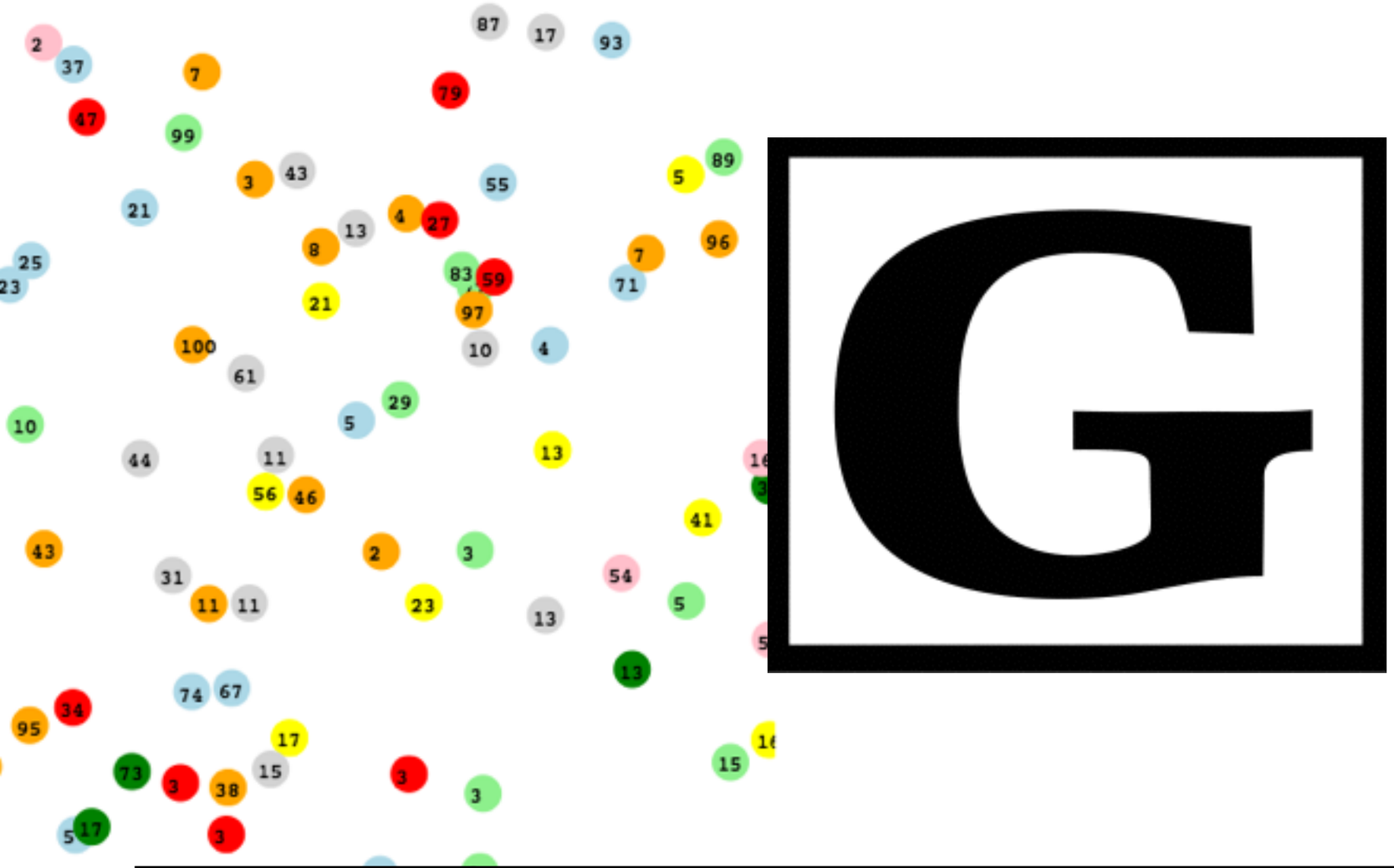
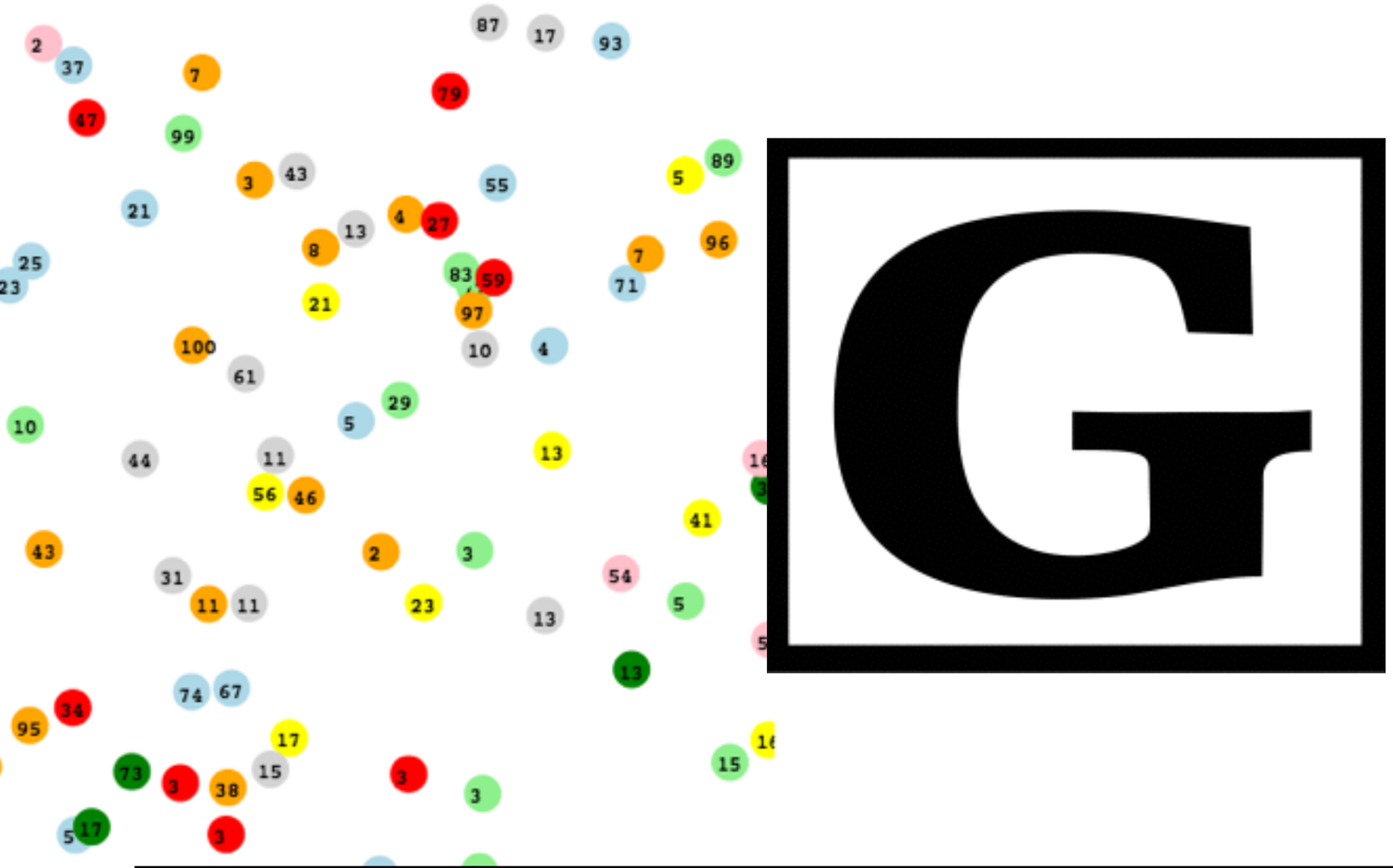
Unconventional Programming with Chemical Computing

_ Carin Meier
@gigasquid



GENERAL PROGRAMMERS



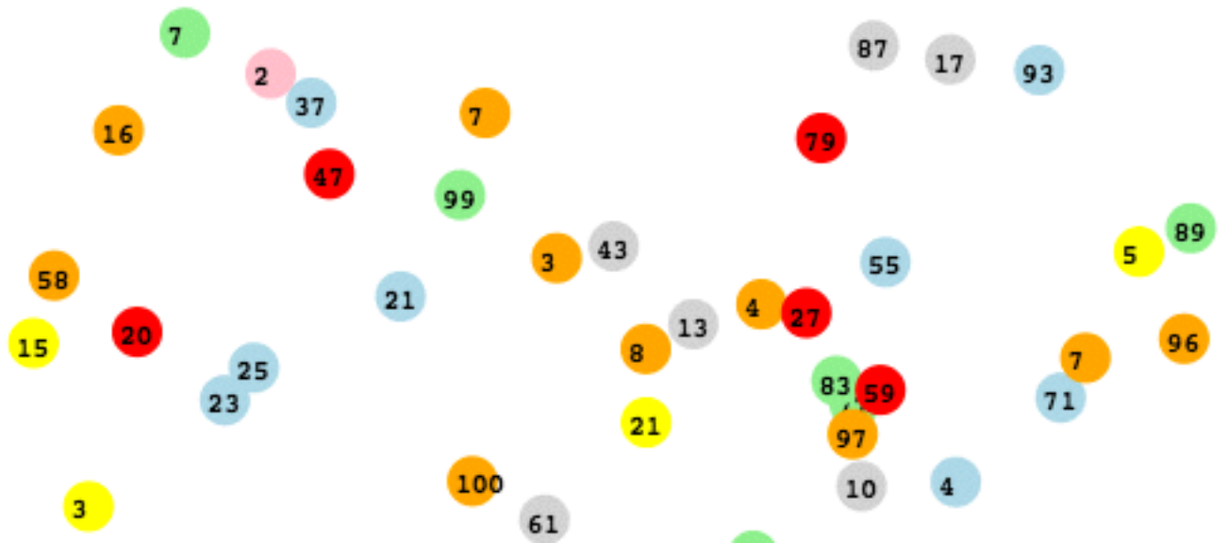
GENERAL PROGRAMMERS

Intense Sequences of LISP Code

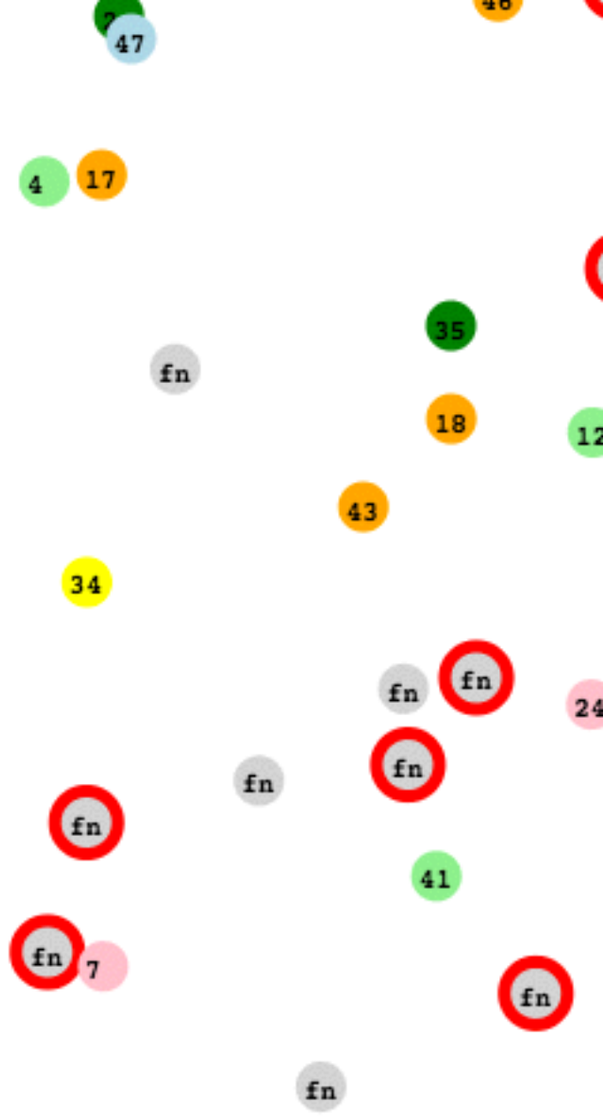
Starring Papers

Chemical Computing

by Peter Dittrich



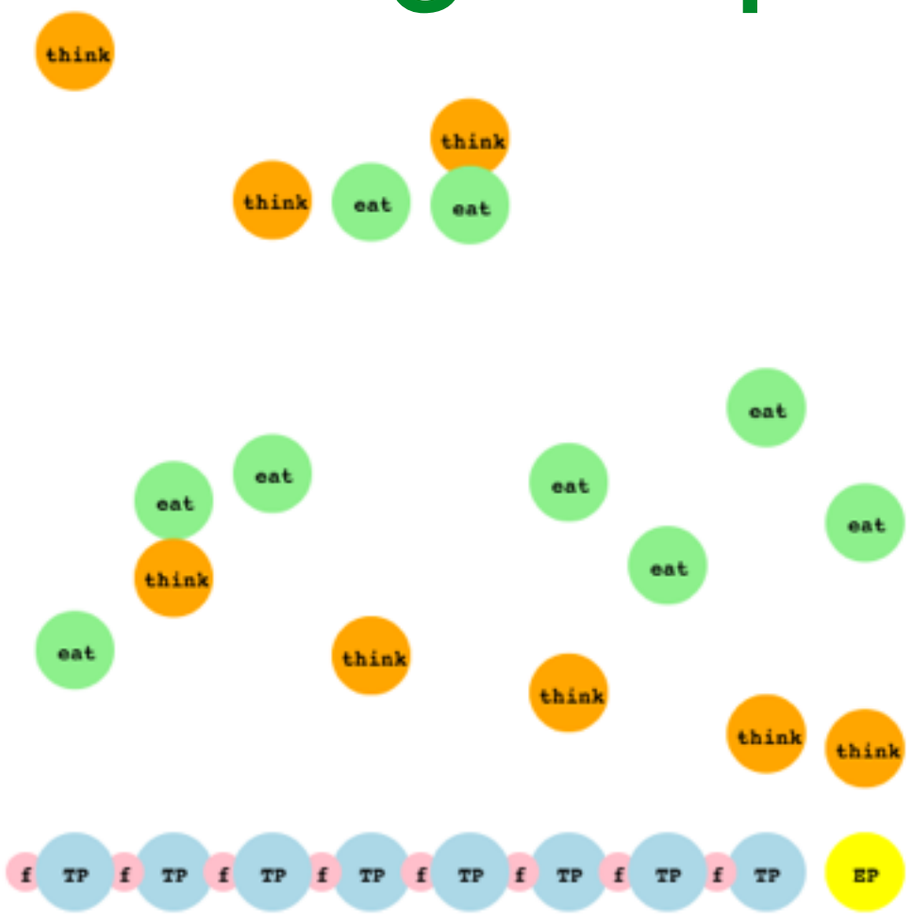
Starring Papers



Higher-Order Chemical Programming Style

by J.P. Banâtre, P. Fradet, Y. Radenac

Starring Papers



Principles of Chemical Programming

by J.P. Banâtre, P. Fradet, Y. Radenac

Starring Papers

Programming Self-Organizing Systems with the Higher-Order Chemical Language

by J.P. Banâtre, P. Fradet, Y. Radenac

Narrator

Carin Meier
aka @gigasquid
author of Living Clojure
works at Cognitect

Prologue

Prologue

Cutting the lawn

Prologue

Cutting the lawn

Programming Unconventional Programming Paradigms

Prologue

TREE BRANCH

Prologue

Prologue

Grass is computing

Tree is computing

I am computing

Prologue

All Living Things Process Information with Chemical Reactions on the Molecular Level

Prologue

Endocrine System with Hormones

Adaptive Defense Immune System

Signal Processing in Bacteria

Wait

Are we going to be programming with chemicals?

No

Although that would be cool too

Abstract Chemical

We are going to be using the metaphor of molecules and reactions

Wait

What am I going to get out of this talk?

Here is the exciting part ….

I DO NOT KNOW

and

That is awesome!

Cross Fertilization

Computer Programming Nature & Biology

Cross Fertilization

Computer Programming Nature & Biology

RESEARCH

NEW IDEAS

INNOVATION

Cross Fertilization

Computer Programming Nature & Biology

RESEARCH

NEW IDEAS

INNOVATION

What might inspire you?

The Reaction

The Reaction

Compare two examples

Traditional Primes

Primes with Prime Reaction

Traditional Primes

```
(defn is-prime? [n]
  (let [possible-factors (range 2 n)
        remainders (map #(mod n %) possible-factors)]
    (not (some zero? remainders))))

(is-prime? 5)
;; -> true

(is-prime? 6)
;; -> false
```

WAIT!

What language is that with all the parens?

Clojure

Hitchhiker's Guide to Clojure

Don't Worry About the Parens

Clojure

Dynamic

Functional

Java Interop

Concurrency

```
(def cat "cat")
;=> "cat"
```

Clojure

Dynamic

Functional

Java Interop

Concurrency

```
(defn say-hello [name]
  (str "hello " name))

(say-hello "molecule")
;=> "hello molecule"
```

Clojure

Dynamic

Functional

Java Interop

Concurrency

```
(class "molecule")
 ;=> java.lang.String

(.toUpperCase "molecule")
;=> "MOLECULE"
```

Clojure

Dynamic

Functional

Java Interop

Concurrency

Immutable Data

Vars

Refs

Atoms

Agents

ClojureScript

Dynamic

Functional

JavaScript Interop

Concurrency

```
(js/alert “Hi there alert!”)
```

Traditional Primes

```
(defn is-prime? [n]

                                                )
```

Traditional Primes

```
(defn is-prime? [n]
  (let [possible-factors (range 2 n)
        remainders (map #(mod n %) possible-factors)]
                                   ))
```

Traditional Primes

```
(defn is-prime? [n]
  (let [possible-factors (range 2 n)
        remainders (map #(mod n %) possible-factors)]
    (not (some zero? remainders))))
```

Traditional Primes

```
(defn is-prime? [n]
  (let [possible-factors (range 2 n)
        remainders (map #(mod n %) possible-factors)]
    (not (some zero? remainders))))

(is-prime? 5)
;; -> true

(is-prime? 6)
;; -> false
```

Traditional Primes

```
(defn gen-primes [n]
  (filter is-prime? (range 2 (inc n))))

(gen-primes 100)
 ;=> (2 3 5 7 11 13 17 19 23 29 31 37 41
43 47 53 59 61 67 71 73 79 83 89 97)
```

Prime Reaction

Prime Reaction

Prime Reaction

```
(defn prime-reaction [[a b]]
  (if (and (> a b)
           (zero? (mod a b)))
    [(/ a b) b]
    [a b]))
```

Prime Reaction

```
(defn prime-reaction [[a b]]
  (if (and (> a b)
           (zero? (mod a b)))
    [(/ a b) b]
    [a b]))

(prime-reaction [6 2])
;; -> [3 2]

(prime-reaction [5 2])
;; -> [5 2]
```

Prime Reaction

```
(def molecules (range 2 101))
```

Prime Reaction

```
(def molecules (range 2 101))

(defn mix-and-react [mols]
  (let [mixed (partition 2 (shuffle mols))
        reacted (map prime-reaction mixed)]
    (flatten reacted)))
```

Prime Reaction

```
(def molecules (range 2 101))

(defn mix-and-react [mols]
  (let [mixed (partition 2 (shuffle mols))
        reacted (map prime-reaction mixed)]
    (flatten reacted)))

(take 10 (mix-and-react molecules))
;; -> (37 48 87 46 38 91 68 13 39 33)
```

Prime Reaction

```
(defn reaction-cycle [n]
  (loop [i n
         mols molecules]
    (if (zero? i)
      mols
      (recur (dec i) (mix-and-react mols)))))


(take 10 (reaction-cycle 100))
;; -> (2 2 11 23 2 2 5 3 79 17)
```

Prime Reaction

```
(let [reactions (reaction-cycle 10000)]
  (-> reactions distinct sort))
;; -> (2 3 5 7 11 13 17 19 23 29 31 37 41
43 47 53 59 61 67 71 73 79 83 89 97)
```

Gamma Chemical Programming

Reaction Rules on multisets of elements

Reaction is condition + action

Execution: replacement with result elements

Result is solution is “steady state”

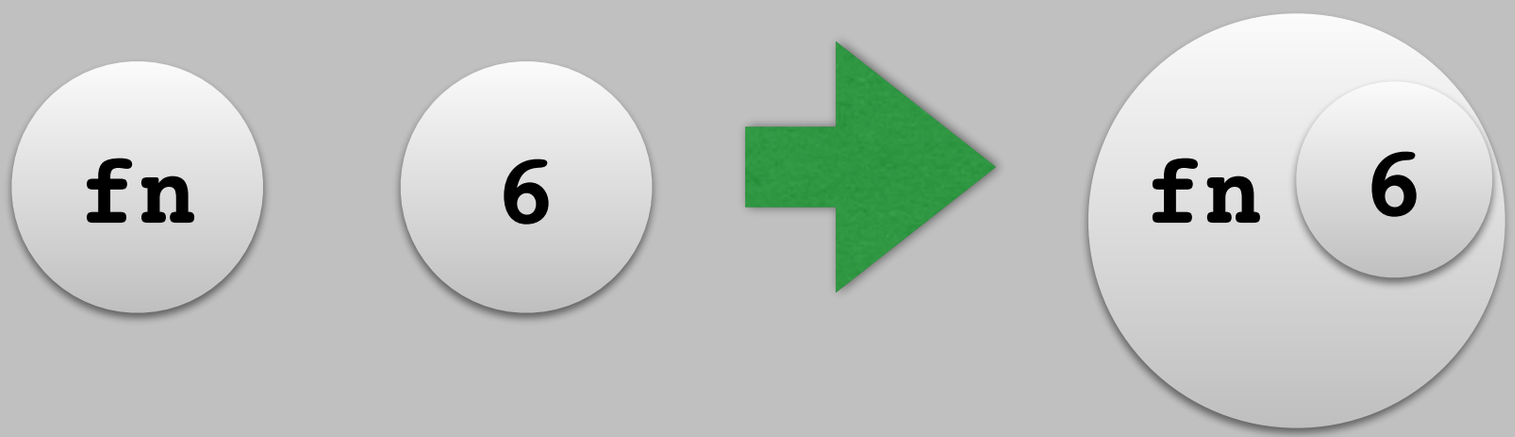
More Gamma Max

Gamma Demo in ClojureScript with core.async

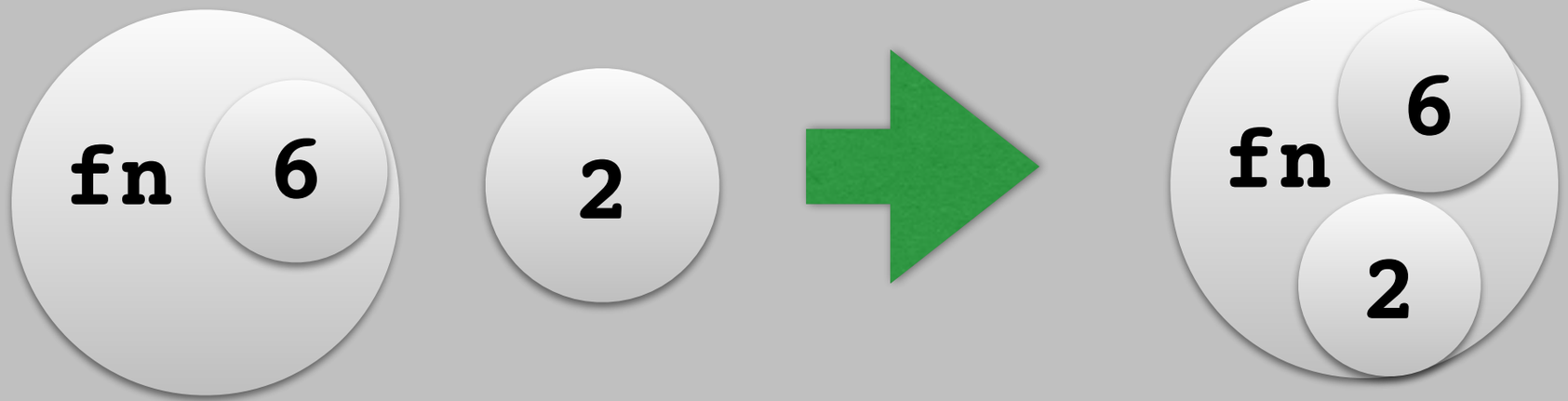
Higher Order

What if we made the reaction functions molecules too?

Higher Order



Higher Order



Higher Order

Hatching

Prime Reaction

```
(defn prime-reaction [a b]
  (if (and (> a b)
           (zero? (mod a b)))
    [(/ a b) b]
    [a b]))
```

Max Reaction

```
(defn max-reaction [a b]
  (if (> a b) [a a] [a b]))
```

Higher Order – Max

Hatching

Higher Order

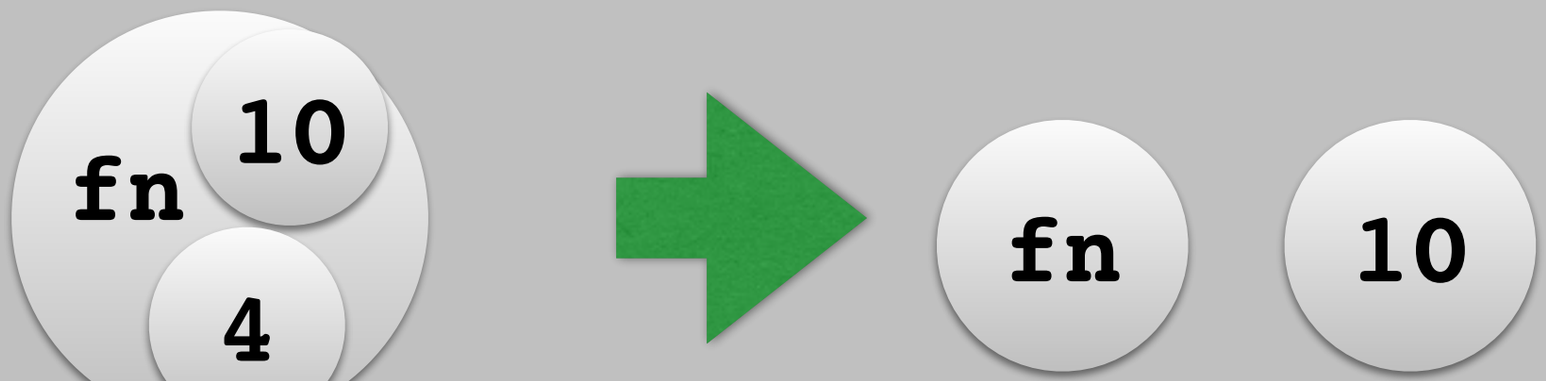
Can control the solution set size by the reaction function

Reducing Reaction Function

Max Reaction Reducing

```
(defn max-reaction-reducing [a b]
  (if (> a b) [a] [a b]))
```

Max Reducing



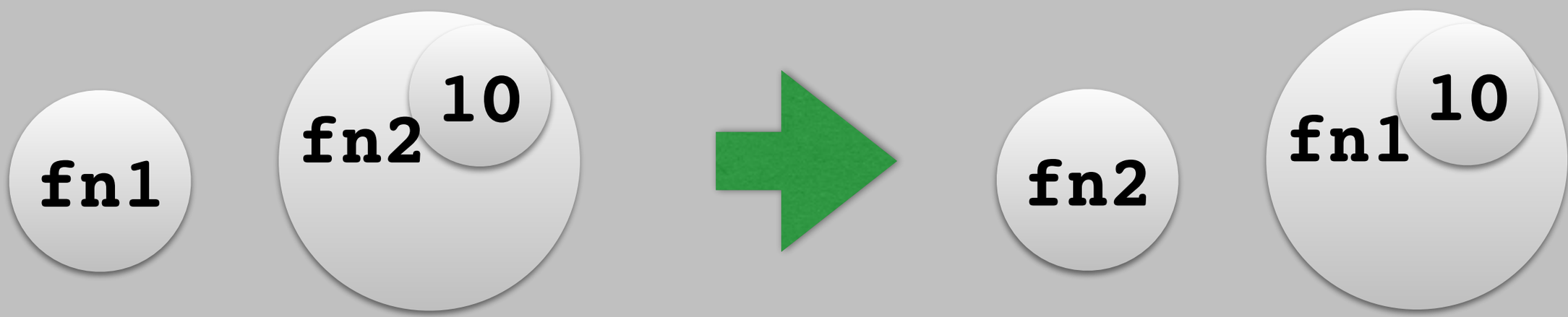
Hatching

Reducing Reactions

Need more “stirring” in our simulation

Allow fns to exchange captured vals

Higher Order



Capture Value Exchange Reaction

Higher Order Demo

in

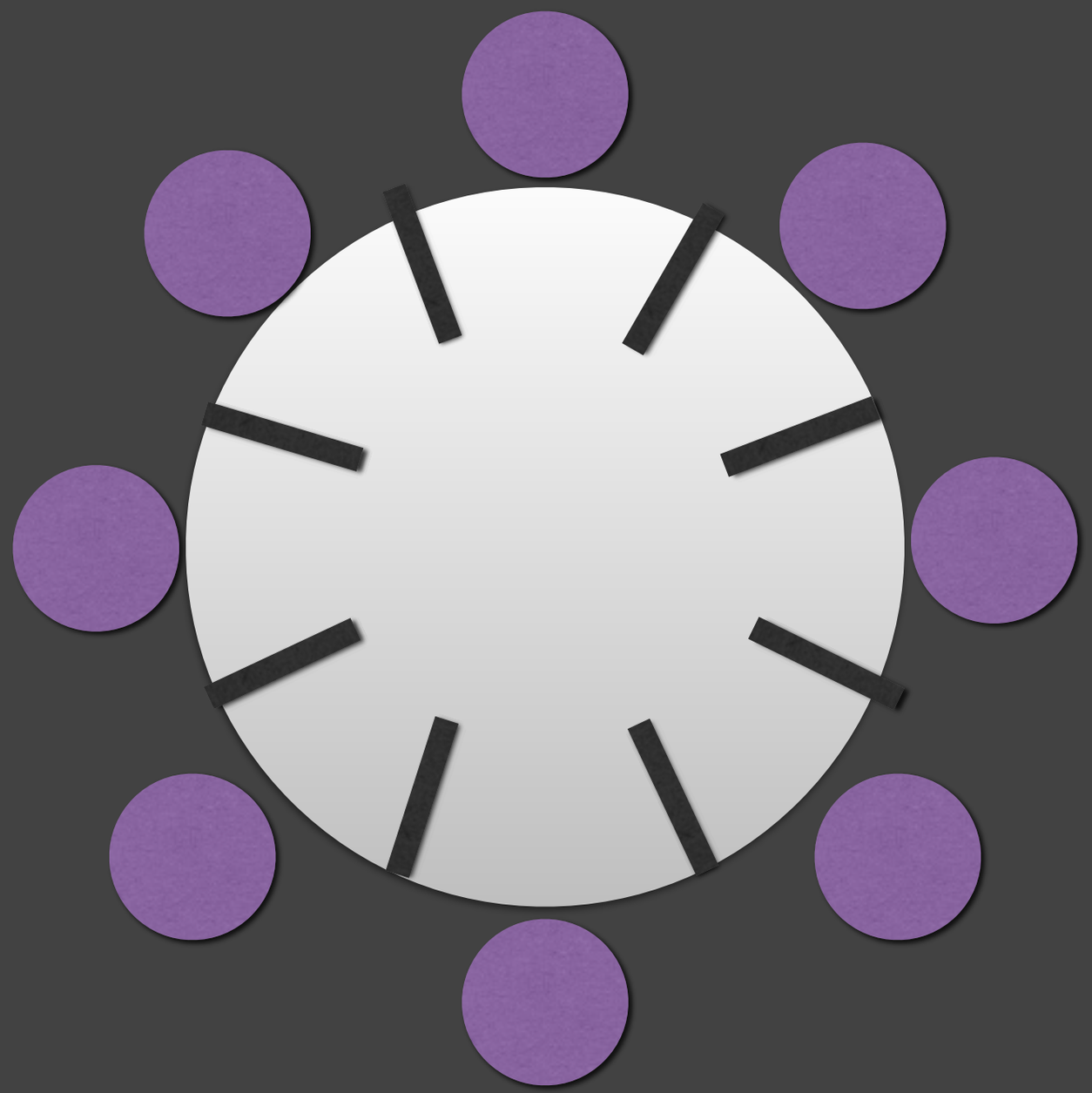
ClojureScript

Concurrency

No Sequential Processing

Doing things concurrently

Dining Philosophers



Dining Philosophers

Dining Philosophers

Dining Philosophers Demo in ClojureScript

Self Organizing

Simple behaviors combine to create systems

Self Organizing

Simple behaviors combine to create systems

Ant Colonies

Self Organizing

Simple behaviors combine to create systems

Ant Colonies

Bees

Self Organizing

Simple behaviors combine to create systems

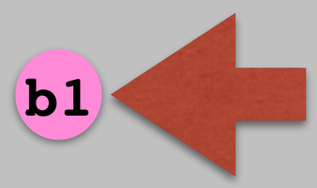
Ant Colonies

Bees

Mail Systems

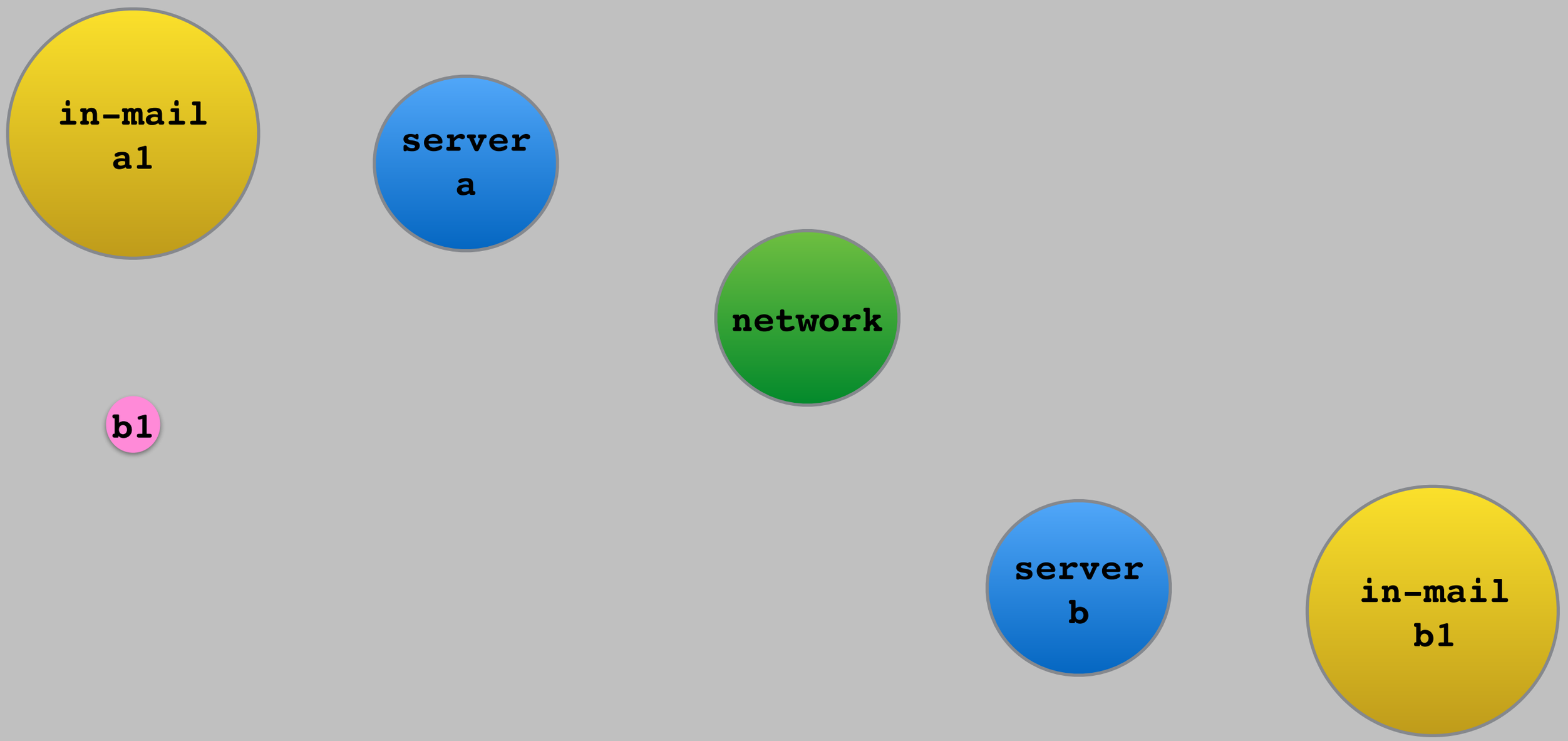
Mail System

Mail System

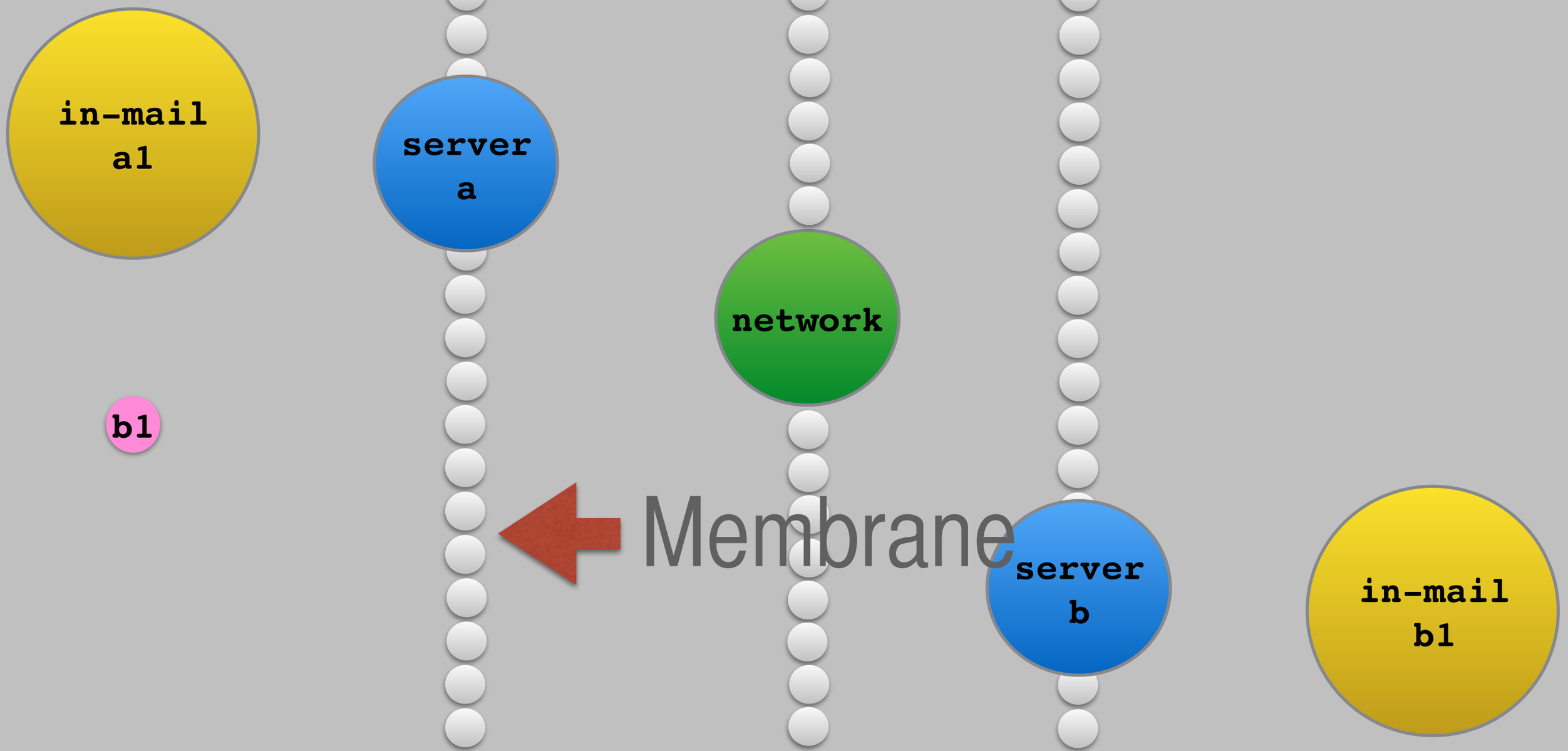


Mail Message

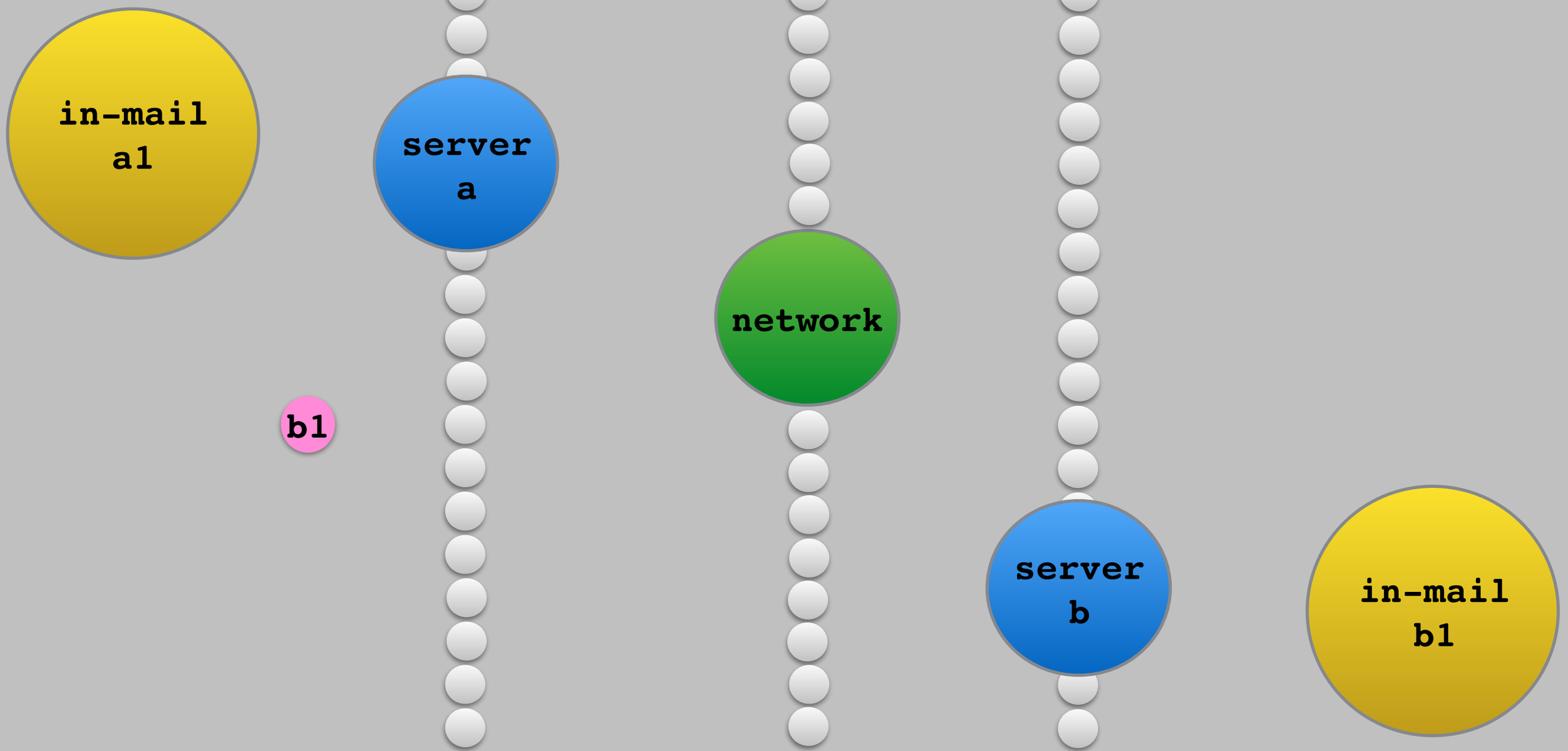
Mail System



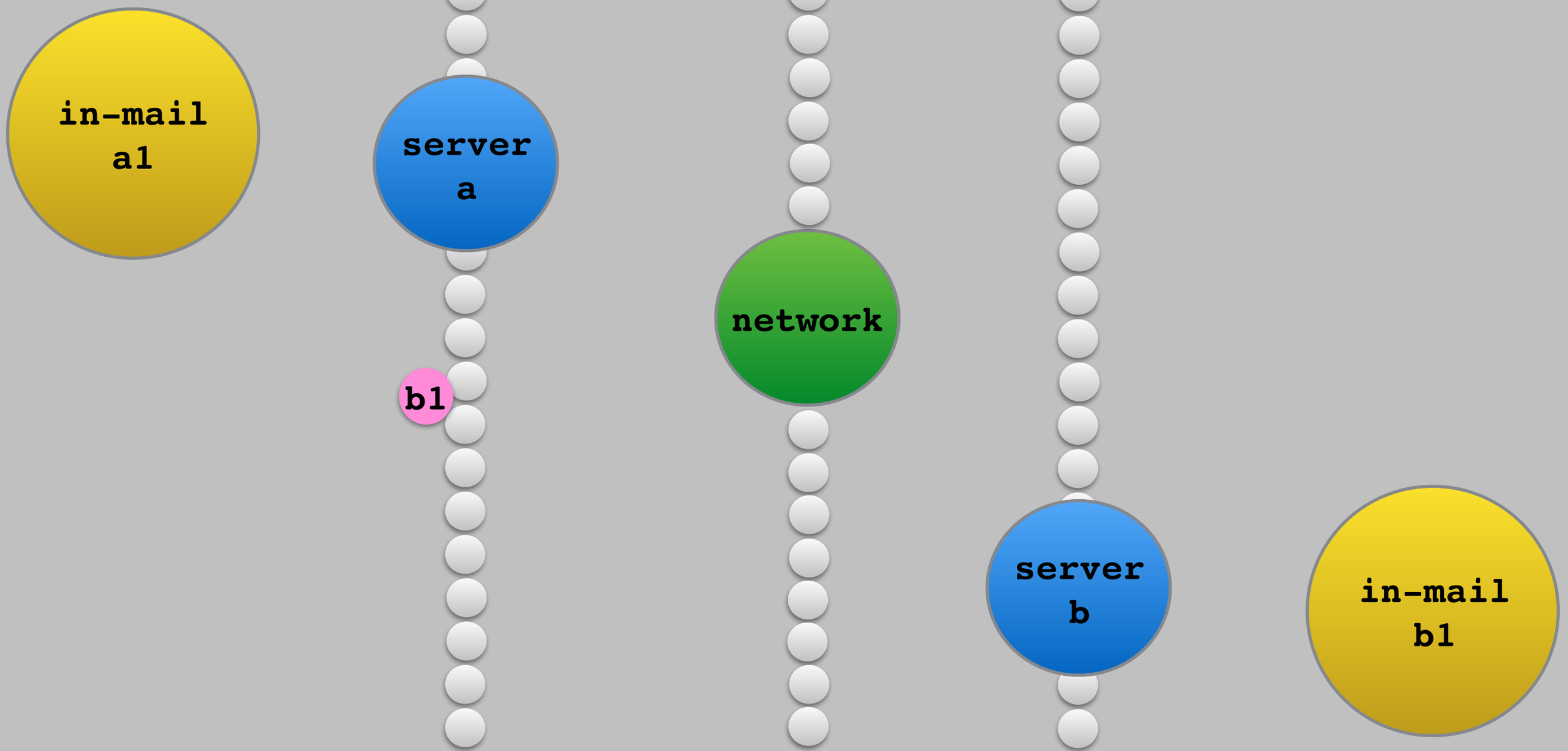
Mail System



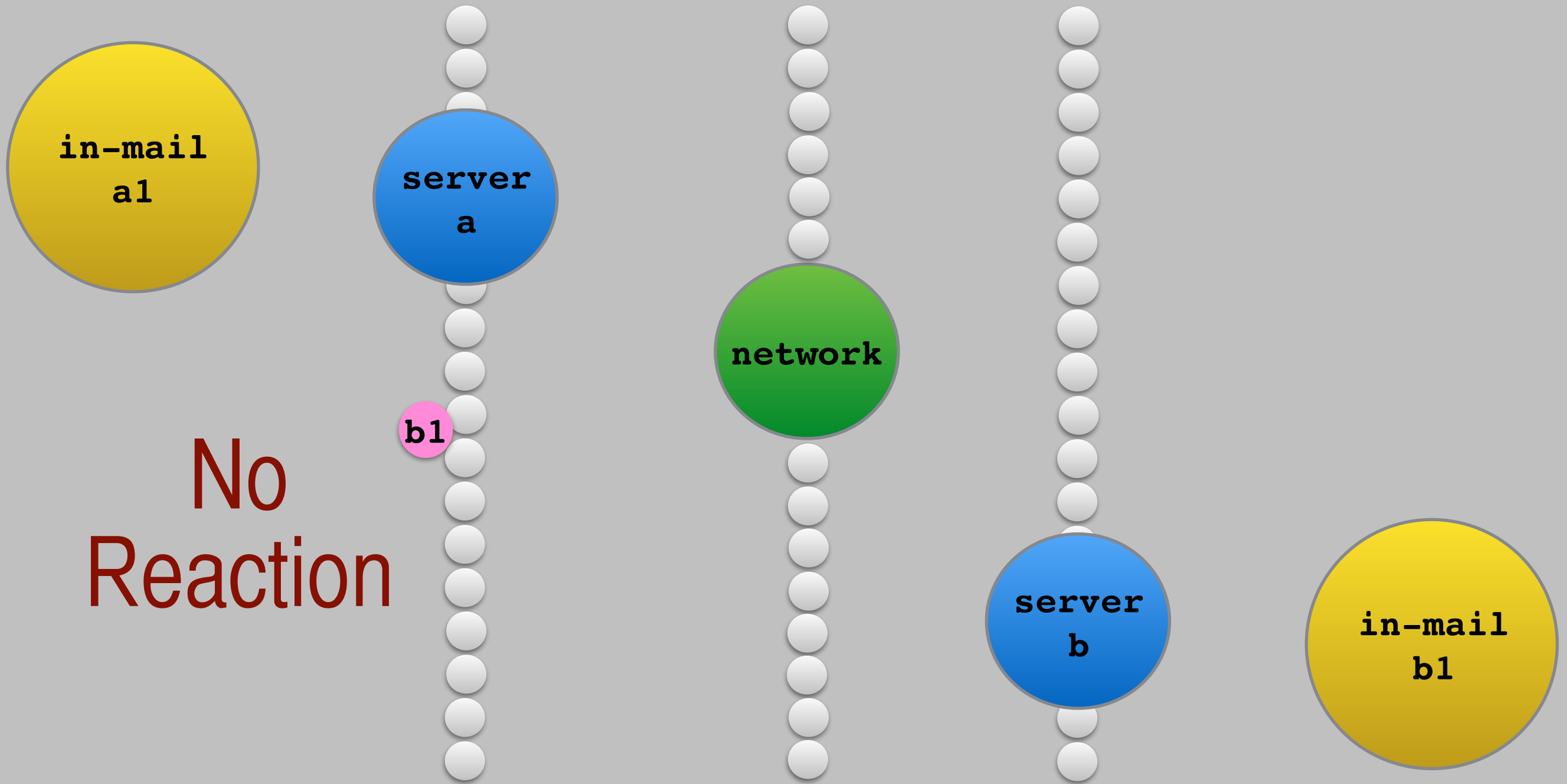
Mail System



Mail System

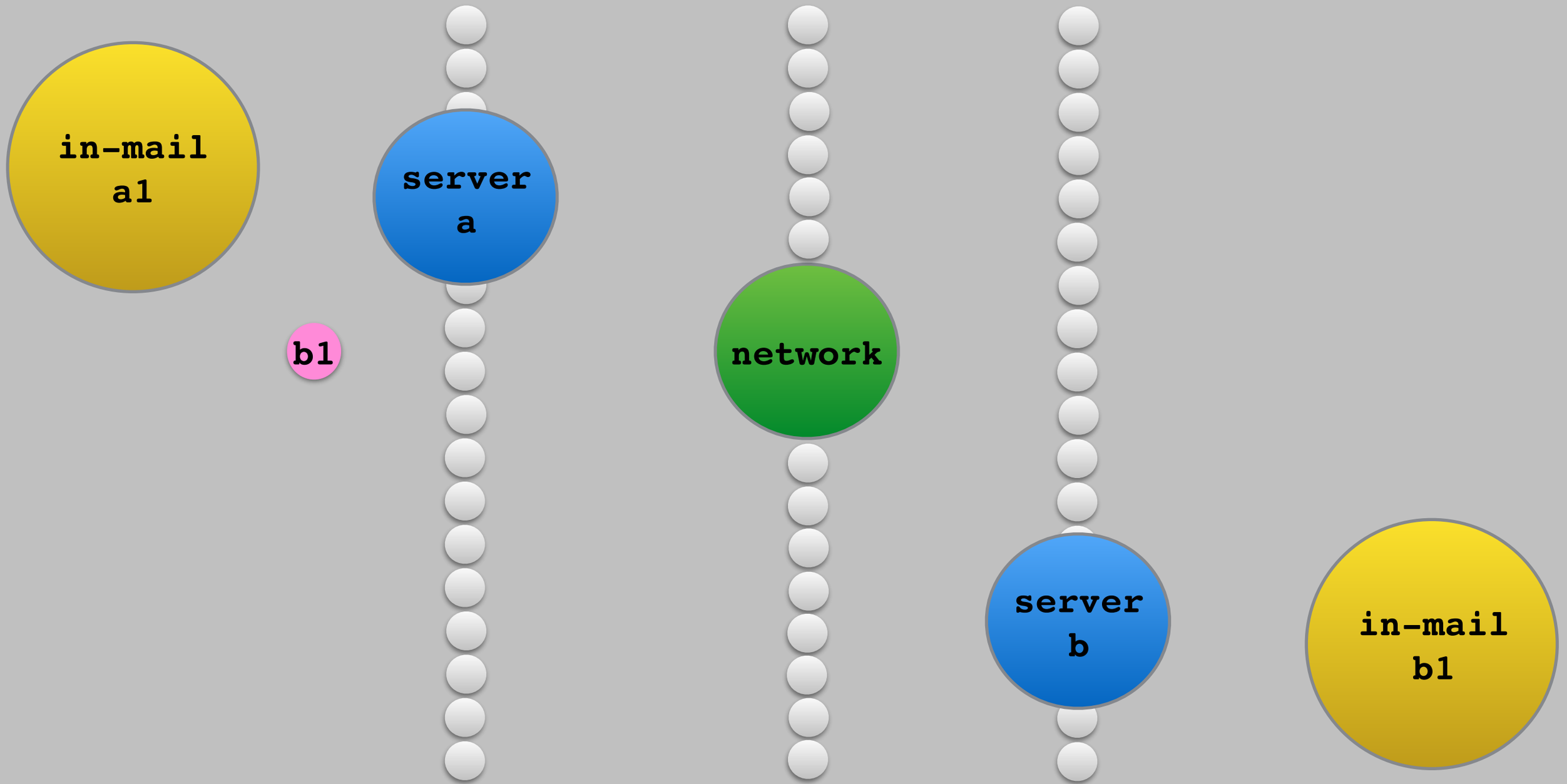


Mail System

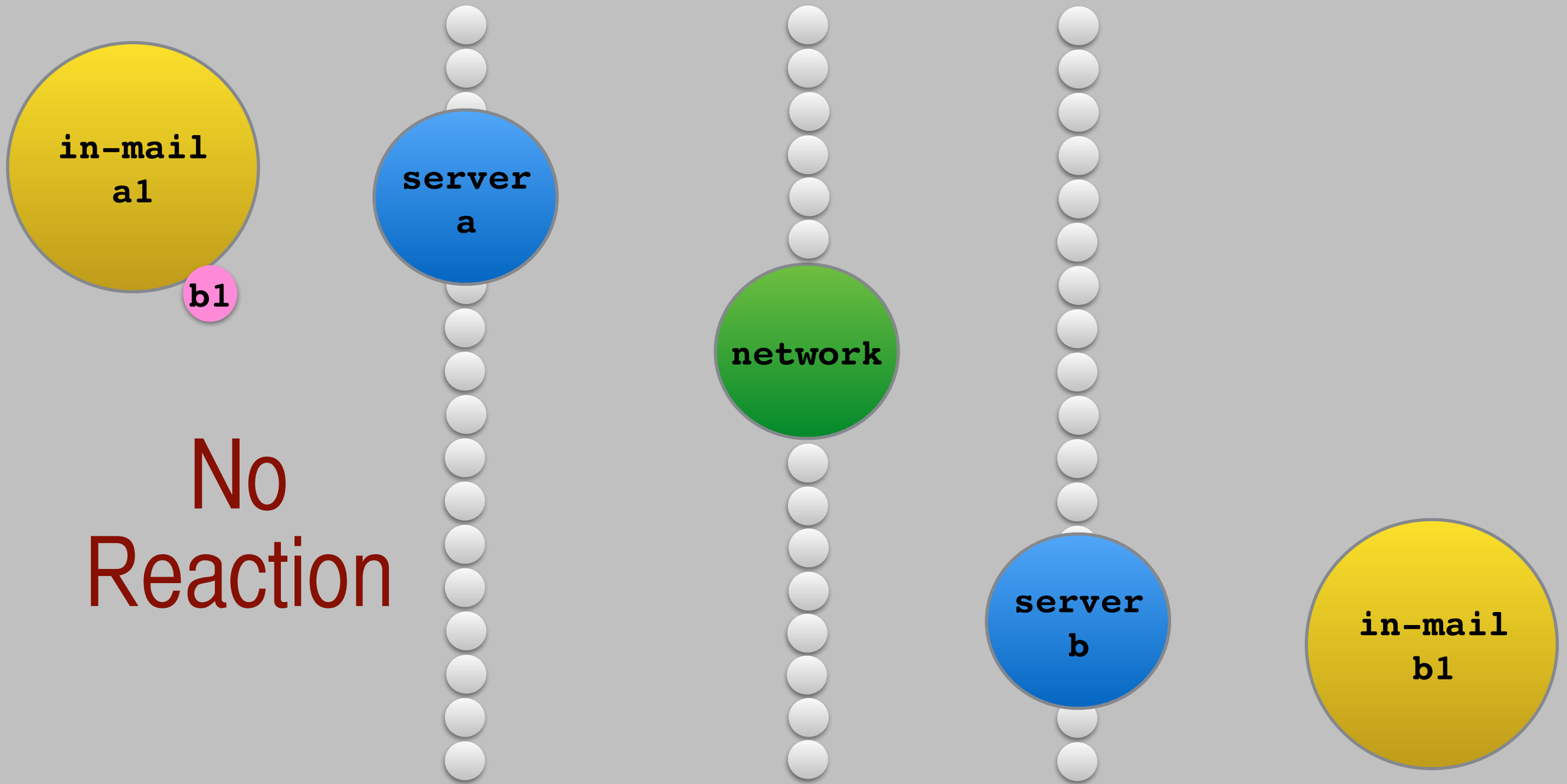


No
Reaction

Mail System

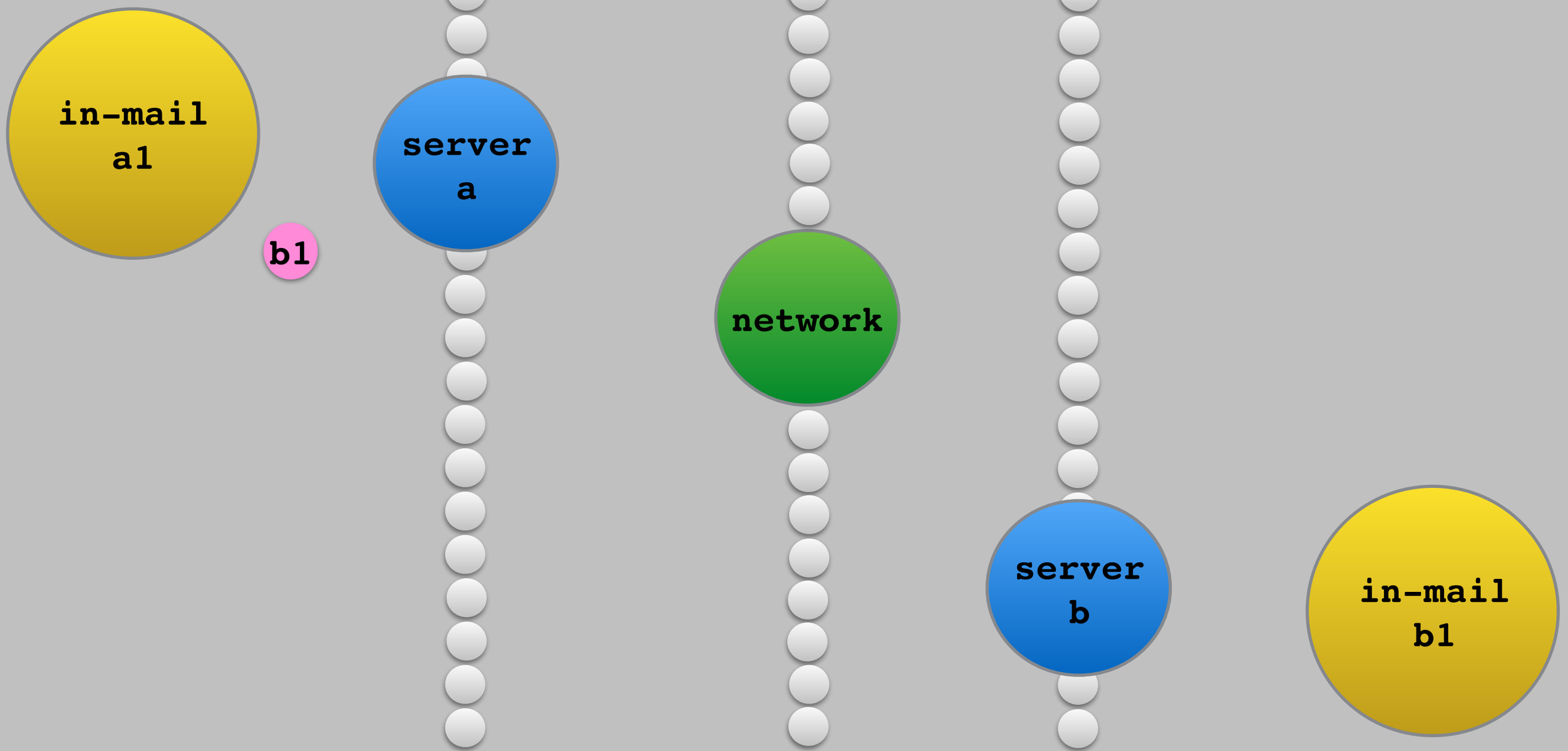


Mail System



No
Reaction

Mail System



Mail System

in-mail
a1

server
a

b1

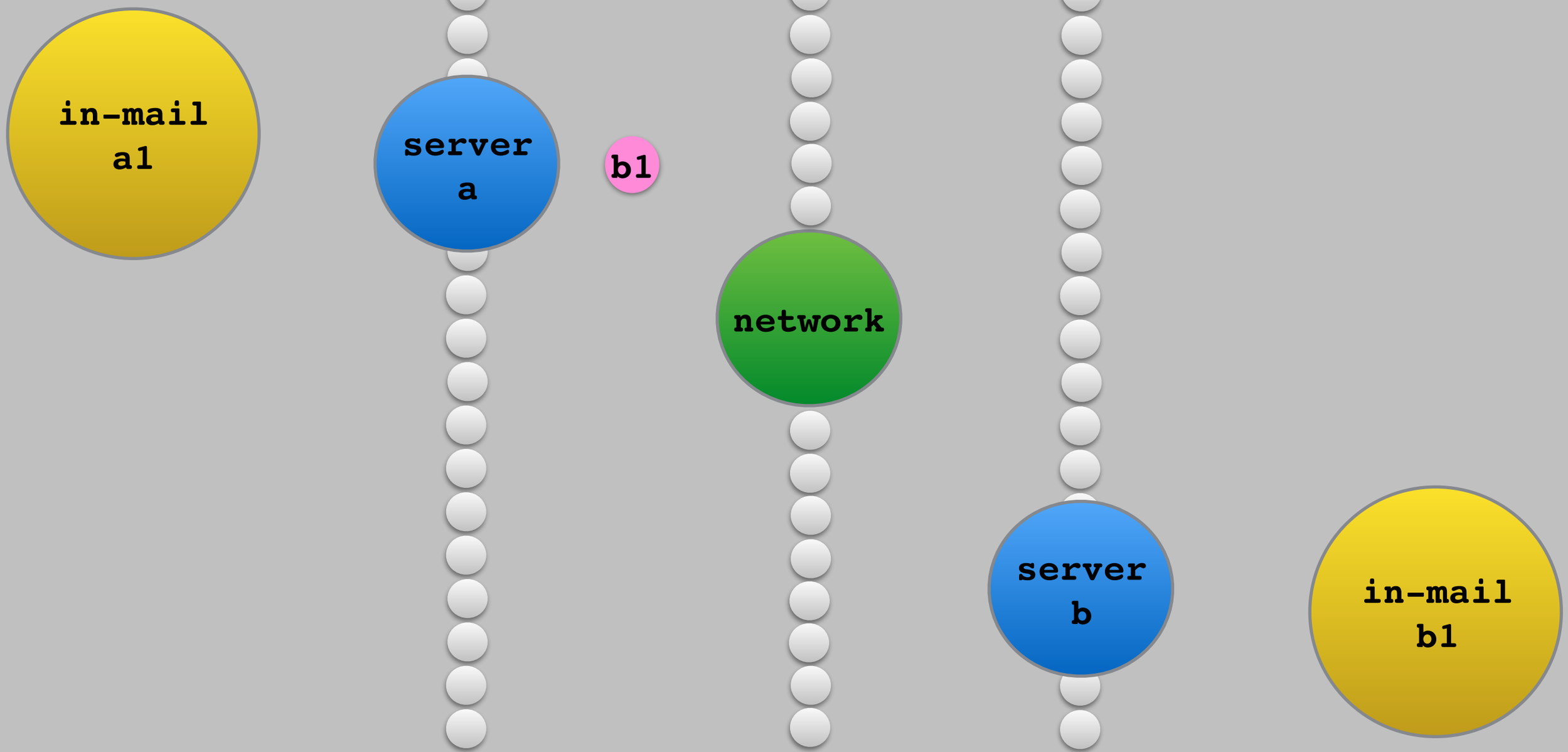
network

server
b

in-mail
b1

Reaction!

Mail System



Mail System

in-mail
a1

server
a

b1

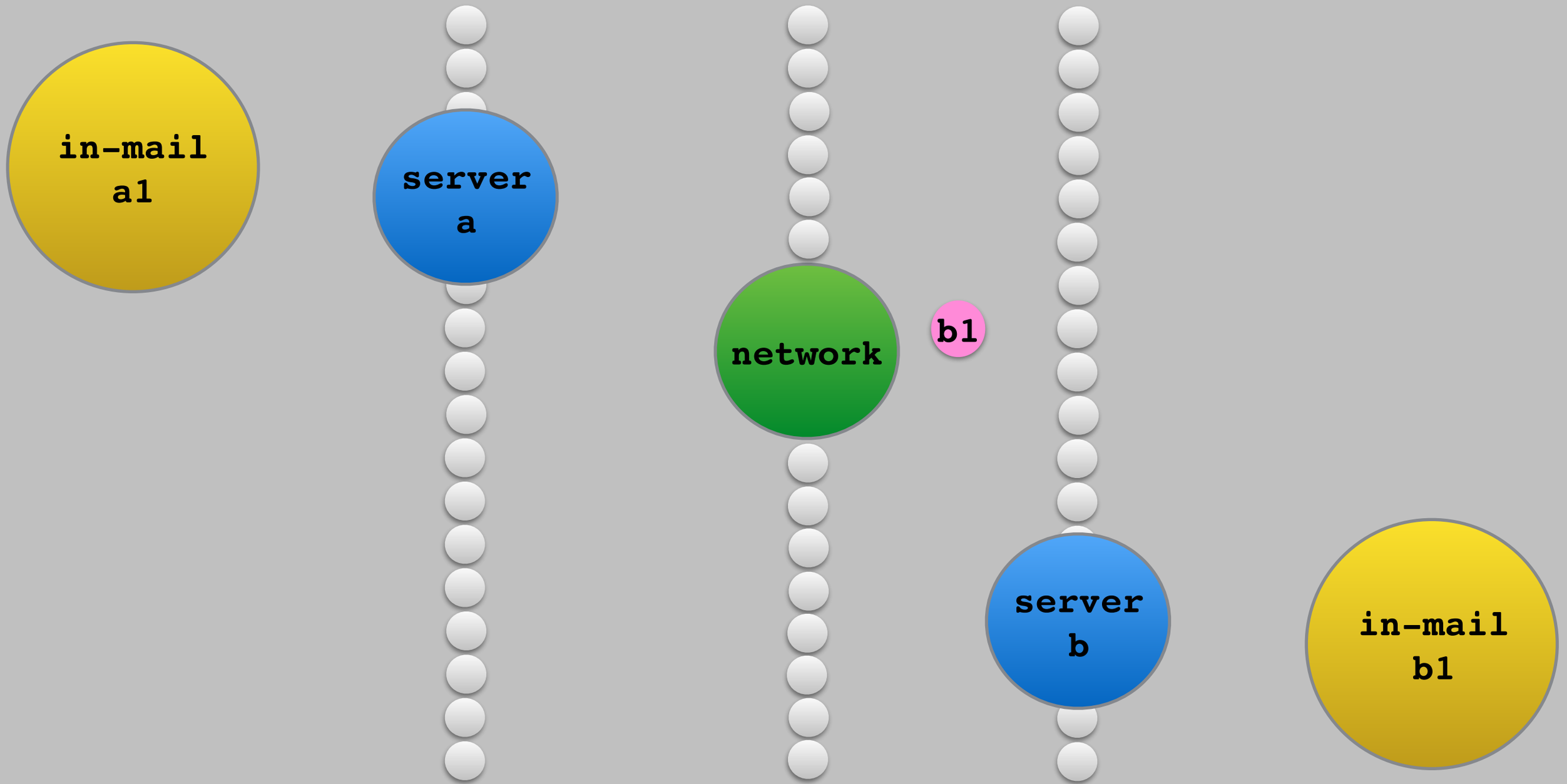
network

Reaction!

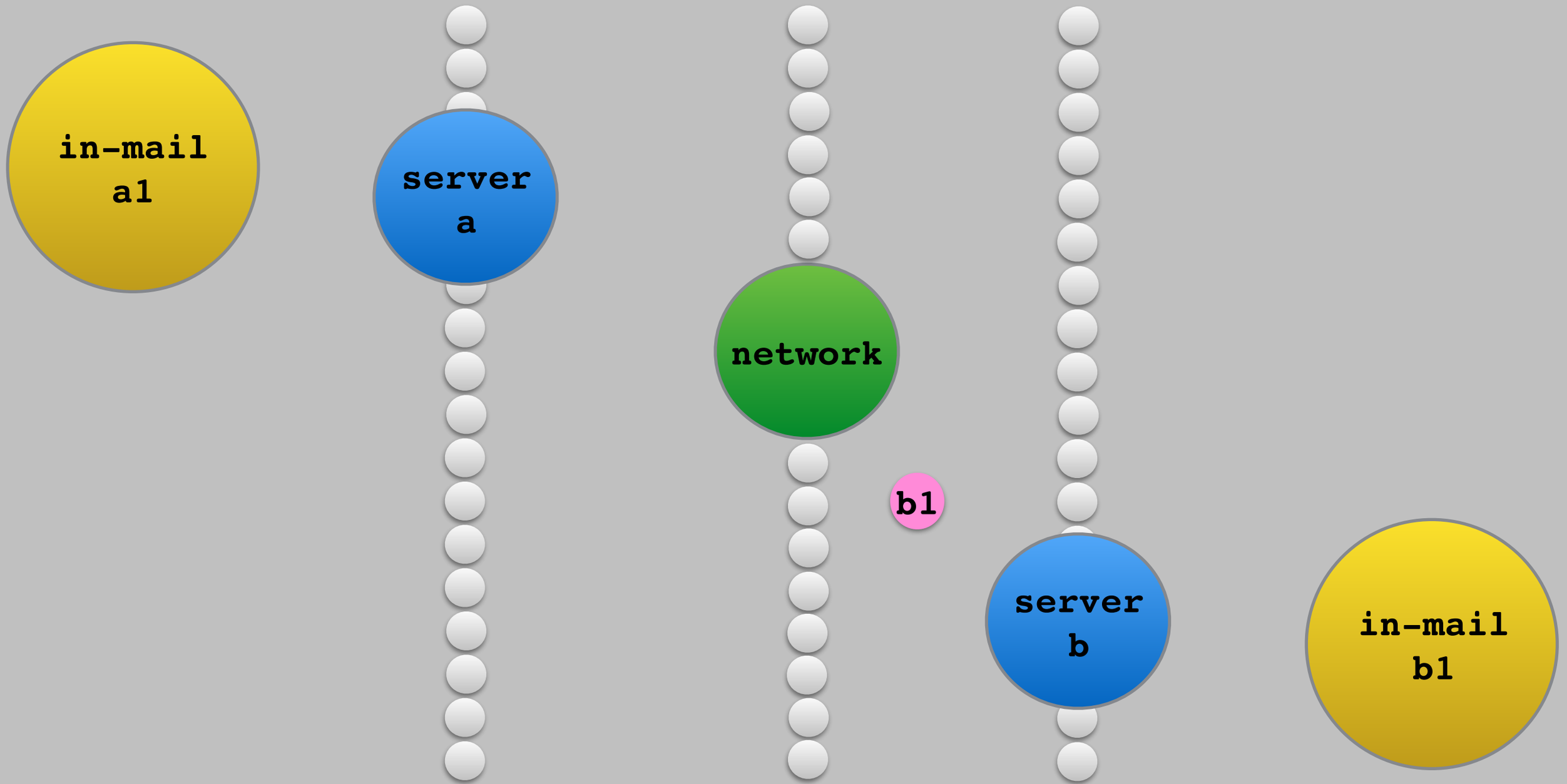
server
b

in-mail
b1

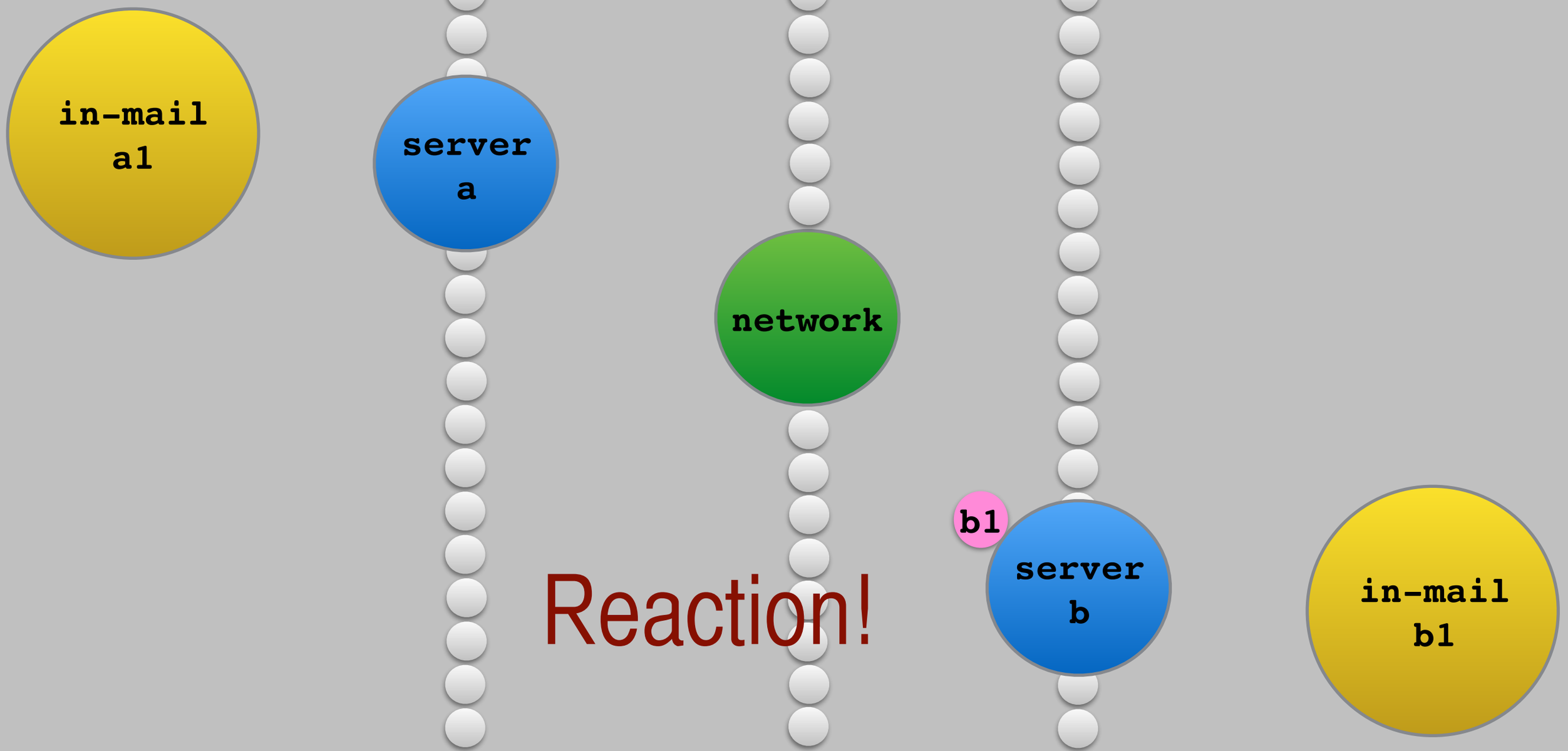
Mail System



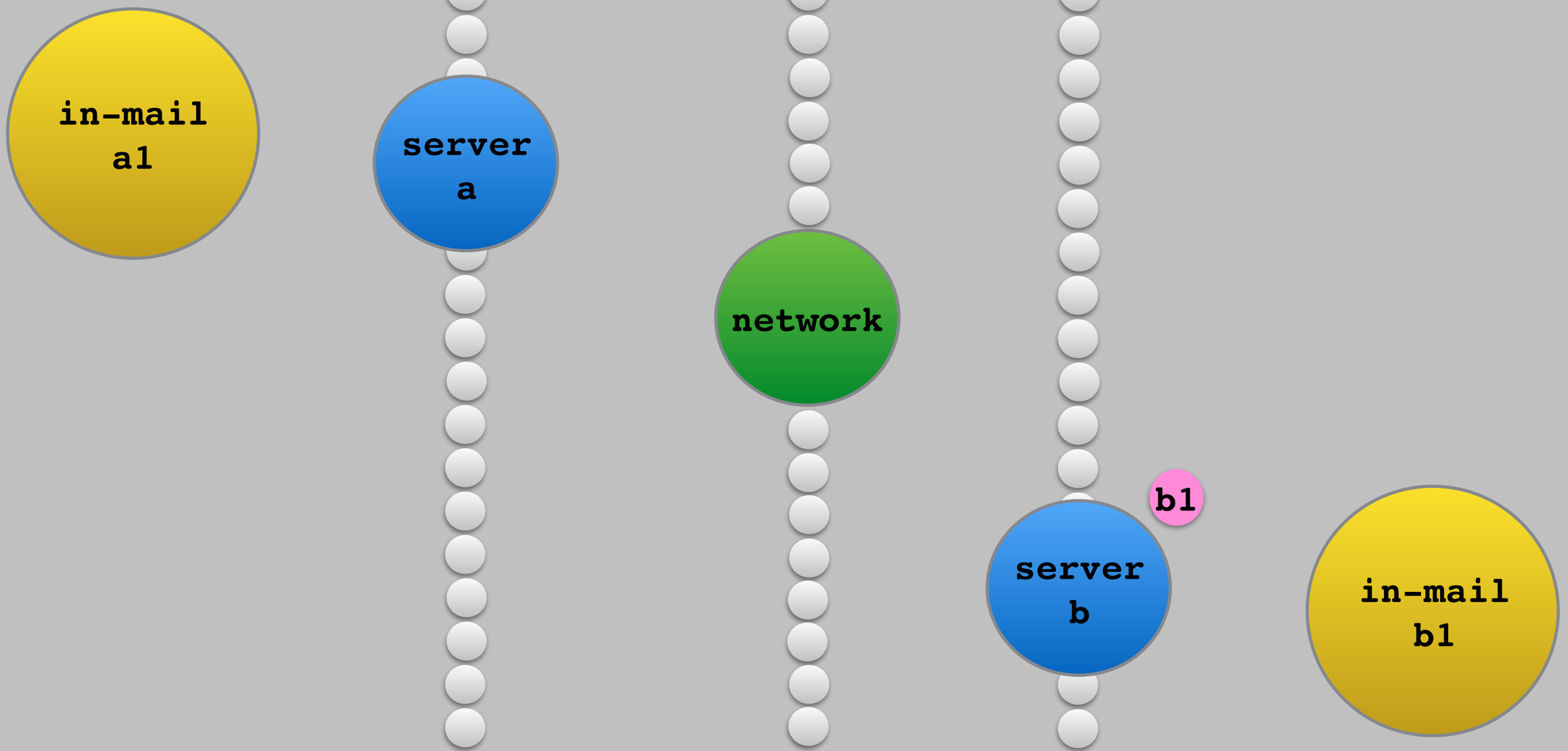
Mail System



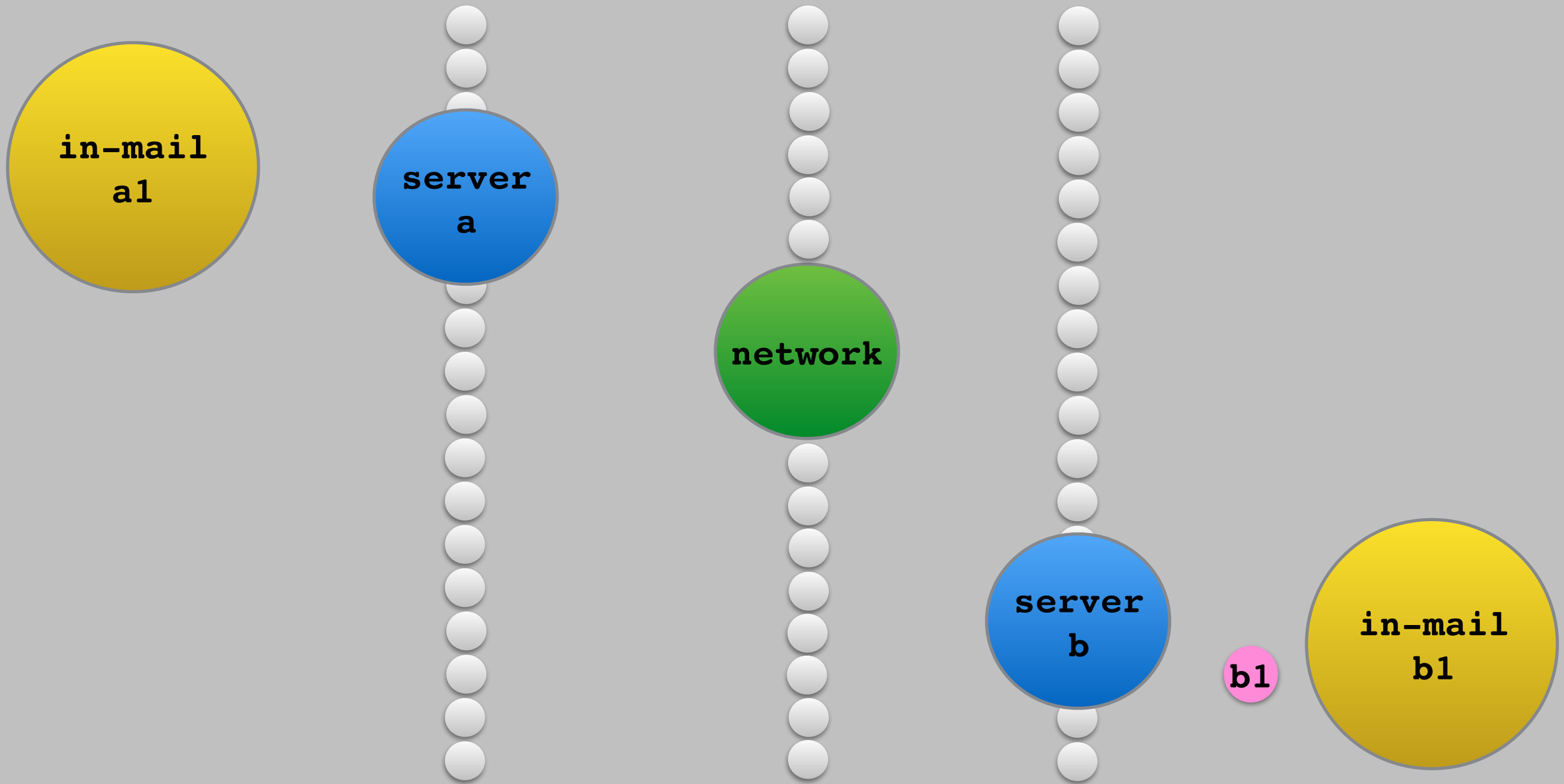
Mail System



Mail System



Mail System



Mail System

in-mail a1

server a

network

server b

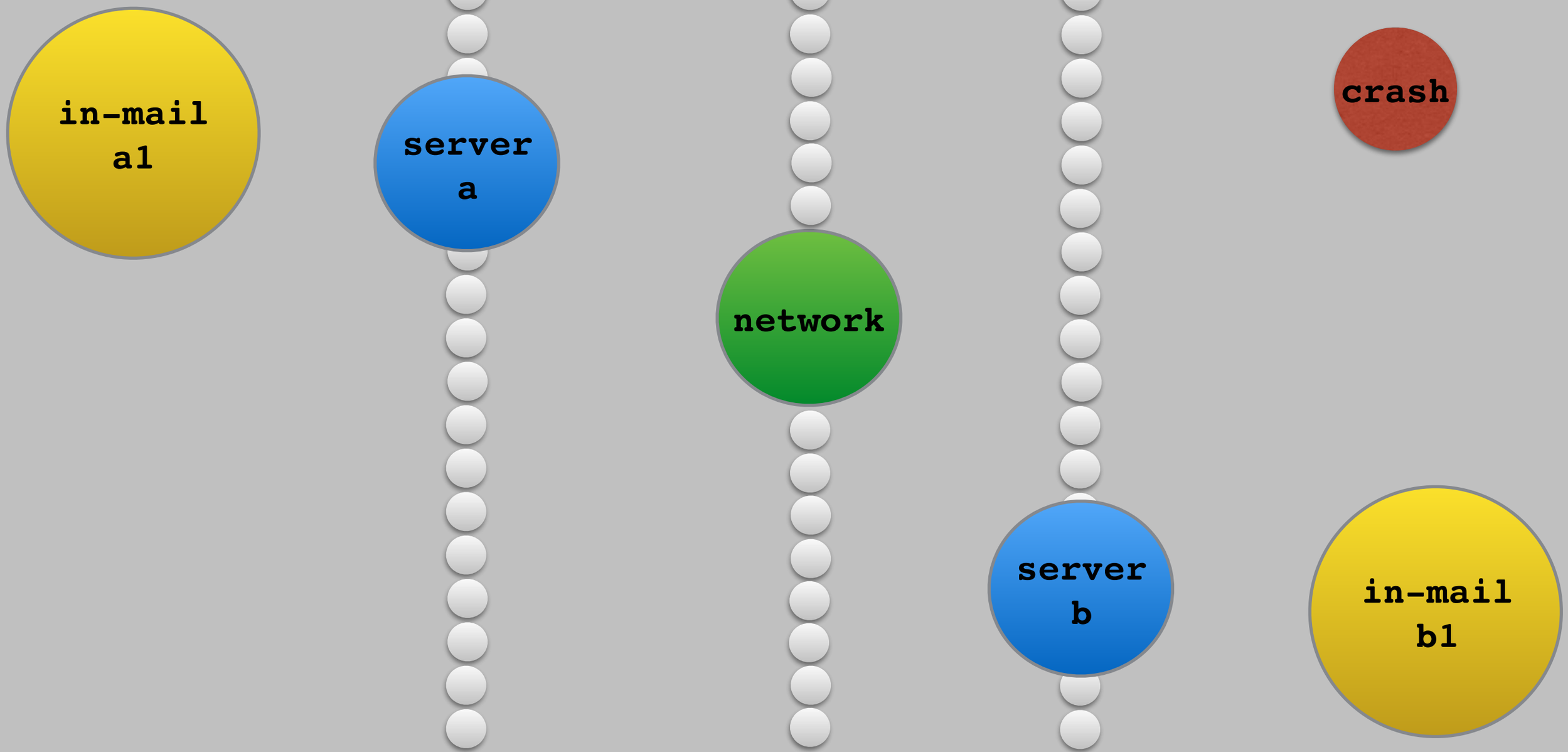
in-mail b1

Reaction!
Mail
Received

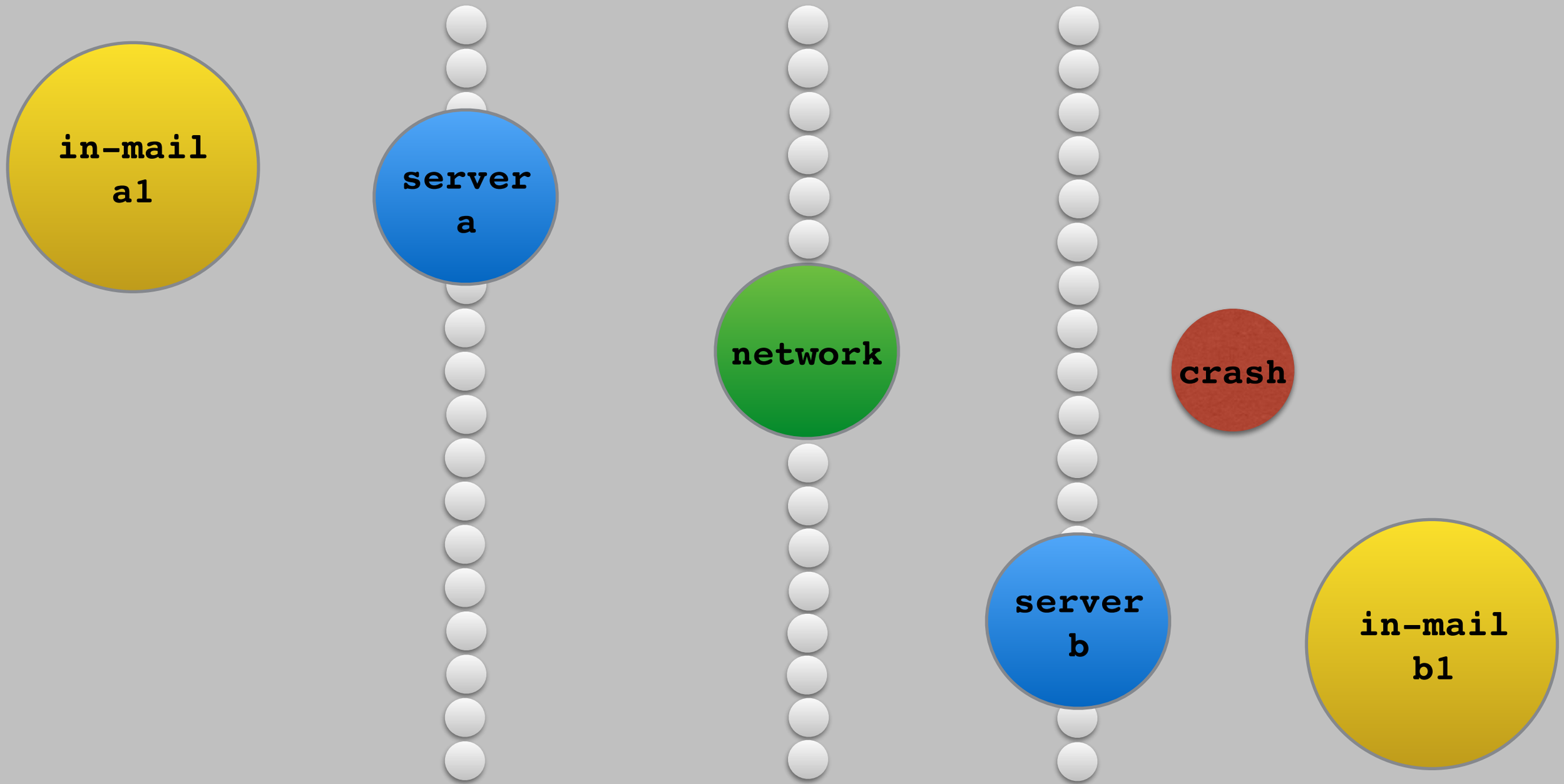
Self Healing

Simple behaviors for resilient systems

Mail System



Mail System



Mail System

in-mail a1

server a

network

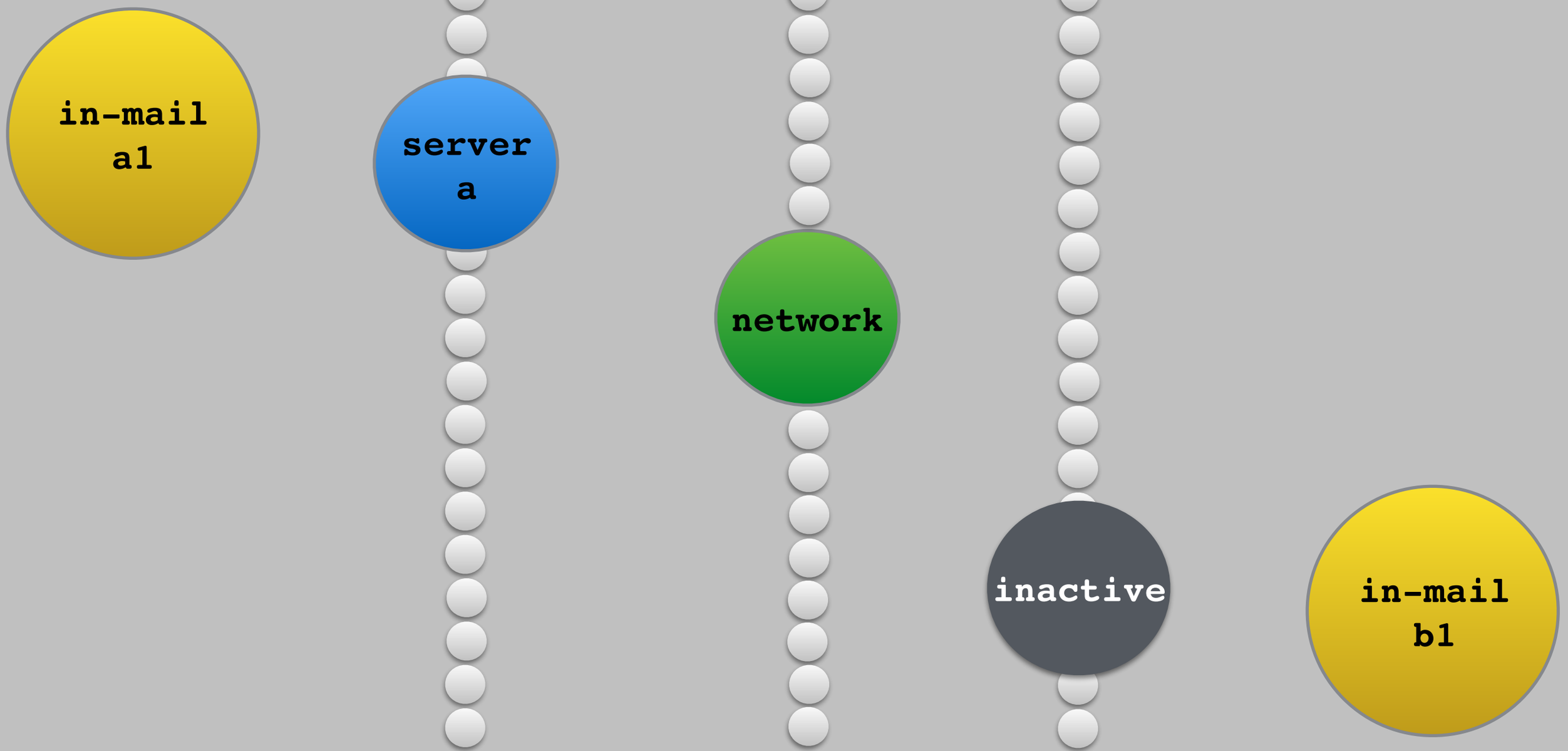
server b

crash

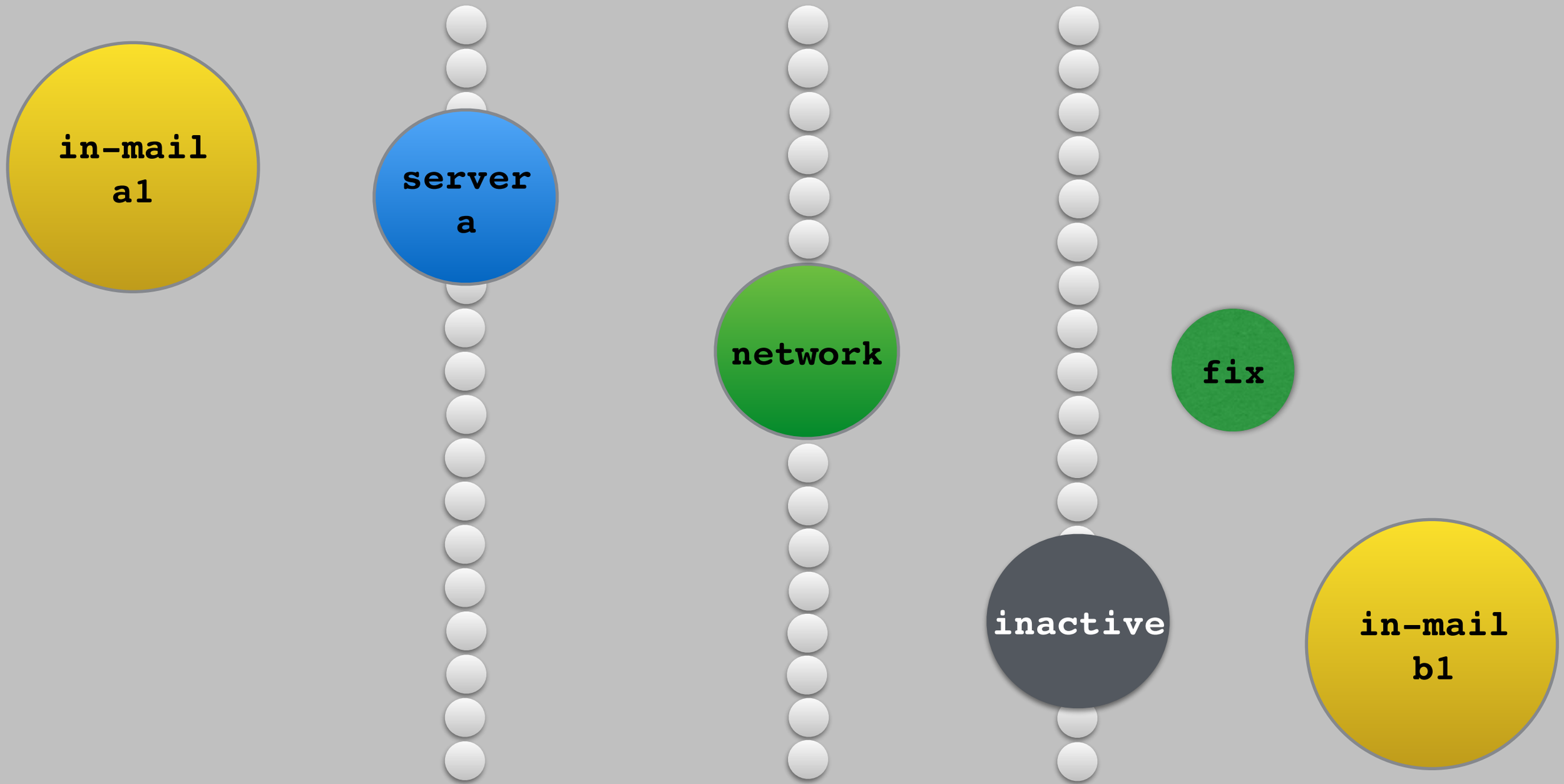
in-mail b1

Reaction Server Crash!

Mail System



Mail System



Mail System

in-mail a1

server a

network

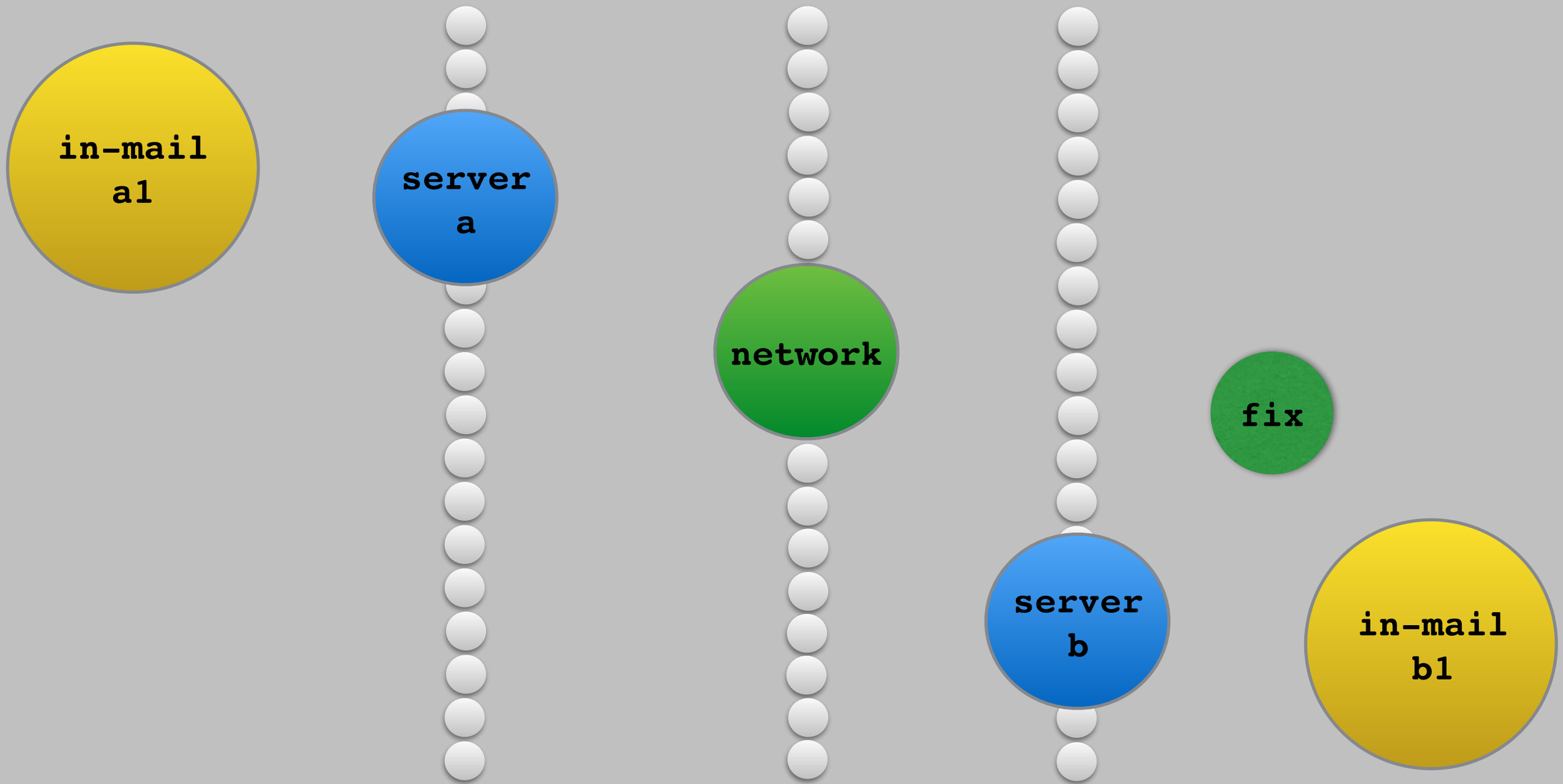
inactive

fix

in-mail b1

Reaction Server Repair!

Mail System



Mail System Demo

in

ClojureScript

Summary

Chemical Programming is about Reactions

Summary

Chemical Programming is about Reactions

Reaction Rules are simple and elegant

Summary

Chemical Programming is about Reactions

Reaction Rules are simple and elegant

Reactions eliminate *incidental sequentiality*

Summary

Chemical Programming is about Reactions

Reaction Rules are simple and elegant

Reactions eliminate *incidental sequentiality*

No sequentiality -> CONCURRENCY

Summary

Chemical Programming is about Reactions

Reaction Rules are simple and elegant

Reactions eliminate *incidental sequentiality*

No sequentiality -> CONCURRENCY

Simple behaviors can build robust systems

Summary

Nature knows what it is doing

Thank you!

github.com/gigasquid/chemical-computing