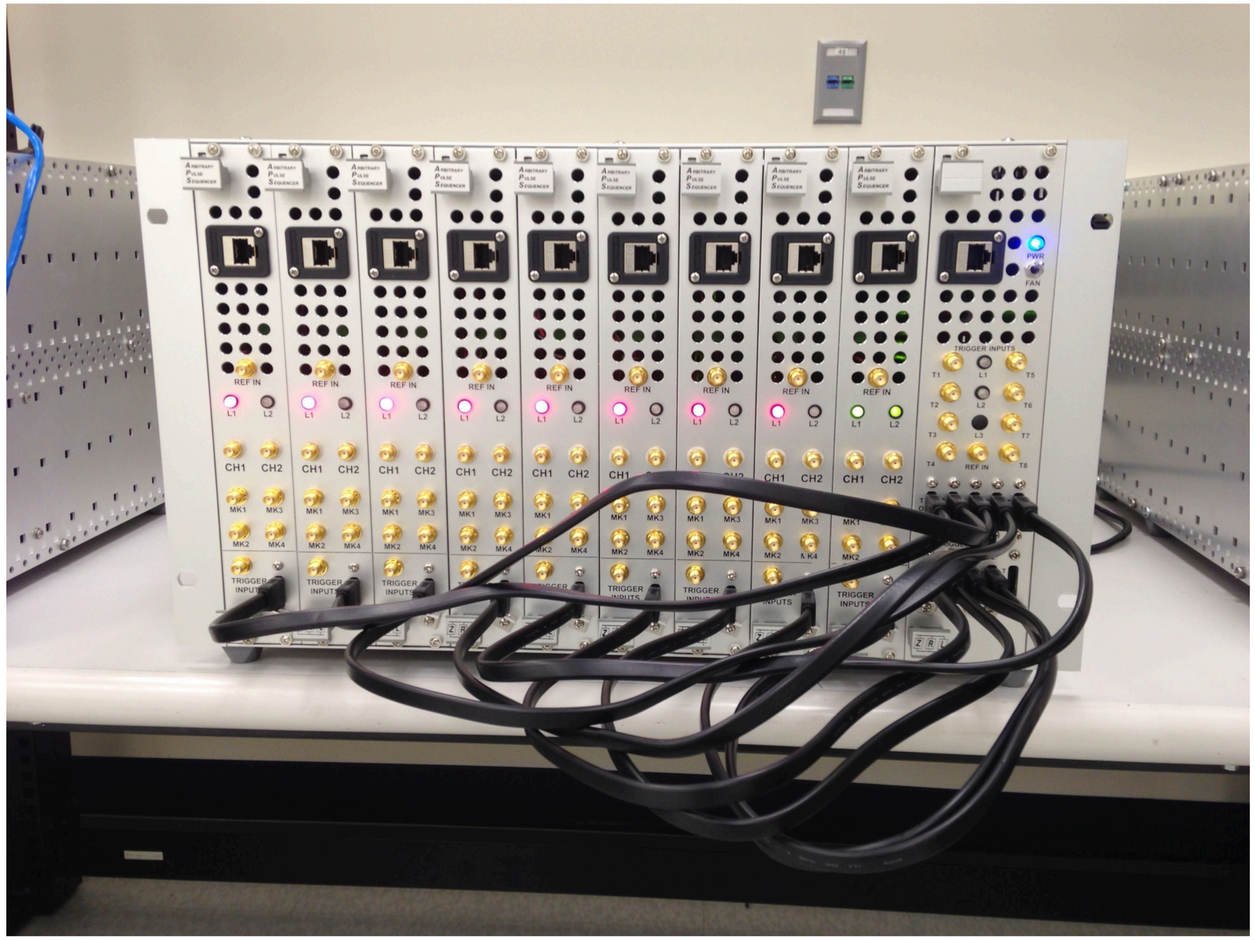
aether

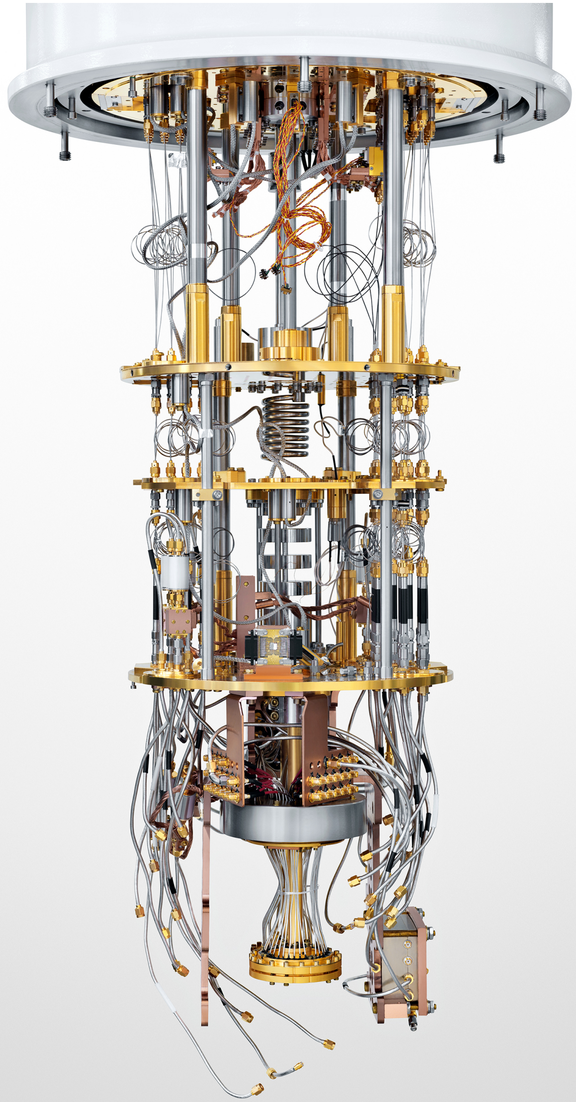
Distributed system emulation in Common Lisp

Eric Peterson and Peter Karalekas @ European Lisp Symposium XIV, 3 May 2021

Motivation: Steering quantum electronics



↑ Typical control electronics, Ryan et al., BBN

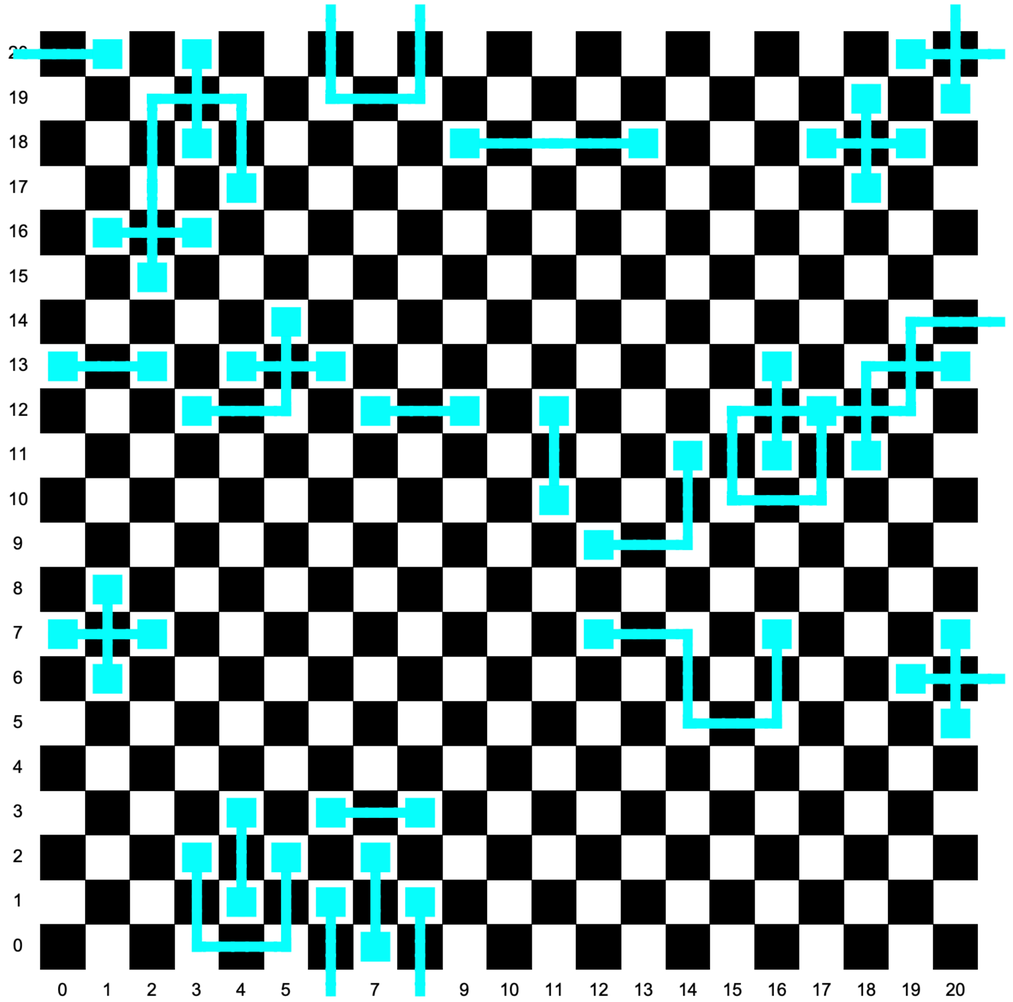


“Chandelier”, Rigetti Comp. →

Motivation: Steering quantum electronics

The basic problem

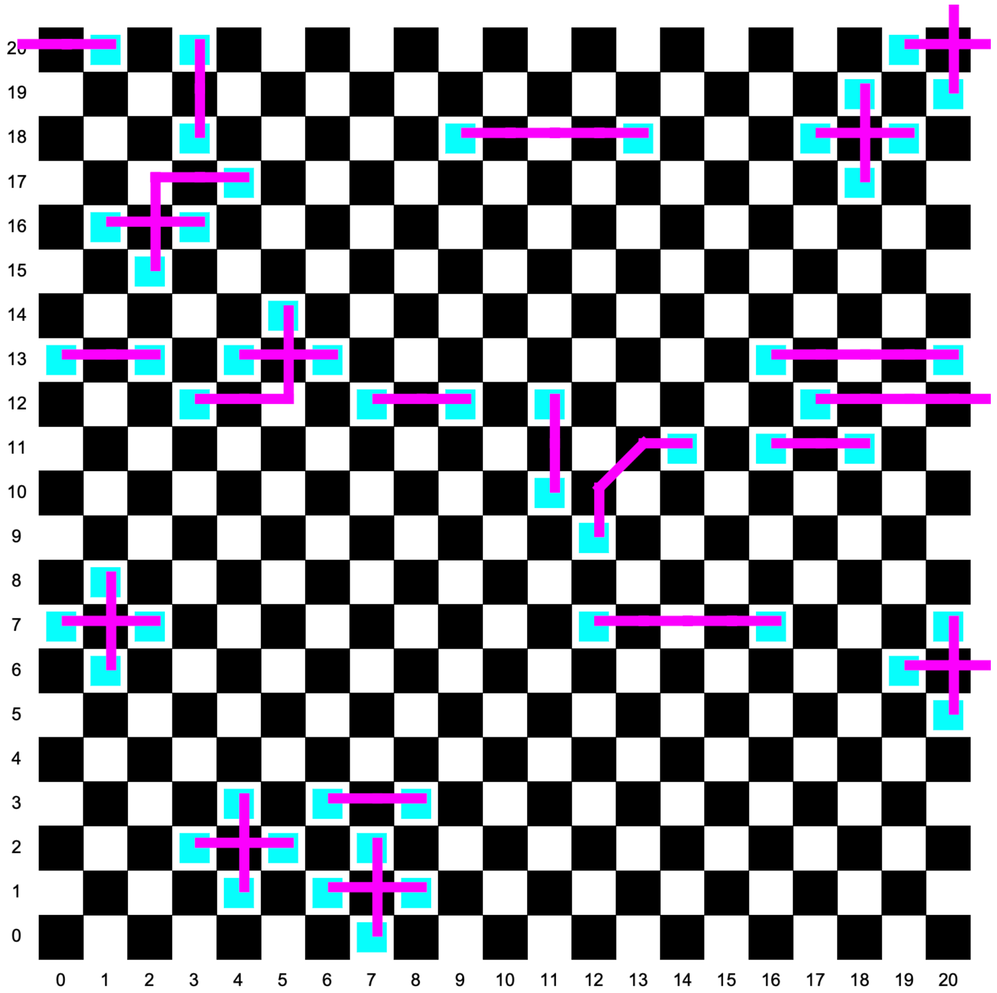
- **Situation:** Quantum errors look like lines, but the electronics only get notified about endpoints.



Motivation: Steering quantum electronics

The basic problem

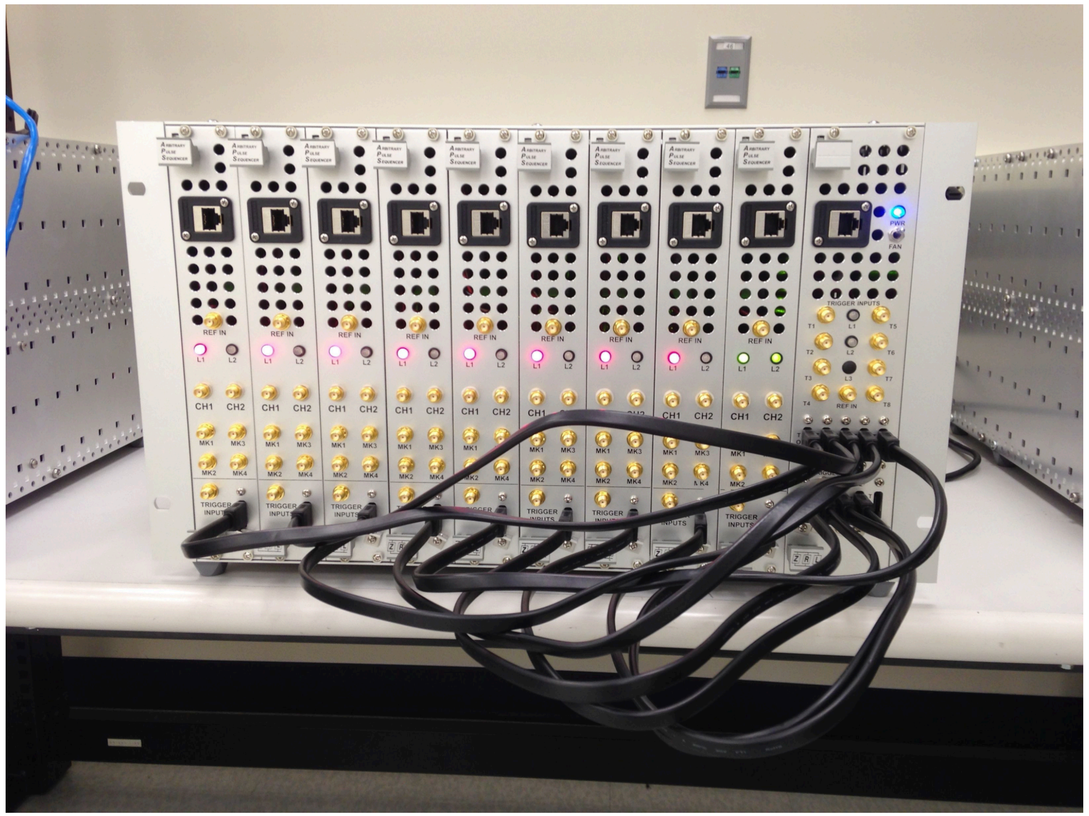
- **Situation:** Quantum errors look like lines, but the electronics only get notified about endpoints.
- **Task:** Reconstruct the lines given just the endpoints. (Ambiguous, but tolerates approximate answers.)



Motivation: Steering quantum electronics

The proposed solution: the Blossom algorithm, sorta

- **Situation:** Quantum errors look like lines, but the electronics only get notified about endpoints.
- **Task:** Reconstruct the lines given just the endpoints. (Ambiguous, but tolerates approximate answers.)
- **Engineering:** Run on this kind of system → and maintain bounded resource use even as it scales.



Towards aether

Objectives

- **Software:** Make precise Fowler's proposal for an algorithmic solver
- **Hardware:** Emulate its execution on "generic" hardware
- **Instrumentation:** Check the claimed properties w/r/t resource usage
- **More hardware:** Increase emulation fidelity, uncover architecture restrictions

aether

Distributed system emulation in CL

- **Time-domain simulation:** Flexibly process discrete time-ordered events
- **Networking:**
  - “Physical” courier layer for simulating congestion, routing, …
  - “Logical” message layer for robust communication between actors
- **Actor framework:** Heterogeneous components in communication
  - Describe transitions for individual hardware components
  - Describe hardware-opaque application components

aether by example

Coloring a line w/ three colors

```
(defclass process-coloring (process)
  ((color     :type (integer 3) ...)
   (neighbors :type list ...)))  ; of addresses
```

aether by example

Coloring a line w/ three colors

```
(defclass process-coloring (process)
  ((color     :type (integer 3) ...)
   (neighbors :type list ...)))  ; of addresses

(defstruct (message-color-query
            (:include message)))

(define-rpc-handler handle-message-color-query
    ((process process-coloring)
     (message message-color-query)
     now)
  (process-coloring-color process))

(define-message-dispatch process-coloring
  (message-color-query
   'handle-message-color-query))
```

aether by example

Coloring a line w/ three colors

```
(defclass process-coloring (process)
  ((color     :type (integer 3) ...)
   (neighbors :type list ...)))  ; of addresses

(defstruct (message-color-query
            (:include message)))

(define-rpc-handler handle-message-color-query
    ((process process-coloring)
     (message message-color-query)
     now)
  (process-coloring-color process))

(define-message-dispatch process-coloring
  (message-color-query
   'handle-message-color-query))

(define-process-upkeep
    ((process process-coloring) now) (START)
  (process-continuation process `(QUERY)))
```

```
(define-process-upkeep
    ((process process-coloring) now) (IDLE)
  (process-continuation process `(IDLE)))
```

aether by example

Coloring a line w/ three colors

```
(defclass process-coloring (process)
  ((color     :type (integer 3) ...)
   (neighbors :type list ...)))  ; of addresses

(defstruct (message-color-query
            (:include message)))

(define-rpc-handler handle-message-color-query
    ((process process-coloring)
     (message message-color-query)
     now)
  (process-coloring-color process))

(define-message-dispatch process-coloring
  (message-color-query
   'handle-message-color-query))

(define-process-upkeep
    ((process process-coloring) now) (START)
  (process-continuation process `(QUERY)))
```

```
(define-process-upkeep
    ((process process-coloring) now) (IDLE)
  (process-continuation process `(IDLE)))

(define-process-upkeep
    ((process process-coloring) now) (QUERY)
  (let (listeners)
    (with-slots (color neighbors)
        process
      (setf color (random 3))
      (setf listeners
            (send-message-batch
             #'make-message-color-query
             neighbors))
      (with-replies (replies) listeners
        (when (member color replies)
          (process-continuation process `(QUERY))
          (finish-with-scheduling))
        (process-continuation process `(IDLE))))))
```

aether by example

Coloring a line w/ three colors

```
(dolist (node-count '(2 4 8 16 32 64 128 256 512))
  ;; ... loop over trials for statistical average ...

  (let (couriers nodes simulation canaries)
    ;; ... instantiate sim, couriers ...

    ;; install courier events
    (loop :for courier :across couriers
          :do (simulation-add-event simulation
                (make-event :callback courier
                            :time 0)))

    ...))
```

aether by example

Coloring a line w/ three colors

```
(dolist (node-count '(2 4 8 16 32 64 128 256 512))
  ;; ... loop over trials for statistical average ...

  (let (couriers nodes simulation canaries)
    ;; ... instantiate sim, couriers ...

    ;; install courier events
    (loop :for courier :across couriers
          :do (simulation-add-event simulation
                (make-event :callback courier
                            :time 0)))

    ;; build nodes within couriers
    (dotimes (j node-count)
      (let ((*local-courier* (aref couriers j)))
        (setf (aref nodes j)
              (spawn-process 'process-coloring))
        (simulation-add-event
          simulation
          (make-event :callback (aref nodes j)
                      :time 0))))

   ...))
```

aether by example

Coloring a line w/ three colors

```
(dolist (node-count '(2 4 8 16 32 64 128 256 512))
  ;; ... loop over trials for statistical average ...

  (let (couriers nodes simulation canaries)
    ;; ... instantiate sim, couriers ...

    ;; install courier events
    (loop :for courier :across couriers
          :do (simulation-add-event simulation
                (make-event :callback courier
                            :time 0)))

    ;; build nodes within couriers
    (dotimes (j node-count)
      (let ((*local-courier* (aref couriers j)))
        (setf (aref nodes j)
              (spawn-process 'process-coloring))
        (simulation-add-event
          simulation
          (make-event :callback (aref nodes j)
                      :time 0))))

    ;; ... set up each node's neighbors ...
```

```
;; define simulation stopping condition
(dolist (node nodes)
  (push (lambda (now)
          (equalp `((IDLE))
                  (process-command-stack node)))
        canaries))

;; spin up simulation, run until everyone's stopped
(simulation-run
   simulation
   :canary (apply #'canary-all canaries))

...))
```

aether by example

Coloring a line w/ three colors

```
(dolist (node-count '(2 4 8 16 32 64 128 256 512))
  ;; ... loop over trials for statistical average ...

  (let (couriers nodes simulation canaries)
    ;; ... instantiate sim, couriers ...

    ;; install courier events
    (loop :for courier :across couriers
          :do (simulation-add-event simulation
                (make-event :callback courier
                            :time 0)))

    ;; build nodes within couriers
    (dotimes (j node-count)
      (let ((*local-courier* (aref couriers j)))
        (setf (aref nodes j)
              (spawn-process 'process-coloring))
        (simulation-add-event
          simulation
          (make-event :callback (aref nodes j)
                      :time 0))))

    ;; ... set up each node's neighbors ...

    ;; define simulation stopping condition
    (dolist (node nodes)
      (push (lambda (now)
              (equalp `((IDLE))
                      (process-command-stack node)))
            canaries))

    ;; spin up simulation, run until everyone's stopped
    (simulation-run
       simulation
       :canary (apply #'canary-all canaries))

    ;; get the stopping time
    (simulation-horizon simulation)))
```

```
; Coloring   4 nodes took  8.952 ticks on average (σ = 6.7618)
; Coloring   8 nodes took 13.898 ticks on average (σ = 8.1636)
; Coloring  16 nodes took 20.114 ticks on average (σ = 10.161)
; Coloring  32 nodes took 24.602 ticks on average (σ = 10.003)
; Coloring  64 nodes took 30.482 ticks on average (σ = 10.125)
; Coloring 128 nodes took 34.502 ticks on average (σ = 9.1873)
; Coloring 256 nodes took 39.608 ticks on average (σ = 9.9049)
; Coloring 512 nodes took 44.954 ticks on average (σ = 9.9954)
```

`aether` in Practice

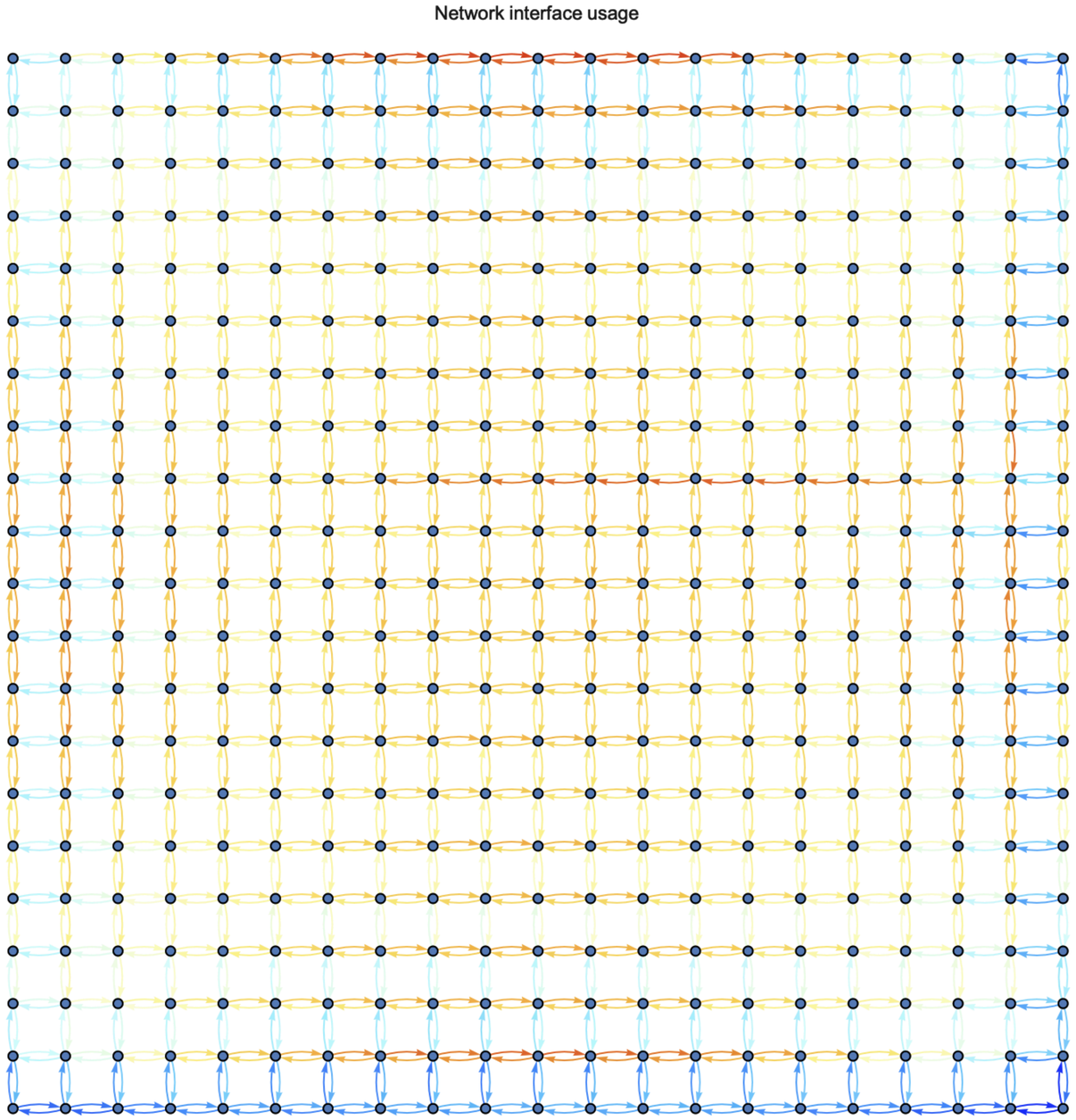
Debugging

- **Swank / SLIME:**
  - Common Lisp already has a really nice debugging system!
  - Direct manipulation of world state when you want it
- **Structured logging:**
  - Programmatically trace call effects
  - Inspect temporal ordering / temporal windows
- **Dereferencing:**
  - Break address / actor opacity
- **Time-domain manipulation:**
  - Inject new events and latency to explore race conditions

aether in Practice

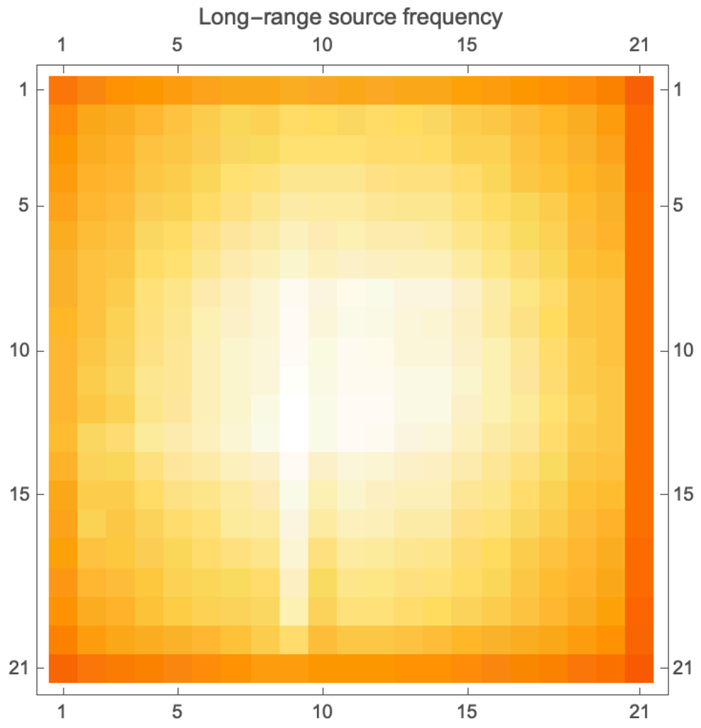
Instrumentation

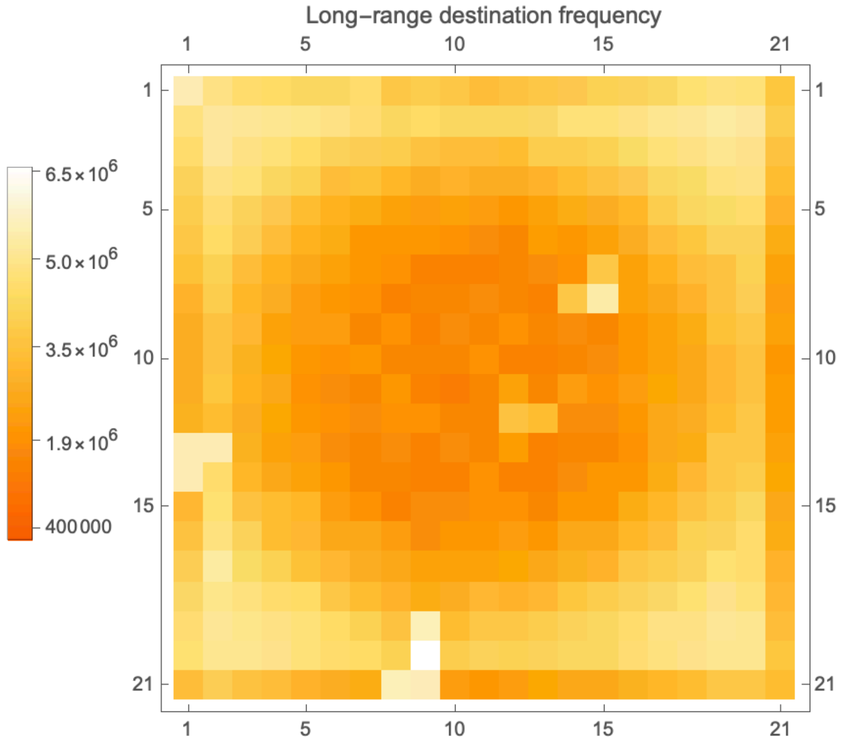
- Network pressure:
  - Message counting
  - Message queue depth
  - Interface use
- Computational pressure:
  - Live (/ non-blocked) process count
  - Actor command hit counts

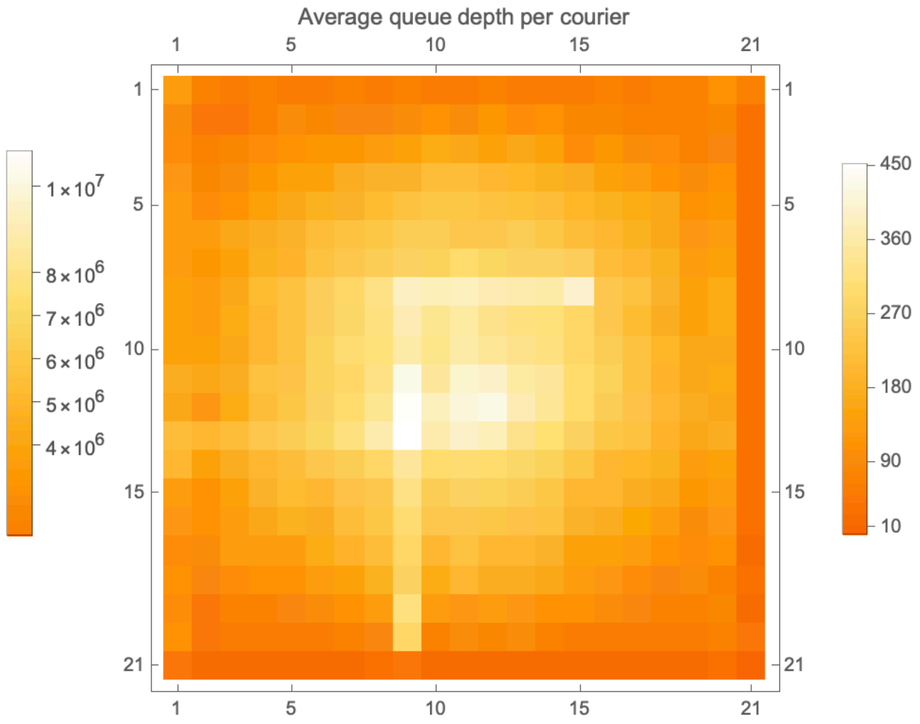


aether in Practice

Instrumentation







Thank you!

https://github.com/dtqec/aether