Distributed secure programming with Spritely Goblins

Christopher Lemmer Webber

https://dustycloud.org/

Fediverse: https://octodon.social/@cwebber

Birdsite: https://twitter.com/dustyweb

Spritely? Goblins?

Seeing is believing?

(well, using is even better but this is a talk)

TERMINAL PHASE DEMO GOES HERE

DISTRIBUTED CHAT DEMO GOES HERE

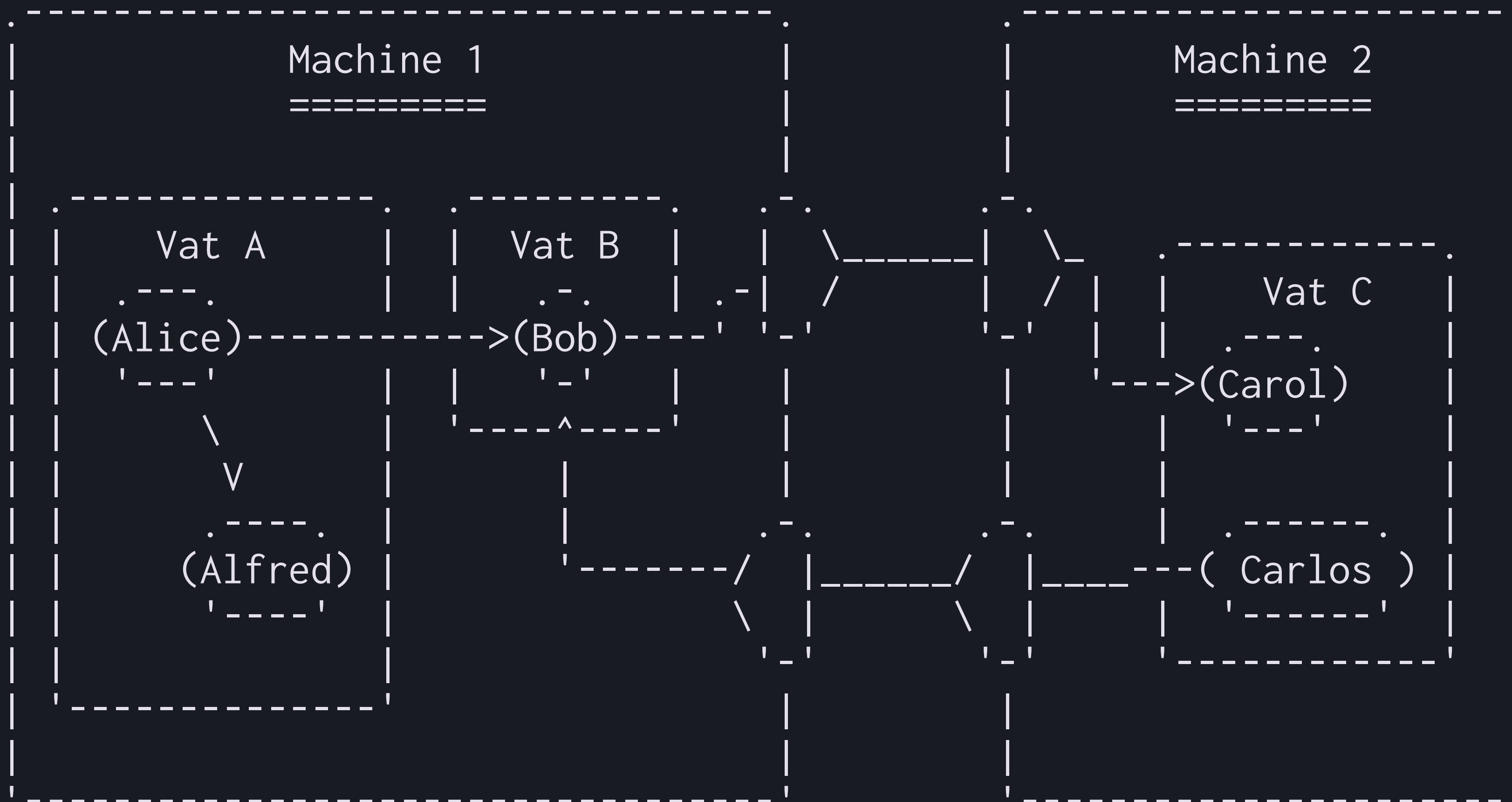
Distributed Programming Environment

Quasi-Functional State Management

Easy Transactions

Time Travel Included

Object Capability ("ocap") Security




```
(machine (vat (actormap {refr: (mactor object)})))
```

(Condensed) Goblins' Heritage

- **~193X:** λ
- **~1958:** (λ () ...)
- **~1972:** Smalltalk
- **~1973:** Actors
- **~1975:** Scheme
- **~1995:** Joule
- **~1997:** E

Carl Hewitt & co: the (classic) actor model

In response to a message, can:

```
.--------------------------------------------------------.
| Actors                         |               Goblins |
|--------------------------------+-----------------------|
|                                |                       |
|                                |                       |
|                                |                       |
'--------------------------------------------------------'
```

Carl Hewitt & co: the (classic) actor model

In response to a message, can:

Actors	Goblins
Send messages / invoke	

Carl Hewitt & co: the (classic) actor model

In response to a message, can:

Actors	Goblins
Send messages / invoke	(<- ...) or ($ ...)

Carl Hewitt & co: the (classic) actor model

In response to a message, can:

```
.---------------------------------------------.
| Actors                  |           Goblins |
|-------------------------+-------------------|
| Send messages / invoke  | (<- ...) or ($ ...) |
| Create actors           |                   |
|                         |                   |
'---------------------------------------------'
```

Carl Hewitt & co: the (classic) actor model

In response to a message, can:

```
.-----------------------------------------------.
| Actors                    |           Goblins |
|---------------------------+-------------------|
| Send messages / invoke    | (<- ...) or ($ ...) |
| Create actors             |        (spawn ...) |
|                           |                   |
'-----------------------------------------------'
```

Carl Hewitt & co: the (classic) actor model

In response to a message, can:

Actors	Goblins
Send messages / invoke	(<- ...) or ($ ...)
Create actors	(spawn ...)
Designate next behavior	

Carl Hewitt & co: the (classic) actor model

In response to a message, can:

Actors	Goblins
Send messages / invoke	(<- ...) or ($ ...)
Create actors	(spawn ...)
Designate next behavior	(bcom ...)

Carl Hewitt & co: the (classic) actor model

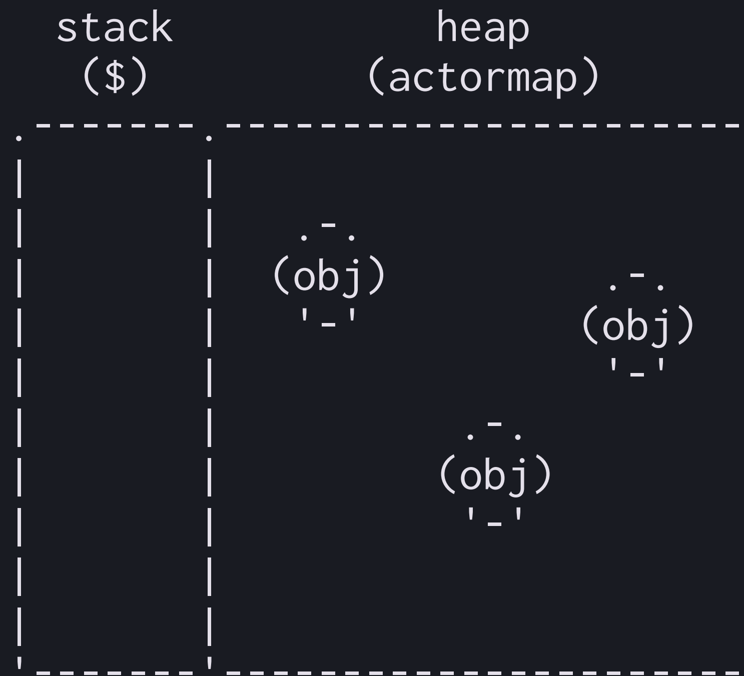
In response to a message, can:

```
.-----------------------------------------------.
| Actors                    |           Goblins |
|---------------------------+-------------------|
| Send messages / invoke    | (<- ...) or ($ ...) | WTF, why two?!
| Create actors             |       (spawn ...) |
| Designate next behavior   |        (bcom ...) |
'-----------------------------------------------'
```

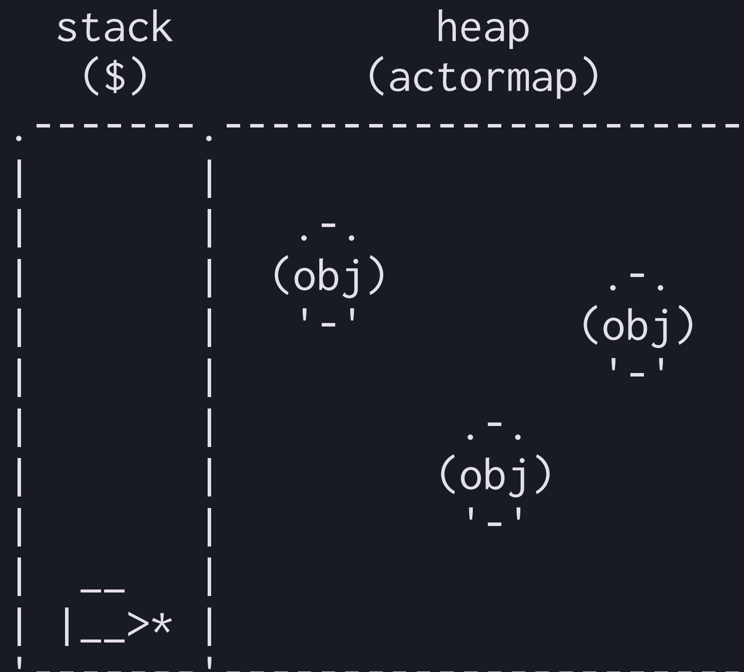
The Synchronous Call-Return Worldview

- **~193X:** λ
- **~1958:** (λ () ...)
- **~1972:** Smalltalk
- **~1973:** Actors
- **~1975:** Scheme
- **~1995:** Joule
- **~1997:** E (... kinda)

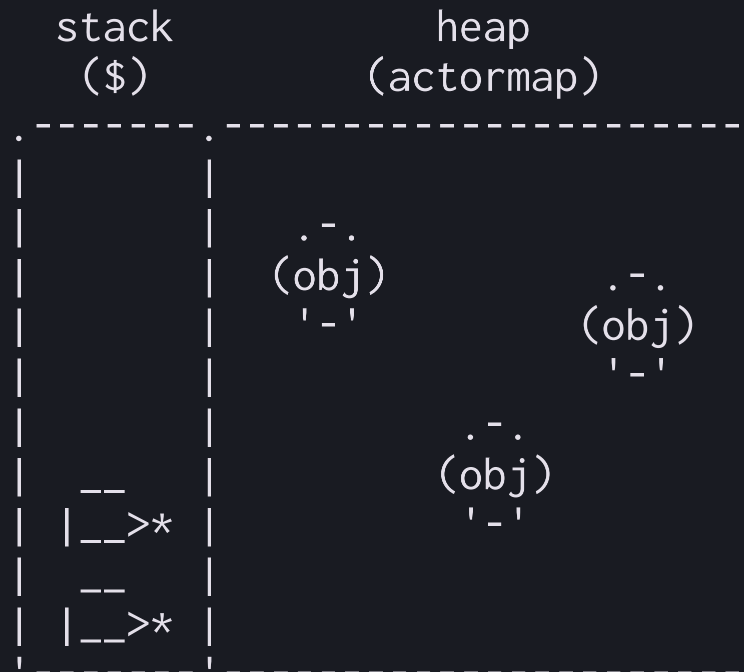
The Synchronous Call-Return Worldview



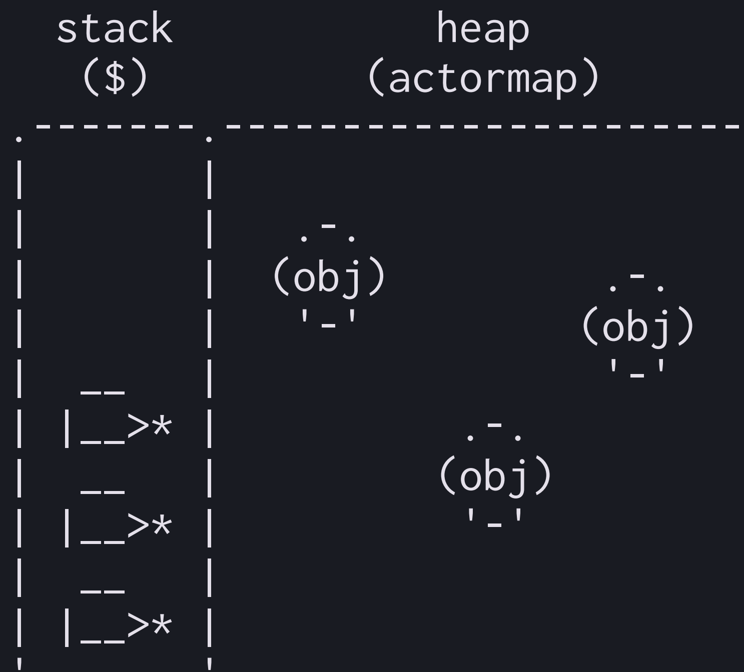
The Synchronous Call-Return Worldview



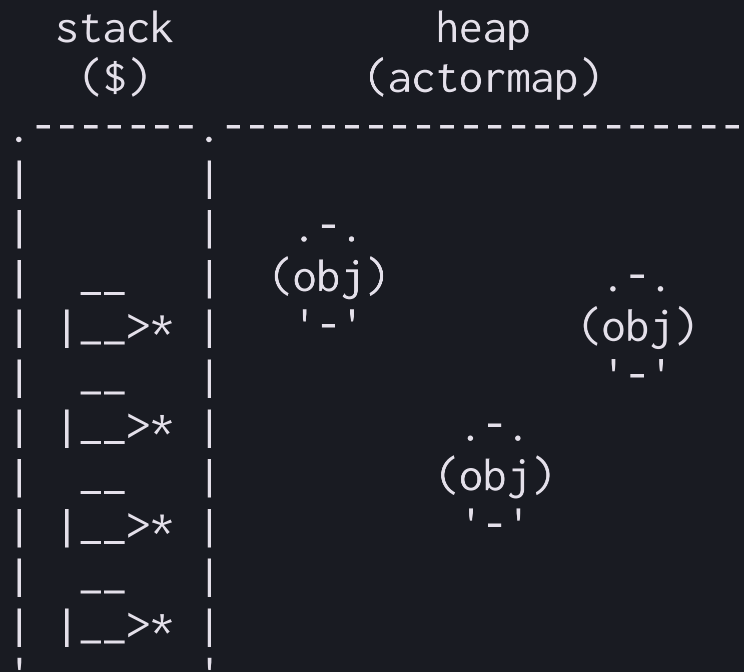
The Synchronous Call-Return Worldview



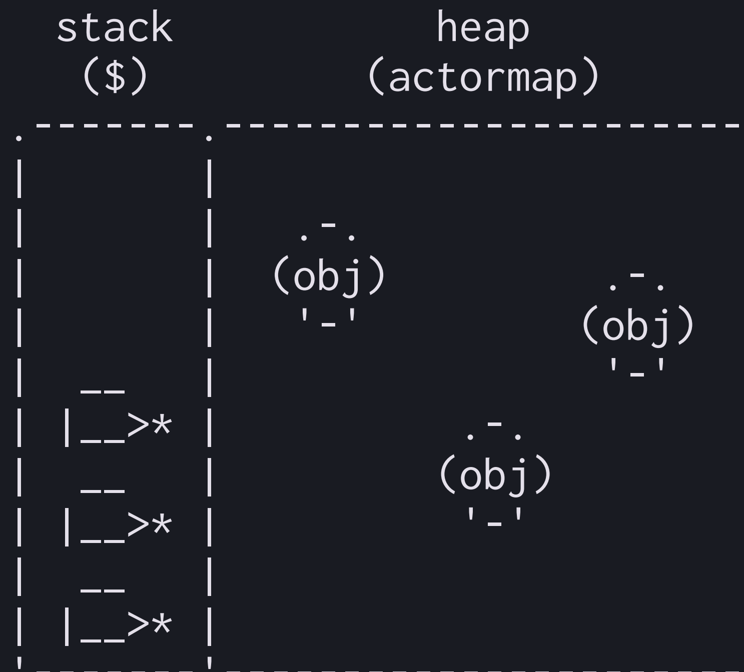
The Synchronous Call-Return Worldview



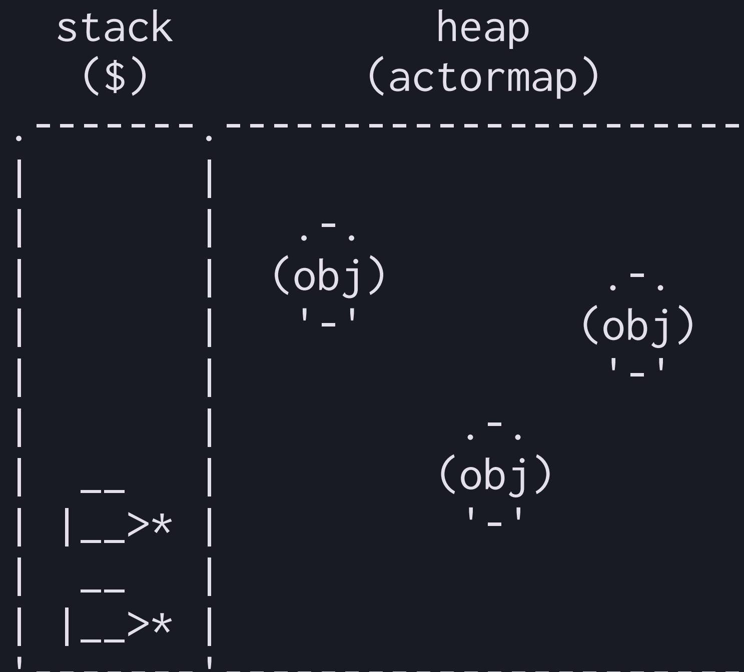
The Synchronous Call-Return Worldview



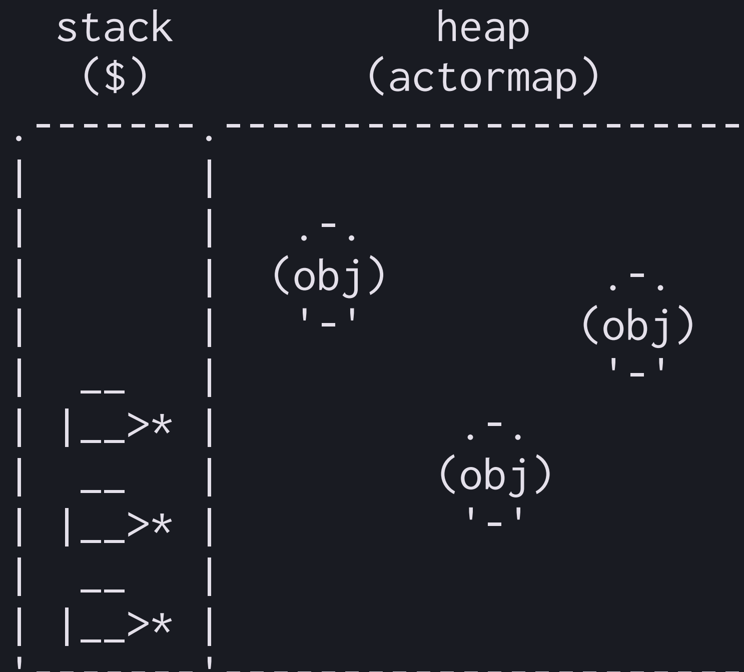
The Synchronous Call-Return Worldview



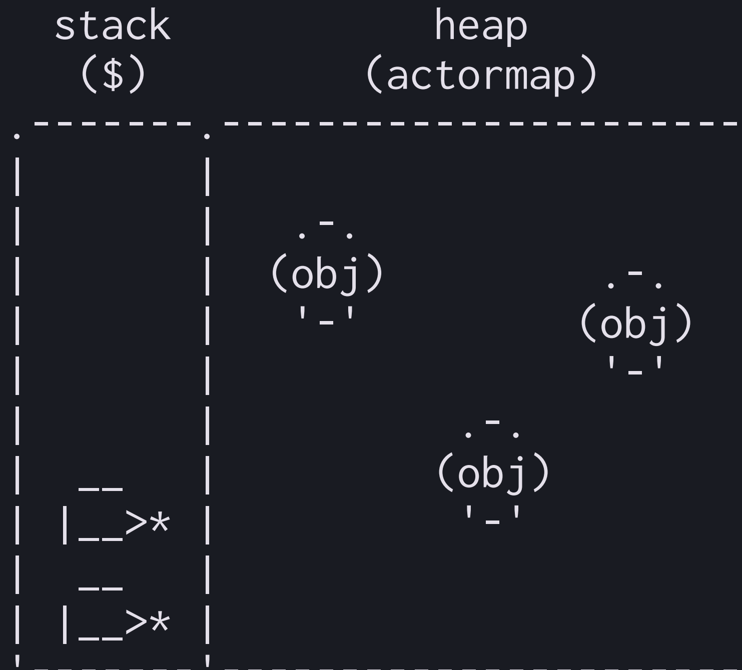
The Synchronous Call-Return Worldview



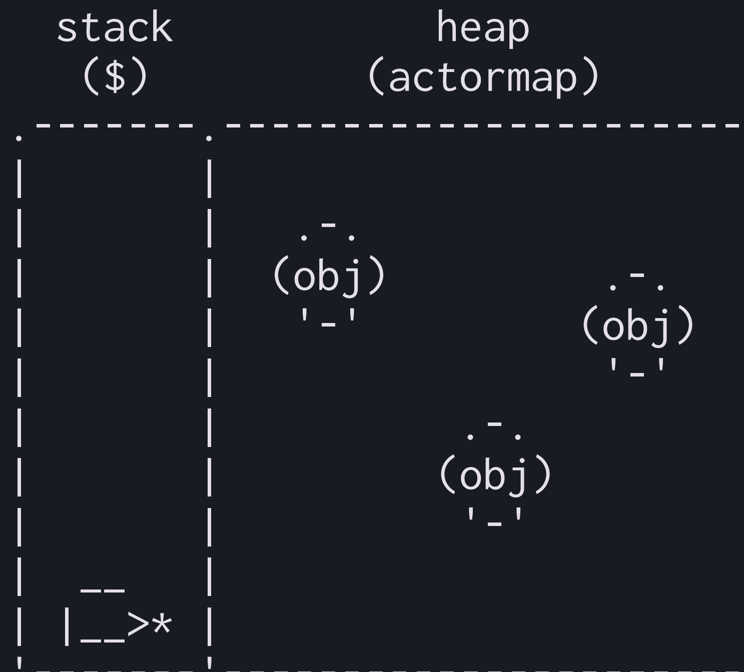
The Synchronous Call-Return Worldview



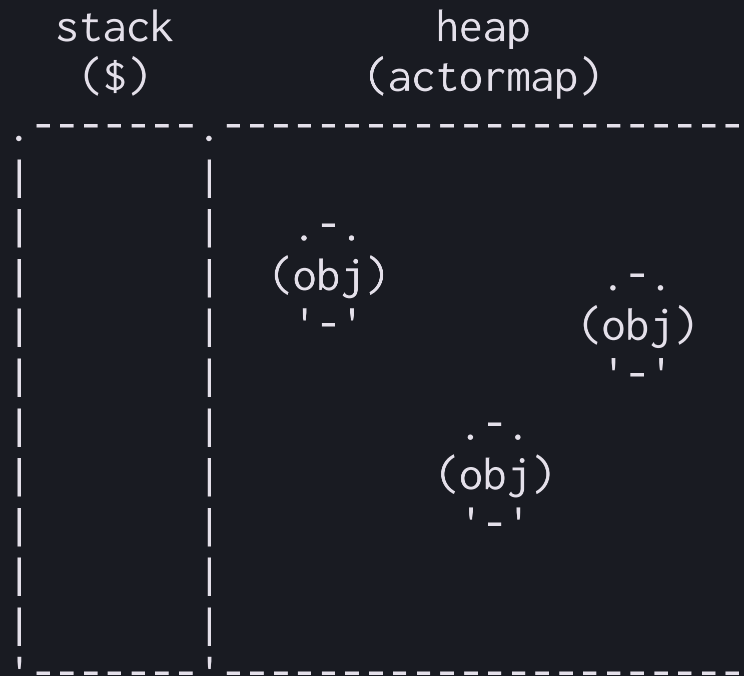
The Synchronous Call-Return Worldview



The Synchronous Call-Return Worldview



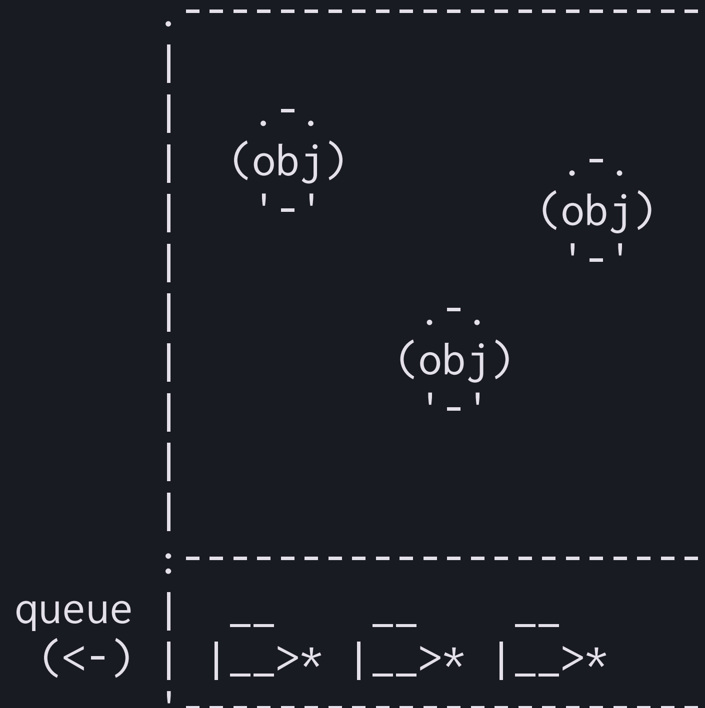
The Synchronous Call-Return Worldview



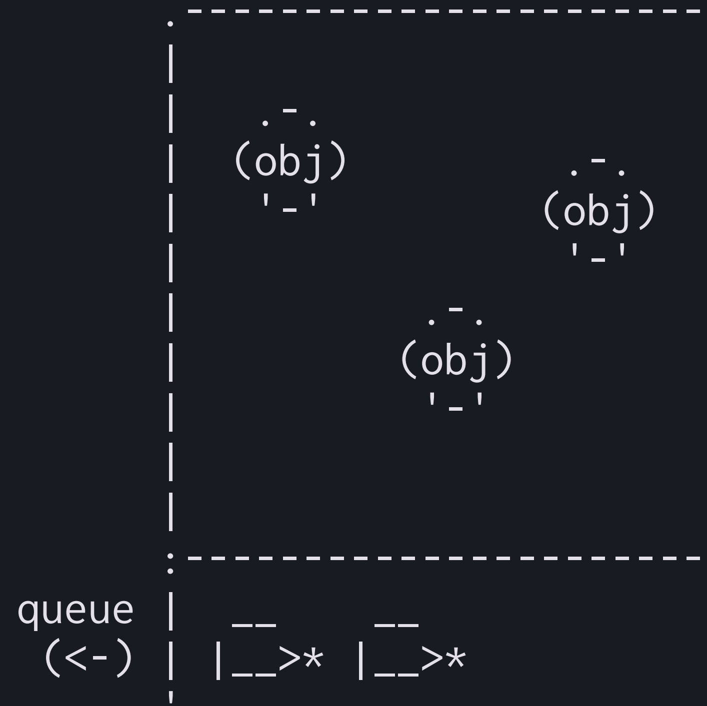
The Eventual Send Worldview

- **~193X:** λ
- **~1958:** (λ () ...)
- **~1972:** Smalltalk
- **~1973:** Actors
- **~1975:** Scheme
- **~1995:** Joule
- **~1997:** E (... kinda)

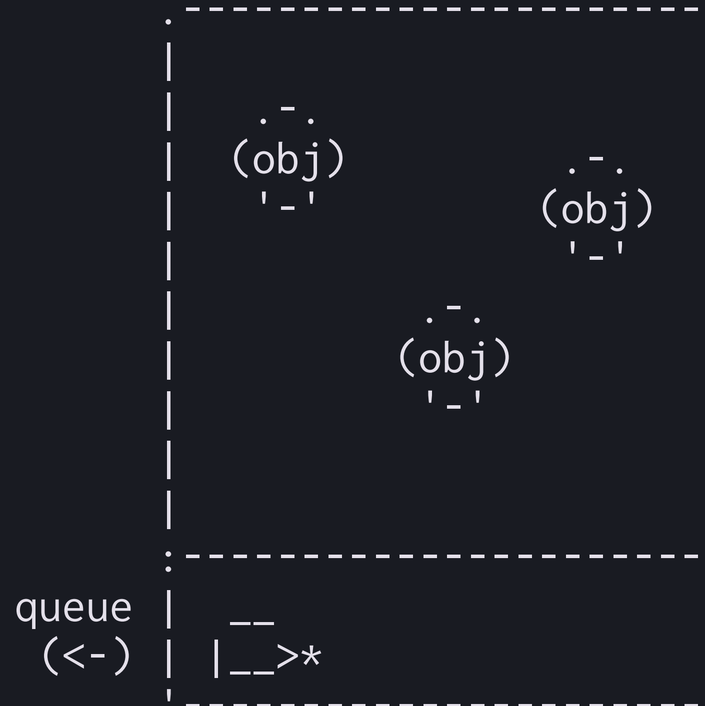
The Eventual Send Worldview



The Eventual Send Worldview

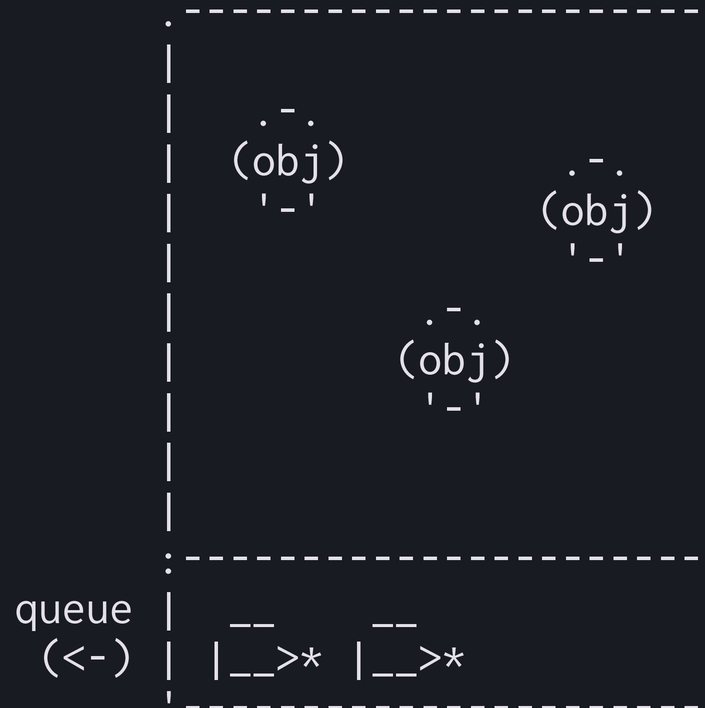


The Eventual Send Worldview

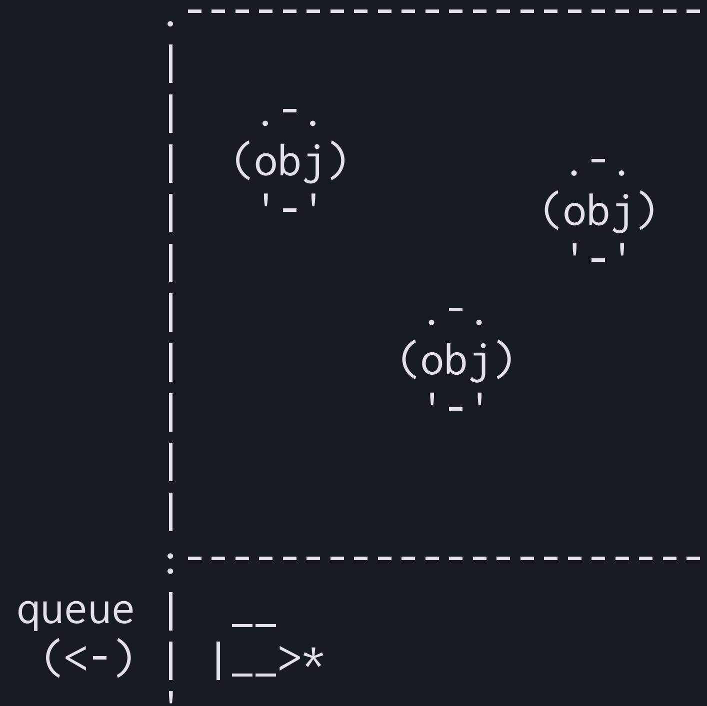


The Eventual Send Worldview

The Eventual Send Worldview



The Eventual Send Worldview

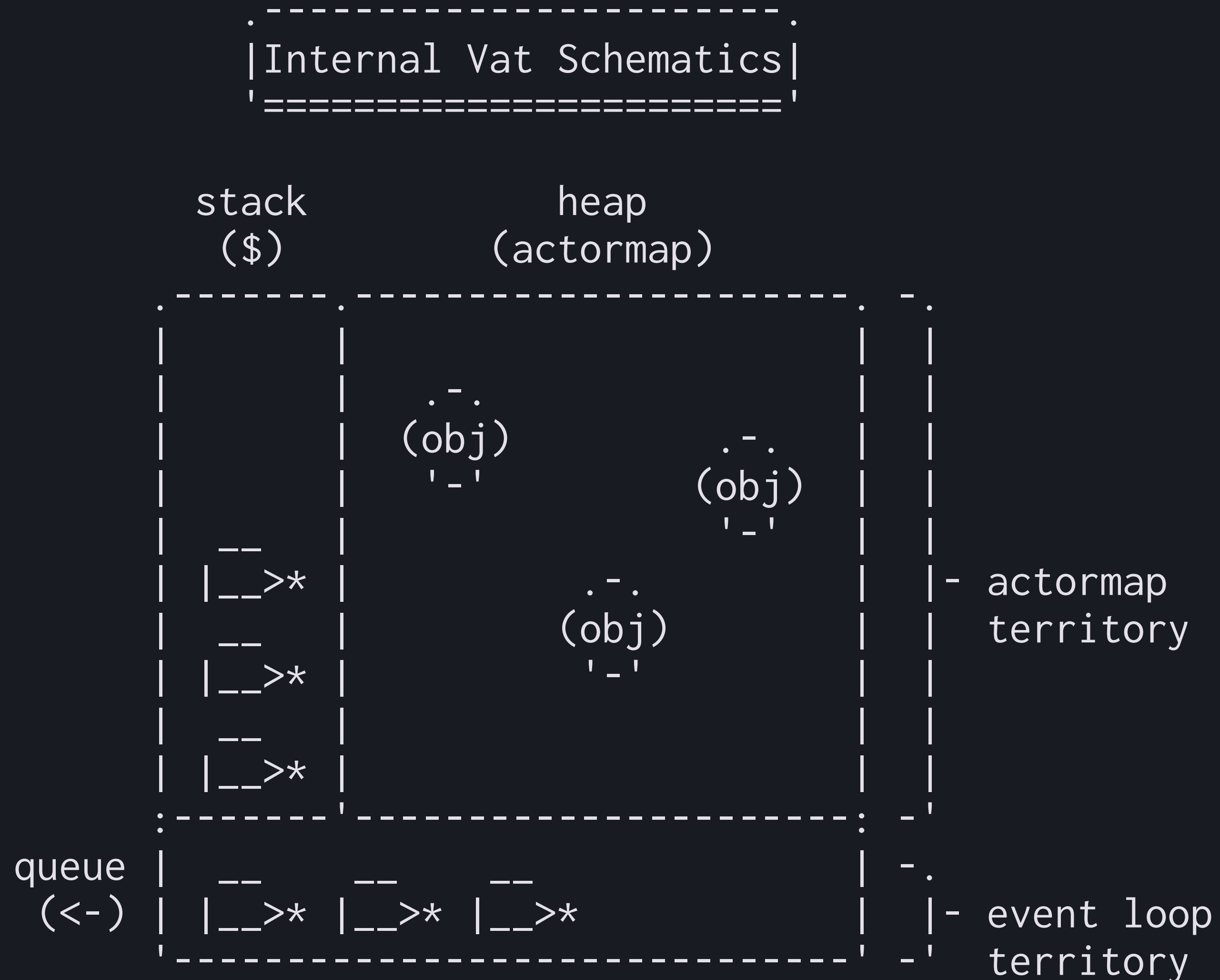


The Eventual Send Worldview

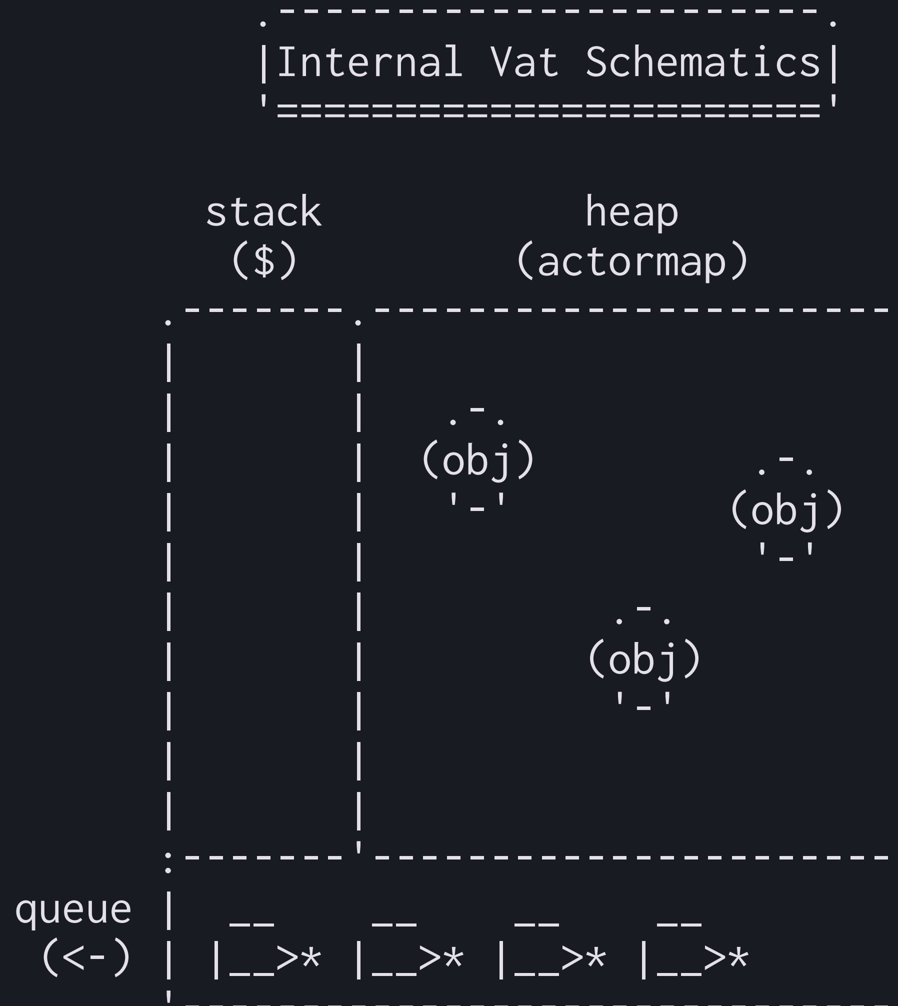
The Hybrid Worldview ("The Vat Model")

- **~193X:** λ
- **~1958:** (λ () ...)
- **~1972:** Smalltalk
- **~1973:** Actors
- **~1975:** Scheme
- **~1995:** Joule
- **~1997:** E
- **~2020:** Goblins

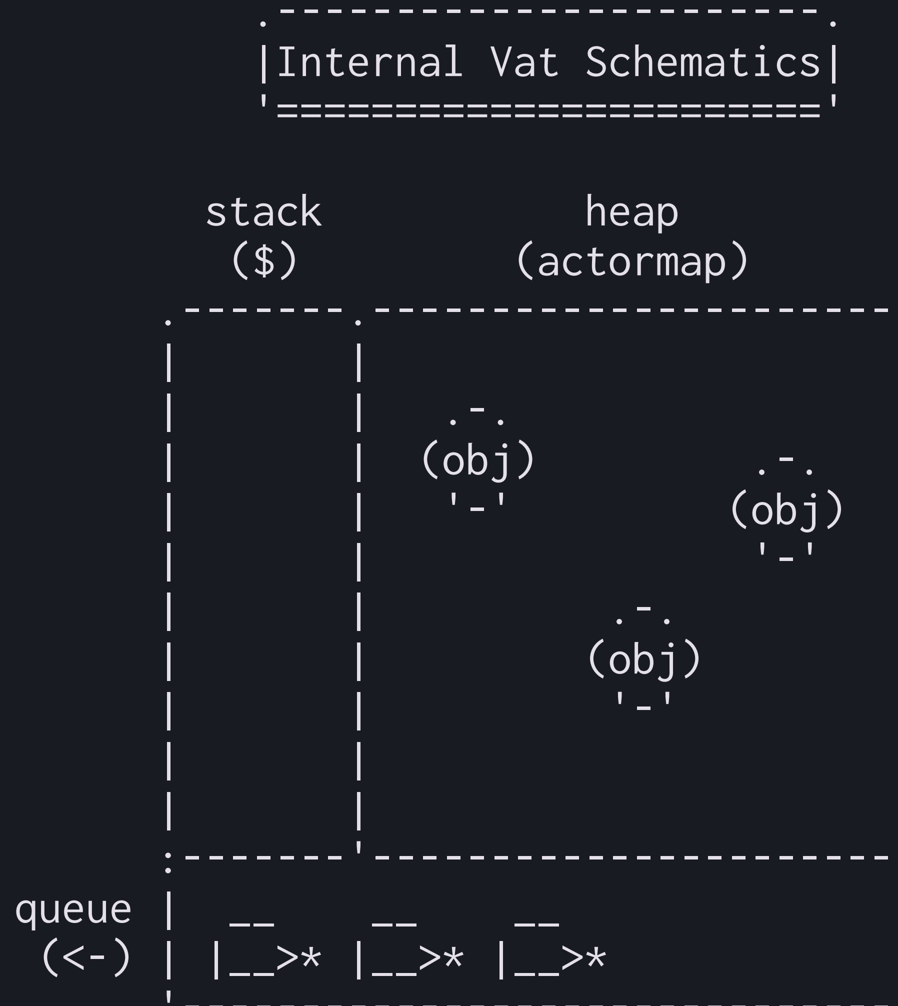
The Hybrid Worldview ("The Vat Model")



The Hybrid Worldview ("The Vat Model")



The Hybrid Worldview ("The Vat Model")

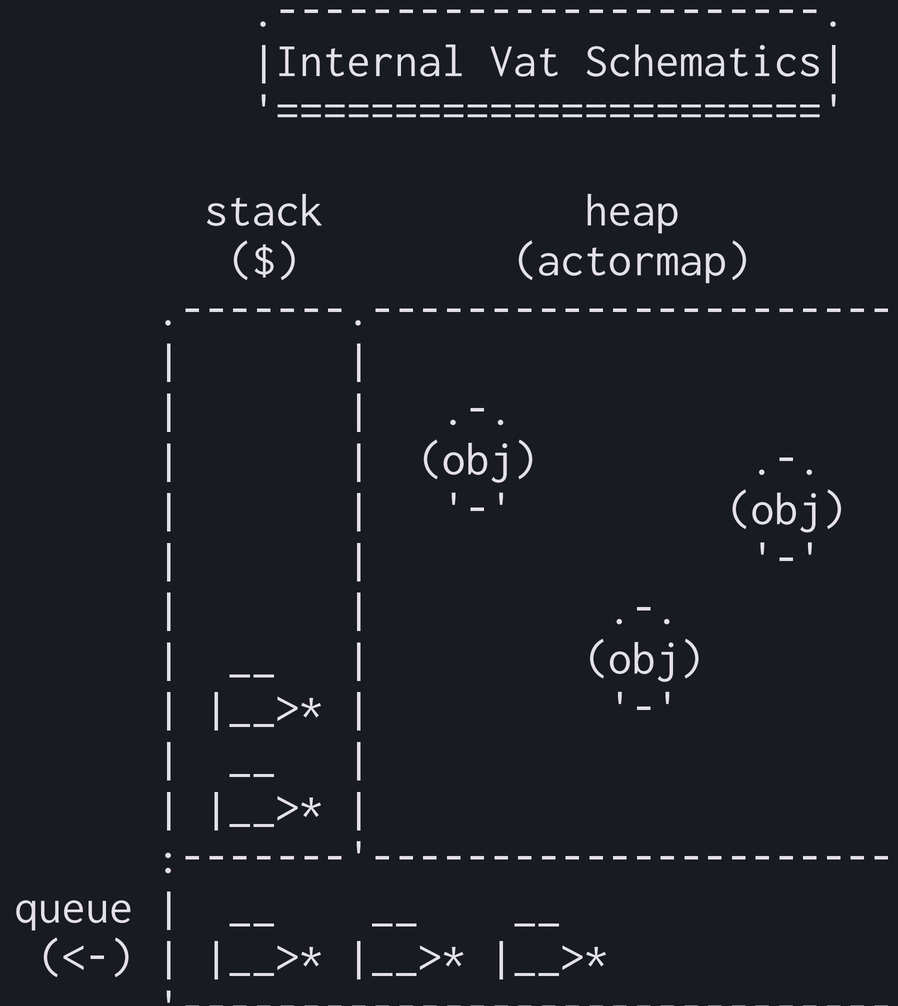


The Hybrid Worldview ("The Vat Model")

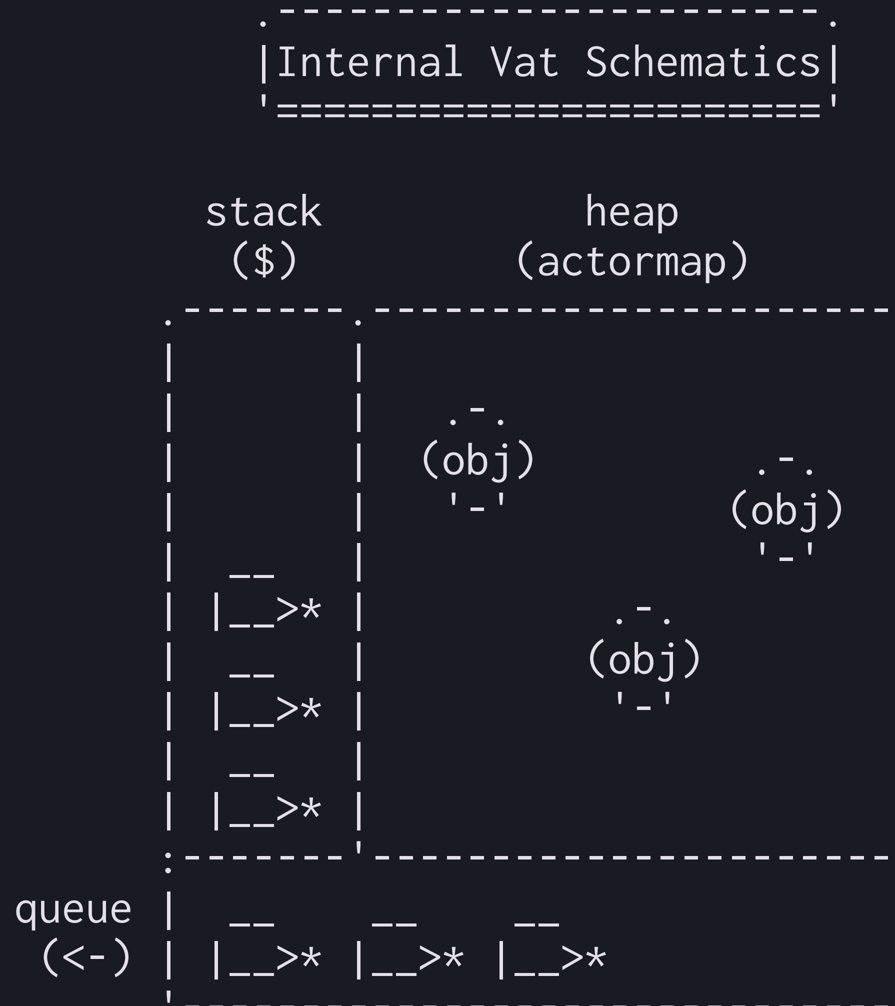
```
                .----------------------.
                |Internal Vat Schematics|
                '======================='

             stack           heap
              ($)         (actormap)
            .-------.----------------------.
            |       |                      |
            |       |   .-.                |
            |       |  (obj)         .-.   |
            |       |   '-'         (obj)  |
            |       |                '-'   |
            |       |          .-.         |
            |       |         (obj)        |
            |       |          '-'         |
            |   __  |                      |
            |  |__>*|                      |
            :-------'----------------------:
      queue |   __    __    __             |
       (<-) |  |__>* |__>* |__>*           |
            '------------------------------'
```

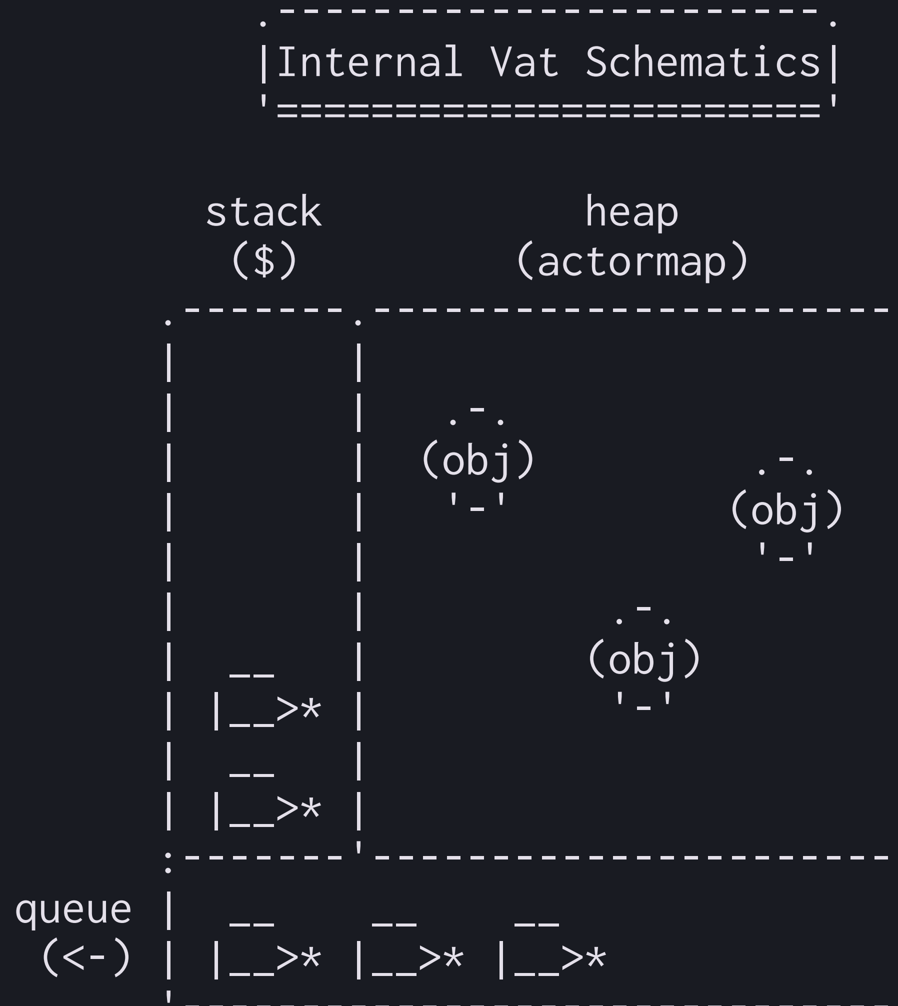
The Hybrid Worldview ("The Vat Model")



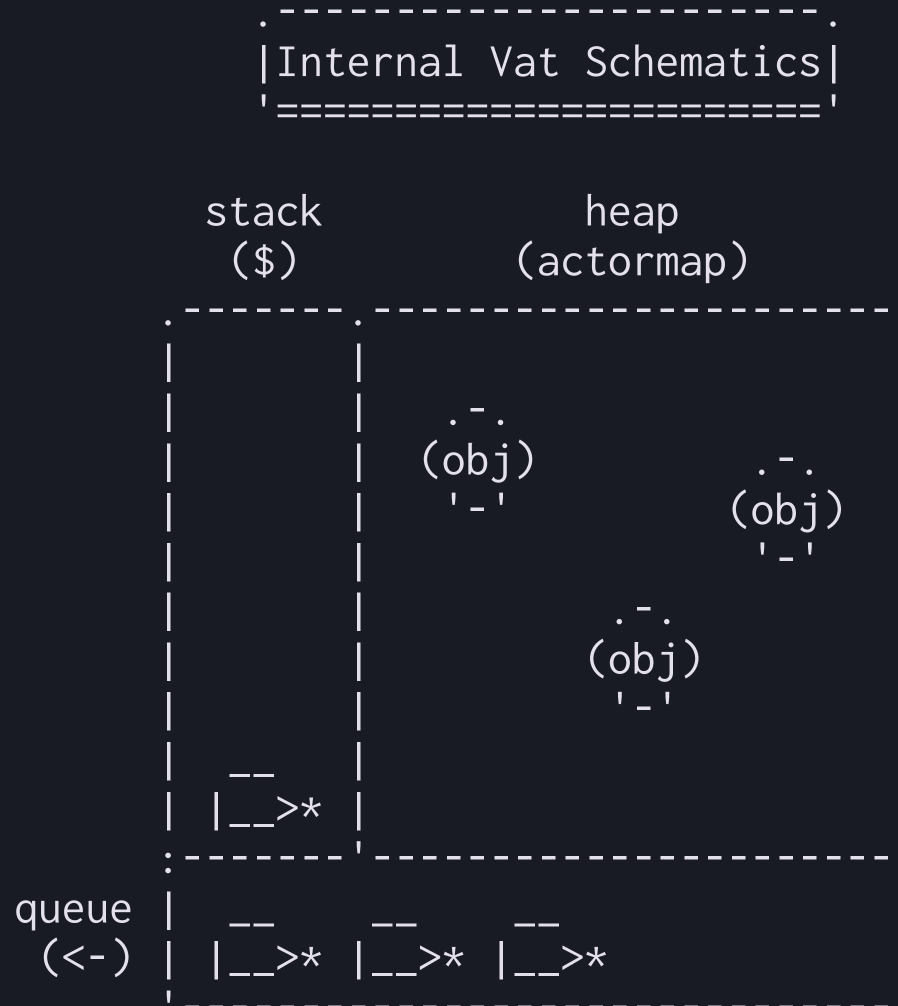
The Hybrid Worldview ("The Vat Model")



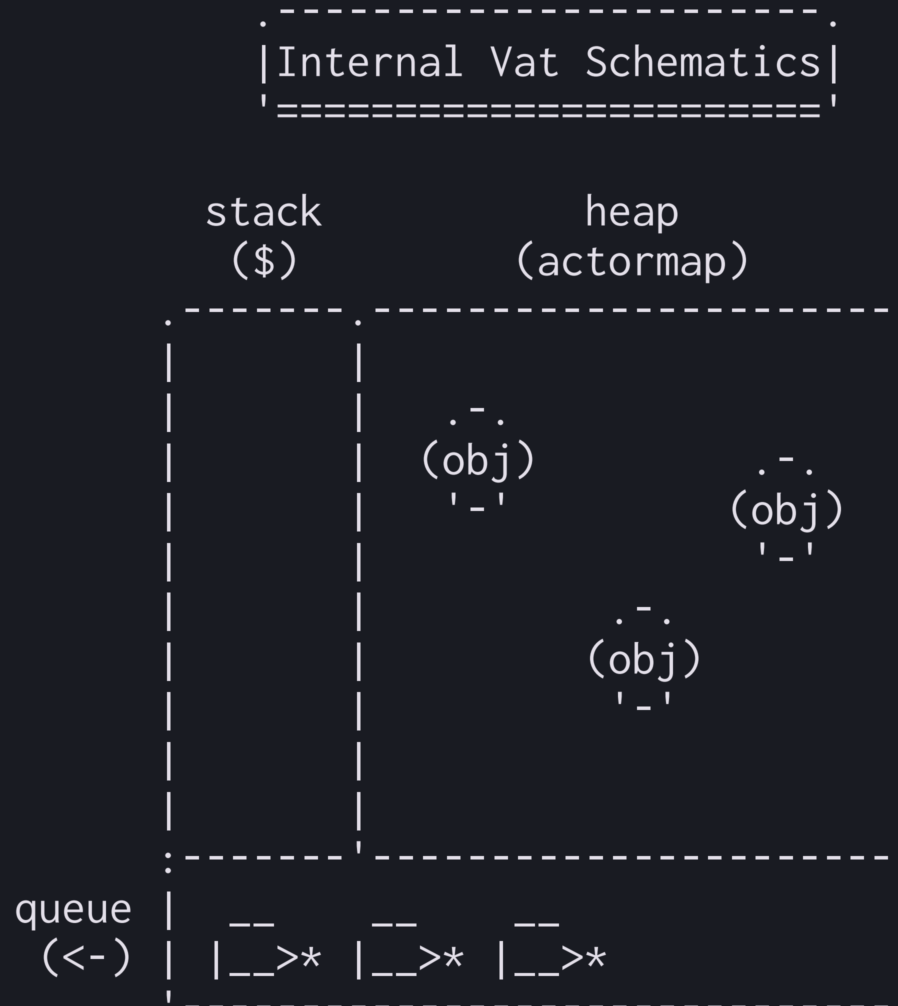
The Hybrid Worldview ("The Vat Model")



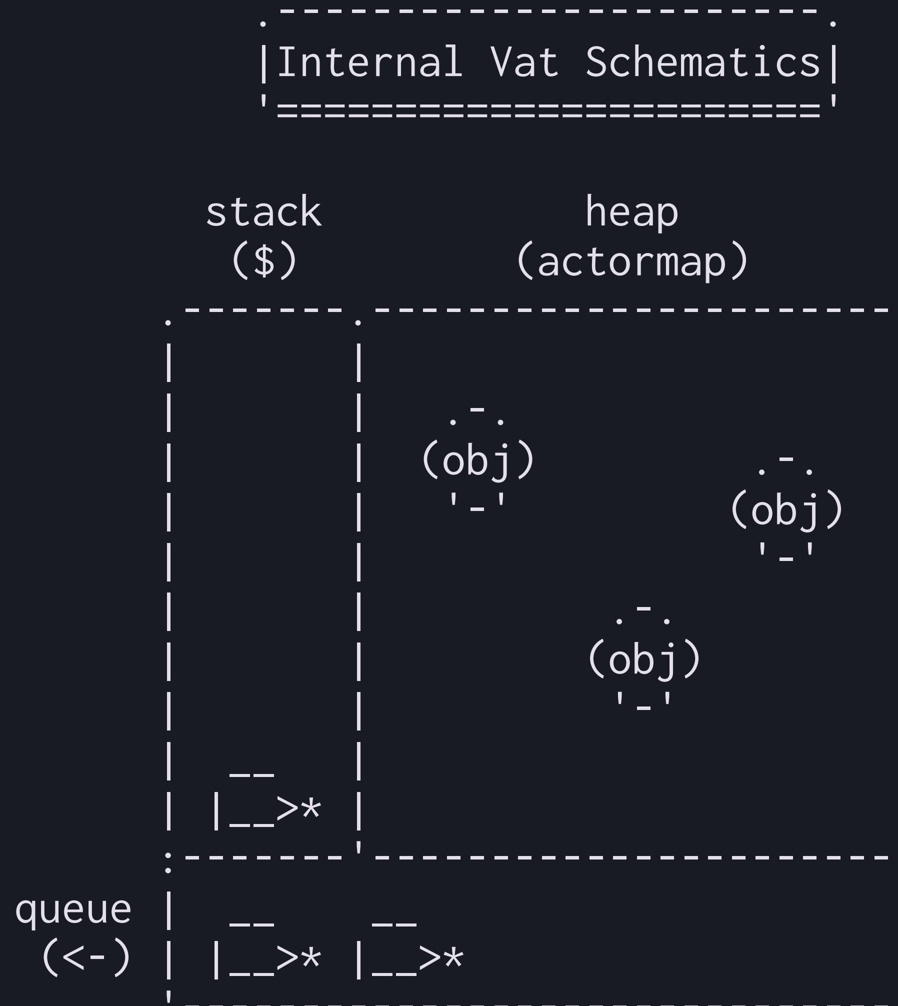
The Hybrid Worldview ("The Vat Model")



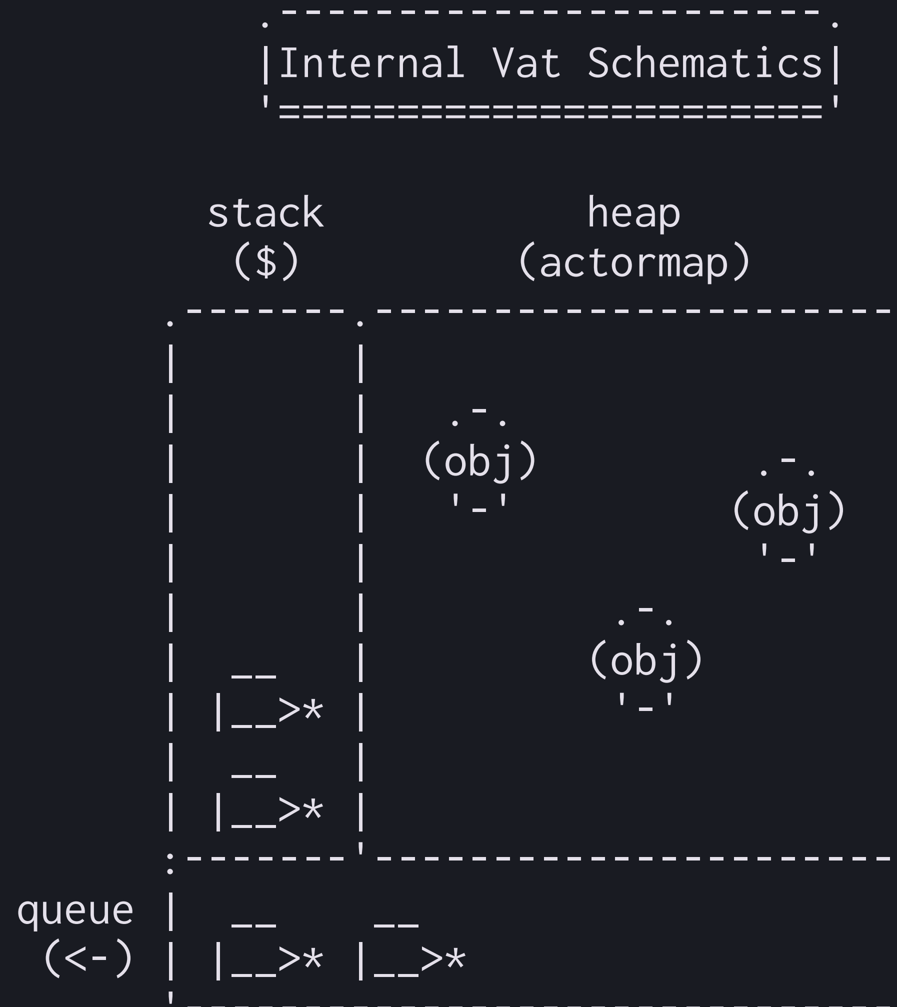
The Hybrid Worldview ("The Vat Model")



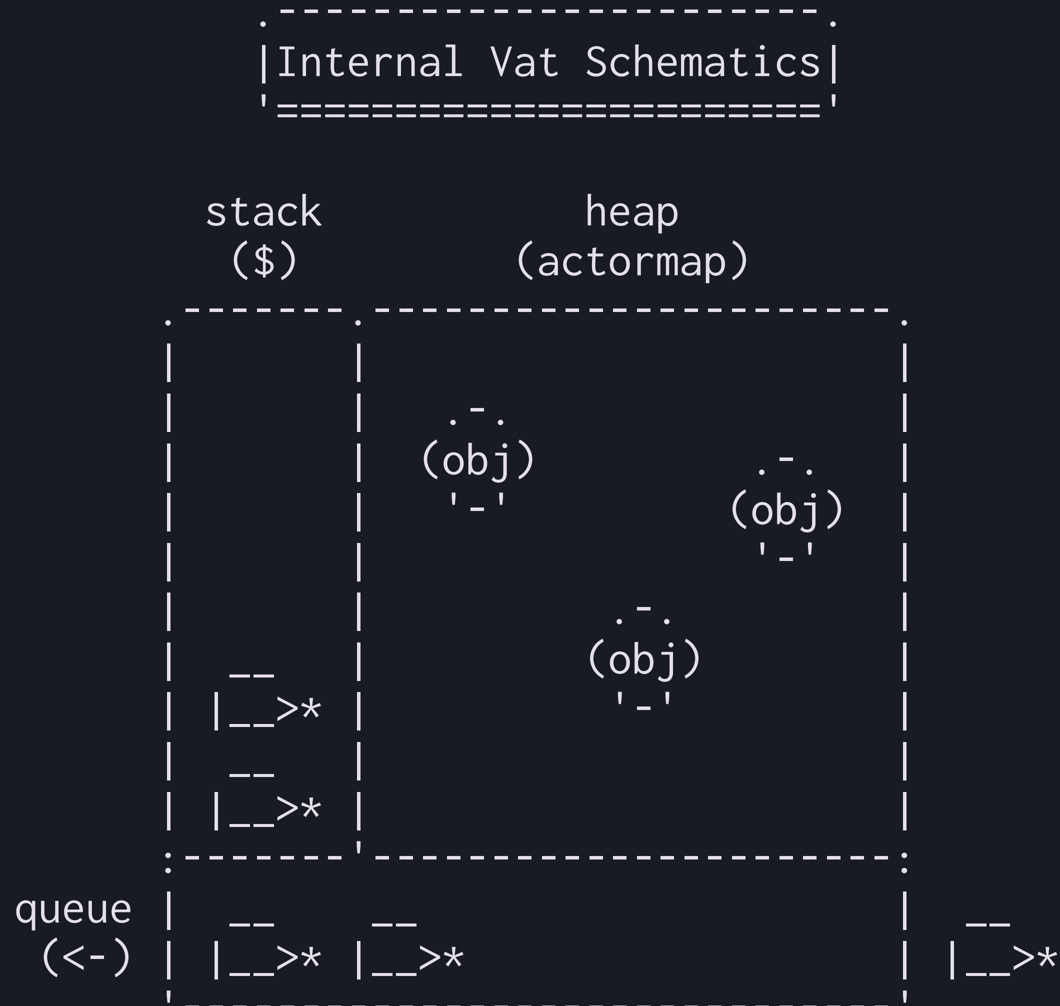
The Hybrid Worldview ("The Vat Model")



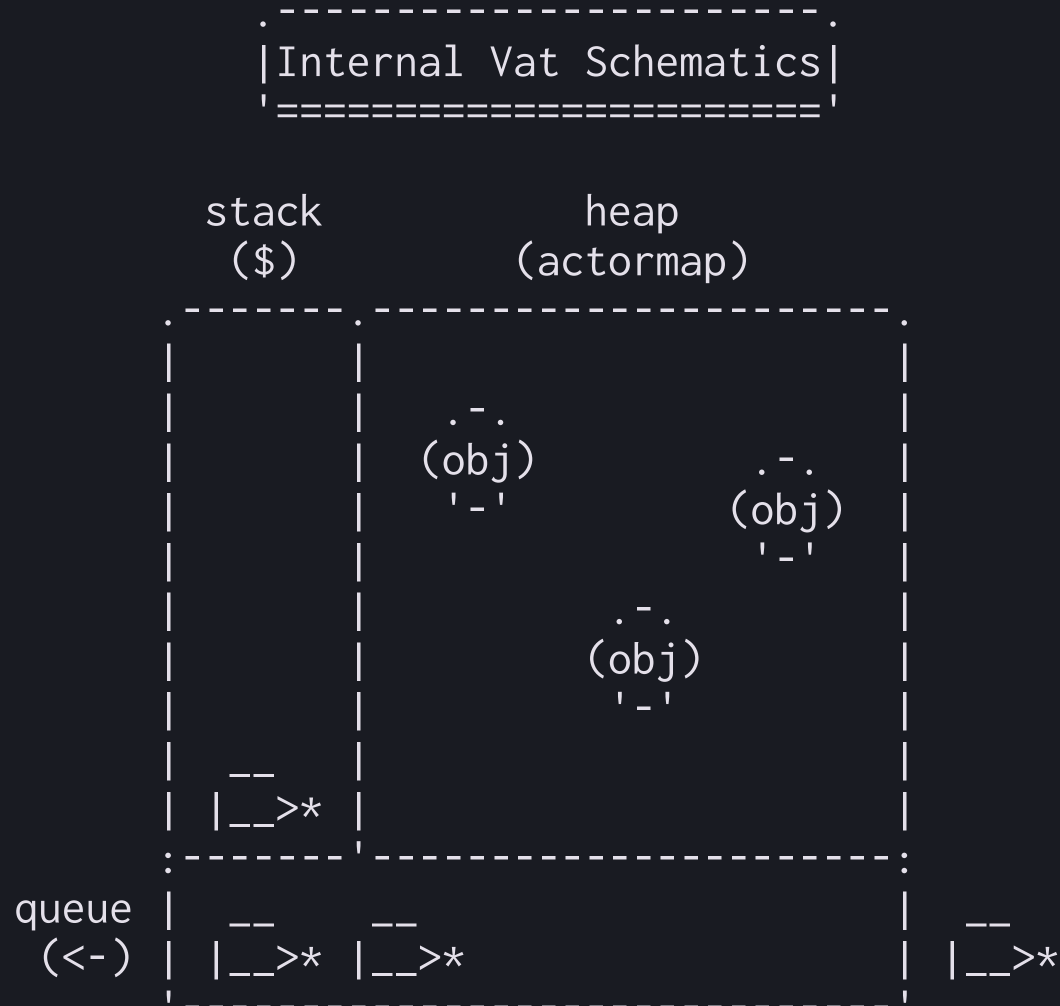
The Hybrid Worldview ("The Vat Model")



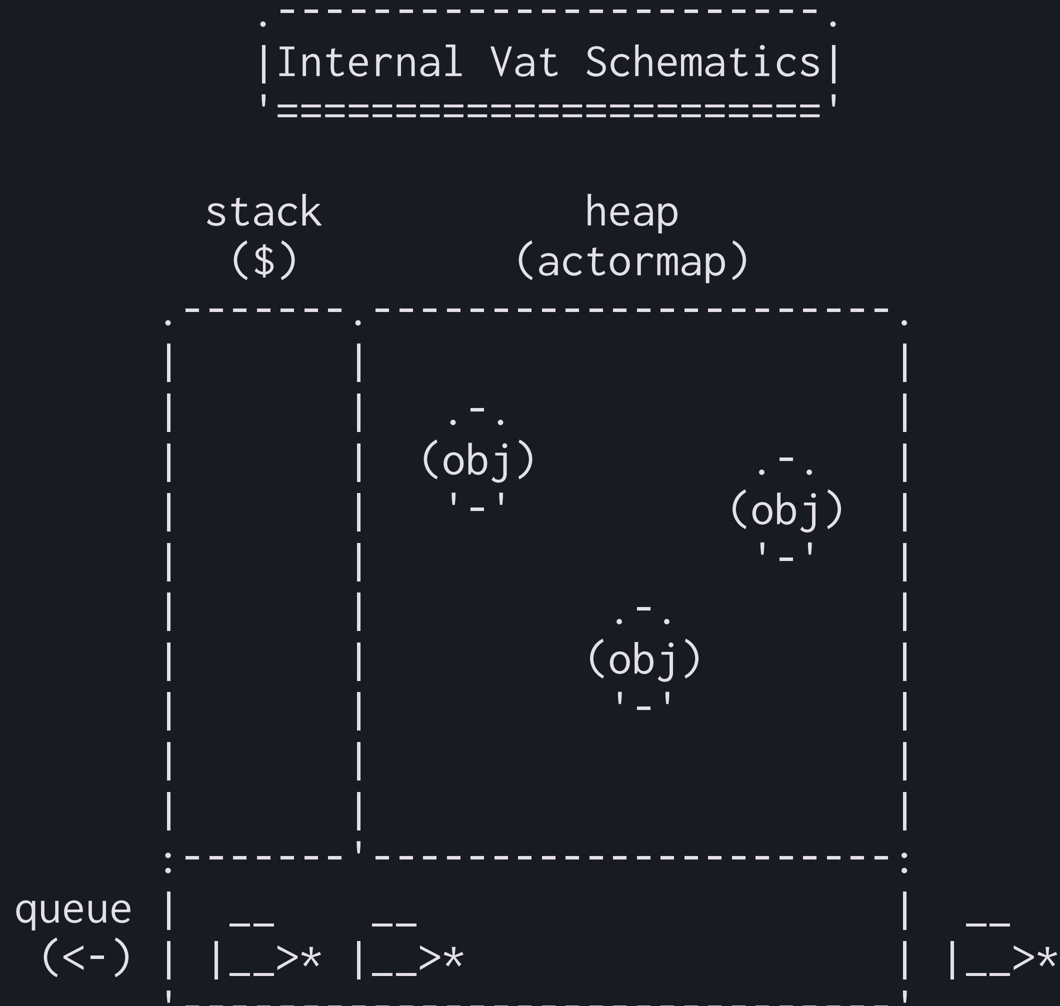
The Hybrid Worldview ("The Vat Model")



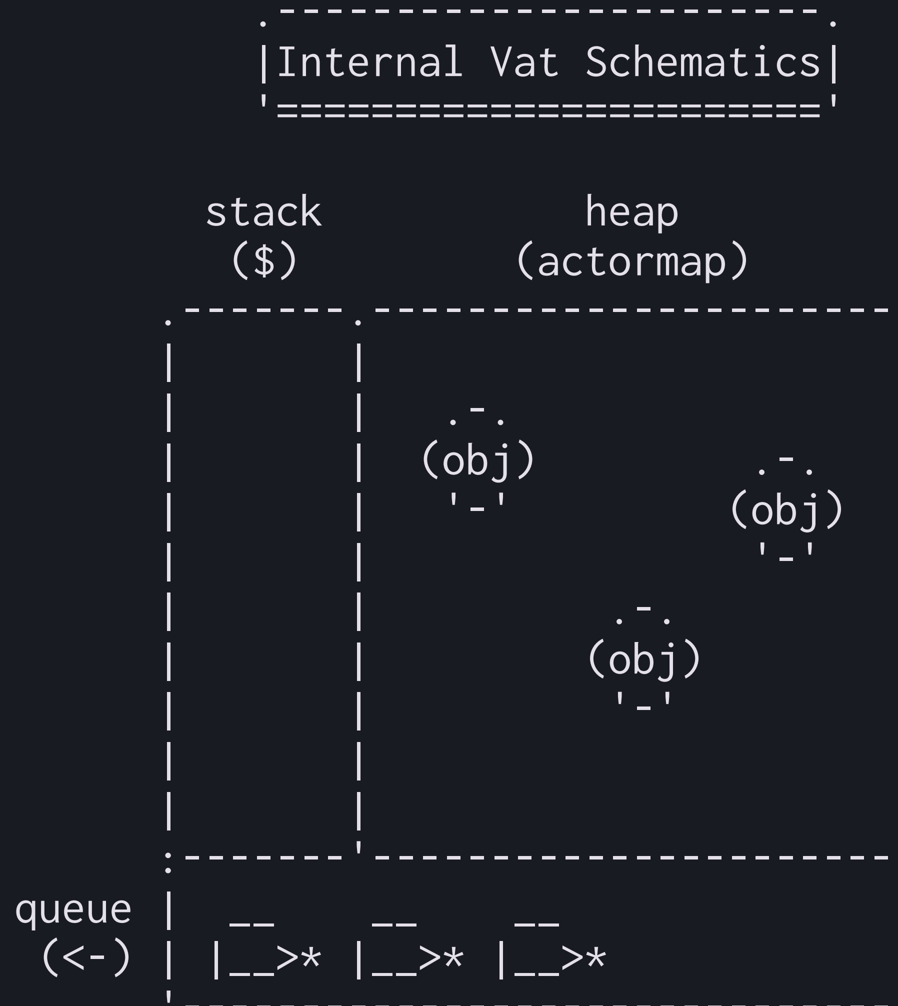
The Hybrid Worldview ("The Vat Model")



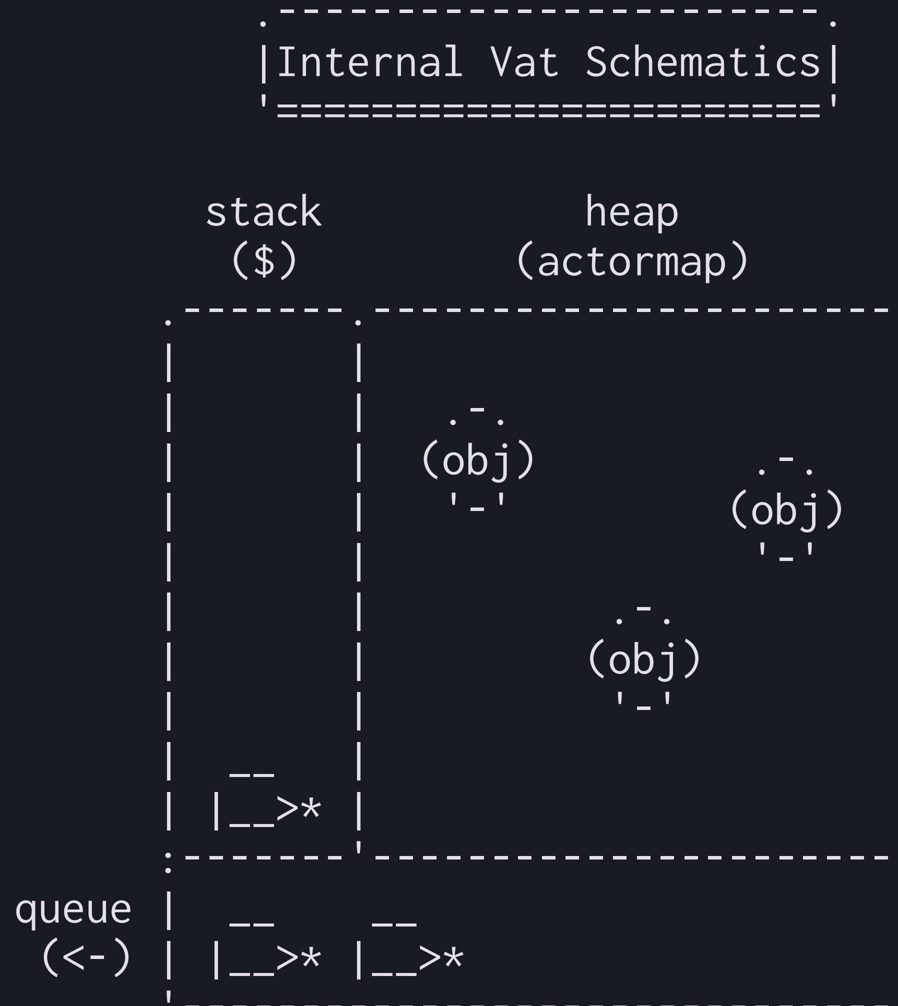
The Hybrid Worldview ("The Vat Model")



The Hybrid Worldview ("The Vat Model")

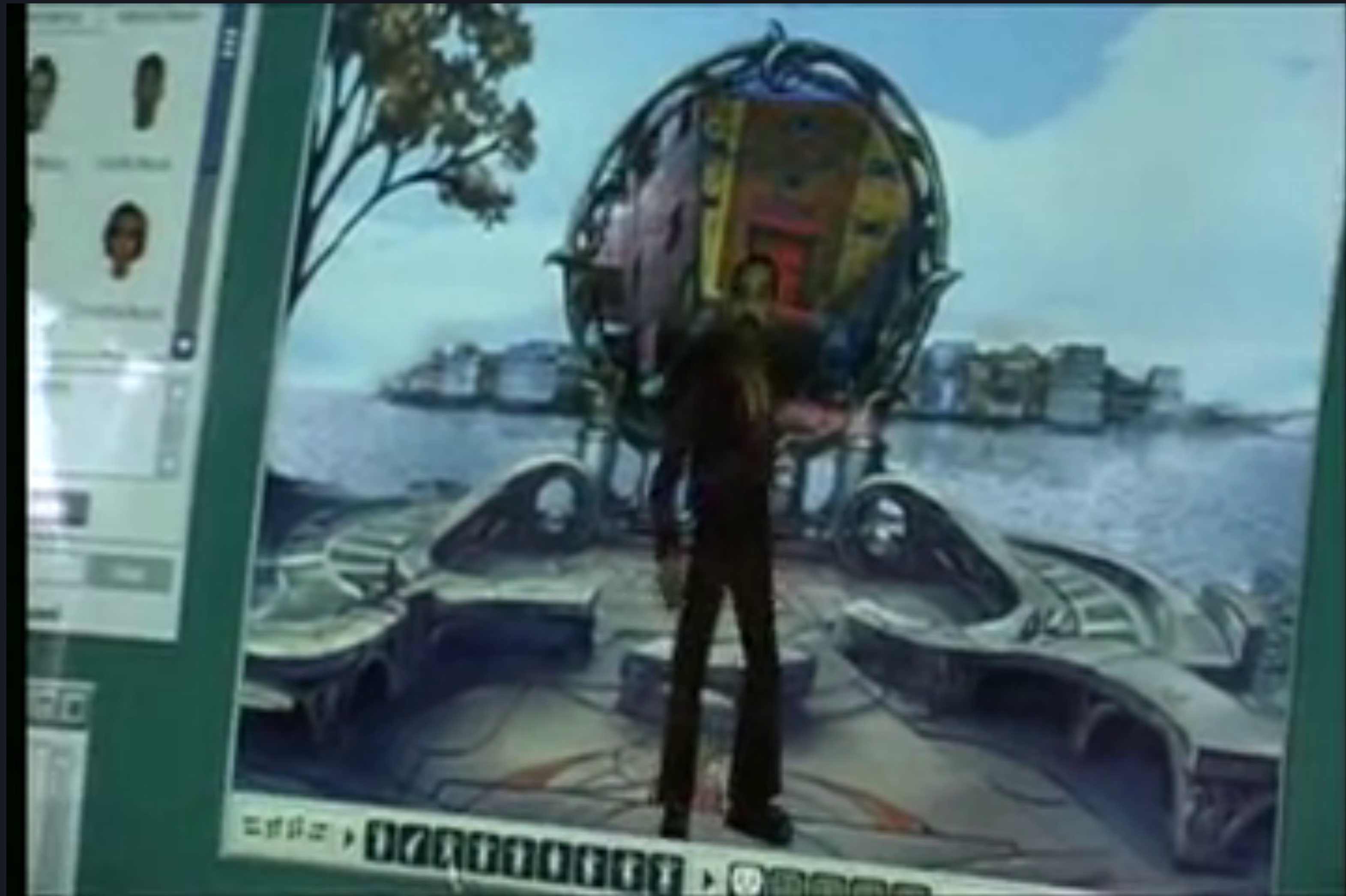


The Hybrid Worldview ("The Vat Model")



The Vat Model Combines Both Worlds

Call-Return ($)	Eventual Send (<-)
Same vat only	Any actor, anywhere
Immediate	Eventual
Return values	Promises
Transactions	Distributed w/o deadlocks



Open Source Distributed Capabilities

Welcome to *ERights.org*, home of ***E***,
the secure distributed persistent language
for capability-based smart contracting.

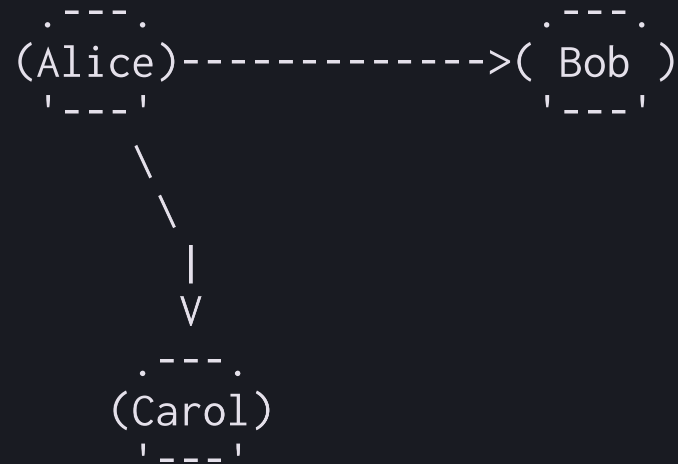
Quick Start | What's New? | What's ***E***?
Smart Contracts | History & Talks | Feedback

[California Home] [Mirror in Virtual Tonga]

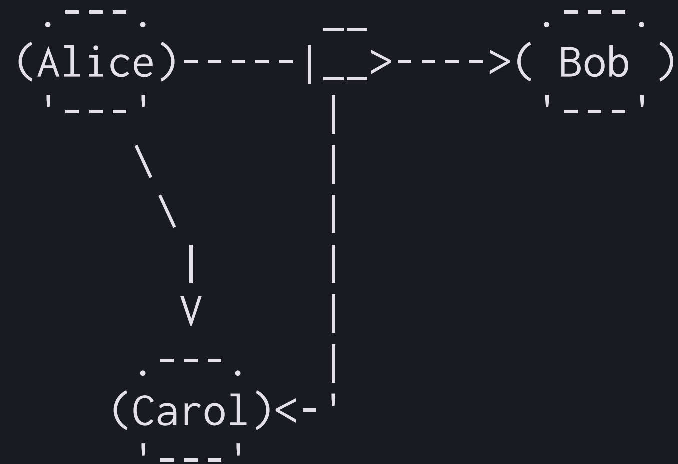
We do not influence the course of events by persuading people that we are right when we make what they regard as radical proposals. Rather, we exert influence by keeping options available when something has to be done at a time of crisis.

--Milton Friedman

Object Capability Security



Object Capability Security



Object Capability Security

Lambda: The Ultimate Security Model

A.I. Memo No. 1564

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

March 1996

A Security Kernel Based on the Lambda Calculus

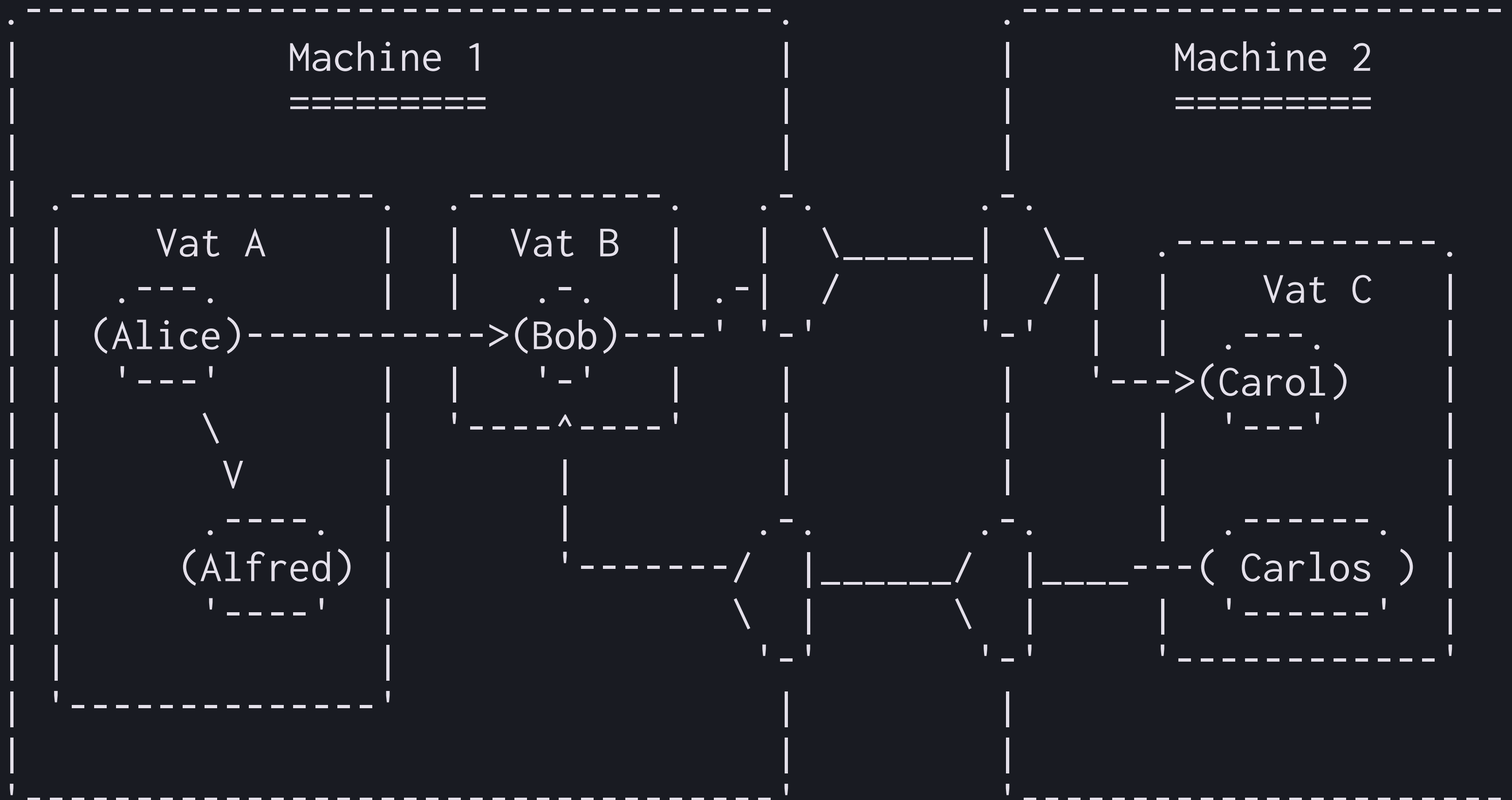
Jonathan A. Rees

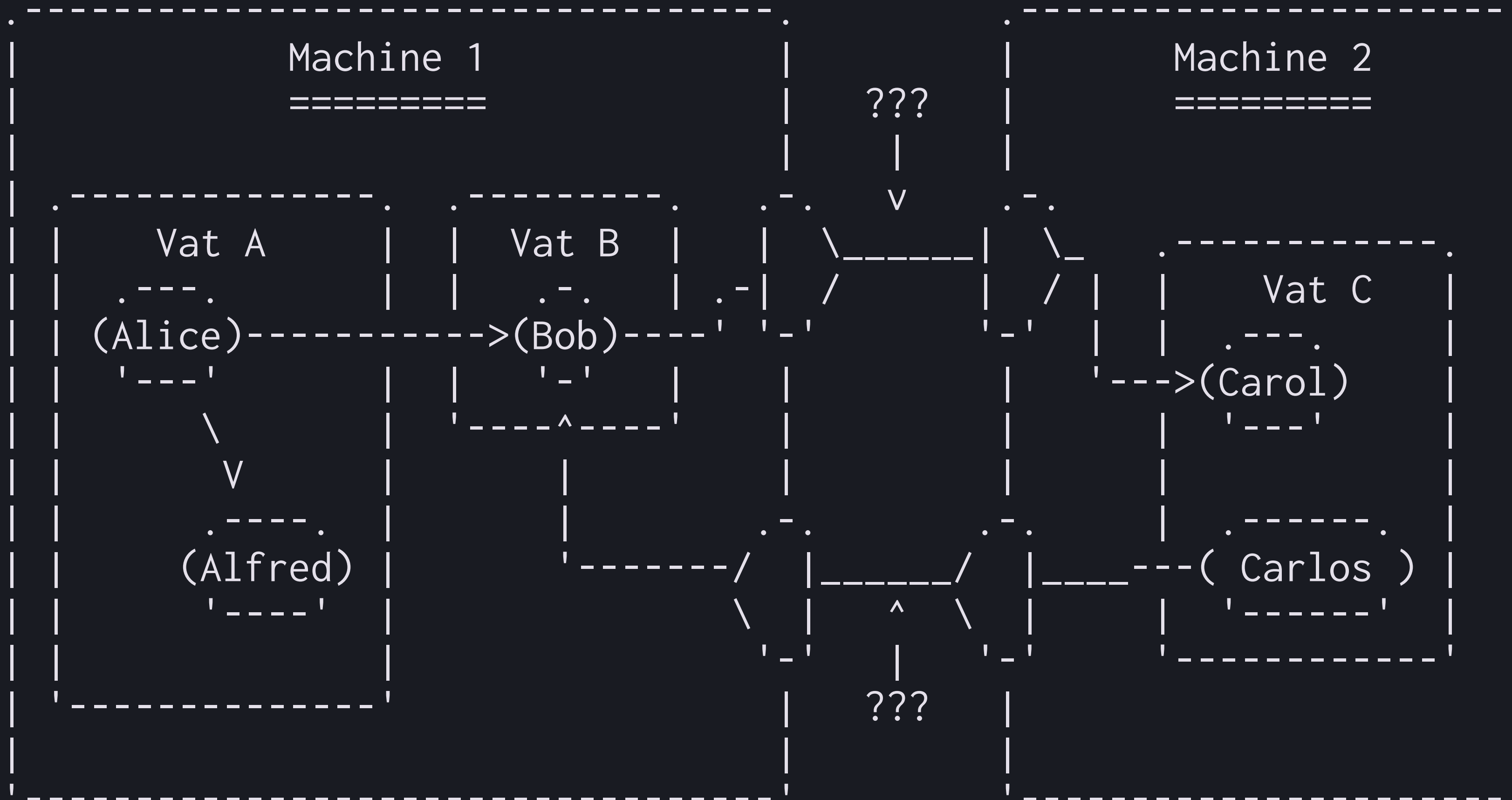
This publication can be retrieved by anonymous ftp to publications.ai.mit.edu.

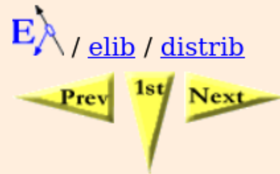
Abstract

Cooperation between independent agents depends upon establishing a degree of security. Each of the cooperating agents needs assurance that the cooperation will not endanger resources of value to that agent. In a computer system, a computational mechanism can assure safe cooperation among the system's users by mediating resource access according to desired security policy. Such a mechanism, which is called a *security kernel*, lies at the heart of many operating systems and programming environments.

What's in your scope? That's your authority.


Machine 1
Vat A
Alice
Alfred
Vat B
Bob
Machine 2
Vat C
Carol
Carlos


Machine 1
Machine 2
???
Vat A
Vat B
Vat C
(Alice)
(Bob)
(Carol)
(Alfred)
(Carlos)
???



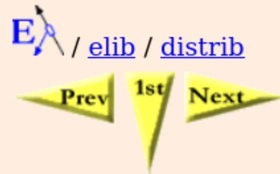
CapTP:

The Capability Transport Protocol

Overview

The Four Tables	The "data structure" defining the semantics of the 2-vat CatTP protocol
Resolving RemotePromises	Explains the automatically generated whenMoreResolved messages, and how they resolve remote unresolved references (RemotePromises).
Three-Vat Granovetter Introduction	What happens when Alice, Bob, and Carol are in three separate vats?
Distributed Acyclic Garbage Collection	How we collect what distributed garbage we can.
Preparing for the Pervasive Possibility of Partition	Cleaning up after the show's over.
Protocol Parameter Types	Defines the types used in the protocol declarations below.

Distributed (acyclic) garbage collection!



/ elib / distrib

CapTP:

The Capability Transport Protocol

Overview

The Four Tables	The "data structure" defining the semantics of the 2-vat CatTP protocol
Resolving RemotePromises	Explains the automatically generated whenMoreResolved messages, and how they resolve remote unresolved references (RemotePromises).
Three-Vat Granovetter Introduction	What happens when Alice, Bob, and Carol are in three separate vats?
Distributed Acyclic Garbage Collection	How we collect what distributed garbage we can.
Preparing for the Pervasive Possibility of Partition	Cleaning up after the show's over.
Protocol Parameter Types	Defines the types used in the protocol declarations below.

Promise pipelining!

Convenient *and* network efficient!

```
(define (^greeter bcom my-name)
  (lambda (your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name)))
```

```
(define (^greeter bcom my-name)
  (lambda (your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name)))

; Make a new instance of ^greeter
(define my-greeter
  (spawn ^greeter "Alice"))
```

```
(define (^greeter bcom my-name)
  (lambda (your-name)
    (format "Hello ~a, my name is ~a!" your-name my-name)))

; Make a new instance of ^greeter
(define my-greeter
  (spawn ^greeter "Alice"))

($ my-greeter "Bob")
; => "Hello Bob, my name is Alice!"
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))

(define my-counter
  (spawn ^counter))
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))

(define my-counter
  (spawn ^counter))

($ my-counter 'get-count)
; => 0
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))

(define my-counter
  (spawn ^counter))

($ my-counter 'get-count)
; => 0

($ my-counter 'incr)
($ my-counter 'get-count)
; => 1
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))

(define my-counter
  (spawn ^counter))

($ my-counter 'get-count)
; => 0

($ my-counter 'incr)
($ my-counter 'get-count)
; => 1

($ my-counter 'incr)
($ my-counter 'incr)
($ my-counter 'get-count)
; => 3
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))

(define (^counting-greeter bcom my-name)
  (define counter
    (spawn ^counter))
  (lambda (your-name)
    ($ counter 'incr)
    (format "Hello ~a, my name is ~a! [call #~a]"
            your-name my-name ($ counter 'get-count))))
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))

(define (^counting-greeter bcom my-name)
  (define counter
    (spawn ^counter))
  (lambda (your-name)
    ($ counter 'incr)
    (format "Hello ~a, my name is ~a! [call #~a]"
            your-name my-name ($ counter 'get-count))))

(spawn ^counting-greeter "Alice")
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))

(define (^counting-greeter bcom my-name)
  (define counter
    (spawn ^counter))
  (lambda (your-name)
    (| counter 'incr)
    (format "Hello ~a, my name is ~a! [call #~a]"
            your-name my-name (| counter 'get-count))))

(spawn ^counting-greeter "Alice")

(| my-counting-greeter "Bob")
; => "Hello Bob, my name is Alice! [call #1]"
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))

(define (^counting-greeter bcom my-name)
  (define counter
    (spawn ^counter))
  (lambda (your-name)
    (| counter 'incr)
    (format "Hello ~a, my name is ~a! [call #~a]"
            your-name my-name (| counter 'get-count))))

(spawn ^counting-greeter "Alice")

(| my-counting-greeter "Bob")
; => "Hello Bob, my name is Alice! [call #1]"

(| my-counting-greeter "Betsy")
; => "Hello Betsy, my name is Alice! [call #2]"
```

```
(define (^counter bcom [count 0])
  (methods
   [(incr)
    (bcom (^counter bcom (add1 count)))]
   [(get-count)
    count]))

(define (^counting-greeter bcom my-name)
  (define counter
    (spawn ^counter))
  (lambda (your-name)
    ($ counter 'incr)
    (format "Hello ~a, my name is ~a! [call #~a]"
            your-name my-name ($ counter 'get-count))))

(spawn ^counting-greeter "Alice")

($ my-counting-greeter "Bob")
; => "Hello Bob, my name is Alice! [call #1]"

($ my-counting-greeter "Betsy")
; => "Hello Betsy, my name is Alice! [call #2]"

($ my-counting-greeter "Billy")
; => "Hello Billy, my name is Alice! [call #3]"
```

GREETER WHICH CALLS COUNTER GOES HERE

CALLING W/ ASYNC MESSAGE PASSING GOES HERE

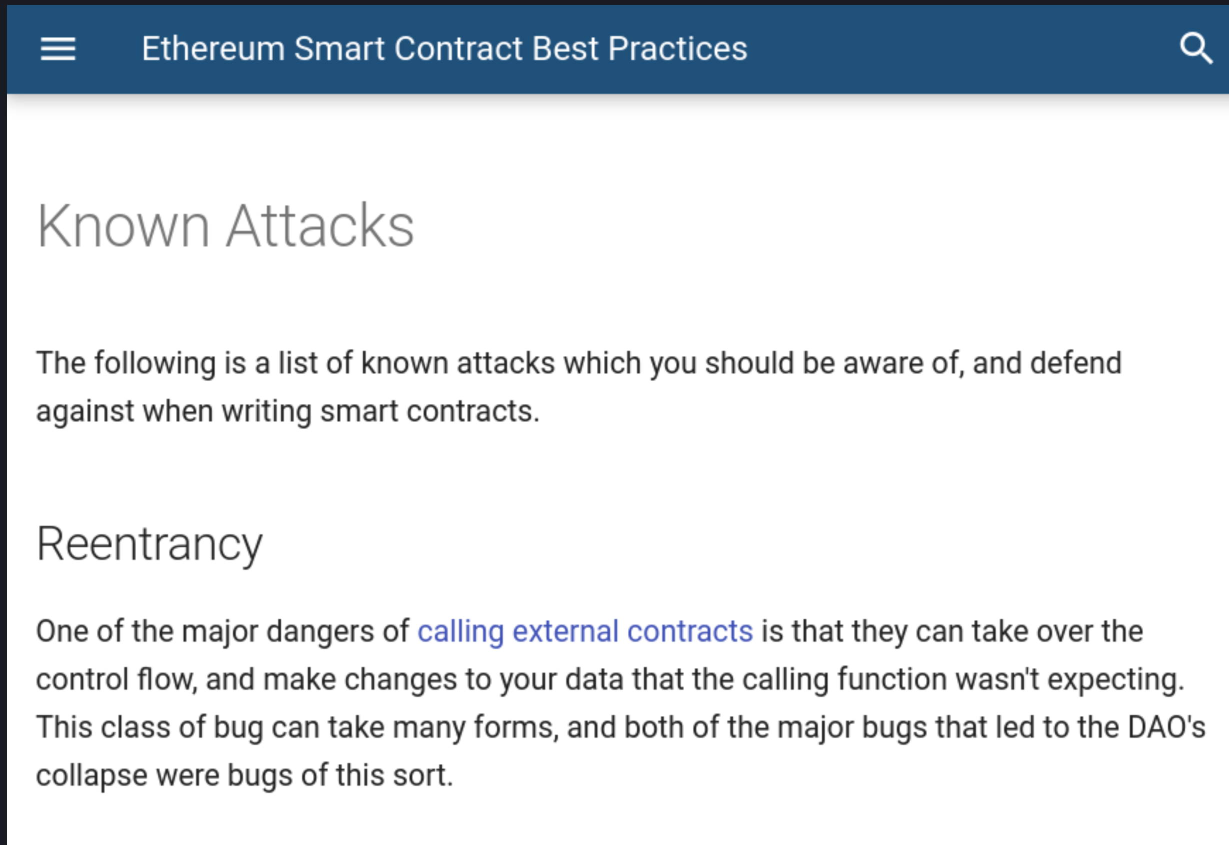
PROMISE PIPELINING EXAMPLE GOES HERE

Promise pipelining!

Machines grow faster and memories grow larger. But the speed of light is constant and New York is not getting any closer to Tokyo.

– Mark S. Miller

Why promises instead of coroutines?



Simple money in 25 lines of code!

```
(define (^mint bcom)
  (define-values (decr-seal decr-unseal decr-sealed?)
    (make-sealer-triplet 'mint))
  (define (^purse bcom initial-balance)
    (define-cell balance
      initial-balance)
    (define (<=-balance? amount)
      (<= amount ($ balance)))
    (define/contract (decr amount)
      (-> (and/c integer? (>=/c 0) <=-balance?)
          any/c)
      ($ balance (- ($ balance) amount)))
    (define/contract (deposit-method amount src)
      (-> (and/c integer? (>=/c 0)) any/c any/c)
      ((decr-unseal ($ src 'get-decr)) amount)
      ($ balance (+ ($ balance) amount)))
    (methods
     [(get-balance) ($ balance)]
     [(sprout) (spawn ^purse 0)]
     [deposit deposit-method]
     [(get-decr) (decr-seal decr)]))
  (define/contract (fiat-make-purse initial-balance)
    (-> (and/c integer? (>=/c 0)) any/c)
    (spawn ^purse initial-balance))
  (methods [new-purse fiat-make-purse]))
```