Purely Functional Programming Order-Insensitive Asynchronous Message Passing

Seyed H. HAERI (Hossein) & Peter Van Roy

UCLouvain, Belgium

19 Jun 2020

ICE 2020 -- 13th Interaction and Concurrency Experience

www.discotec.org/2020/ice

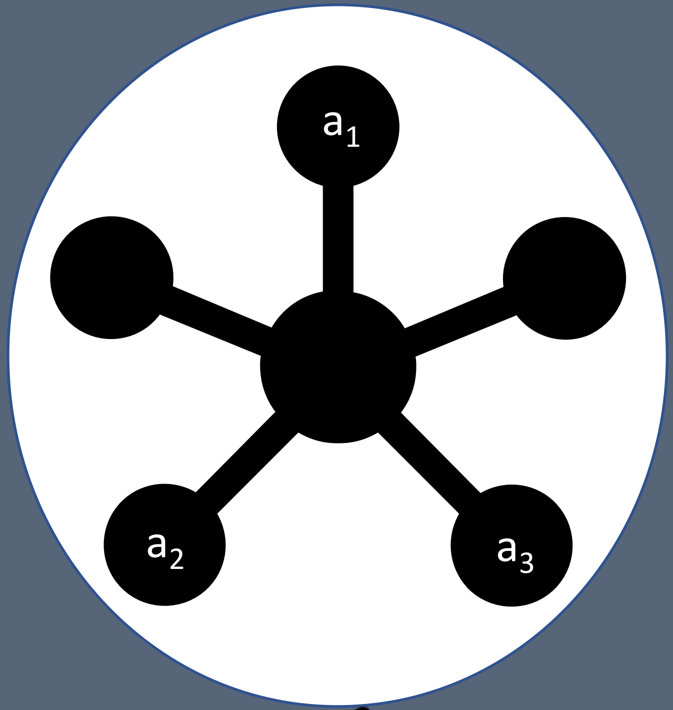
UCLouvain

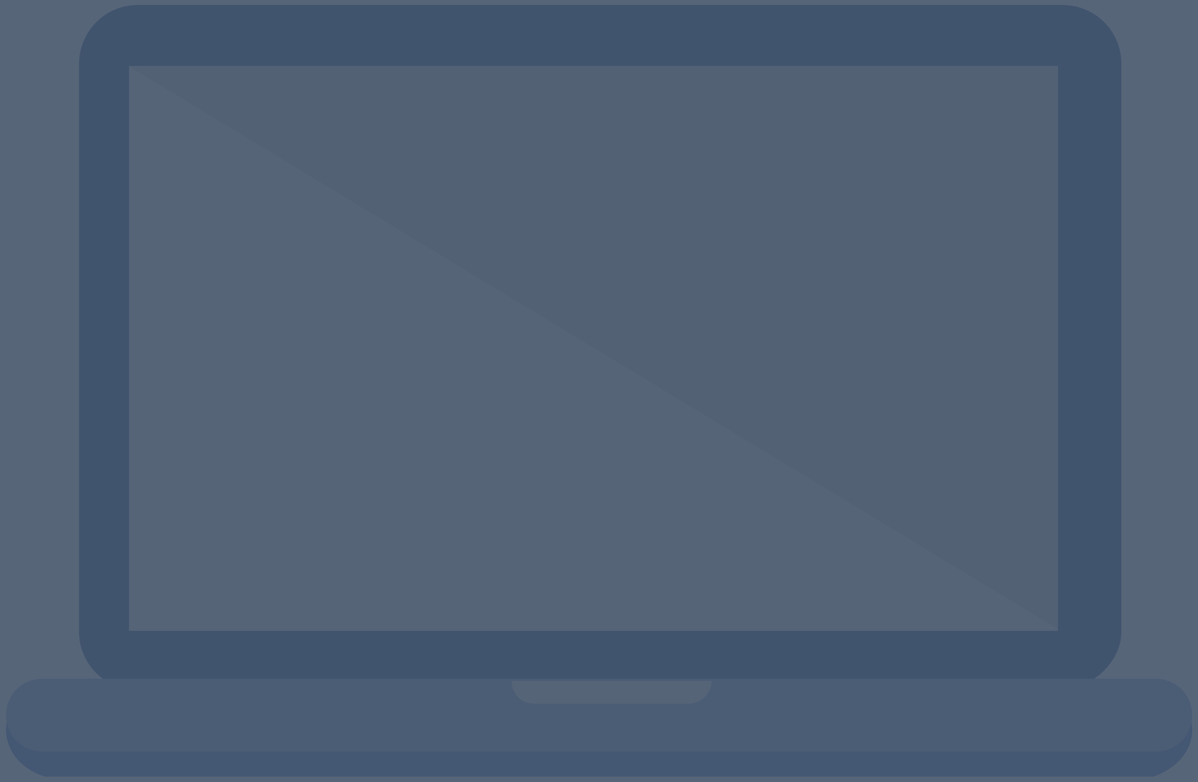
icteam Institute of Information and Communication Technologies, Electronics and Applied Mathematics

LIGHTKONE Lightweight computation for networks at the edge

INPUT | OUTPUT

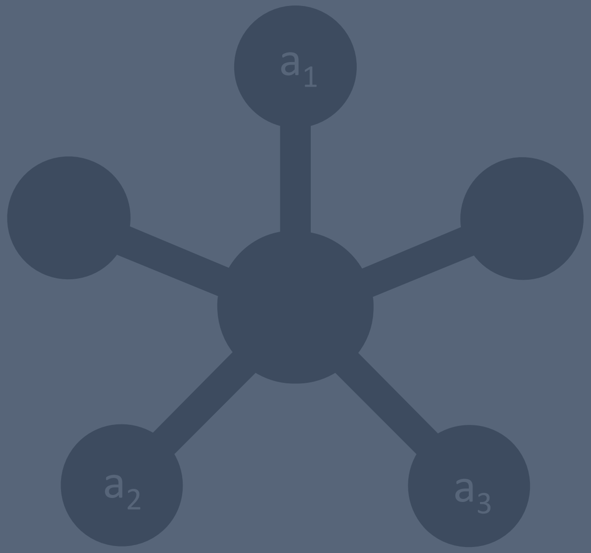
What's a distributed system?

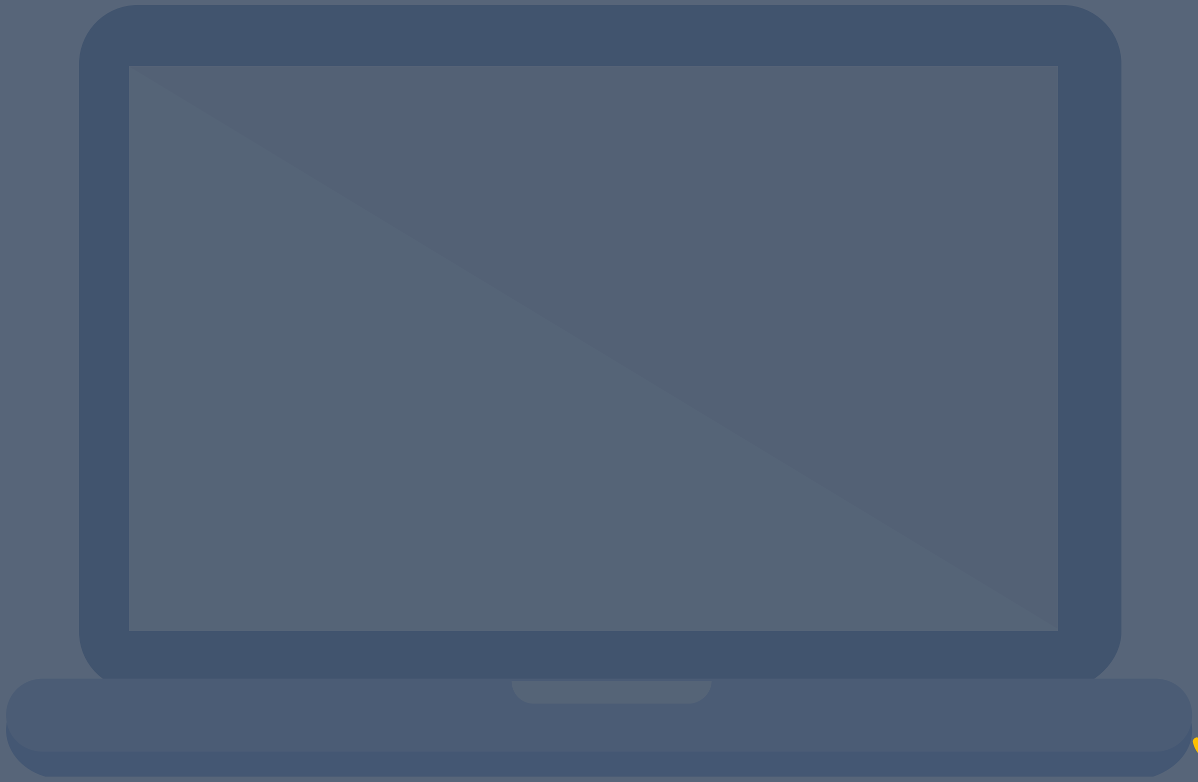




A collection of network connected nodes…

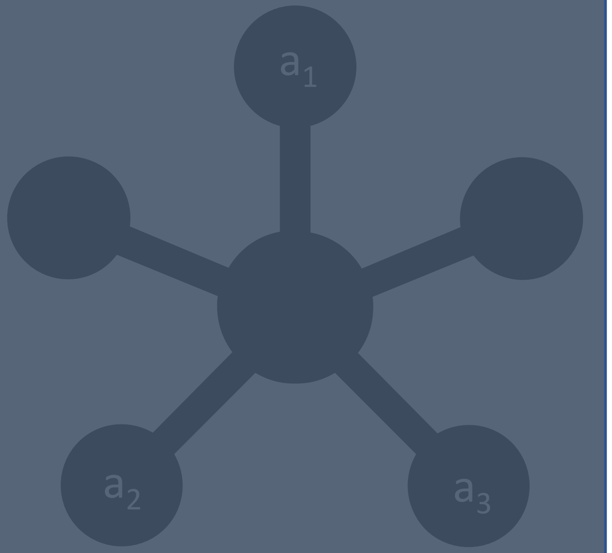
What's a distributed system?

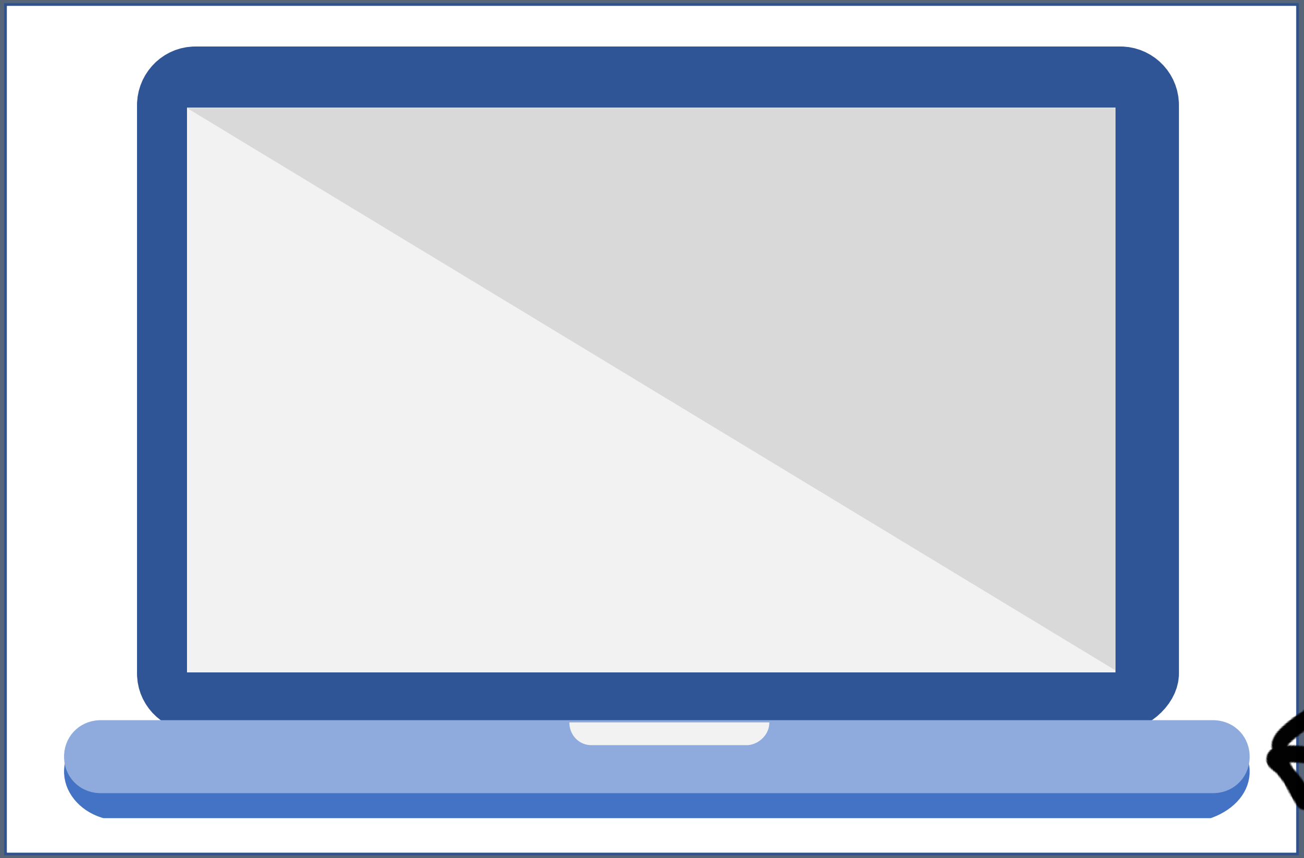




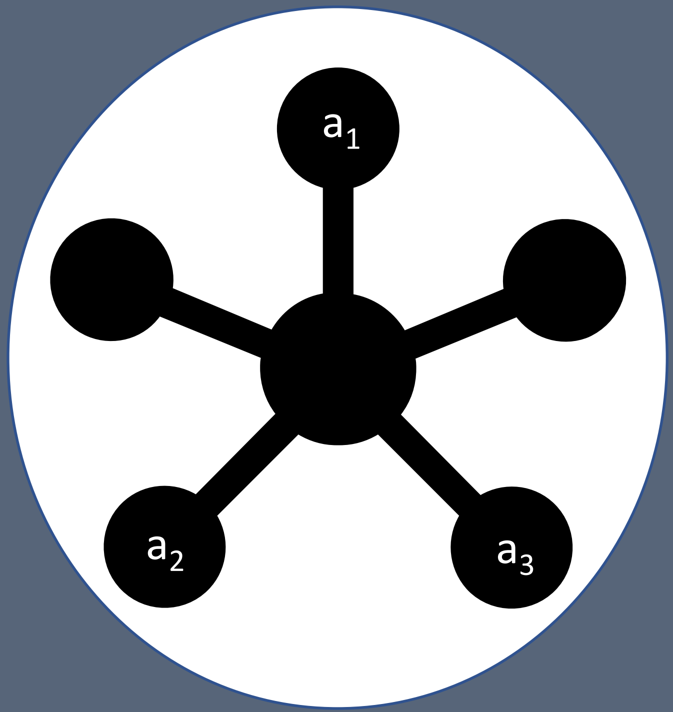
... that from a user's point of view...

What's a distributed system?



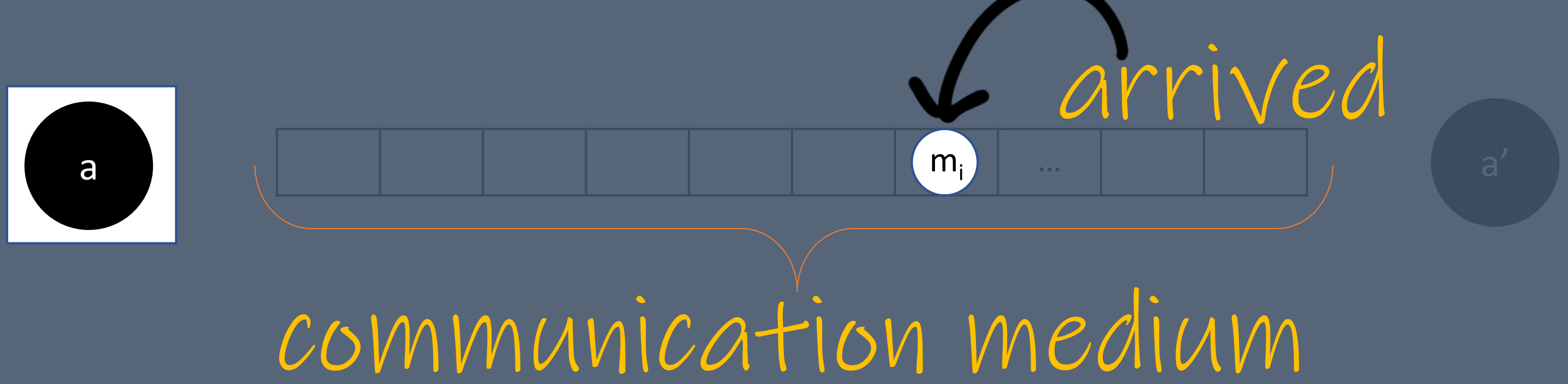


... looks like a single system.



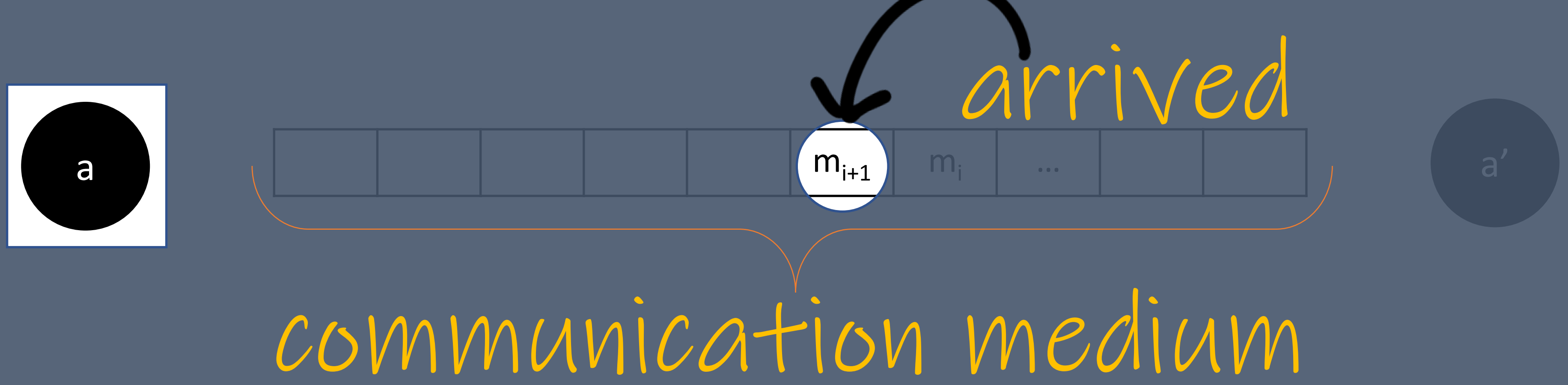
Nodes talk to one another.

→ Message-Passing



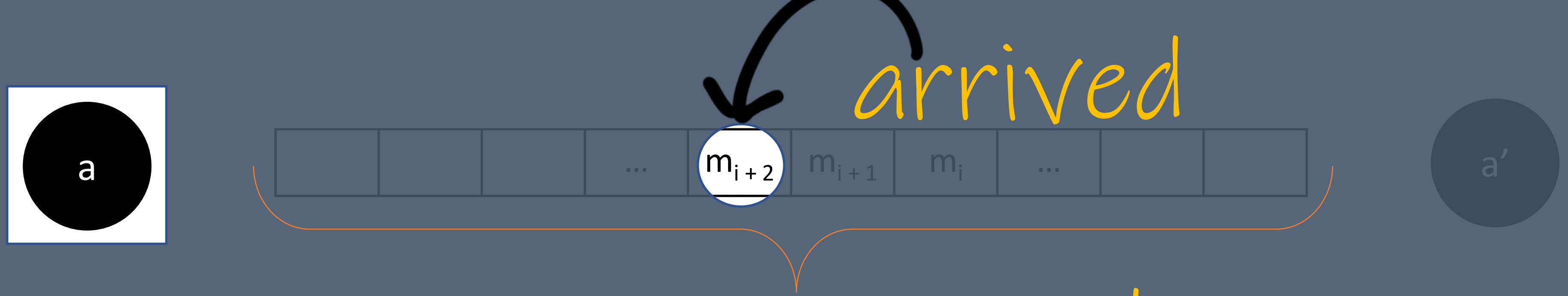
- Node a keeps sending

➔ …, m_i, m_{i+1}, m_{i+2}, … keep adding to the communication medium



- Node a keeps sending

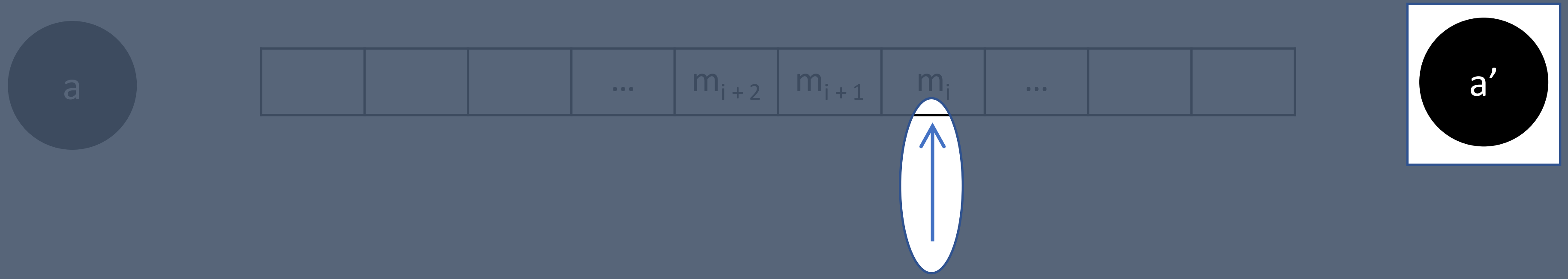
➔ …, m_i, m_{i+1}, m_{i+2}, … keep adding to the communication medium



communication medium

- Node a keeps sending

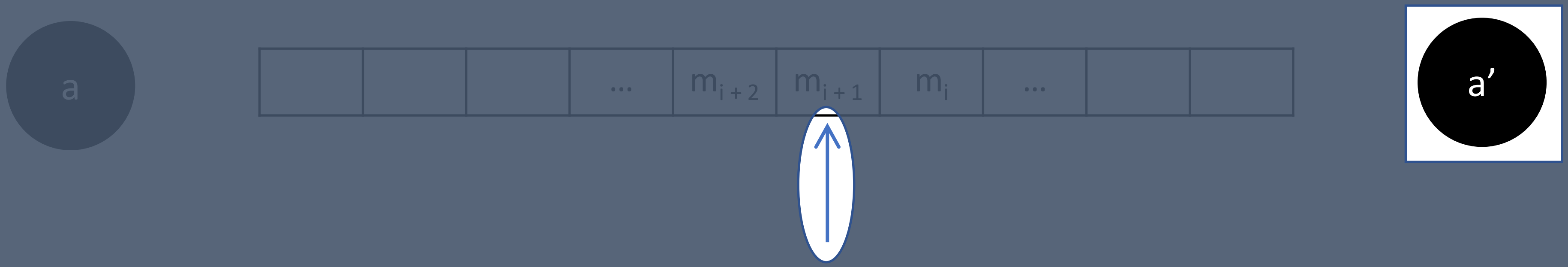
➔ …, m_i, m_{i+1}, m_{i+2}, … keep adding to the communication medium



reading indicator of a'

- Node a' keeps reading them

  ➔ Updating its indicator



reading indicator of a’

- Node a’ keeps reading them

  ➔ Updating its indicator

- Common Judgment: “Change of State”
  ➔ <u>Impurity</u>
- Multiple Senders
- Asynchronous
- Order-Sensitivity:
  “Served in the (Causal) Order of Arrival”

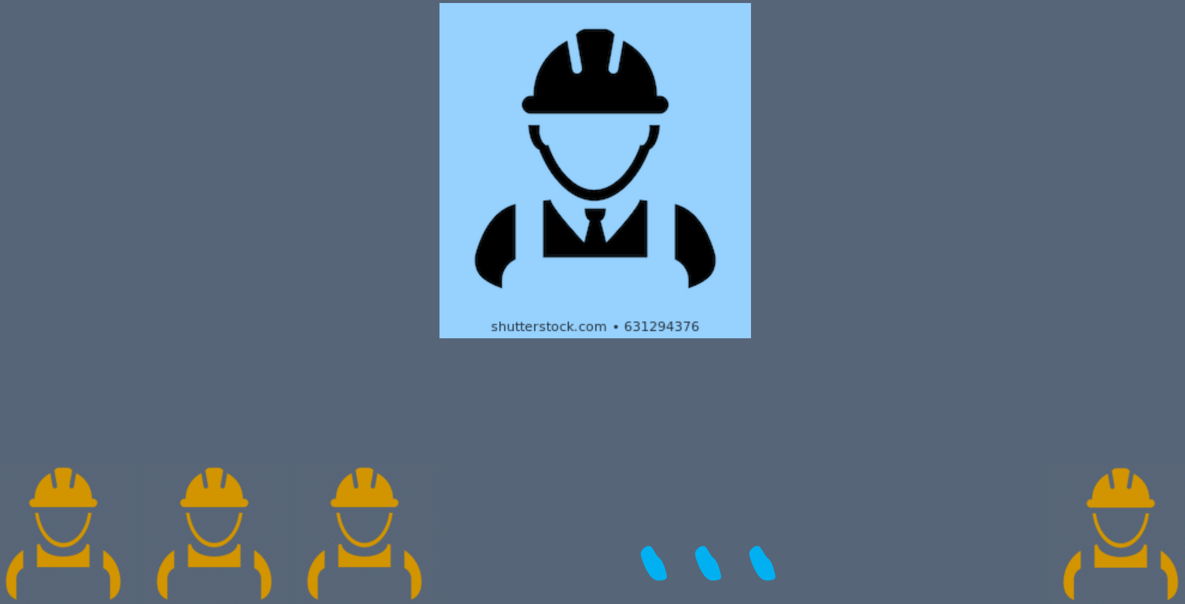
What if the order is insignificant?

- Remote futures can be used.
- And, they are pure.
- λ(refut) Formal Model
  - Simple **let**-Notation for Remote Futures
  - Examples to Come

What is purity?

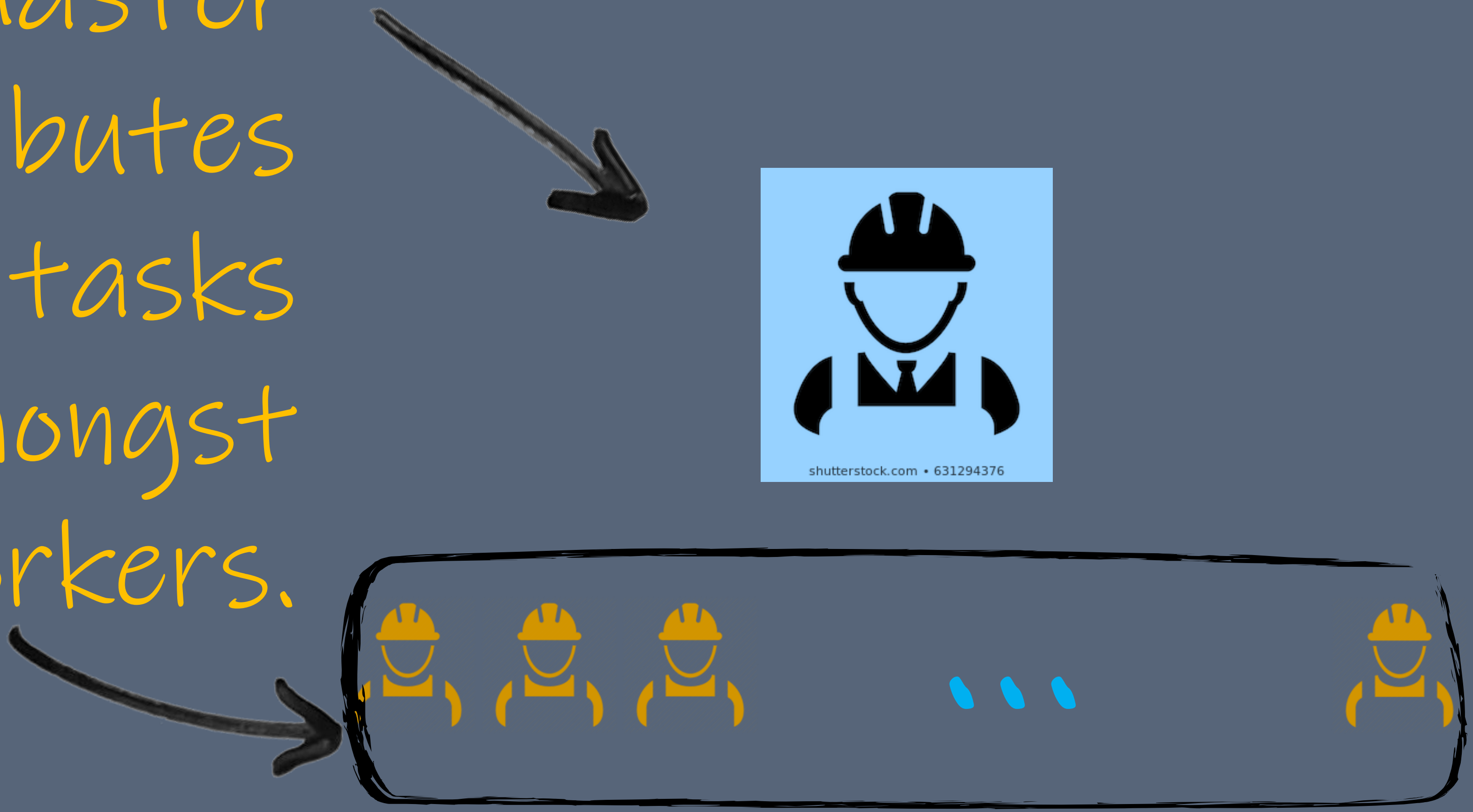
- Very Diverse Comprehensions
- Ours
  - Simply: Lack of Side-Effects + Determinism
  - More Formally: Pure Functional Programming
  - Formally: λ(refut) ~ Untyped λ-Calculus

Example



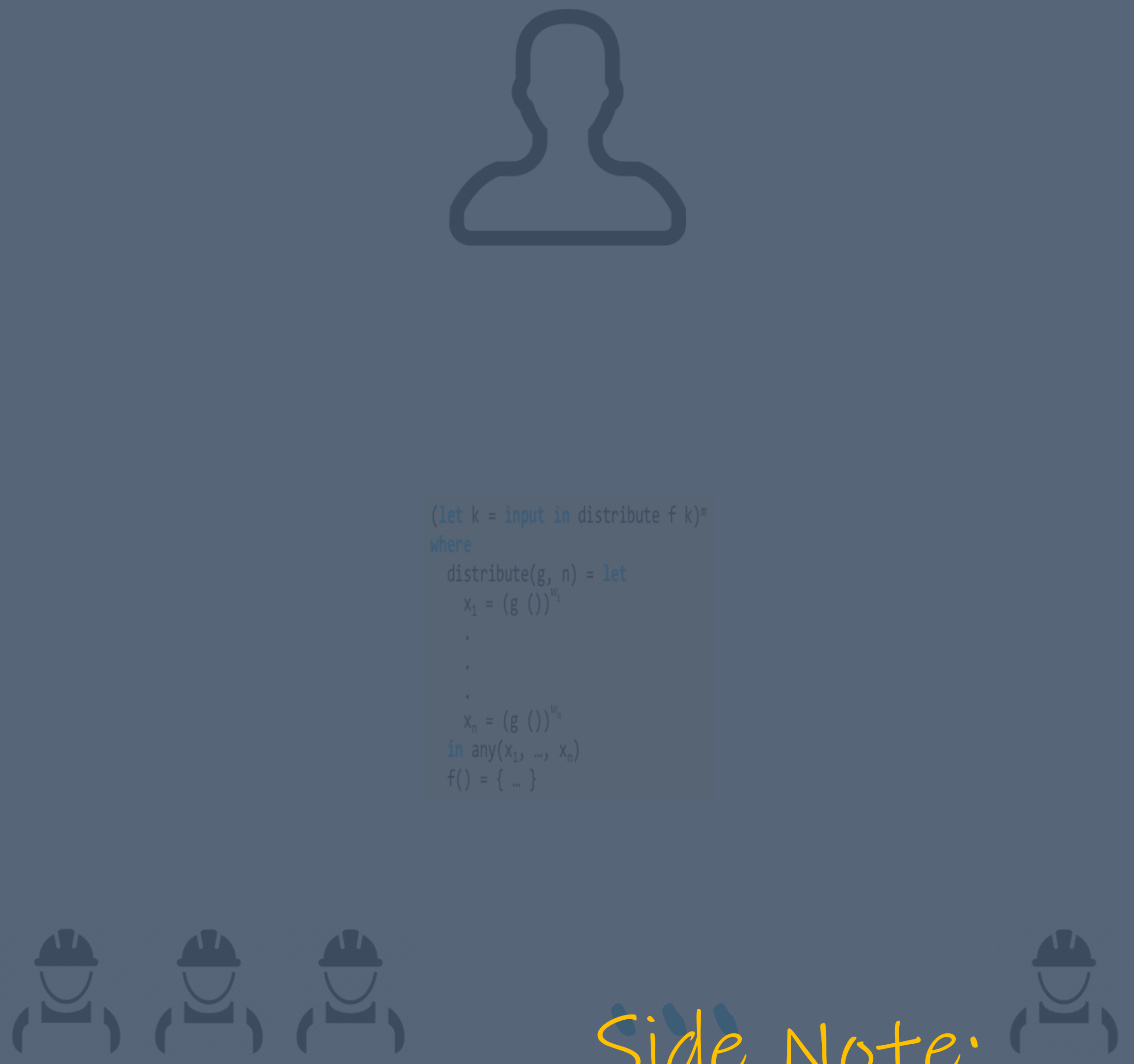
Master-Worker Scenario

Master distributes tasks amongst workers.



```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^{w_1}
    .
    .
    .
    x_n = (g ())^{w_n}
  in any(x_1, …, x_n)
  f() = { … }
```

A Piece of λ(refut) Code (Our Formal Model)



```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^{w_1}
    .
    .
    .
    x_n = (g ())^{w_n}
  in any(x_1, …, x_n)
  f() = { … }
```

Side Note:

refut in λ(refut) is fo

remote futures.

```
(let k = input in distribute f k)^m
where
    distribute(g, n) = let
        x_1 = (g ())^{w_1}
        .
        .
        .
        x_n = (g ())^{w_n}
      in any(x_1, …, x_n)
    f() = { … }
```

Running Part

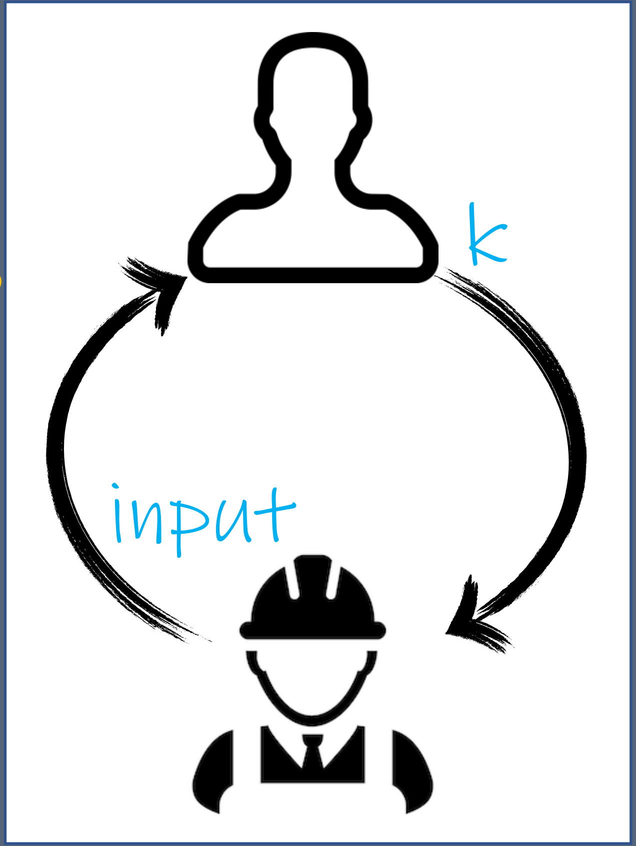
Definitions

```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^w_1
    .
    .
    .
    x_n = (g ())^w_n
  in any(x_1, …, x_n)
  f() = { … }
```

Running on
"m"
(The Master Node)

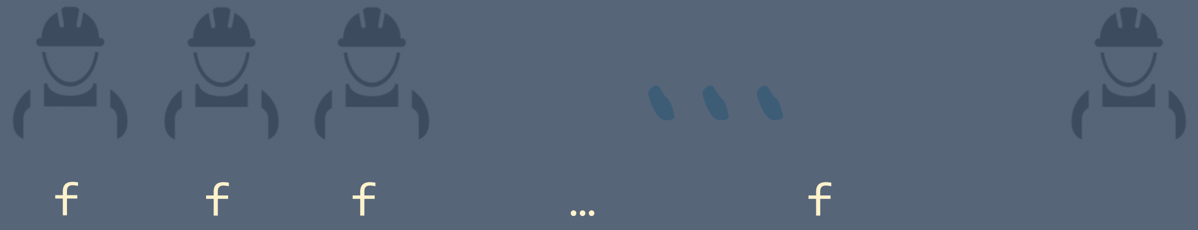
```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^w_1
    .
    .
    .
    x_n = (g ())^w_n
  in any(x_1, …, x_n)
  f() = { … }
```

The master inputs "k" (#workers) from the user.



```
(let k = input in distribute f k)^m
where
   distribute(g, n) = let
      x_1 = (g ())^w_1
      .        …
      .        …
      .        …
      x_n = (g ())^w_n
   in any(x_1, …, x_n)
   f() = { … }
```

The master invokes k different copies of "f" (in parallel)...

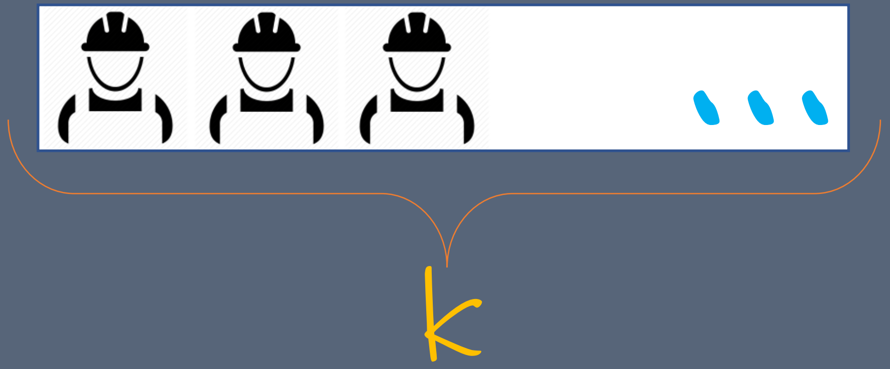


```
(let k = input in distribute f k)m
where
  distribute(g, n) = let
    x1 = (g ())
    .
    .
    .
    xn = (g ())
  in any(x1, …, xn)
  f() = { … }
```

w_1

w_n

… on k different workers.



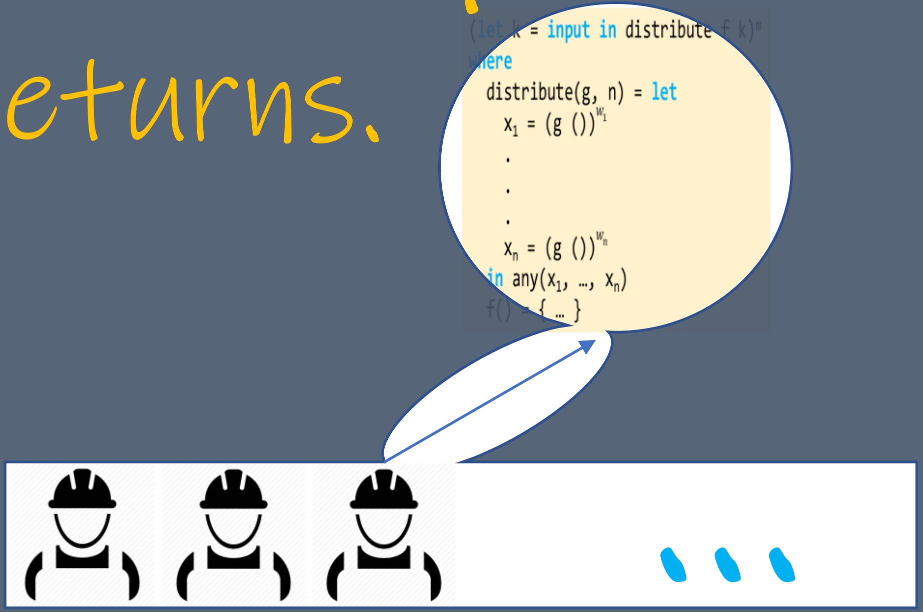
```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x1 = (g ())^w1
    .
    .
    .
    xn = (g ())^wn
  in any(x1, …, xn)
  f() = { … }
```

The master keeps references to those invocations…

```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x1 = (g ())^w1
    .
    .
    .
    xn = (g ())^wn
  in any(x1, …, xn)
  f() = { … }
```

```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^{w_1}
    .
    .
    .
    x_n = (g ())^{w_n}
  in any(x_1, …, x_n)
  f() = { … }
```

… and, returns the result as soon as **any** of those returns.

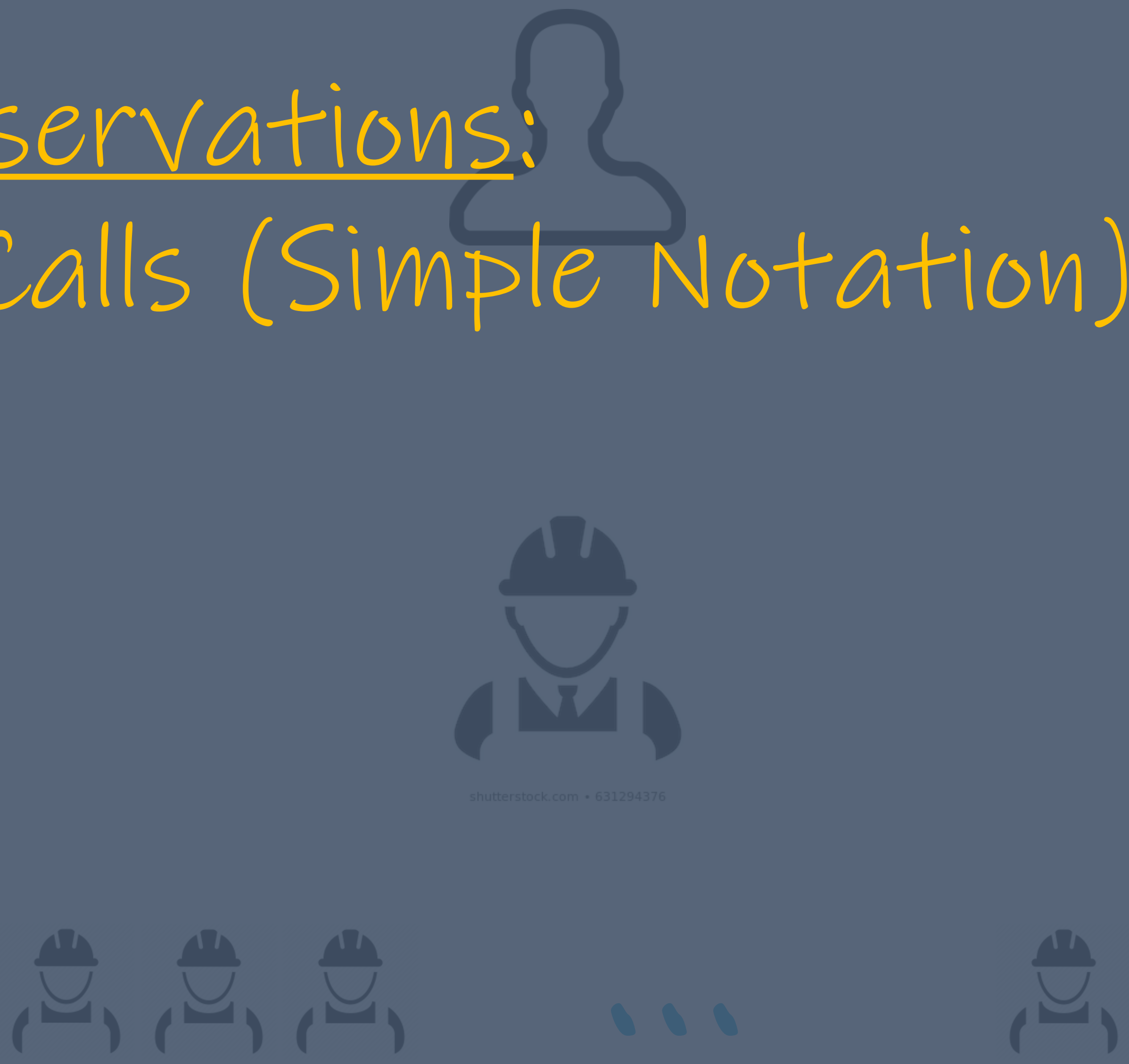


```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^{w_1}
    .
    .
    .
    x_n = (g ())^{w_n}
  in any(x_1, …, x_n)
  f() = { … }
```

λ(refut) Observations:

1. Remote Calls (Simple Notation)

```
(let k = input in distribute(f, k))^m
where
  distribute(g, n) = let
    x_1 = (g ())^{w_1}
    .      …
    .      …
    .      …
    x_n = (g ())^{w_n}
  in any(x_1, …, x_n)
  f() = { … }
```



λ(refut) Observations:

1. Remote Calls (Simple Notation)
2. Remote Futures (Familiar **let**-bindings)

```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^{w_1}
    .
    .
    .
    x_n = (g ())^{w_n}
  in any(x_1, …, x_n)
  f() = { … }
```

```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^{w_1}
    .
    .
    .
    x_n = (g ())^{w_n}
  in any(x_1, …, x_n)
  f() = { … }
```

Side-Effects

```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^w_1
    .
    .
    .
    x_n = (g ())^w_n
  in any(x_1, …, x_n)
  f() = { … }
```

1 Input/Output
Have fun w/ your IO monad!

2 Nondeterminism
Have fun w/ your favourite effect system!

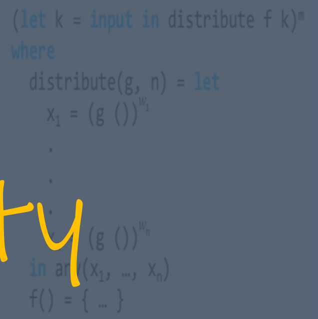
```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^w_1
    .
    .
    .
    x_n = (g ())^w_n
  in any(x_1, …, x_n)
  f() = { … }
```

Take-Home Lesson:

- Message-Passing **Pure**!
- Order of Message Processing Insignificant

➔ Order-Insensitivity

```
(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^{w_1}
    .
    .
    .
    x_n = (g ())^{w_n}
  in any(x_1, …, x_n)
  f() = { … }
```



(let k = input in distribute f k)^m
where
  distribute(g, n) = let
    x_1 = (g ())^{w_1}
    (g ())^{w_n}
  in any(x_1, …, x_n)
  f() = { … }

More in the paper:

- Earlier Version Online:

  http://hdl.handle.net/2078.1/229005
- More Examples of Order-Insensitive Message Passing
- Formal Syntax and (Reduction) Semantics of λ(refut)