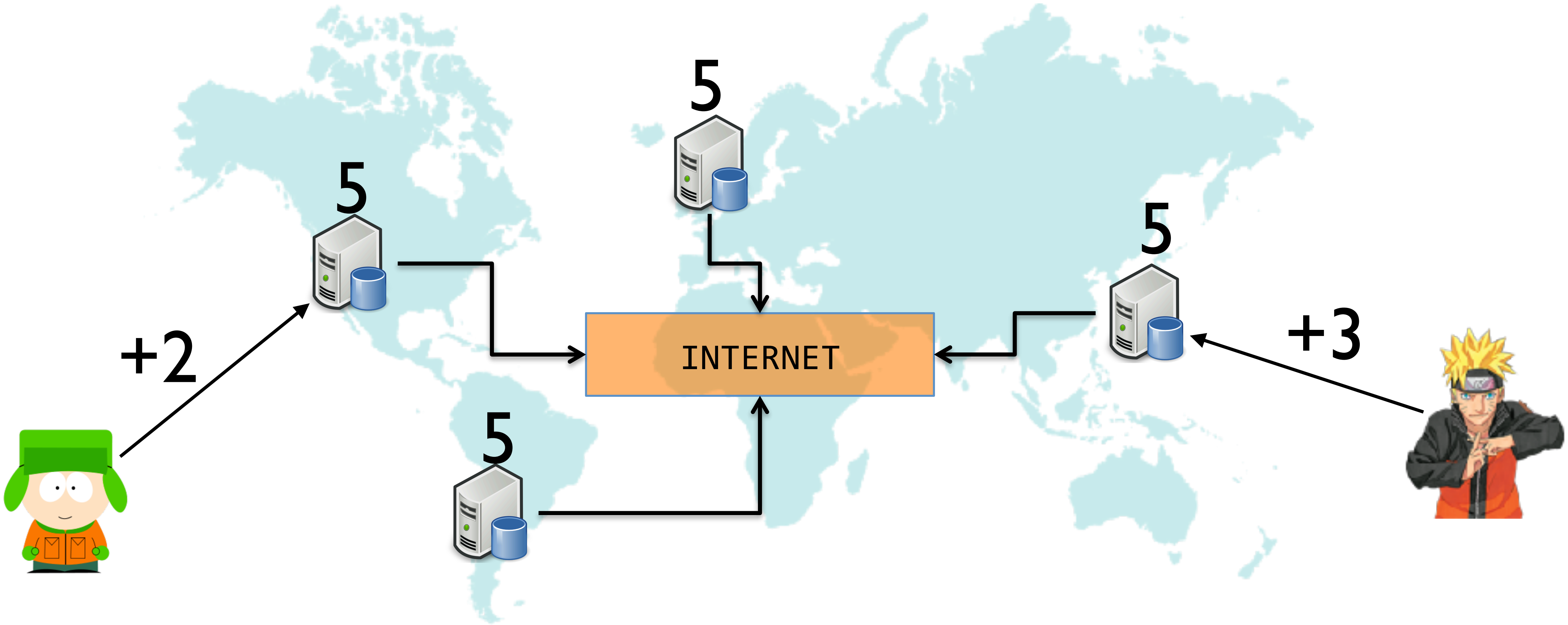
```
module Counter: sig
  type t
  val add: int → t → t
  val mult: int → t → t
  val read: t → int
end = struct
  type t = int
  let add x v = v + (abs x)
  let mult x v = v * (abs x)
  let read v = v
end
```

- Written in idiomatic functional style.
- No special reasoning principles required.
- Easily composable, e.g., `Counter.t List.t`

Replicated Counter



Replicated Counter



Adding replication to counter is non trivial!

Adding replication to counter is non trivial!

- Atomically applying updates to replicas requires expensive distributed transactions.
- Lazily applying updates leads to non-convergence since `add` and `mult` do not commute.

A common practice is to adopt the second approach, but re-engineer the datatype in terms of *commutative effects* to avoid non-convergence

Replicated Monotonic Counter Datatype in OCaml

```
module Counter: sig
  type t
  type eff
  val add: int -> t-> eff
  val mult: int -> t -> eff
  val apply: eff -> t -> t
  val read: t -> int
end = struct
  type t = int
  type eff = Add of int
  let add x v = Add (abs x)
  let mult x v = Add (v * (abs x - 1))
  let apply (Add x) v = x + v
  let read v = v
end
```

A new type for Counter effects

Counter operations return Counter effects

An apply function to apply Counter effects at local & remote replicas

Replicated Monotonic Counter Datatype in OCaml

```
module Counter: sig
  type t
  type eff
  val add: int -> t-> eff
  val mult: int -> t -> eff
  val apply: eff -> t -> t
  val read: t -> int
end = struct
  type t = int
  type eff = Add of int
  let add x v = Add (abs x)
  let mult x v = Add (v * (abs x - 1))
  let apply (Add x) v = x + v
  let read v = v
end
```

`Add` is the only Counter effect

The effect of `mult` operation is also expressed via `Add`

Since `Add` commutes with `Add`, counter states on all replicas eventually converge.

However …

Replicated Counter

```
module Counter: sig
  type t
  type eff
  val add: int -> t-> eff
  val mult: int -> t -> eff
  val apply: eff -> t -> t
  val read: t -> int
end = struct
  type t = int
  type eff = Add of int
  let add x v = Add (abs x)
  let mult x v = Add (v * (abs x - 1))
  let apply (Add x) v = x + v
  let read v = v
end
```

$\not\asymp$

Simple Counter

```
module Counter: sig
  type t
  val add: int → t → t
  val mult: int → t → t
  val read: t → int
end = struct
  type t = int
  let add x v = v + (abs x)
  let mult x v = v * (abs x)
  let read v = v
end
```

Replicated Counter

```
module Counter: sig
  type t
  type eff
  val add: int -> t-> eff
  val mult: int -> t -> eff
  val apply: eff -> t -> t
  val read: t -> int
end = struct
  type t = int
  type eff = Add of int
  let add x v = Add (abs x)
  let mult x v = Add (v * (abs x - 1))
  let apply (Add x) v = x + v
  let read v = v
end
```

$\not\asymp$

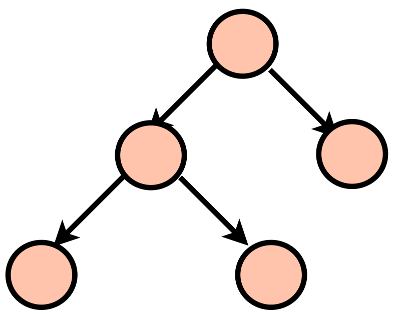
Simple Counter

```
module Counter: sig
  type t
  val add: int → t → t
  val mult: int → t → t
  val read: t → int
end = struct
  type t = int
  let add x v = v + (abs x)
  let mult x v = v * (abs x)
  let read v = v
end
```

- Requires significant re-engineering of libraries, taking into account operational characteristics of replication (i.e., effects).
- Programming model is restrictive: Common polymorphic data structures (e.g., lists) are often non-commutative.
- No polymorphic structures = no compositionality. Each new data structure has to be engineered from scratch.

we need…

A declarative programming model for replicated functional data types.



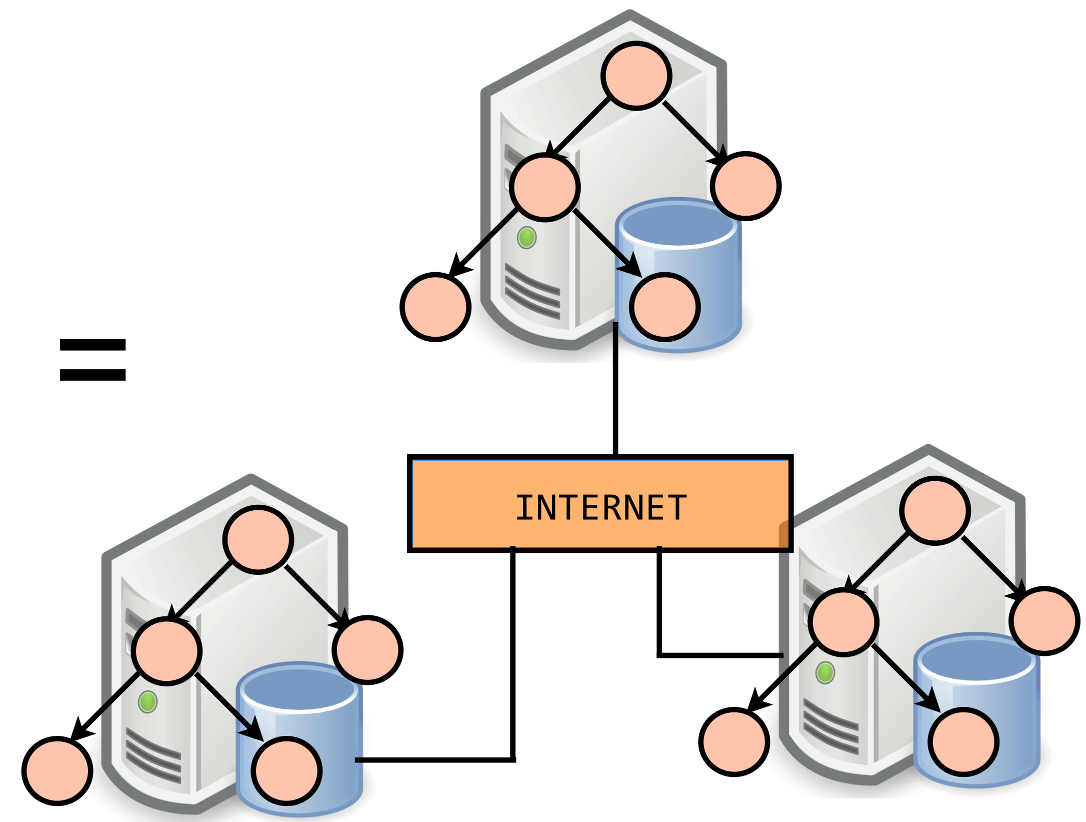
Functional data type

+

```
module Mk_replicated :
  functor(T:Serializable) ->
  sig
    ...
  end
```

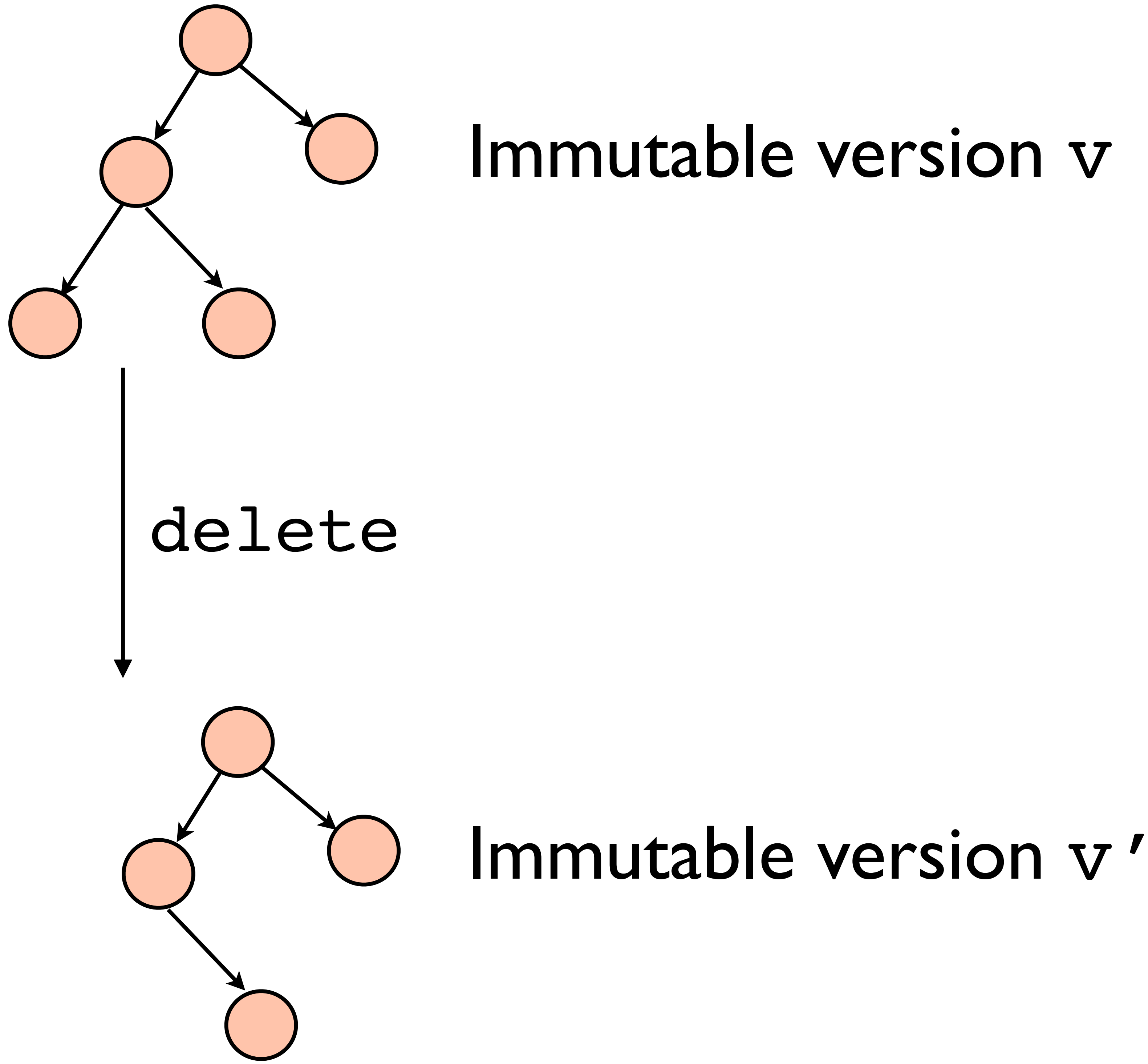
Declarative semantics of replication

=

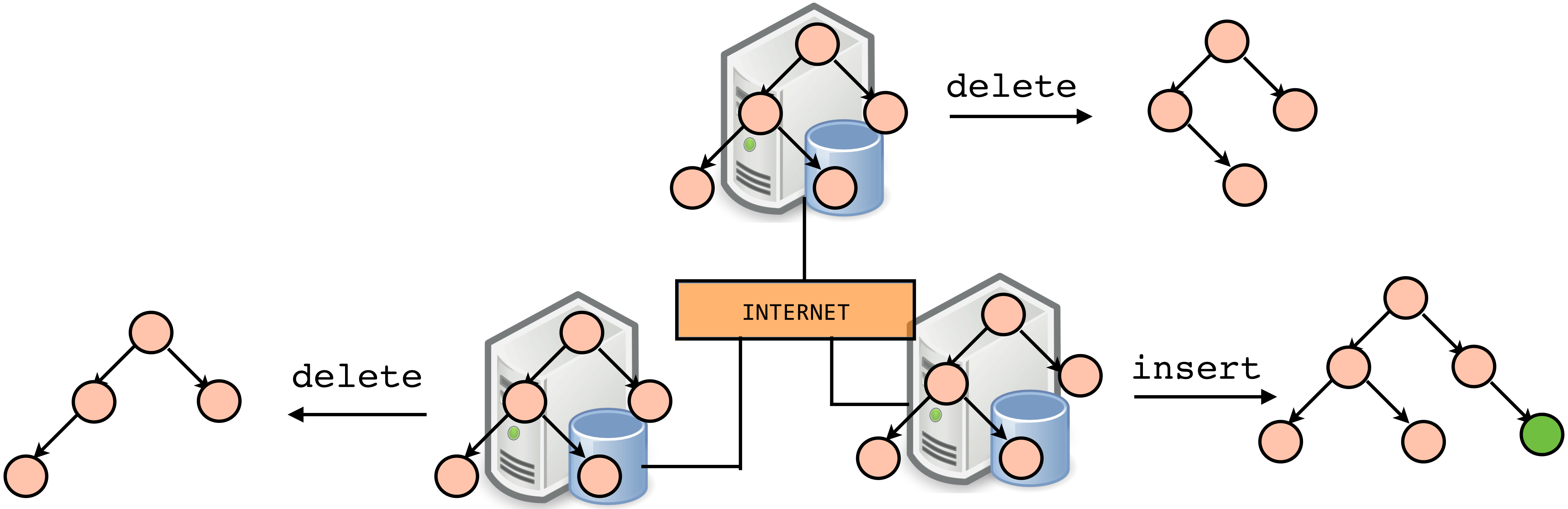


Replicated functional data type

Functional Data Types : Persistence & Immutability

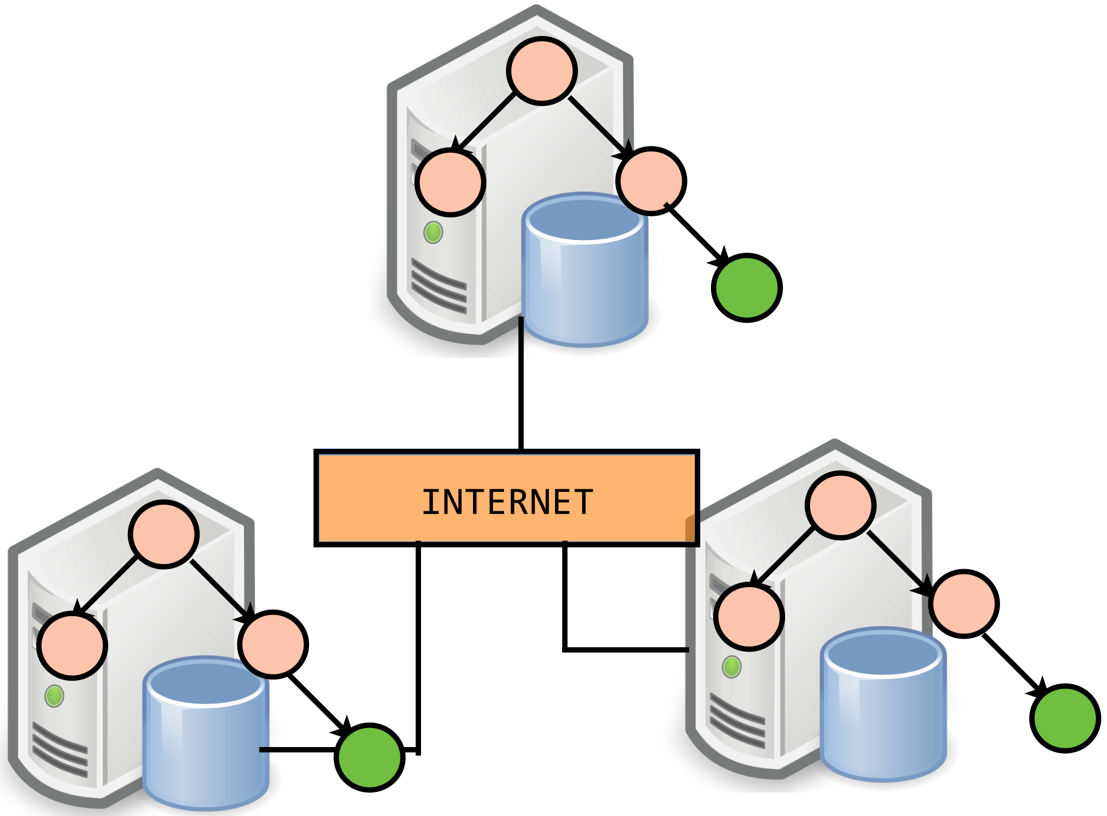


Functional Data Types



Multiple concurrent versions!

Functional Data Types → Functional Replicated Data Types

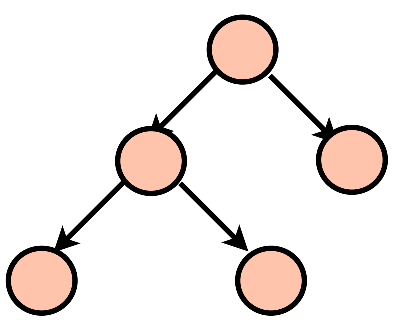


Multiple concurrent versions!

merge

Single replicated version

Functional Data Types → Functional Replicated Data Types



Functional data type

\+

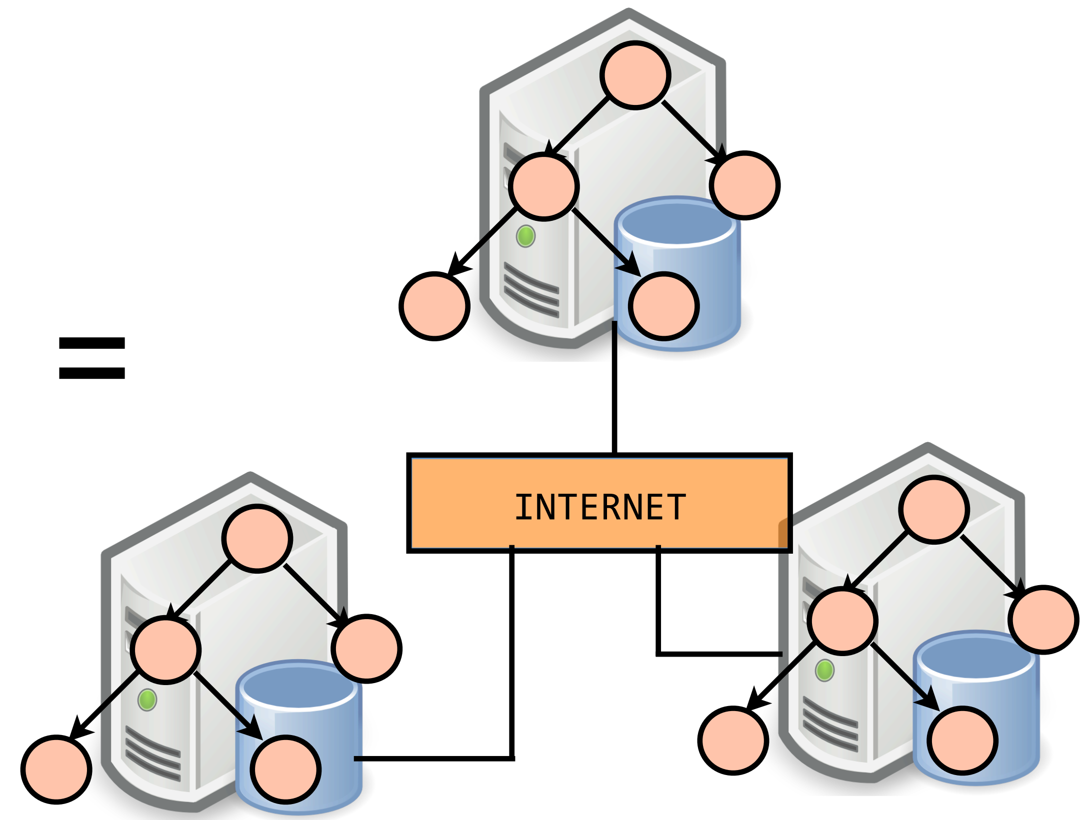
```
module Mk_replicated :
  functor(T:Serializable) ->
  sig
    ...
  end
```

Declarative semantics of replication

=

Semantics of merging concurrent versions!

=



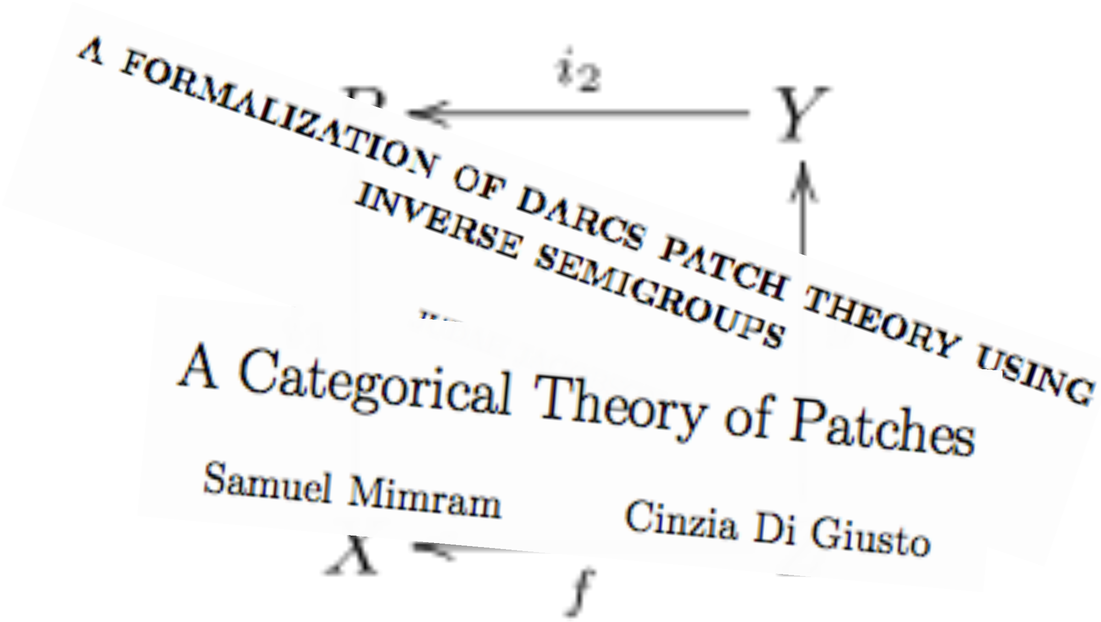
Replicated functional data type

Mergeable Type!

Abstractly…

Merging multiple versions of data

Extensively studied for for unstructured text data



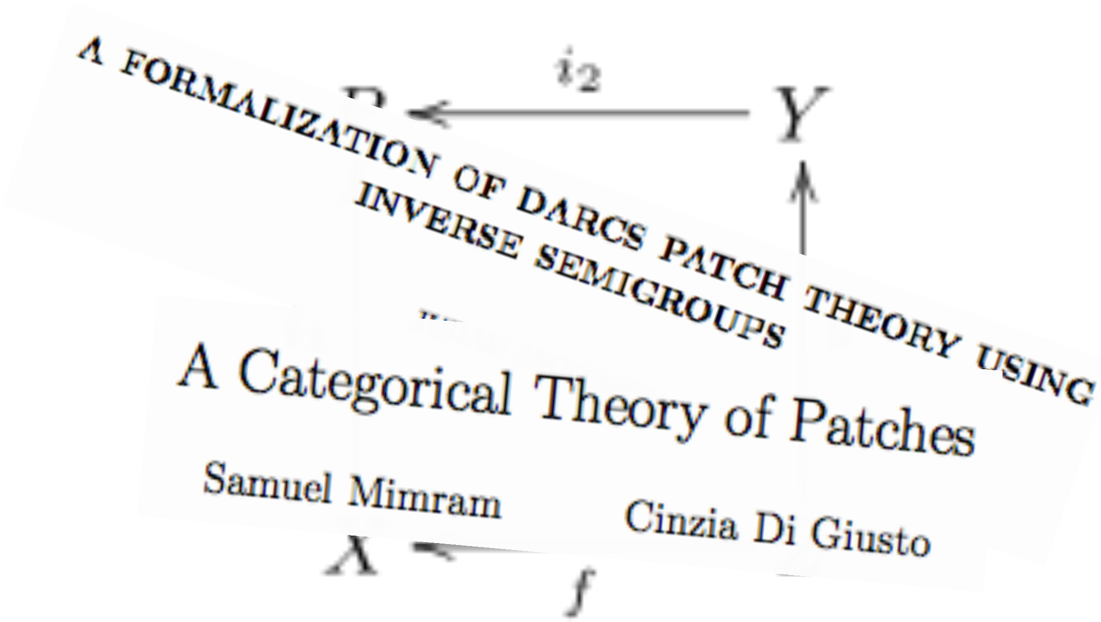
theory & practice

of version control systems

Abstractly…

Merging multiple versions of data

Extensively studied for for unstructured text data



darcs

GitHub

Mergeable Types

Extending
theory & practice
of version control systems
to functional data structures

Abstractly…

Mergeable Types

Extending
theory & practice
of version control systems
to functional data structures

Concretely…

VML: Versioned ML

An OCaml DSL that lets programers create version-controlled replicated data types by promoting ordinary data types to mergeable types.

contributions

VML: Versioned ML

1. A meta-programming framework to convert `Mergeable` types to `Versioned` types needed for replication.
2. A monad to compose concurrent/distributed computations around `Versioned` types.

VML: Versioned ML

1. A meta-programming framework to convert `Mergeable` types to `Versioned` types needed for replication.

```
module type MERGEABLE = sig
  type t
  val merge: t -> t -> t -> t
end
```

three-way merge function

lowest common ancestor (LCA)

concurrent versions

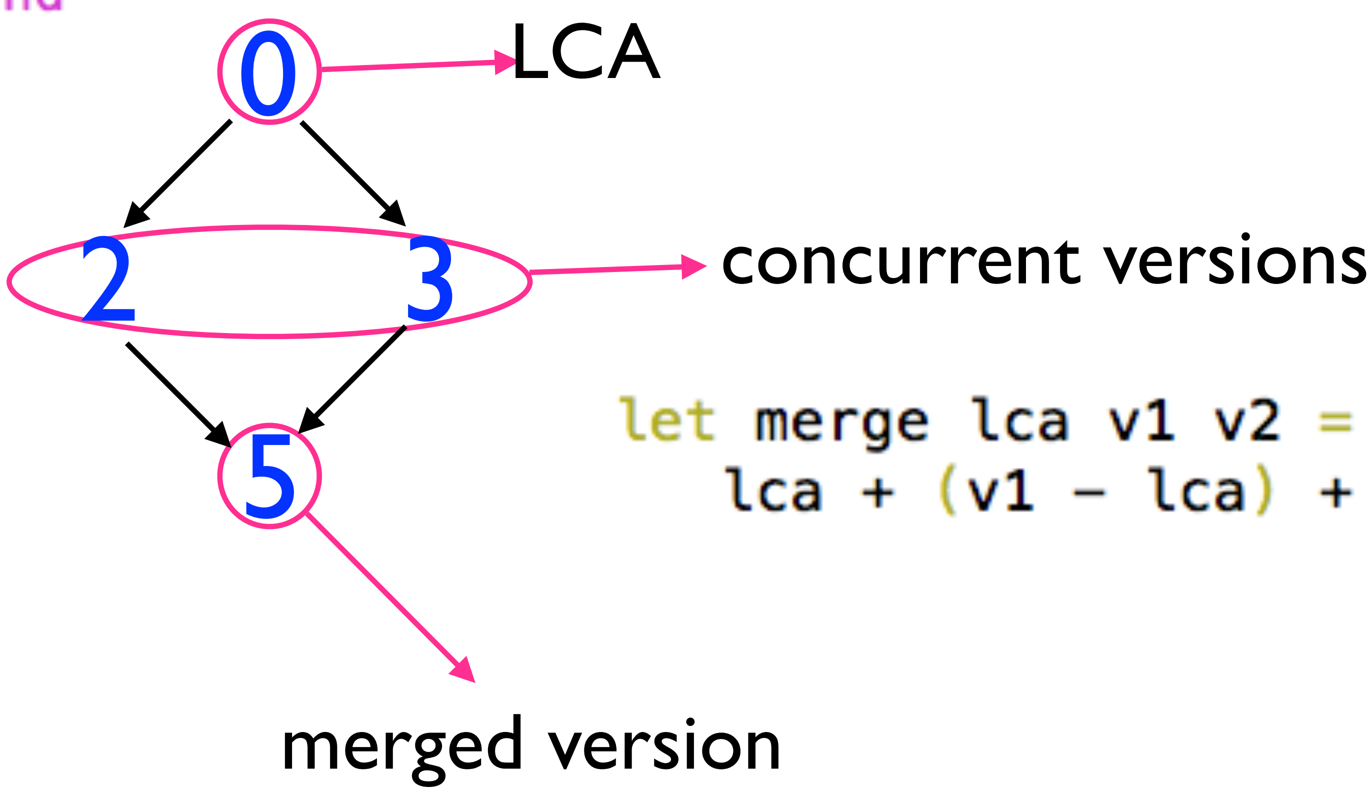
merged version

2. A monad to compose concurrent/distributed computations around `Versioned` types.

VML: Versioned ML

1. A meta-programming framework to convert `Mergeable` types to `Versioned` types needed for replication.

```
module type MERGEABLE = sig
  type t
  val merge: t -> t -> t -> t
end
```




```
let merge lca v1 v2 =
   lca + (v1 - lca) + (v2 - lca)
```

2. A monad to compose concurrent/distributed computations around `Versioned` types.

VML: Versioned ML

1. A meta-programming framework to convert `Mergeable` types to `Versioned` types needed for replication.

```
module type MERGEABLE = sig
  type t
  val merge: t -> t -> t -> t
end
```

```
module type VERSIONED = sig
  type t
  type version
  val of_t: t -> version
  val to_t: version -> t
  val merge: version -> version
                -> version -> version
end
```

three-way merge
for versioned representations.

2. A monad to compose concurrent/distributed computations around `Versioned` types.

VML: Versioned ML

1. A meta-programming framework to convert `Mergeable` types to `Versioned` types needed for replication.
2. A monad to compose concurrent/distributed computations around `Versioned` types.

Versioned State Monad

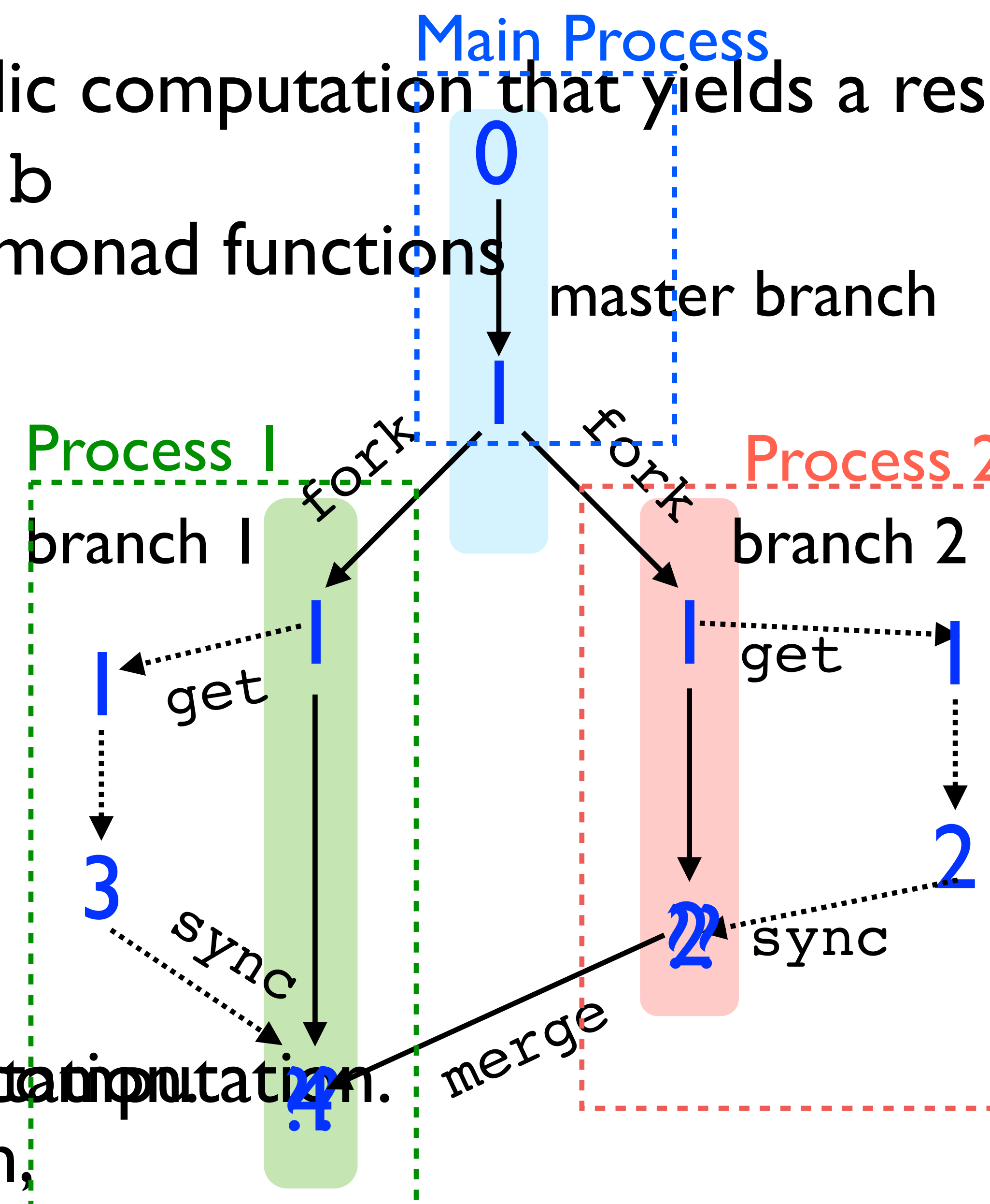
```
let merge lca v1 v2 =
    lca + (v1 - lca) + (v2 - lca)
```

```
module type VST = sig
  module V:VERSIONED
  type 'b t
  val return : 'b -> 'b t
  val bind : 'b t -> ('b -> 'c t) -> 'c t
  val with_init_version_do: V.t -> 'b t -> 'b
  val fork_version : 'b t -> unit t
  val get_current_version: unit -> V.t t
  val sync_next_version: unit -> V.t -> V.t t
end
```

merge 1 3 2 = 1 + (3-1) + (2-1) = 4

A monadic computation that yields a result of type 'b

Standard monad functions

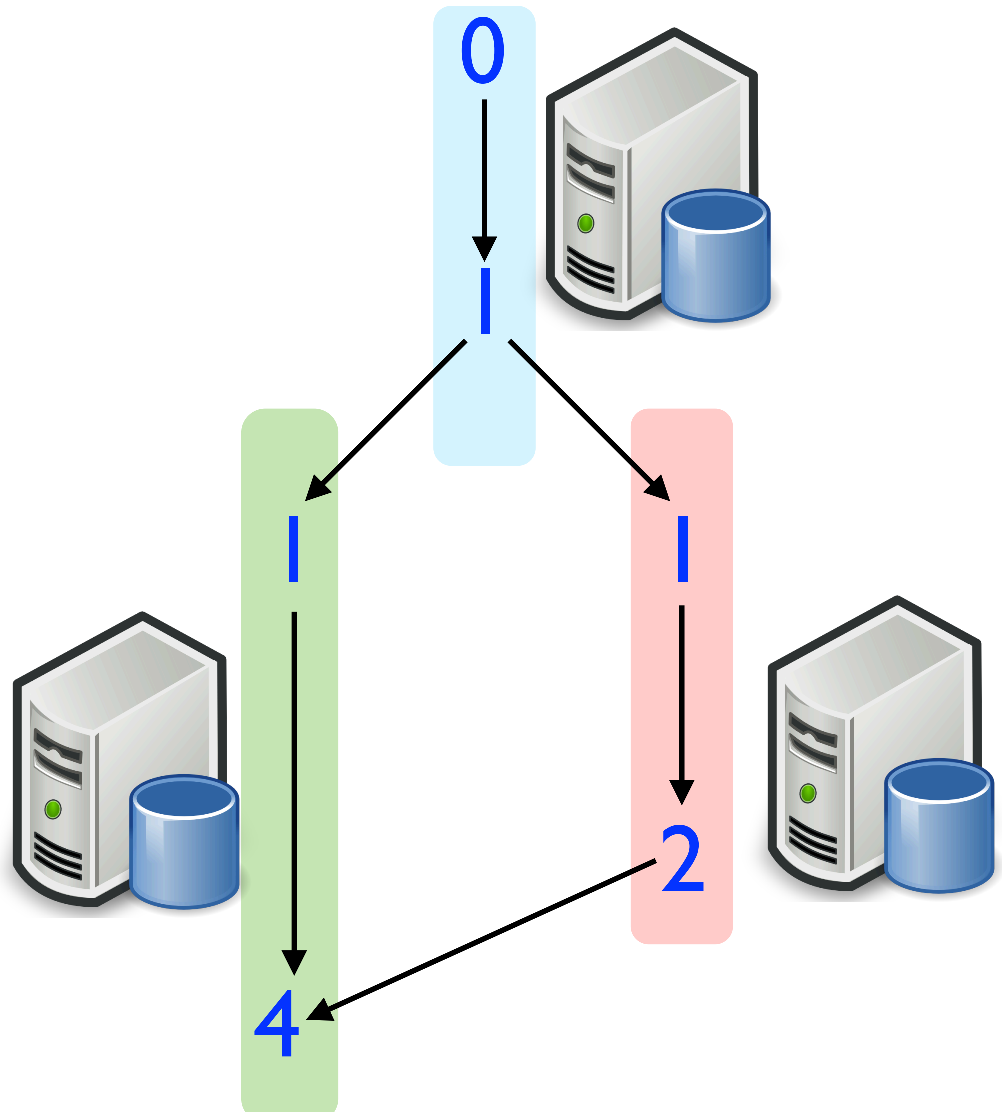


1 is their lowest common ancestor (LCA).

Get the latest version on the current branch (read it into process's local heap)

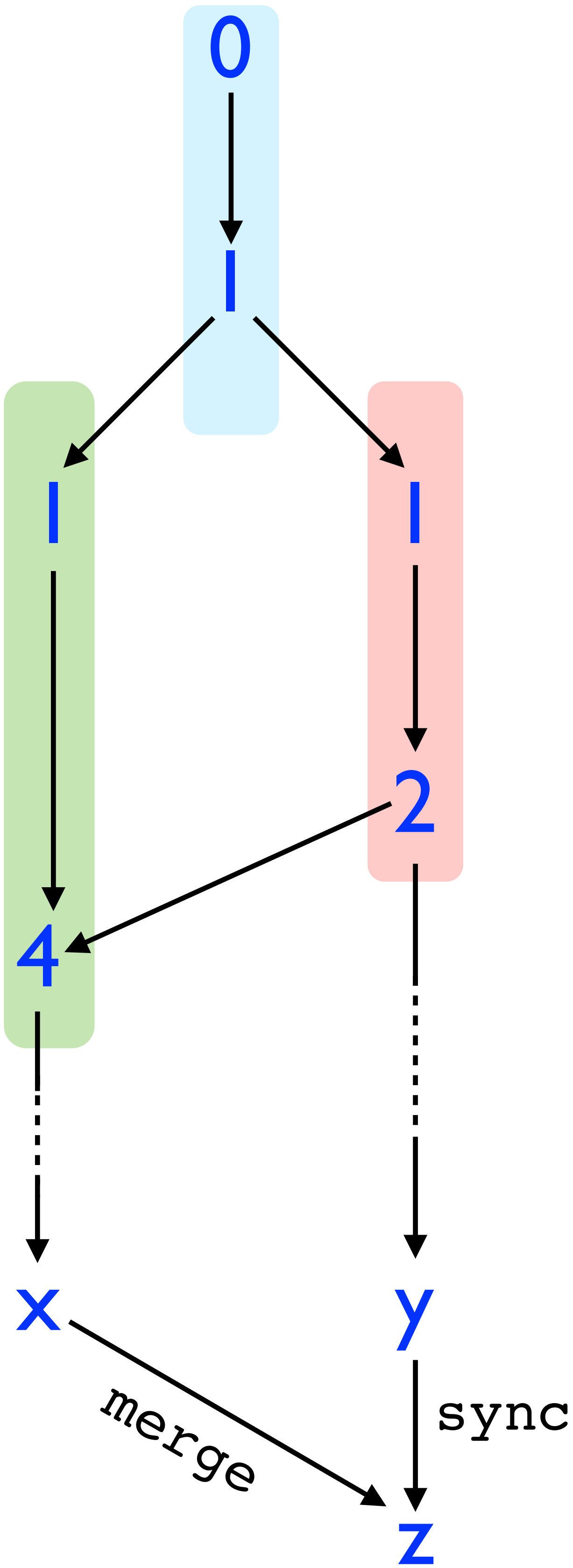
Sync this local version with latest versions on concurrent branches. Install the next version on the current branch.

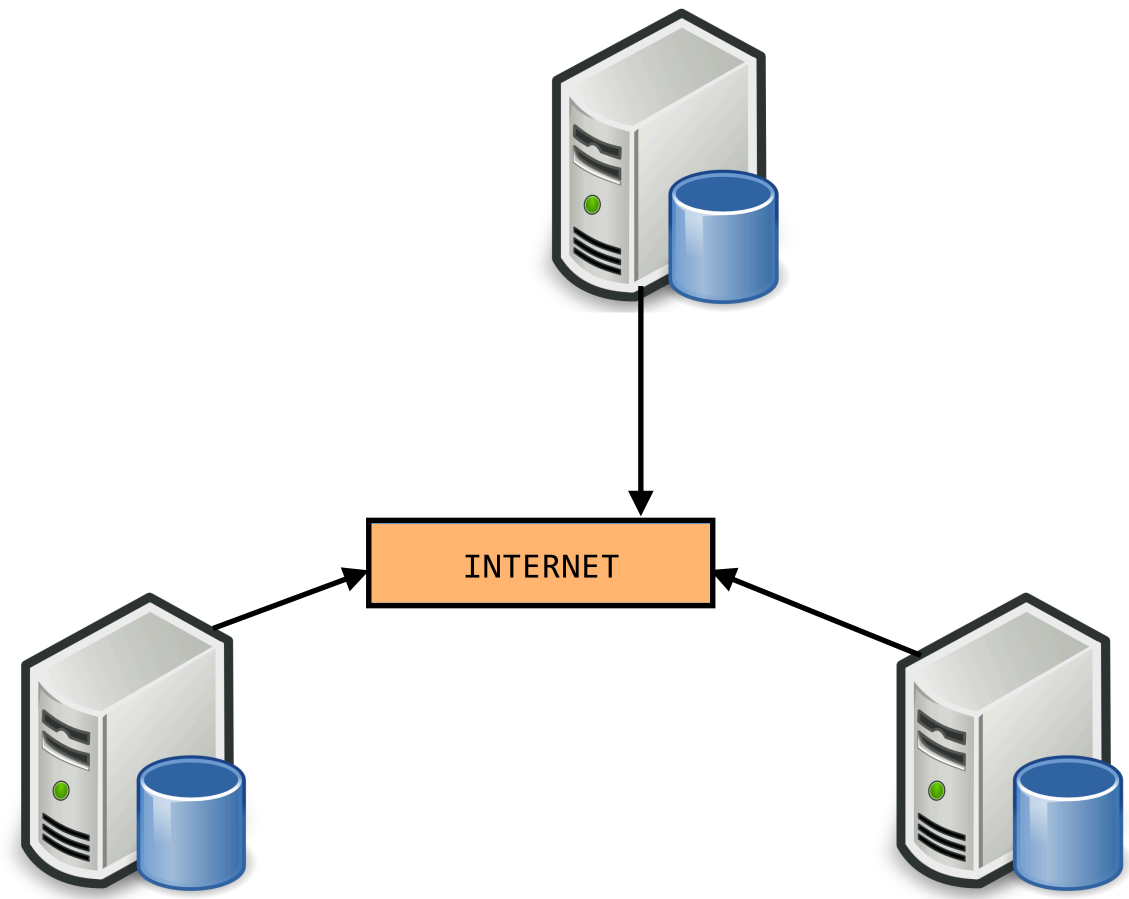
VML: Versioned ML



Branching Structure

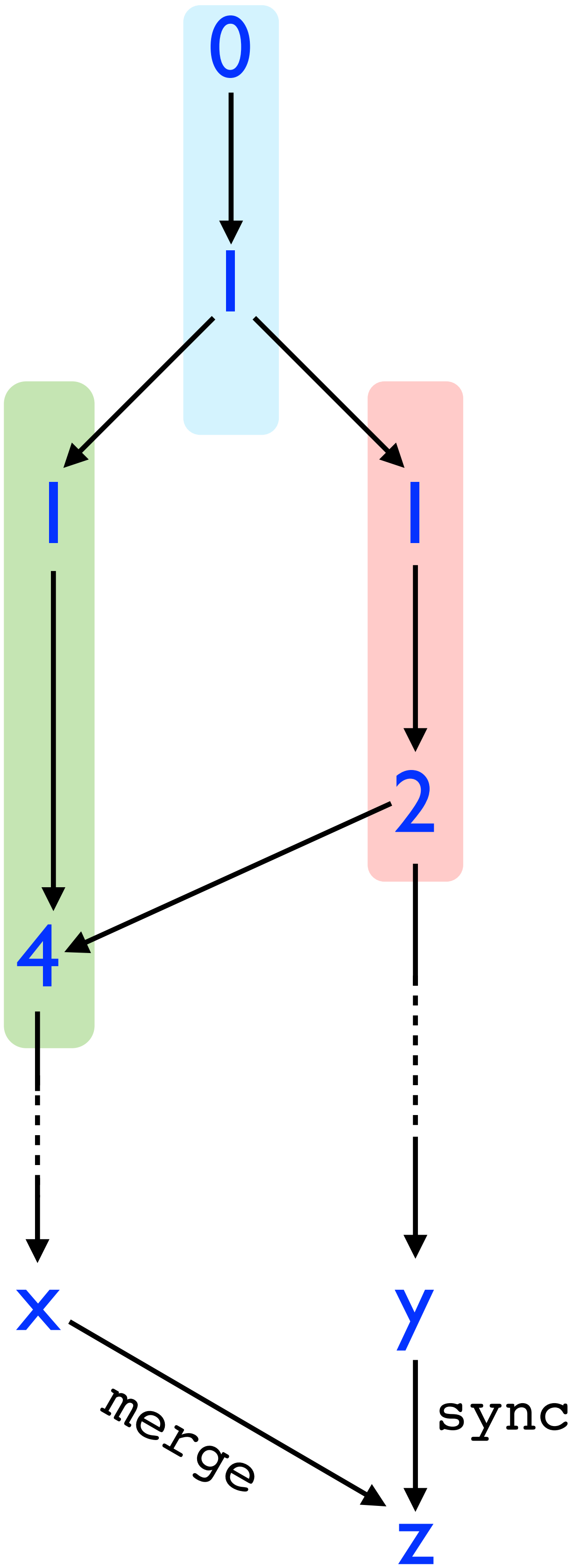
VML: Versioned ML





Mergeable Types guarantee that branches are always mergeable!

VML: Versioned ML



Mergeable Types guarantee that branches are always mergeable!*

*Conditions apply

Branches are mergeable only if a three-way merge function can be used to merge them.

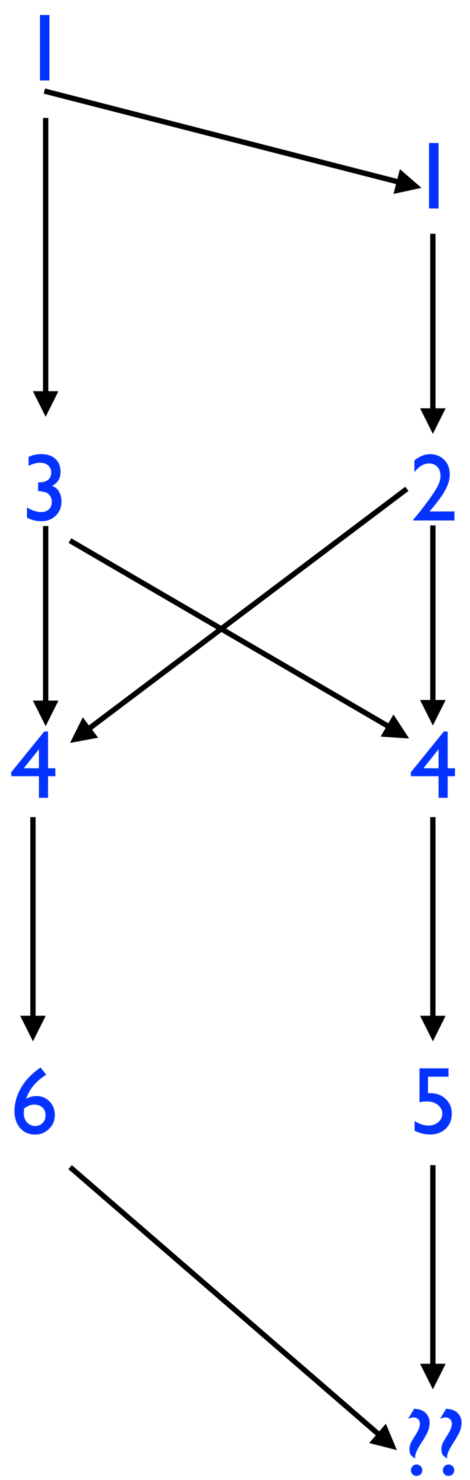
```
module type MERGEABLE = sig
  type t
  val merge: t -> t -> t -> t
end
```

LCA

What if there are two LCAs?

Possible!

VML: Versioned ML



6 & 5 have two LCAs: 3 & 2!

Mergeable Types guarantee that branches are always mergeable!*

*Conditions apply

Branches are mergeable only if a three-way merge function can be used to merge them.

```
module type MERGEABLE = sig
  type t
  val merge: t -> t -> t -> t
end
```

LCA

What if there are two LCAs?

Possible!

e.g., criss-cross merges

VML: Versioned ML

1. A meta-programming framework to convert `Mergeable` types to `Versioned` types needed for replication.
2. A monad to compose concurrent/distributed computations around `Versioned` types.
3. A runtime that ensures progress and convergence in the presence of network partitions.

formalized

THEOREM 4.10 (**Progress**). *In a legal branching history H produced by the operational semantics, if two branches, b_i and b_j are not mergeable (as per Def. 4.8), then there exists a sequence of fork and merge operations (between mergeable versions) that can be performed on H to yield a* [illegible] *b_j are mergeable.*

4. A standard library of mergeable types, including polymorphic containers, such as lists, ropes and trees.

compositionality

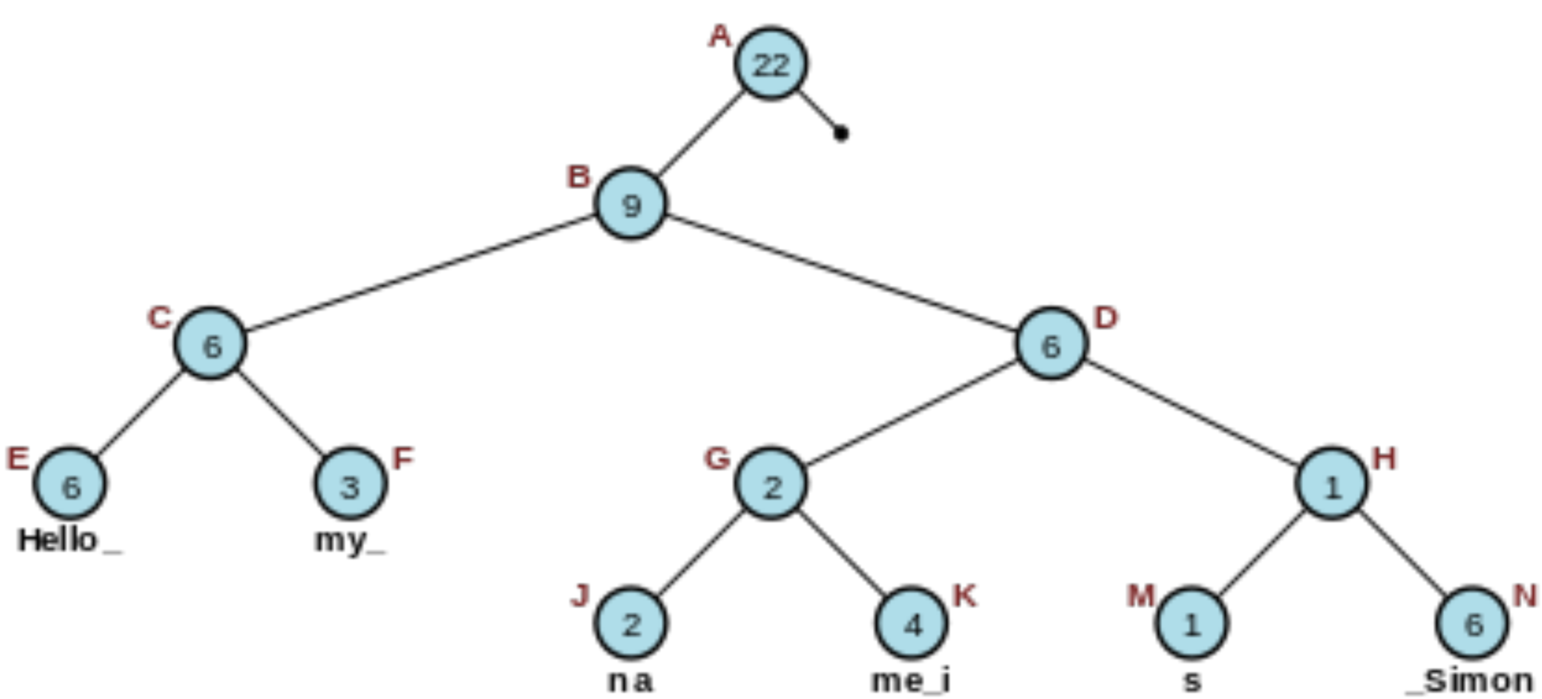
VML: Versioned ML

4. A standard library of mergeable types, including polymorphic containers, such as lists, ropes and trees.

Usually list of chars = strings

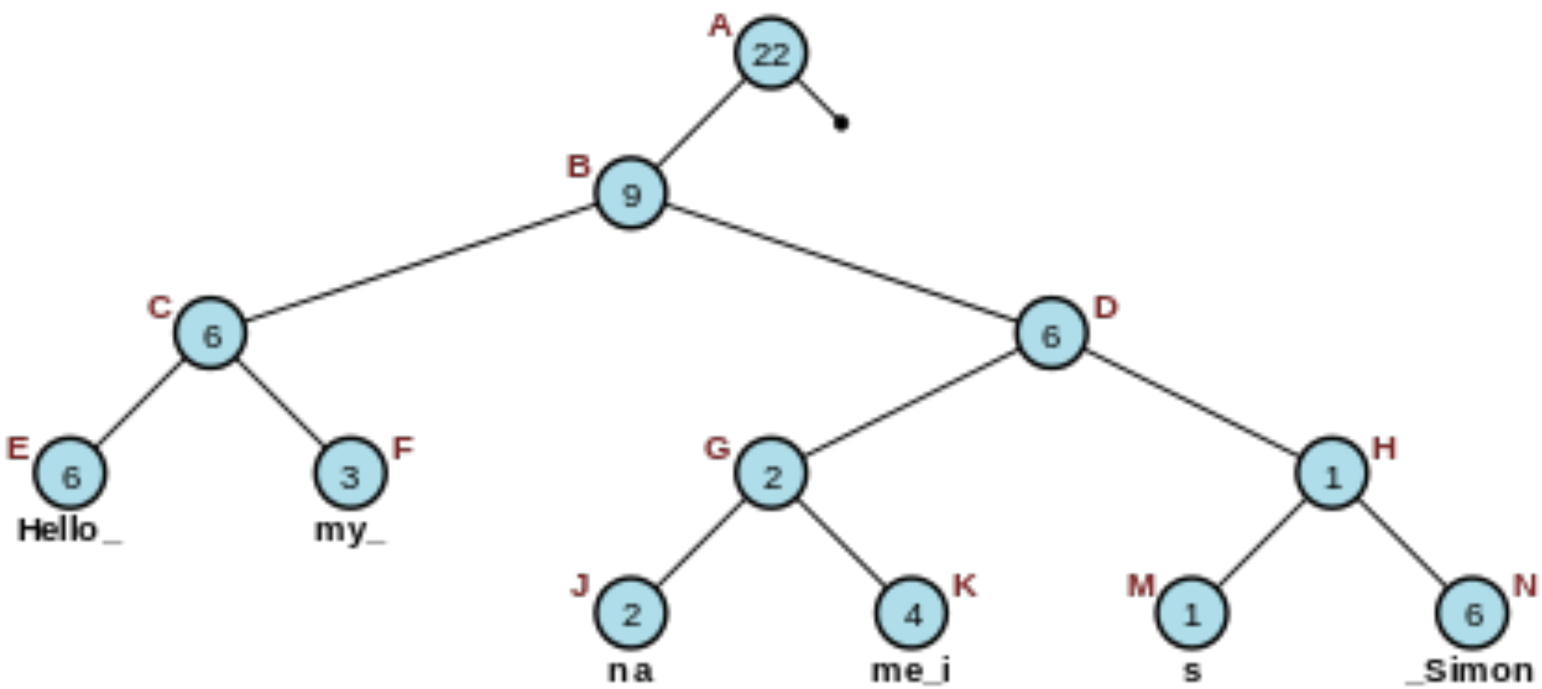
Data structure to store a list of values.
Efficient `concat`, `split`, `lookup`, and `insert` at an index.

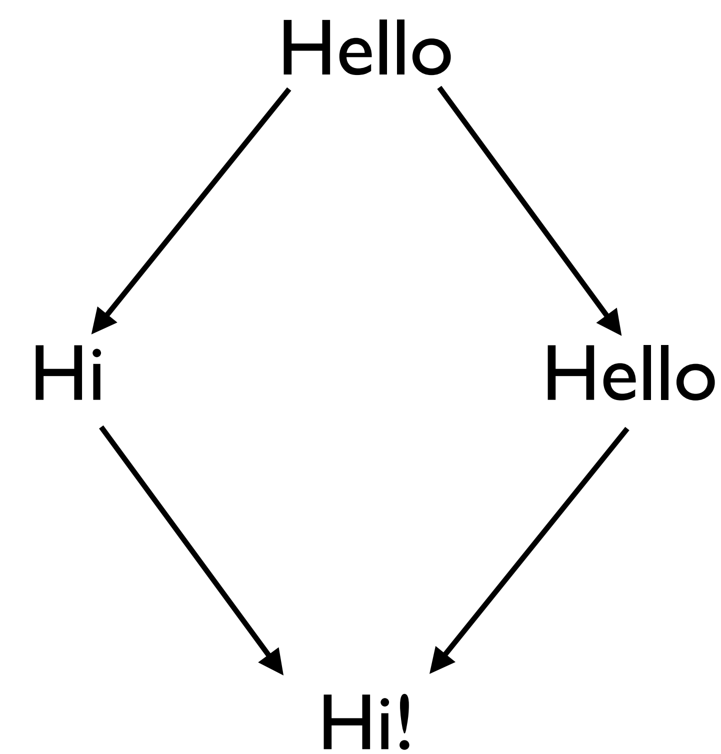
Essentially a binary tree that stores substring indexes in internal nodes, and actual strings at leaf nodes.



VML: Versioned ML

4. A standard library of mergeable types, including polymorphic containers, such as lists, ropes and trees.





Mergeable rope = rope + `merge` function

Merge boils down to merging strings

Wagner-Fischer + Operational Transformation

Wagner–Fischer algorithm

From Wikipedia, the free encyclopedia

In computer science, the **Wagner–Fischer algorithm** is a dynamic programming algorithm that computes the edit distance between two strings of characters.

Implementation

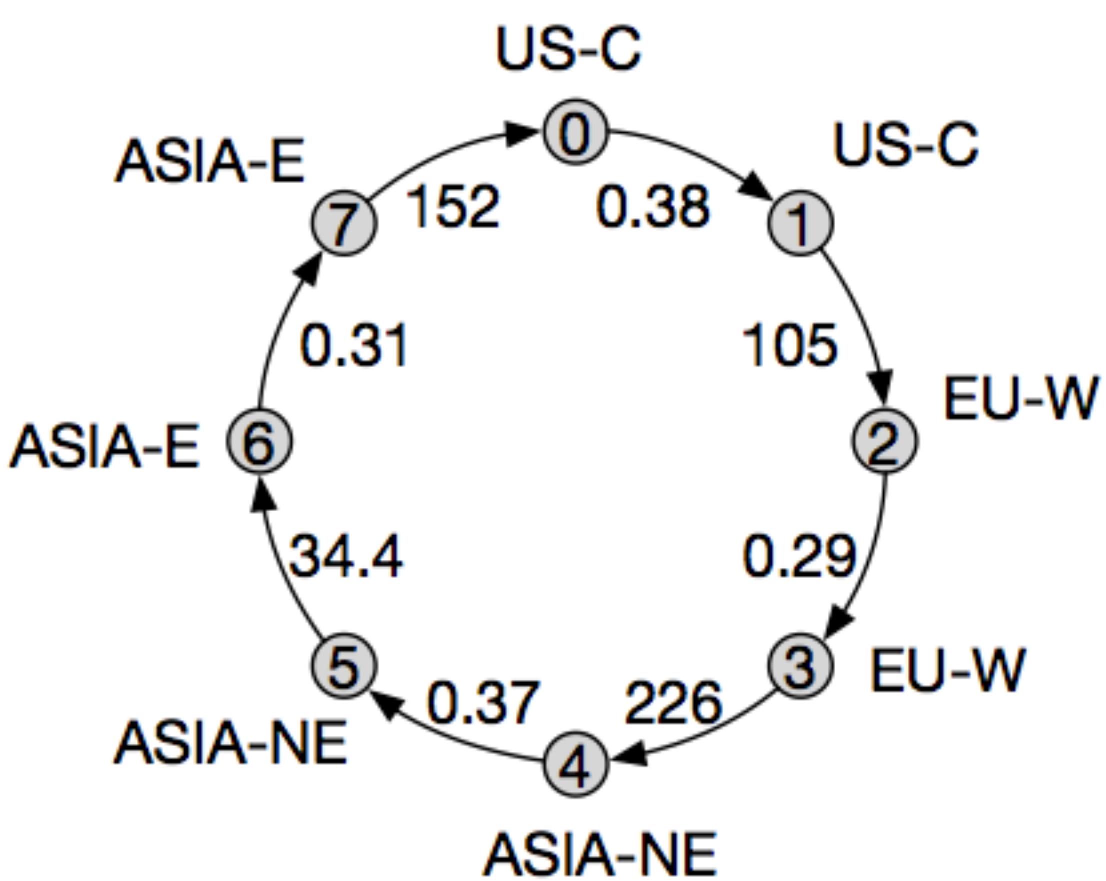
VML is realized on top of Irmin, a persistent multi-versioned store.

- i.e., a git-like store
- offers content-addressable heap abstraction.

  address = f(content)

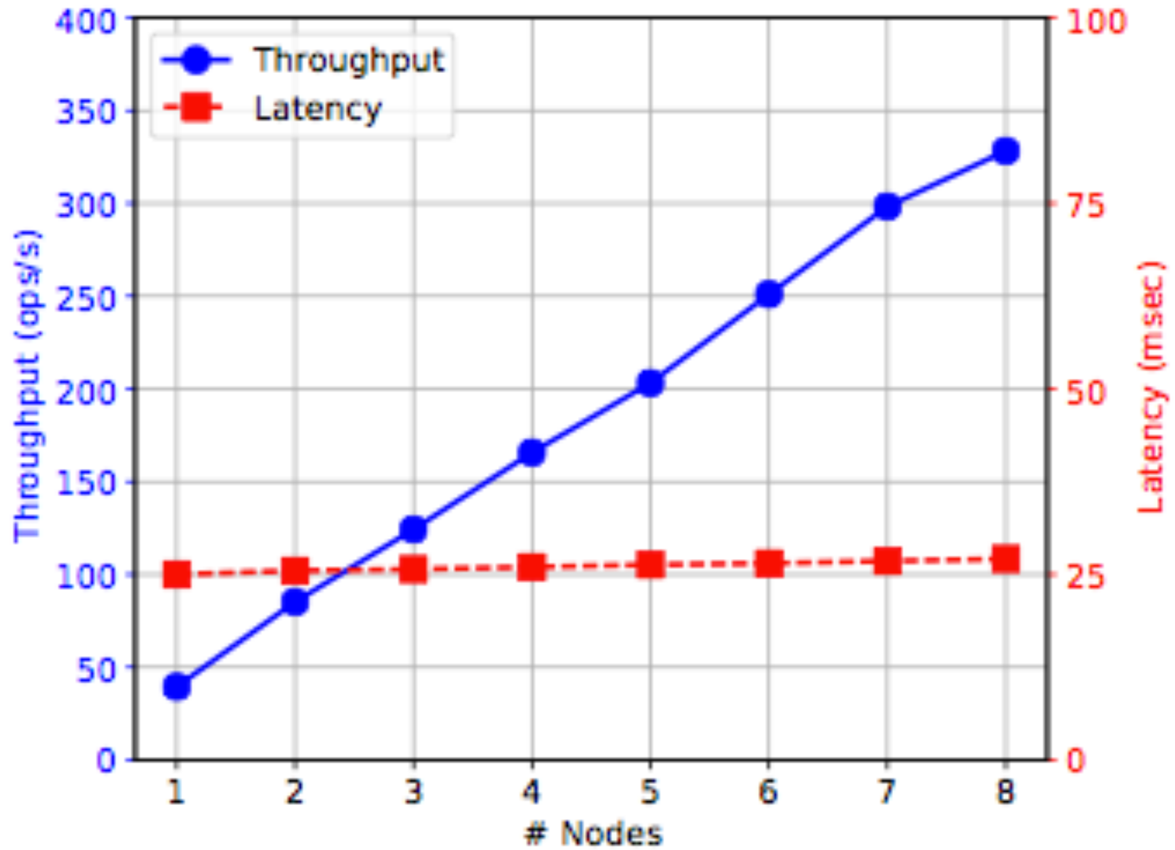
Evaluation

- Research Questions:
  1. Is VML practical enough to build geo-distributed eventually consistent applications?
  2. How close are we to horizontal scalability?
- Application: collaborative text editing application implemented with mergeable ropes.
- Benchmark workload: a 1576-word document. 85% insertions,



- Setup: 8-node geo-distributed Google Compute cluster arranged in ring.
- Each node performs edits locally, and periodically (1s) synchronizes with its successor.
- We measured latency-throughput while increasing nodes from 1 to 8.

Results



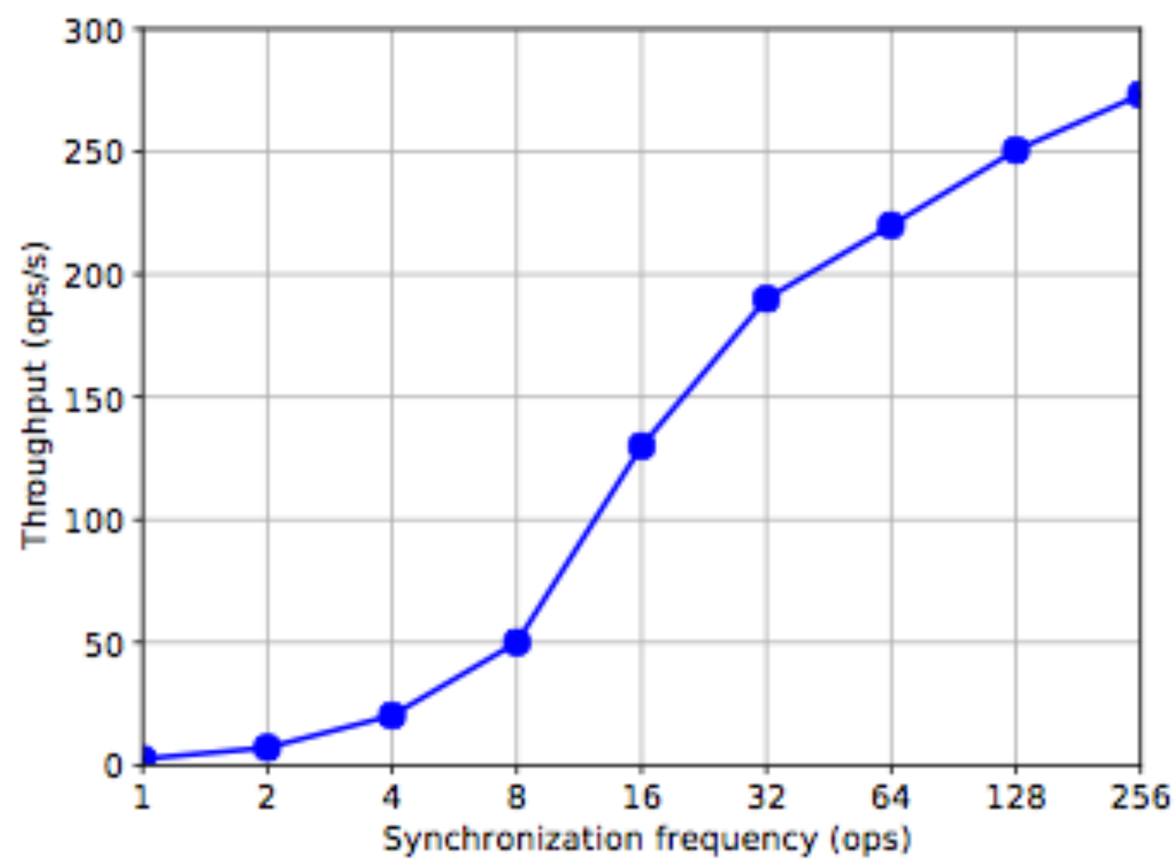
(b) Scalability: Overall throughput of the cluster and latency of each operation.

Mergeability =>

Asynchrony =>

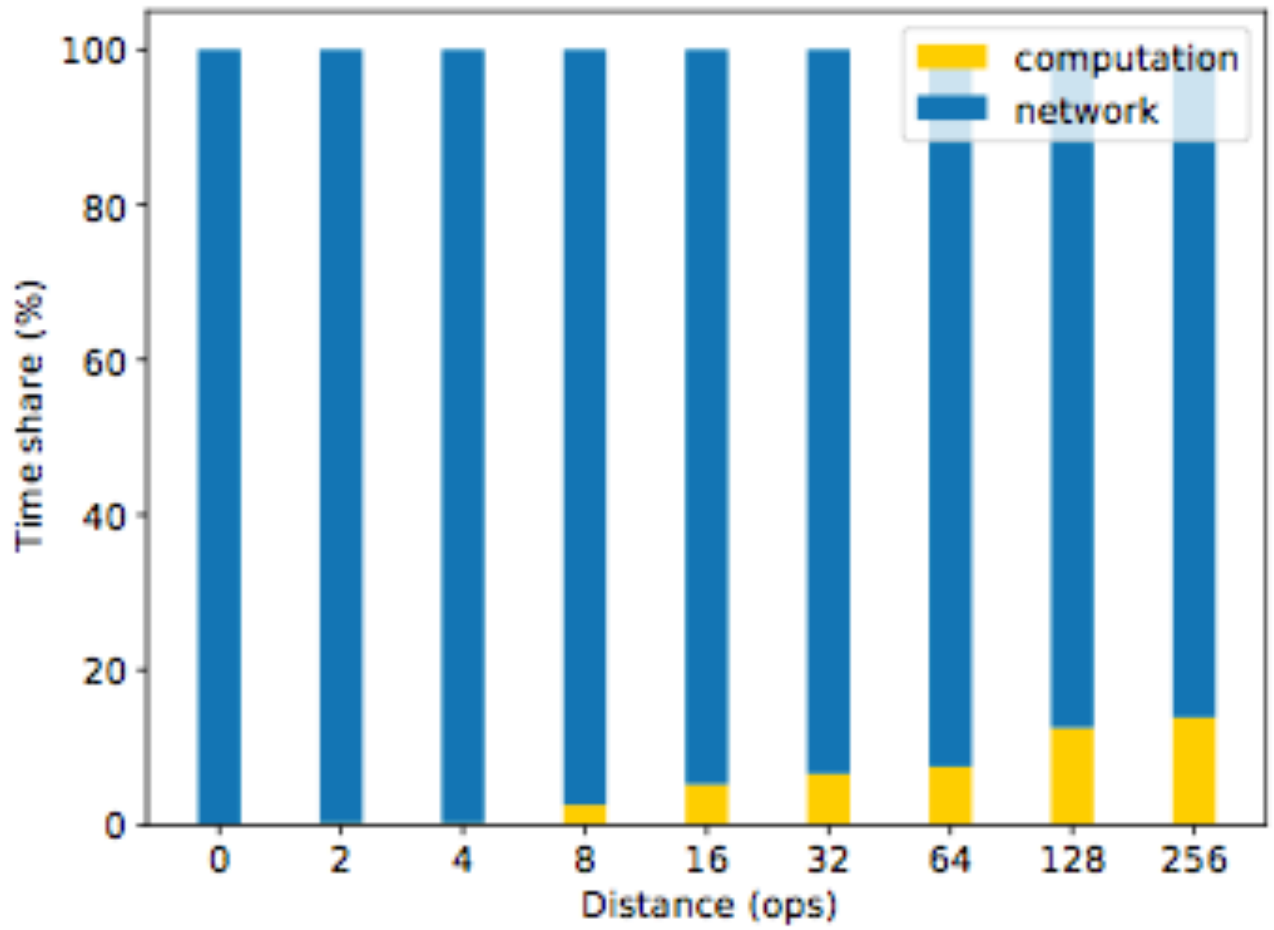
Near-horizontal scalability …

… which is not possible if we have to synchronize often.



(c) Synchronization: Overall throughput of the cluster while actively synchronizing with the successor node.

Results



(d) Merge performance: Cost of merging concurrent operations across nodes.

The overhead of merging is quite less when compared to the network latency.

Scope for improvement: Irmin, being a persistent store, flushes every write to the disk. Removing disk latency off the critical path should improve VML performance.

Thanks!

Short paper: http://www.mlworkshop.org/icfp-mlworkshop17-final4.pdf

Full paper: http://gowthamk.github.io/docs/vml.pdf

Code: https://github.com/icfp2017/vml

Replicated Counter

```
module Counter: sig
  type t
  type eff
  val add: int -> t-> eff
  val mult: int -> t -> eff
  val apply: eff -> t -> t
  val read: t -> int
end = struct
  type t = int
  type eff = Add of int
  let add x v = Add (abs x)
  let mult x v = Add (v * (abs x - 1))
  let apply (Add x) v = x + v
  let read v = v
end
```

Encodes each operation in terms of its *effect* on the global state.

INTERNET

Is more

ack of polymorphic types no compoisitionality. Each new data structure has to be engineered from scratch.

- Lack of polymorphic types no compositionality. Each new data structure has to be engineered from scratch.