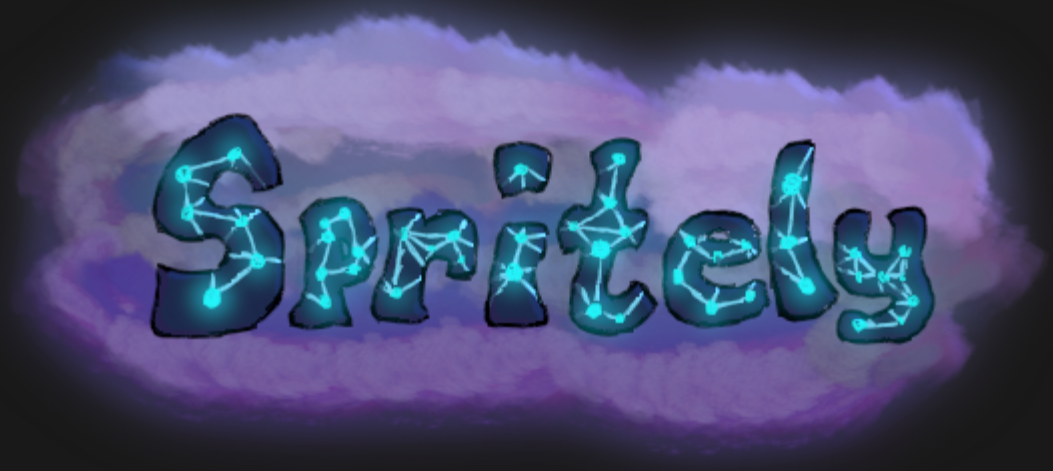
OCAPN

JESSICA TALLON

WHO AM I?

- 👋 I’m Jessica Tallon
- Worked on: ActivityPub
- Developer at the Spritely Institute
- Work on implementation and standardization of OCapN



- US Registered non-profit 501(c)(3)
- Developing new technologies for the decentralized web
- Work on: Goblins, Hoot and OCapN
- Also donate!

OCAPN: WHAT IS IT?

- OCapN: Object Capability Network
- Peer-to-peer communication
- Built on Capability Security
- Built on Actor model
- Secure messaging between actors
- Promises & Promise pipelining
- Network transport agnostic
- Acyclic distributed GC
- Third party Handoffs

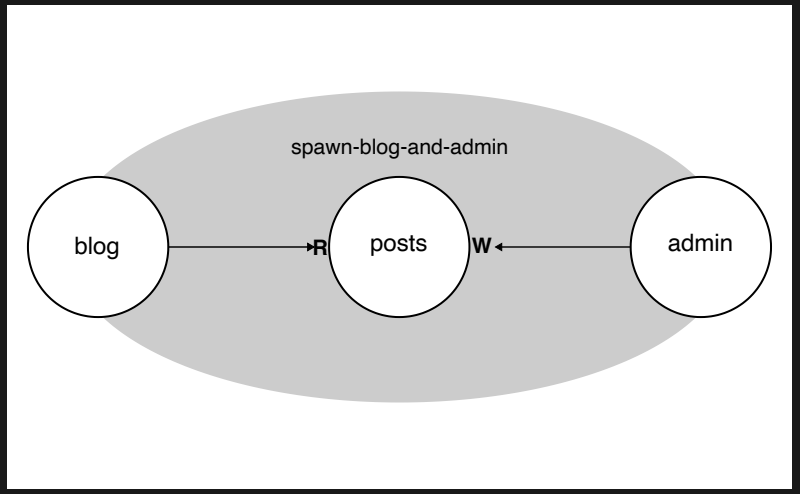
OCAPN: WHY?

- Make building peer-to-peer ergonomic, secure and easy
- Applications can focus on application logic, leaving P2P to OCapN
- Fault tolerant systems
- Applications don’t need to worry about network transport

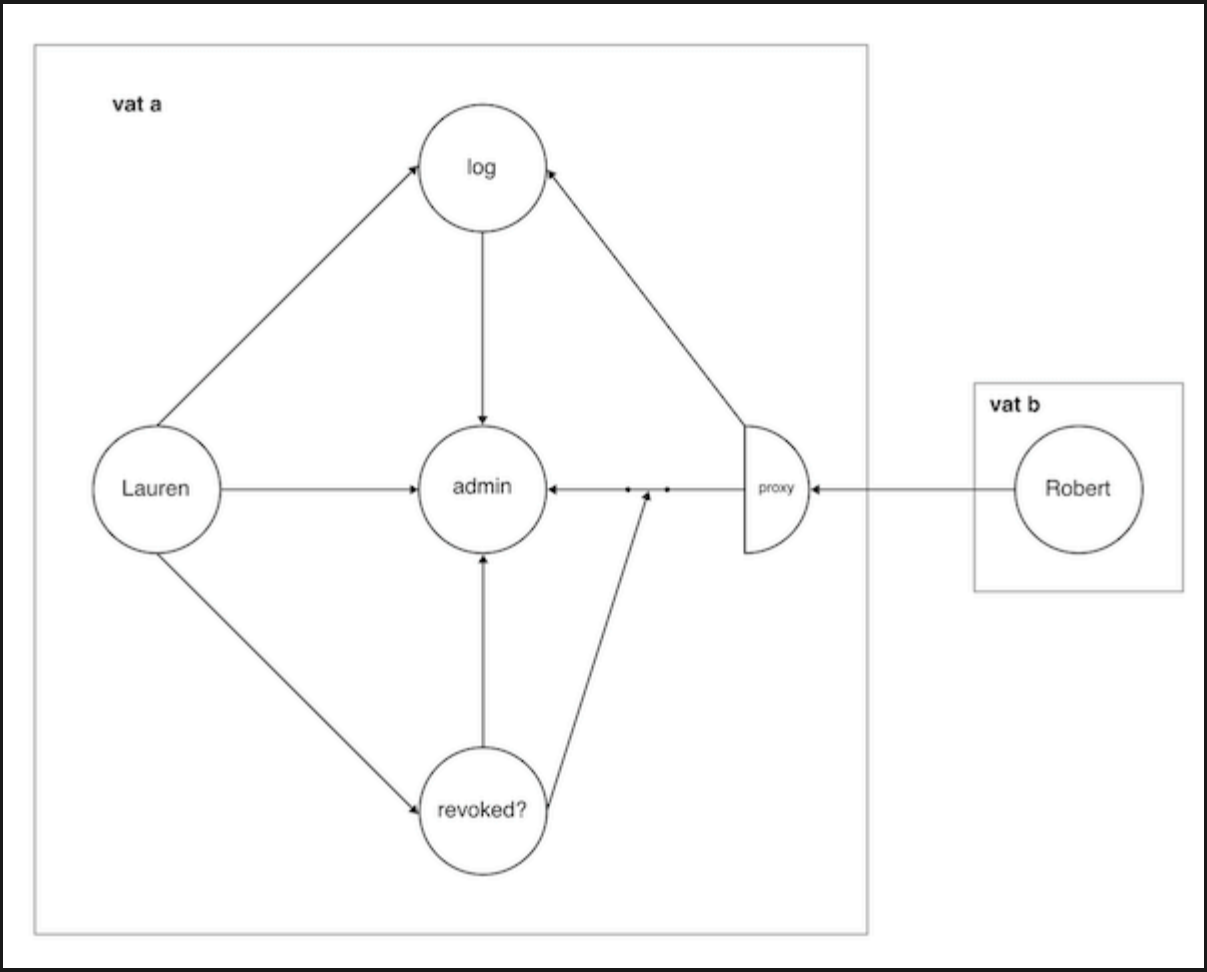
CAPABILITY SECURITY

- “If you don’t have it, you can’t use it”
- Principle of least authority
- ACLs susceptible to confused deputy attacks
- ACLs lend themselves to centralized systems

CAPABILITY SECURITY



CAPABILITY SECURITY



ACTOR MODEL

- Actors communicate with other actors by message passing
- Actors work great concurrently
- Lends itself to capability security approach
- Putting network boundaries within existing actor systems usually trivial
- Fault tolerant

OCAPN: IMPORTS AND EXPORTS

- Actors are imported and exported within a session
- Actors are represented within OCapN as positive integers
- Each session has own set of imports and exports

OCAPN: MESSAGING ACTORS

Operation:

```
<op:deliver to-desc            ; desc:export
            args               ; List
            answer-pos         ; positive integer | false
            resolve-me-desc>   ; desc:import-object | desc:import-pr
```

OCAPN: MESSAGING ACTORS

Goblins:

```
(<-np bob 'hello)
```

OCapN deliver message:

```
<op:deliver <desc:export 1> ; to-desc
            ['hello]        ; arguments
            false           ; answer-pos
            false>          ; resolve-me-desc
```

PROMISES

- Send a message, get a promise back
- Promises may resolve to actor references, concrete values, other promises
- Promises may break with an error
- Promises can be listened to

PROMISES: EXAMPLE BY CODE

```
(define greeting-vow (<- alice 'hello "Bob"))
(on greeting-vow
    (lambda (alice-said)
      (format #f "Alice said: ~a" alice-said)))
```

PROMISES: EXAMPLE

```
;; From A to B
<op:deliver <desc:export 1>           ; to-desc
            ['hello "Bob"]            ; arguments
            false                     ; answer-pos
            <desc:import-object 2>>   ; resolve-me-desc

;; Later on: from B to A
<op:deliver <desc:export 2>
            ['fulfill "Hello Bob, my name is alice"]
            false
            false>
```

PROMISE PIPELINING: CODE

```
(define car-factory-vow (<- factory-builder 'make-factory))
(define car-vow (<- car-factory-vow 'make-car))
(define drive-vow (<- car-vow 'drive))
(on drive-vow
    (lambda (driving-sounds)
      (format #t "Car makes ~a" driving-sounds))
    #:catch
    (lambda (err)
      (format #t "Oh no, something went wrong!")))
```

PROMISE PIPELINING: EXAMPLE

```
;; From peer A to B
<op:deliver <desc:export 5> ['make-factory] 2 false>
<op:deliver <desc:answer 2> ['make-car] 3 false>
<op:deliver <desc:answer 3> ['drive] false <desc:import-object 4>>

;; Later on: B to A
<op:deliver <desc:export 4> ['fulfill "vrooom...."] false false>

;; or an error might have occured!
<op:deliver <desc:export 4> ['break <some-error>] false false>
```

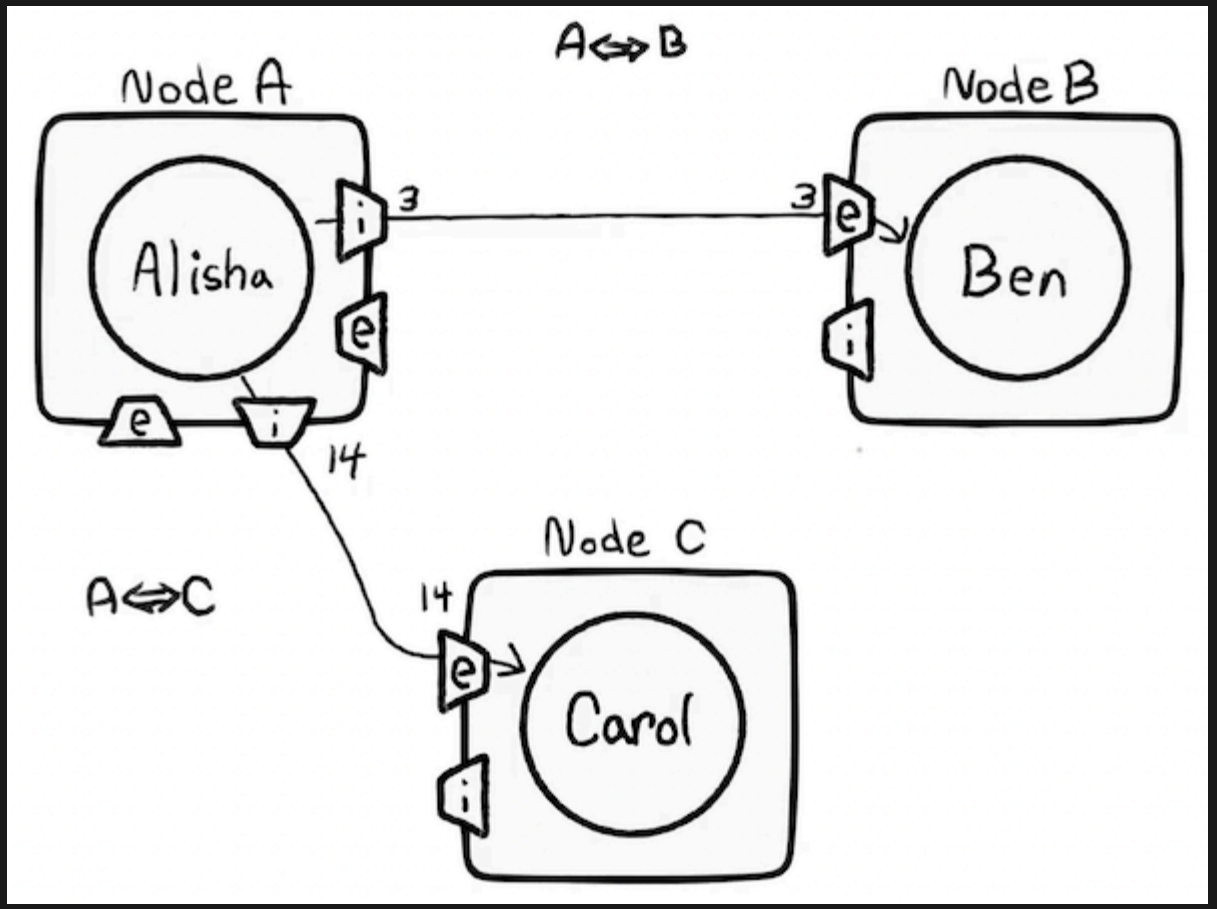
ACYCLIC DISTRIBUTED GC

- We have it!
- Reference count on the wire
- When an object is not needed emit GC operation `op:gc-export` or `op:gc-answer`

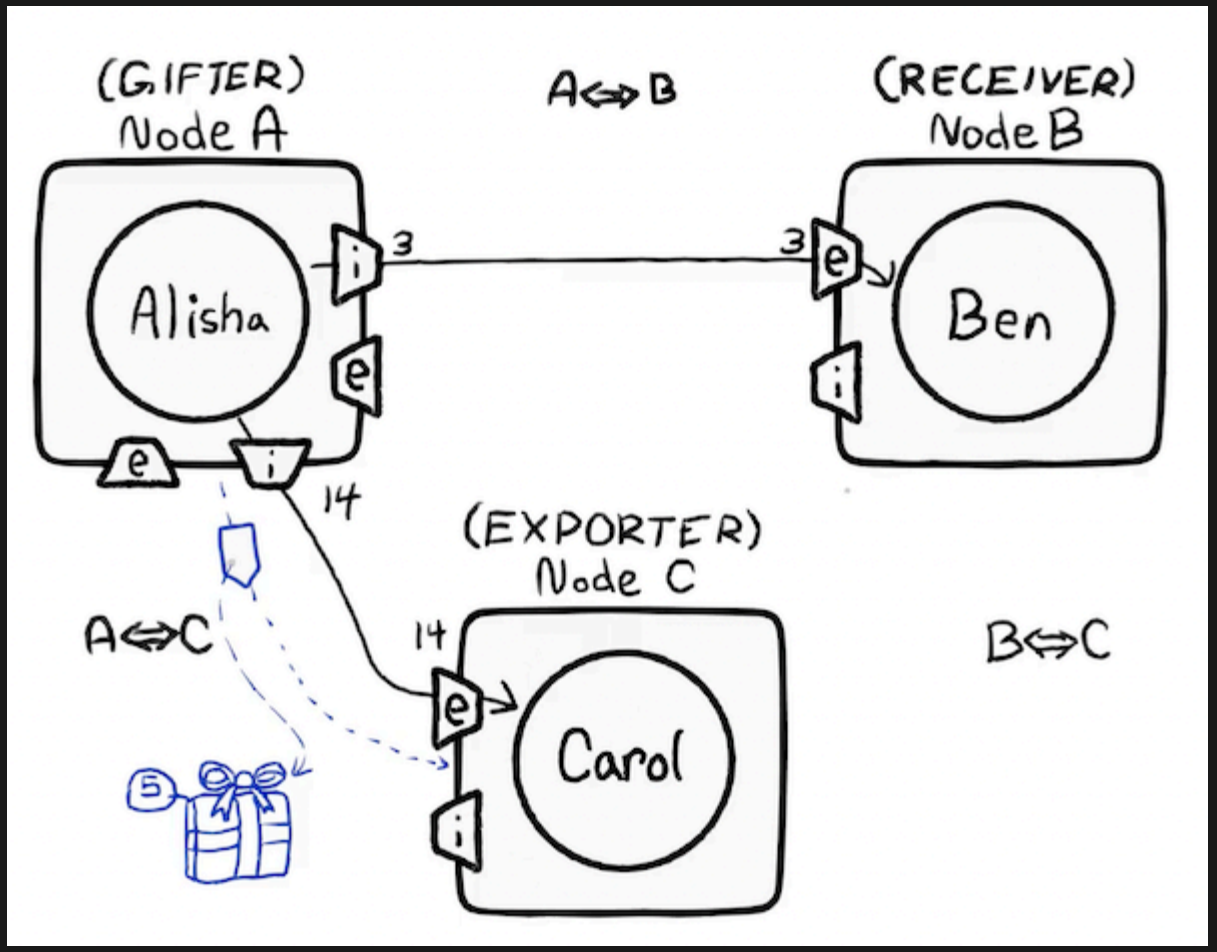
THIRD PARTY HANDOFFS

```
;; From Alisha (Peer A)
;; To: Ben (Peer B)
;; Arguments contain: Carol (Peer C)
(<- ben 'meet carol)
```

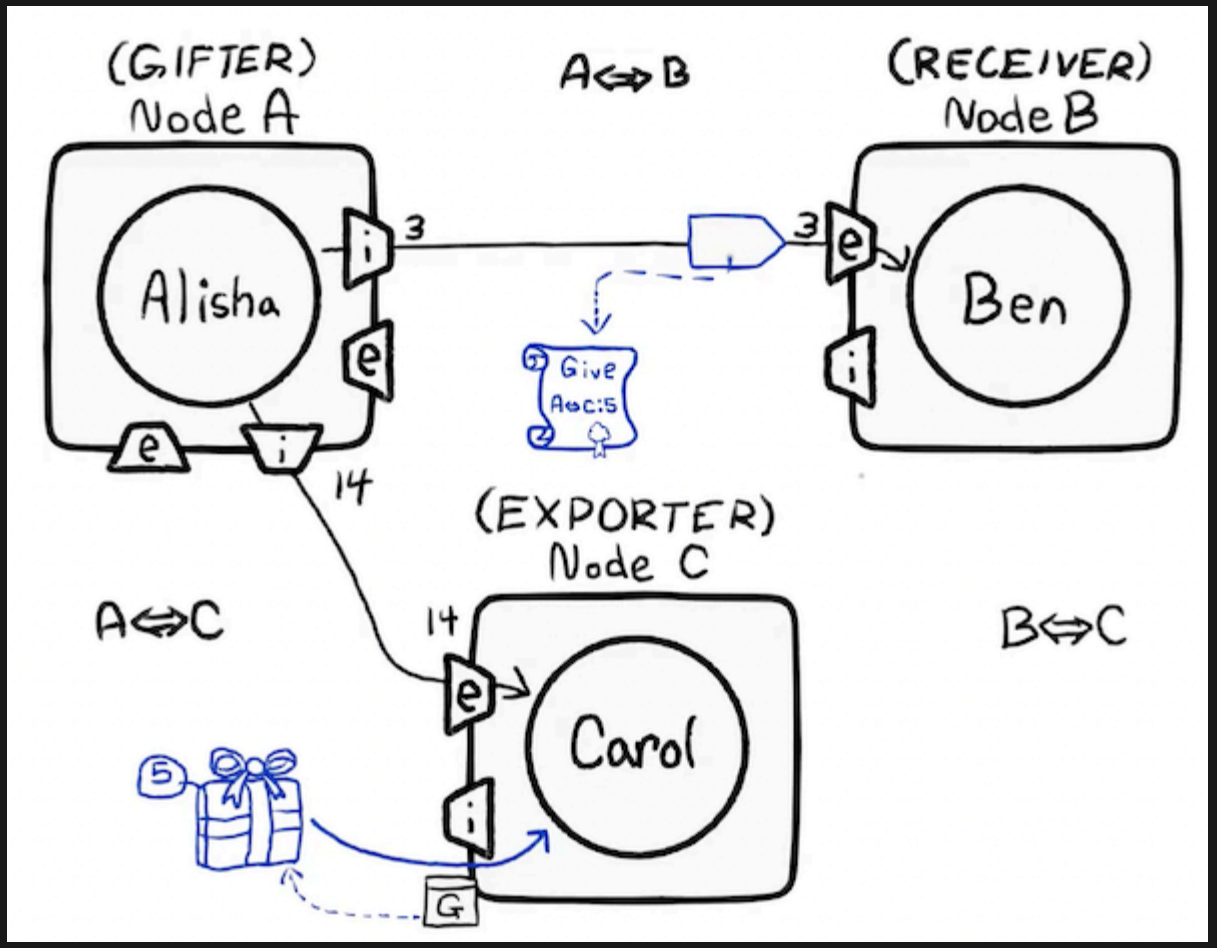
THIRD PARTY HANDOFFS



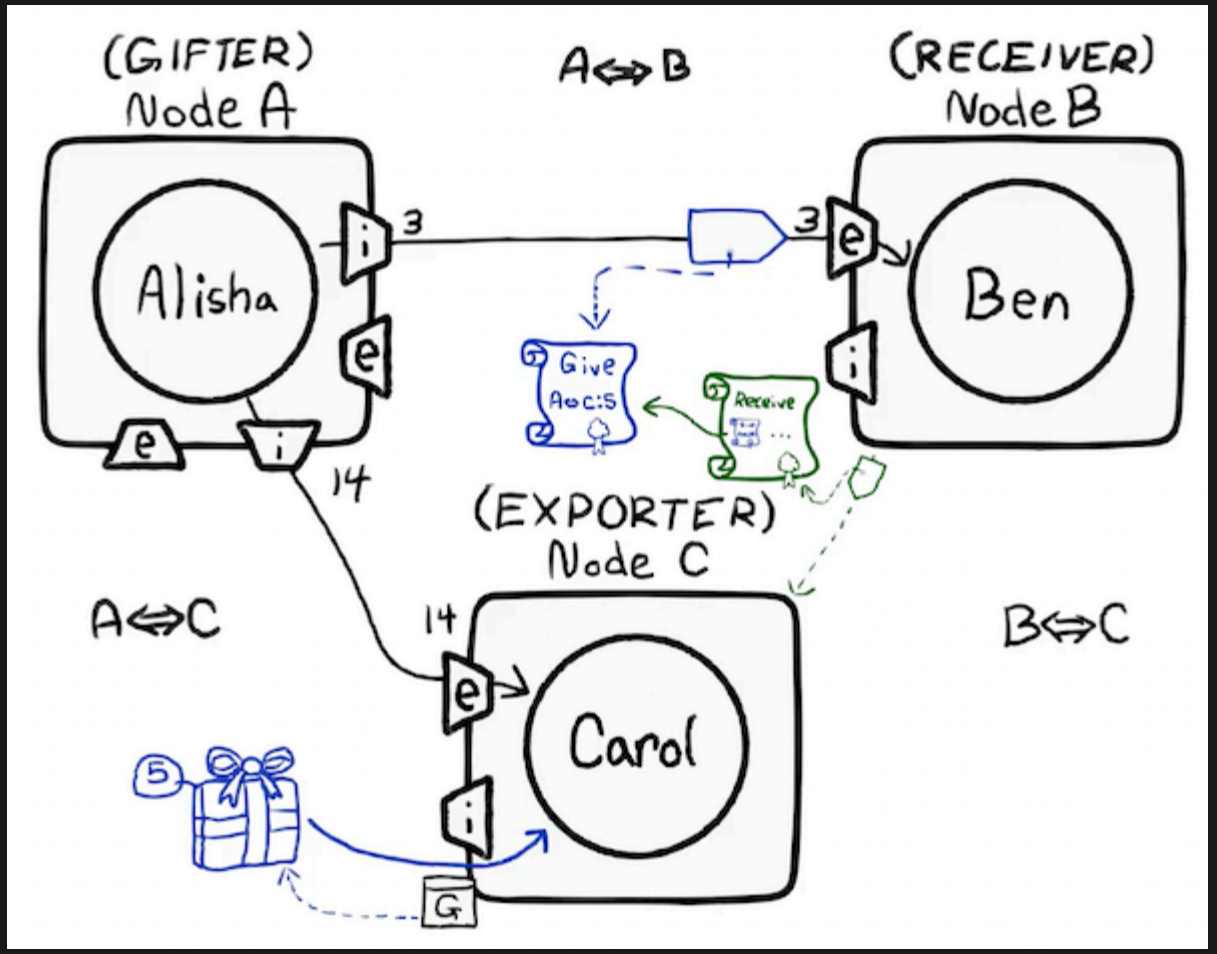
THIRD PARTY HANDOFFS



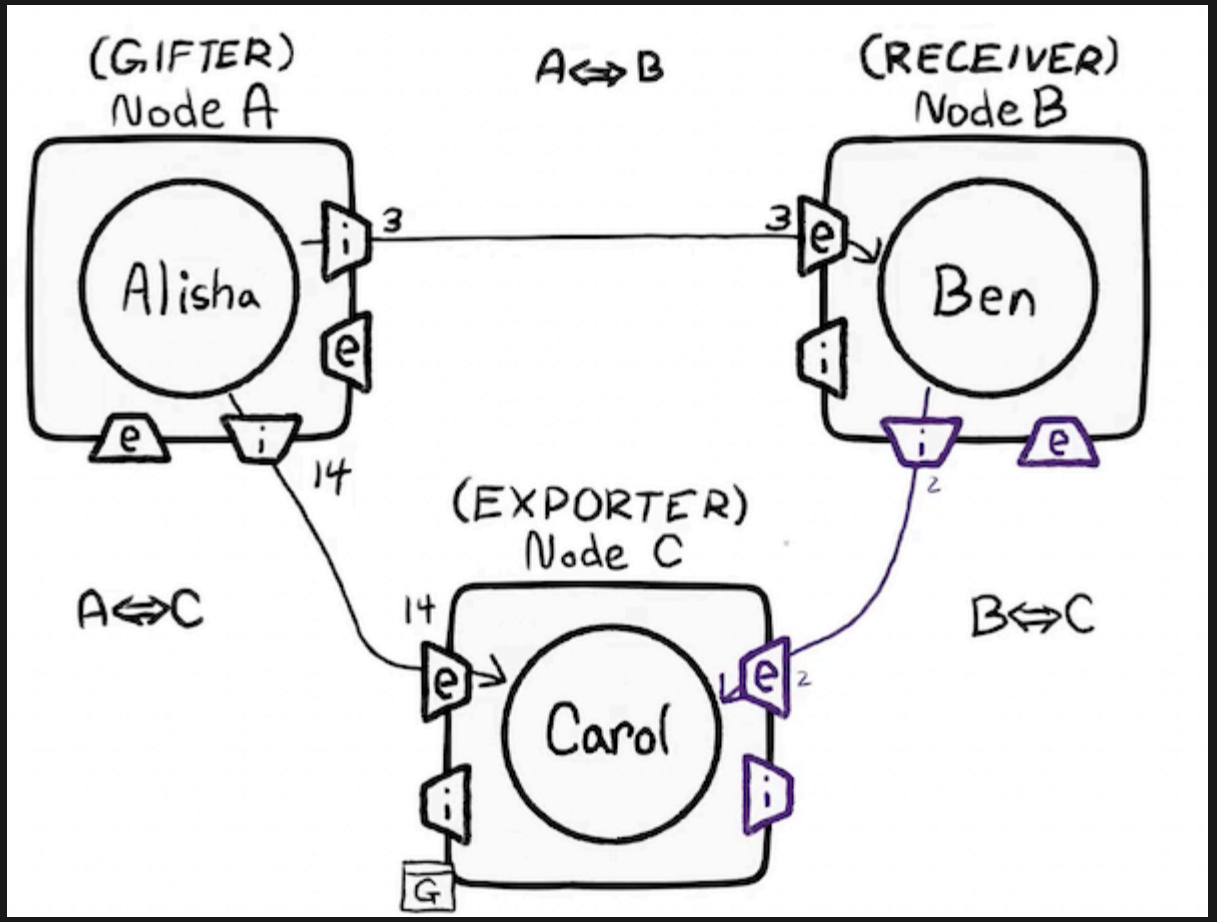
THIRD PARTY HANDOFFS



THIRD PARTY HANDOFFS



THIRD PARTY HANDOFFS



THIRD PARTY HANDOFFS

- Don’t worry if you didn’t understand the quick walkthrough!
- Provides direct peer-to-peer connections instead of proxying
- Uses capability security
- No replay attacks possible
- Provides great developer ergonomics

OCAPN: 3 LAYERS

- **CapTP**: Messaging, Promises, GC, Third party handoffs, errors
- **Netlayers**: agnostic network layer (tor onion, libp2p, websocket, Unix domain sockets, relay, etc)
- **Locators**: Both out-of-band and in-band locations of peers and actors

RELAY

- Peer-to-peer can be difficult in some environments (browsers, behind NATs, etc)
- Building a federated relay
- Built by layering netlayers that exist

OCAPN: STATUS?

- Several implementations (Racket Scheme, Guile Scheme, JS and Dart)
- Pre-standardization phase with OCapN group
- Have draft specifications and implementation guide
- Have a test suite written in Python

QUESTIONS

Any questions?

Also thanks to NLnet for funding a lot of the work