# Why time is evil and what to do about it: designing distributed systems as pure functional programs plus interaction points

Peter Van Roy and Seyed Hossein Haeri
Université catholique de Louvain

PLDS ‘20 Workshop
March 4, 2020

LIGHTKONE
Lightweight computation for networks at the edge



UCL Université catholique de Louvain

# Overview

- Purely functional distributed systems
  - What is functional programming
  - Lambda calculus and its properties
  - **Functional programming can be concurrent**
    - Examples: Kahn networks, pipelines
  - **Distributed systems can be as pure as functional systems**
    - Example: distributed pipeline
- General distributed systems are not pure
  - Purely functional distributed systems are the starting point
  - Add interaction points (“ports”)
  - Add observational purity (“pure blocks”)
  - Preliminary theoretical results
- Ongoing work
  - Designing distributed systems with Piecewise Relative Observable Purity
  - Proofs and applications in progress


LIGHTKONE

Lightweight computation for networks at the edge

# Inspiration

- This work is inspired by the visions of the SyncFree and LightKone EU projects (2013-16, 2017-19)
  - SyncFree: synchronization-free computing
  - LightKone: lightweight computation for networks at the edge (lightkone.eu)
- Can we program distributed systems that achieve consistency using weak synchronization models?
  - ✗ RSMs are distributed data structures using consensus
  - ✓ CRDTs are distributed data structures without consensus
- This work grew out of an attempt to understand what "synchronization-free" means in a fundamental way

# Purely functional distributed systems

# Functional programming

- **Confluent reduction of an initial expression to a final result**

This has very strong mathematical properties that we can use

- For reasoning, debugging, testing, optimization, and maintenance
- For concurrency, parallelism, and distribution
- And there is no efficiency penalty compared to other paradigms

But it can't interact with the real world! Let's see why:

- During the execution, we would like to accept inputs coming from the real world and outputs going back to it
- Functional programming can't do this because the execution of a functional program is a step-by-step reduction of an initial expression to a final result. Reduction steps take time, and the inputs will arrive during this time. The reduction can't use them unless we could put them in the initial expression. But we can't do this, because **the inputs are not known in advance**.

# Lambda (λ) calculus

- Lambda calculus is the core of functional programming
  - We define it and use it for concurrency and distribution
- Syntax
  - x ::= (variables)
  - t ::= x | (λx. t) | ($t_1$ $t_2$)
- Semantics (using substitution operation t[x])
  - (λx. t[x]) → (λy. t[y]) α-conversion
  - ((λx. $t_1$) $t_2$) → $t_1$[x:=$t_2$] β-reduction
  - ((λx. (t x)) → t (if x not free in t) η-conversion

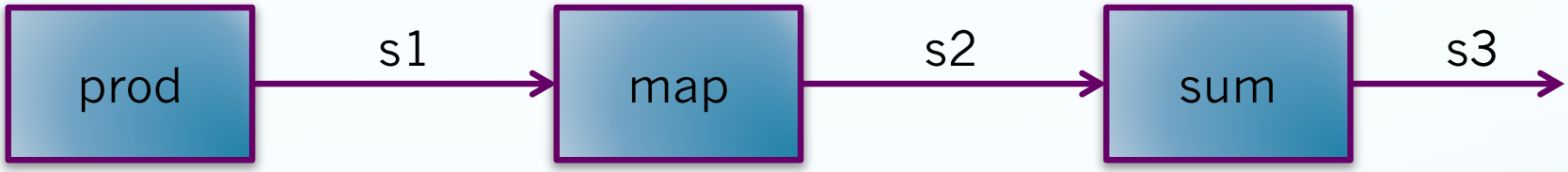
# Properties of λ calculus

- Data types and control structures
  - Data types (lists, records, numbers, etc.) and control structures (if, case, while, etc.) can be added to the λ calculus without changing anything essential
- Confluence
  - **Church-Rosser theorem: Final result of a reduction is the same for all reduction orders (up to variable renaming)**
  - This holds for many variants of the λ calculus
- Functional concurrency (examples will use λ(fut) calculus)
  - To give readable examples, we will use λ(fut), a variation of λ calculus with single-assignment variables that is also confluent
  - λ(fut) easily expresses networks of concurrent agents. An agent has internal state and sends and receives messages from neighbors.

# Functional concurrency (example in λ(fut) calculus)

- Define agents, streams, and threads:
  - Agent = tail-recursive function executing in its own thread
  - Stream = list read by one agent and created by another agent
  - Thread = a restriction on the reductions we are interested in

```
local s1 s2 s3 in
   thread s1=prod(1) end
   thread s2=map(s1,fun (x) x*x end) end
   thread s3=sum(s2,0) end
end
```




```
fun prod(n)
   delay(1000)
   n|prod(n-1)
end
```

```
fun map(s, f)
   case s of h|t then
      f(h)|map(t, f)
   [] nil then
      nil
   end
end
```

```
fun sum(s, a)
   case s of h|t then
      h+a|sum(t, h+a)
   [] nil then
      nil
   end
end
```

LIGHTKONE
Lightweight computation for networks at the edge

UCL Université catholique de Louvain

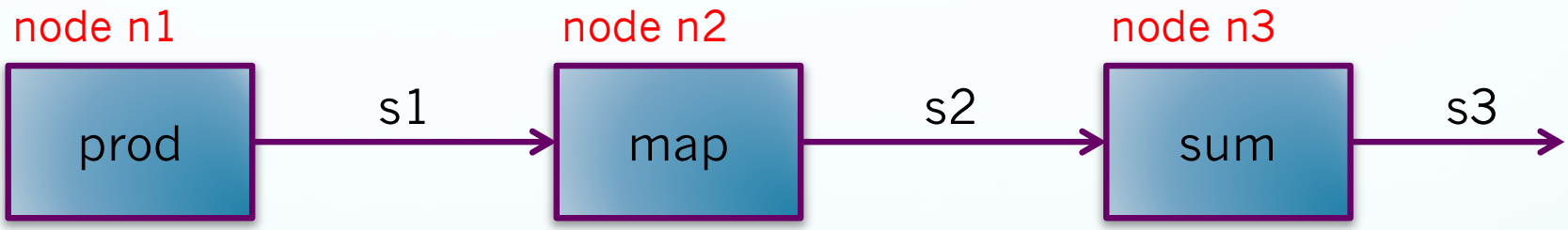
# Distributed λ calculus

- We can easily make functional programming be distributed
  - Consider a set of nodes N with a, b, c, ... ∈ N
- Localize each term on a node
  - x ::= (variables)
  - $t^a ::= x^a \mid (\lambda x.\, t^b)^a \mid (t^b_1\ t^c_2)^a$
  - Terms can reference subterms on other nodes
- Extend the reduction rules to execute on single nodes
  - $(\lambda x.\, t^a[x])^a \rightarrow (\lambda y.\, t^a[y])^a$ — α-conversion
  - $((\lambda x.t^a_1)^a\ t^a_2)^a \rightarrow t^a_1[x:=t^a_2]$ — β-reduction
  - $((\lambda x.\, (t^a\ x^a)^a)^a \rightarrow t^a$ (if x not free in $t^a$) — η-conversion
  - $t^a \rightarrow t^b$ — μ-conversion (mobility)

# Distributed functional concurrency (using λ(fut))

- We put each agent on a node
- This gives a distributed concurrent program that is purely functional
- An agent always knows from where the next input will come

```
local s1 s2 s3 in
   node s1=prod(1) end
   node s2=map(s1,fun (x) x*x end) end
   node s3=sum(s2,0) end
end
```




```
fun prod(n)
   delay(1000)
   n|prod(n-1)
end
```

```
fun map(s, f)
   case s of h|t then
      f(h)|map(t, f)
   [] nil then
      nil
   end
end
```

```
fun sum(s, a)
   case s of h|t then
      h+a|sum(t, h+a)
   [] nil then
      nil
   end
end
```

LIGHTKONE
Lightweight computation for networks at the edge



UCL Université catholique de Louvain

# Distributed λ calculus is pure

**Definition 1.** *The distributed λ-terms are defined as:* $\lambda_d \ni t^a = x^a \mid (\lambda x.t^a)^b \mid (t^a\ t^b)^c$. *Reductions* $. \xrightarrow{d} .$ *on* $\lambda_d$ *follow, where common capture-avoiding measure apply:*

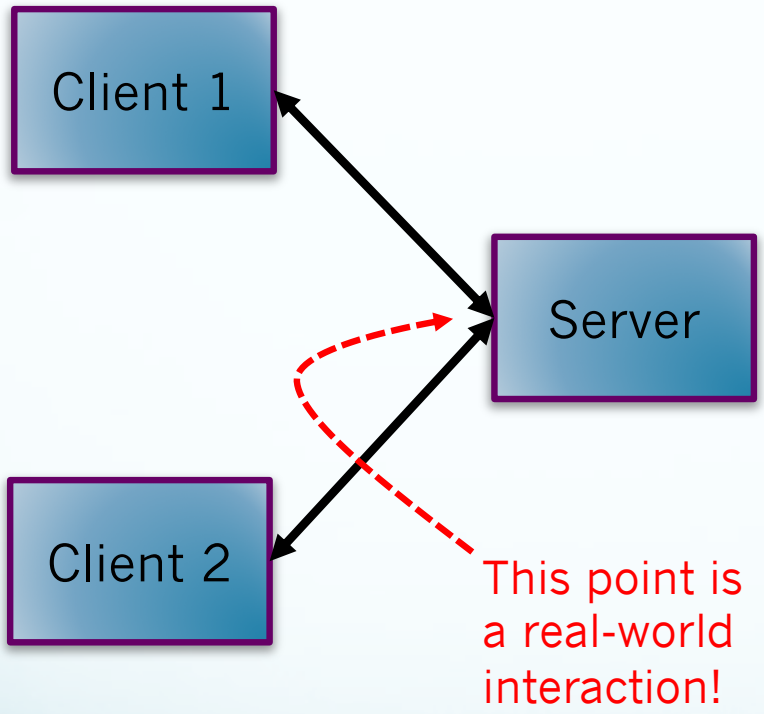
$$(\lambda x.t^a)^a \xrightarrow{d}_{\alpha} (\lambda y.t^a[y^a/x])^a \quad (\alpha_{\mathbf{d}}) \qquad ((\lambda x.t_1^a)^a\ t_2^a)^a \xrightarrow{d}_{\beta} t_1^a[t_2^a/x] \quad (\beta_{\mathbf{d}})$$
$$(\lambda x.(t^a\ x^a)^a)^a \xrightarrow{d}_{\eta} t^a \quad (\eta_{\mathbf{d}}) \qquad t^a \xrightarrow{d}_{\mu} t^b \quad (\mu_{\mathbf{d}}).$$

**Theorem 2.** *For every node a and b, the reduction* $t \overset{o}{\twoheadrightarrow} t'$ *implies* $[\![t]\!]^{+a} \overset{d}{\twoheadrightarrow} [\![t']\!]^{+a}$, *and, the reduction* $t^a \overset{d}{\twoheadrightarrow} t'^b$ *implies* $[\![t^a]\!]^{-} \overset{o}{\twoheadrightarrow} [\![t'^b]\!]^{-}$.

- We prove that a distributed λ reduction is equivalent to a standard λ reduction
- Message delays and reorderings can change the reductions, but according to Church-Rosser this has no effect on the final result

# General distributed systems

# Client/server example




- A client/server cannot be written in purely functional distributed programming
- It is because to satisfy client liveness, the server must accept each incoming request in reasonable time
- Therefore the order of the requests cannot be determined in advance because it depends on client timing
- So the program is nondeterministic
  - There is one interaction point, where the program's result is affected by the real world: where the server receives messages
  - We would like to express this in our calculus

Interaction point = intuitively, a part of the system where the real world affects the program's result

# Expressing interaction points and purity

- The pure distributed lambda calculus can only be used when there is no real-world interaction
  - To be precise: when all inputs are known in advance
- First step: add interaction points
  - There are many ways to do this
    - Distributed λ calculus extended with read and write
    - The λ(fut) calculus extended with **ports (≈ asynch. channels)**
- Second step: add observational purity
  - This allows program transformations to move and hide interaction points, to improve and verify systems
  - The λ(fut) calculus extended with **pure blocks**

# Formal definition of interaction point

- Consider a program's execution as a lambda reduction:
  $e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_m \rightarrow e_n \rightarrow \ldots$
- We define a side effect as an execution step that arbitrarily changes the expression:
  $e_0 \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_m \rightsquigarrow e_n \rightarrow \ldots$
- The step $e_m \rightsquigarrow e_n$ can be defined in various ways:
  - Port (interaction point) : $e_n$ receives a value from an external source (could be an earlier send, or another system)
  - Mutable state: $e_n$ reads state from write earlier in the reduction sequence (where write is an identity function)
  - Failure: $e_n$ drops information from $e_m$ (message loss, partition, node failure)

# Removing interaction points: CRDT example

- Improve databases by removing interaction points
- For example, replace eventual consistency by:
  - strong consistency (quorums). This fixes part of the problem, but successive operations are still nondeterministic. We can improve it by adding causal order to the system, but it's not simple.
  - convergent consistency (Conflict-free Replicated Data Types – CRDTs). A CRDT is a distributed data structure that maintains consistency without needing consensus. Realized in AntidoteDB database.

A CRDT has zero interaction points, whereas a RSM has one interaction point.

# Piecewise relative observable purity

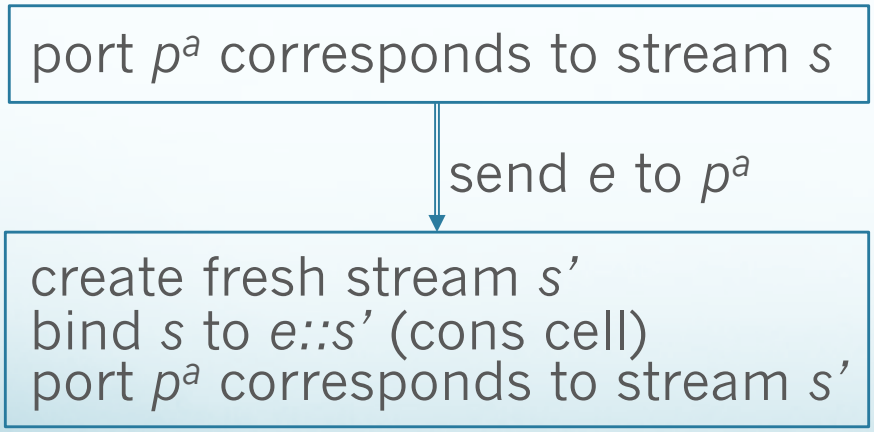
# $\lambda(\text{port})_0$ = $\lambda$ + ports + pure blocks

$$
\begin{array}{l}
e ::= x \mid c \mid \lambda x.e \mid e_1\ e_2 \mid f\ \overline{e} \mid e_1; e_2 \mid e :: s \mid \text{match } s \text{ for } \{x :: s' \Rightarrow e\} \\
\quad \mid \quad \text{send } e \text{ to } p^b \mid \text{pure}^{\overline{a}}\ \{e\}, \\
g ::= e^a \mid \text{port } p^a \mid f(\overline{x}) = \overline{e} \mid g_1 \parallel g_2.
\end{array}
$$

- port $p^a$ creates a channel $s$
  - send $e$ to $p^b$ causes $e$ to appear on the channel $s$
  - Ports are side effects because sends arrive asynchronously
- pure$^{\bar{a}}$ {$e$} creates a pure block
  - A promise to the programmer that $e$ will have no side effects observable from nodes $\bar{a}$ (= $a_1, a_2, \ldots, a_n$)
  - We use pure blocks to do program transformations, such as removing or combining side effects, and purely functional transformations

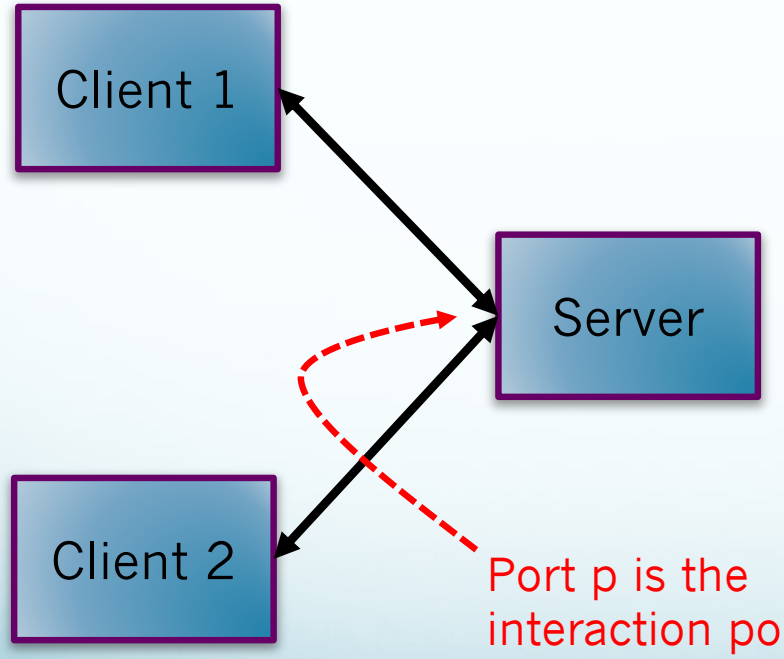
# Ports

- A port is an asynchronous communication channel that is designed to integrate well with λ(fut)
  - The port has a corresponding *stream*, which is a list that is built incrementally by adding messages sent
  - A computation that reads this list will synchronize on new elements appearing
- Semantics:



# Client/server in λ(fut) + ports

- Now we can define a client/server
- $f_c$ and $f_s$ are pure functions
- There is just one interaction point




```
local s p in
   node p=newport(s) server(state,s) end
   node client(state1,p) end
   node client(state2,p) end
   ...  /* as many clients as we need */
end

fun client(state,p)
   send(query(state),p)
   client(fc(state),p)
end

fun server(state,s)
    case s of q|t then
          server(fs(q,state),t)
     [] nil then
          nil
     end
end
```

One interaction point

# Client/server in $\lambda(\text{port})_0$

$$\begin{aligned}&\textsf{port } p^{srv} \parallel (srv\ st_s\ \textsf{stream}(p^{srv})) \parallel (client\ st_1)^{c_1} \parallel (client\ st_2)^{c_2}\\ &\quad\parallel client(st) = \textsf{pure}^{\overline{c}}\ \{\textsf{send}\ (query\ st)\ \textsf{to}\ p^{srv};\ client\ (f_c\ st)\}\\ &\quad\parallel srv\ (st, s) = \textsf{pure}\ \{\textsf{match}\ s\ \textsf{for}\ \{q :: s' \Rightarrow srv\ (f_s\ (q, st), s')\}\}\end{aligned}$$

- The client/server written in $\lambda(\text{port})_0$ syntax
- A client is a recursive function that sends to a port
- The server is a recursive function that reads from the port's stream

# Pure blocks

- A pure block allows to delimit an expression that has no observable side effects (= sends to ports)
- Assume nodes $\bar{a} = a_1, a_2, \ldots, a_n$ for some known $n$

```
pure^ā {
        e1 ;
        e2 ;
        ...
        em ;
}
```

- This is a promise to the programmer that $e_1, e_2, \ldots, e_n$ will not have any observable side effects seen by nodes $a_1, a_2, \ldots, a_n$ .
- This can be checked at run-time or compile-time

# Pure block transformations

- We are working on a theory of pure block transformations. Our first main result is:

**Theorem 2.** *Let* $e_1, e_2 \in E$ *and* $\overline{a} \in \mathfrak{N}$. *Then,* $\forall \tau.\ \overline{a} \notin \Delta^*_{\mathsf{node}}(\tau, e_1)$ *implies* $e_1; \mathsf{pure}^{\overline{a}}\ \{e_2\} \sim^a_\circ \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$. *Likewise,* $\forall \tau.\ \overline{a} \notin \Delta^*_{\mathsf{node}}(\tau, e_2)$ *implies* $\mathsf{pure}^{\overline{a}}\ \{e_1\}; e_2 \sim^a_\circ \mathsf{pure}^{\overline{a}}\ \{e_1; e_2\}$.

- This theorem enables a pure block transformation:

  if "$e_1$ is pure w.r.t. nodes ā"
  *(nodes ā see no side effects from $e_1$)*
  then
  $e_1$; pure$^{\bar{a}}$ {$e_2$} is equivalent to pure$^{\bar{a}}$ {$e_1$;$e_2$}

# Piecewise Relative Observable Purity

- PROP is a design language for distributed systems where programs are specified using λ + ports + pure blocks:
  - Programs are concurrent compositions of pure blocks
  - Pure blocks specify relative to which nodes they are pure
  - Pure blocks can be nested
- Example:

  port $p_1$ || port $p_2$ || …
  || $f_1(x_1, \ldots, x_m)=e_1;\ldots;e_n$ || $f_2(x_1, \ldots, x_m)=e_1;\ldots;e_n$ || …
  || $\text{pure}^{\bar{a}}\ \{e_1;e_2\};\ e_3;\ e_4$
  || $\text{pure}^{\bar{g}}\ \{e_5;\ \text{pure}^{\bar{o}}\{e_6;e_7\}\}$ || …

- We observe that realistic distributed systems are mostly functional
  - Specifying what parts of the system are pure allows powerful transformation and verification techniques
  - This is still in an early stage and we are working hard on making it a practical and useful approach for distributed systems design

# Usefulness of PROP

- Key properties of purity and observational purity
  - Order of interleaving makes no difference
  - Executions are idempotent
- Many possible uses
  - Testing and verifying pure subsystems is simplified
  - Designing systems with awareness of interaction points
  - Program transformations can isolate and reduce side effects, increasing the usefulness
  - Deterministic parallellism is straightforward
  - Speculative execution, which improves performance by trading off computation for communication, is simplified
  - Partial evaluation is possible for pure subsystems

# Conclusions

- There exists a useful purely functional subset of distributed programming
  - Pure distributed computations do not interact with the real world (all inputs are known in advance), but support message asynchrony and reordering
- General distributed programming consists of purely functional distributed programming plus interaction points
  - Many realistic distributed programs need very few interaction points: distributed computations are mostly functional
- We want to design distributed systems explicitly as a purely functional core plus interaction points
  - To enable reasoning about purity, we add pure blocks to the design language
- We are working on a design language for specifying distributed systems, called Piecewise Relative Observable Purity (PROP)
  - We are investigating how this can be used as a tool for helping distributed systems designers
  - We are looking for feedback and suggestions on practical uses for PROP

# Extra information

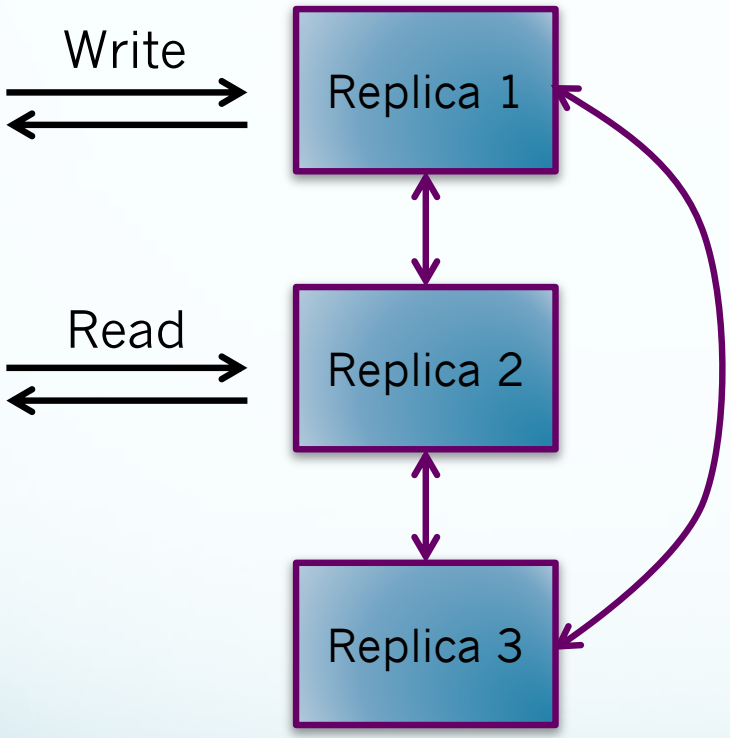
# Read-write distributed λ calculus

- We add read and write operations to the distributed λ calculus
  - Result depends on reduction order and timing, so they are interaction points
  - If the read returns the result of the most recent write, then it's **mutable state**
  - But write and read can also behave like **send and receive**

- Add read and write terms
  - x ::= (variables)
  - $t^a ::= x^a \mid (\lambda x.\, t^b)^a \mid (t^b_1\ t^c_2)^a \mid (\sigma.t^b)^a \mid (\rho x.\, t^b)^a$

- Add two reduction rules
  - $(\lambda x.\, t^a[x]) \rightarrow (\lambda y.\, t^a[y])$ — α-conversion
  - $((\lambda x.t^a_1)\ t^a_2)^a \rightarrow t^a_1[x:=t^a_2]$ — β-reduction
  - $((\lambda x.\, (t^a\ x))^a \rightarrow t^a$ (if x not free in $t^a$) — η-conversion
  - $t^a \rightarrow t^b$ — μ-conversion (mobility)
  - $(\sigma.t^a)^a \rightarrow t^a$ — σ-reduction (store or send, a.k.a. write)
  - $(\rho x.t^a_1)^a \rightarrow t^a_1[x:=t^a_2]$ — ρ-reduction (read or receive)

# Eventual consistency




- Commonly done for performance
  - Requests can be initiated concurrently; multiple requests can be "in flight" simultaneously; replies are returned as quickly as possible
  - Writes are eventually propagated to all replicas; reads are eventually handled by at least one replica
- Consider a replicated database
  - A write is done and immediately followed by a read (without waiting for the write to finish)
  - Does the read see the write?
    - Sometimes yes, sometimes no!
- How should we think about this?
  - Focus on the interaction points! Can we get rid of them?