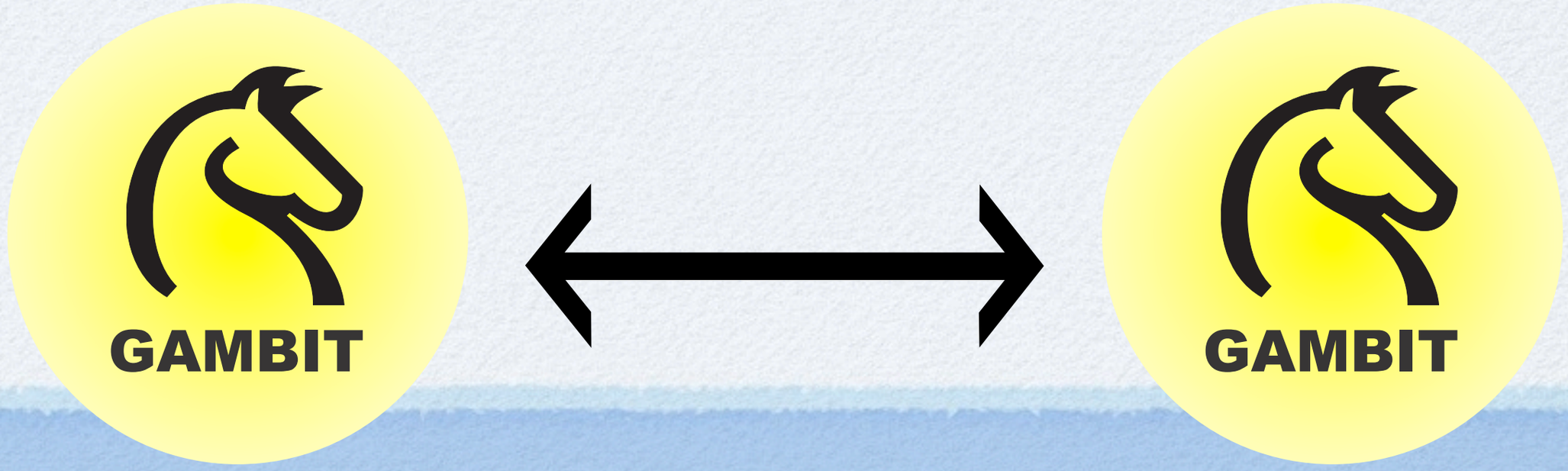


Distributed Computing with Gambit

Marc Feeley

November 24, 2011

Université de Montréal

Overview

- "Teleportation" Distributed Computing demo
- Showcases advanced Gambit Scheme features:
  - Networking
  - Threads
  - Serialization
  - Continuations

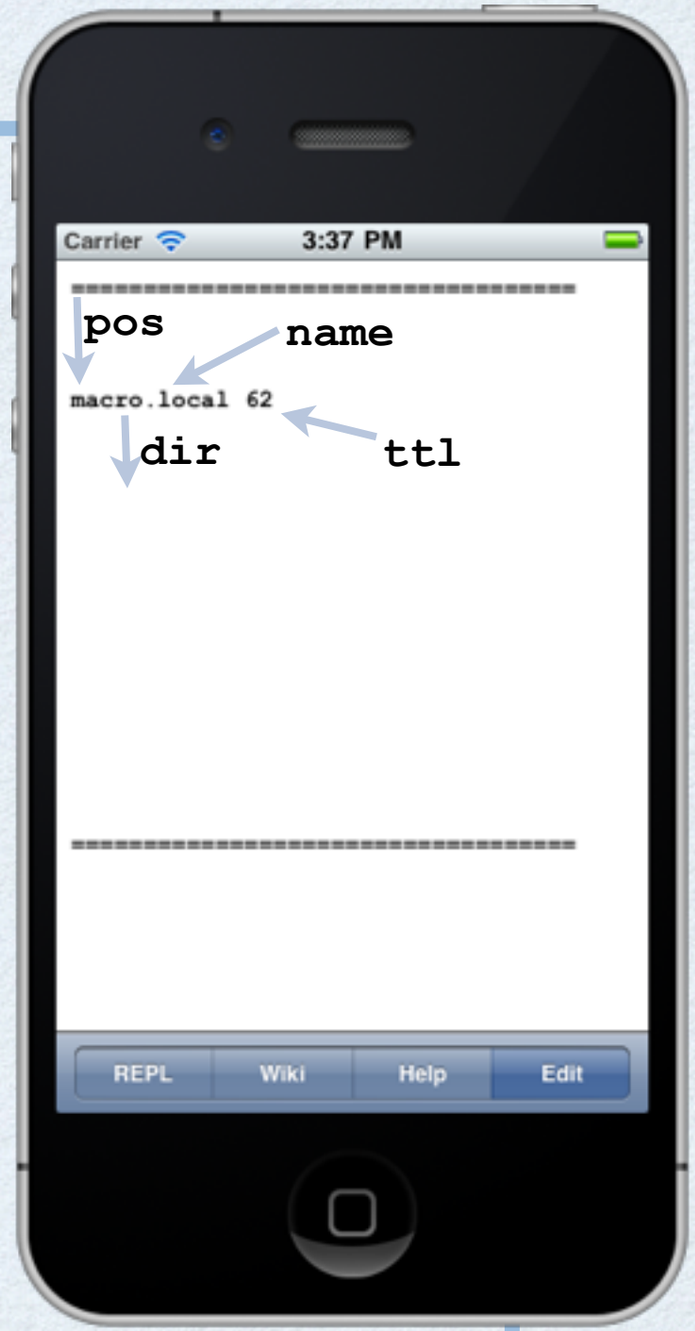
Demo

- Each ***node*** on a LAN spawns a new thread every 20 secs
- The threads ***bounce*** within the screen
- When a thread bounces, it tries to ***teleport*** to the screen of another node
- Threads die after 100 steps

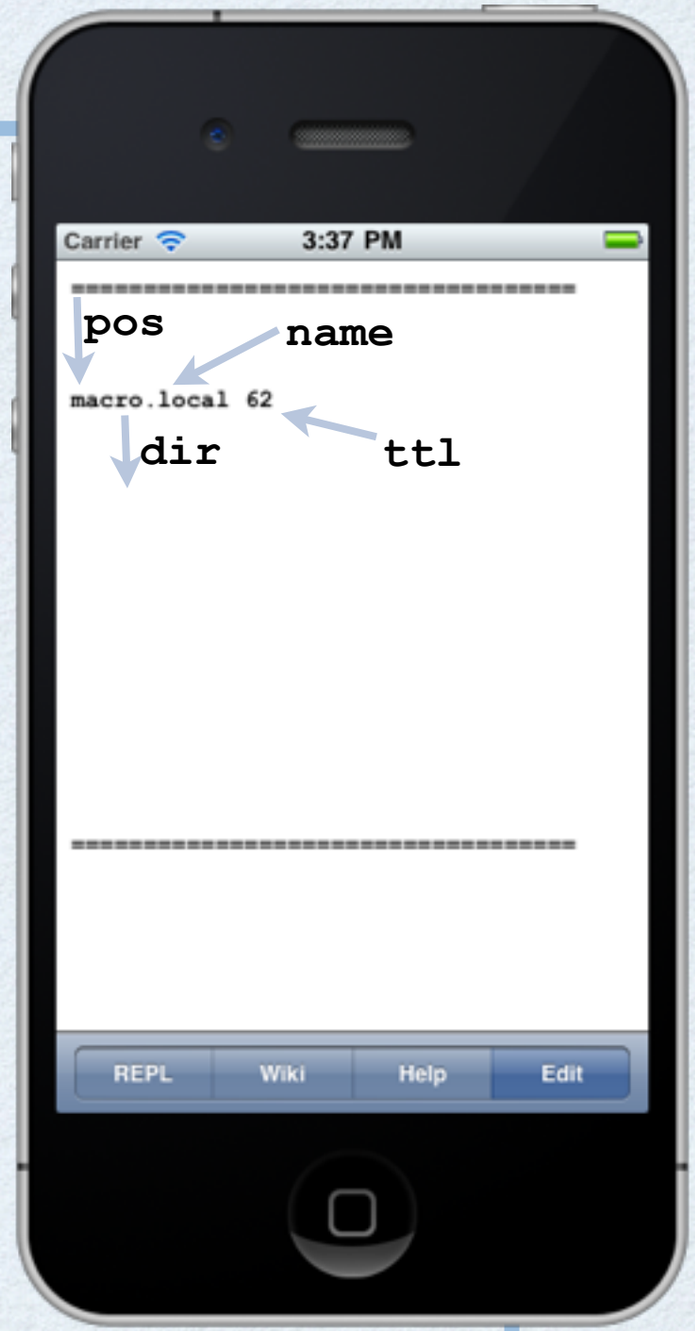
Top-down Code Review

```
(define (go)

  (define (bounce period name)
    ...)

  ;; spawn a bouncing thread every 20 secs

  (let loop ()

    (spawn
     (bounce (+ .2 (* .2 (random-real)))
             (host-name)))

    (sleep 20)

    (loop)))


(go)
```

```
;; Single-node version (no teleportation)

(define (bounce period name)
  (let loop ((pos 2) (dir 1) (ttl 100))
    (if (> ttl 0)
        (begin

          (set-row pos (list name " " ttl))
          (sleep period)
          (set-row pos "")

          (if (and (> pos top) (< pos bot))

              ;; not on fence so just move
              (loop (+ pos dir) dir (- ttl 1))

              ;; must bounce
              (begin


                (loop (- pos dir) (- dir) (- ttl 1)))))))
```



```
;; Multi-node version (with teleportation)

(define (bounce period name)
  (let loop ((pos 2) (dir 1) (ttl 100))
    (if (> ttl 0)
        (begin

          (set-row pos (list name " " ttl))
          (sleep period)
          (set-row pos "")

          (if (and (> pos top) (< pos bot))

              ;; not on fence so just move
              (loop (+ pos dir) dir (- ttl 1))

              ;; must bounce
              (begin

                (let ((n (random-node)))
                  (if (not (eq? n current-node))
                      (begin (swoosh) (teleport n) (bell))))

                (loop (- pos dir) (- dir) (- ttl 1)))))))
```



```
(define (teleport destination)

  ;; get the thread's continuation

  (continuation-capture
   (lambda (k)
     (without-exception ;; guard against rpc failure
      (begin

        ;; try to resume continuation in a new thread
        ;; at the destination in no more than 5 seconds

        (rpc destination
             (begin
               (spawn (continuation-return k #t))
               'done)
             5)

        ;; if teleportation went OK, kill
        ;; the original thread

        (halt))))))
```

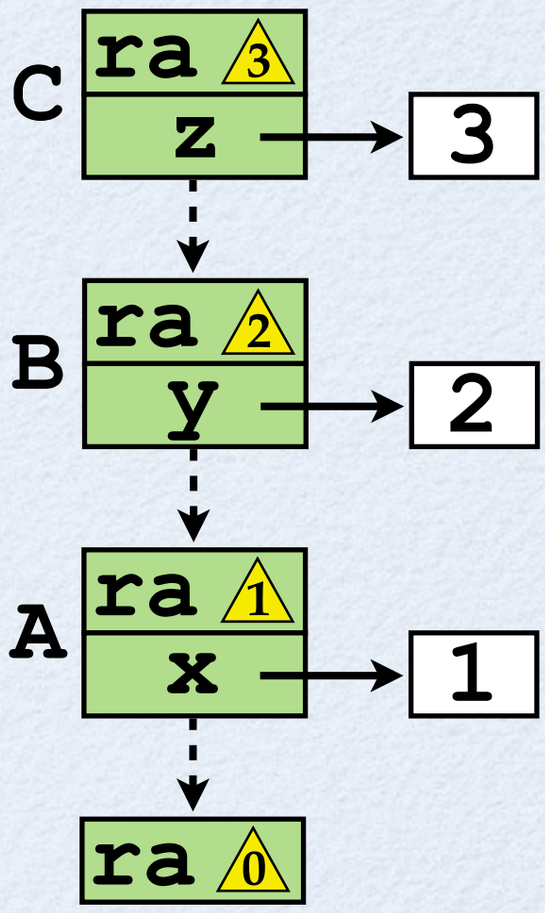
Continuations

- A *continuation* is an object which represents **"the rest of the computation"**
- In most languages, continuations are hidden and correspond to the **call stack**
- In Scheme, continuations are *reified*, meaning the programmer can obtain, store and invoke them explicitly
- In standard Scheme, this is done using **call/cc**
- Gambit has another API: **continuation-capture**

Cont. Example (part 1)

```
(define (C z)
  ...)

(define (B y)
  (* (C 3) y))
          3

(define (A x)
  (+ (B 2) x))
          2

(A 1)
     1
```

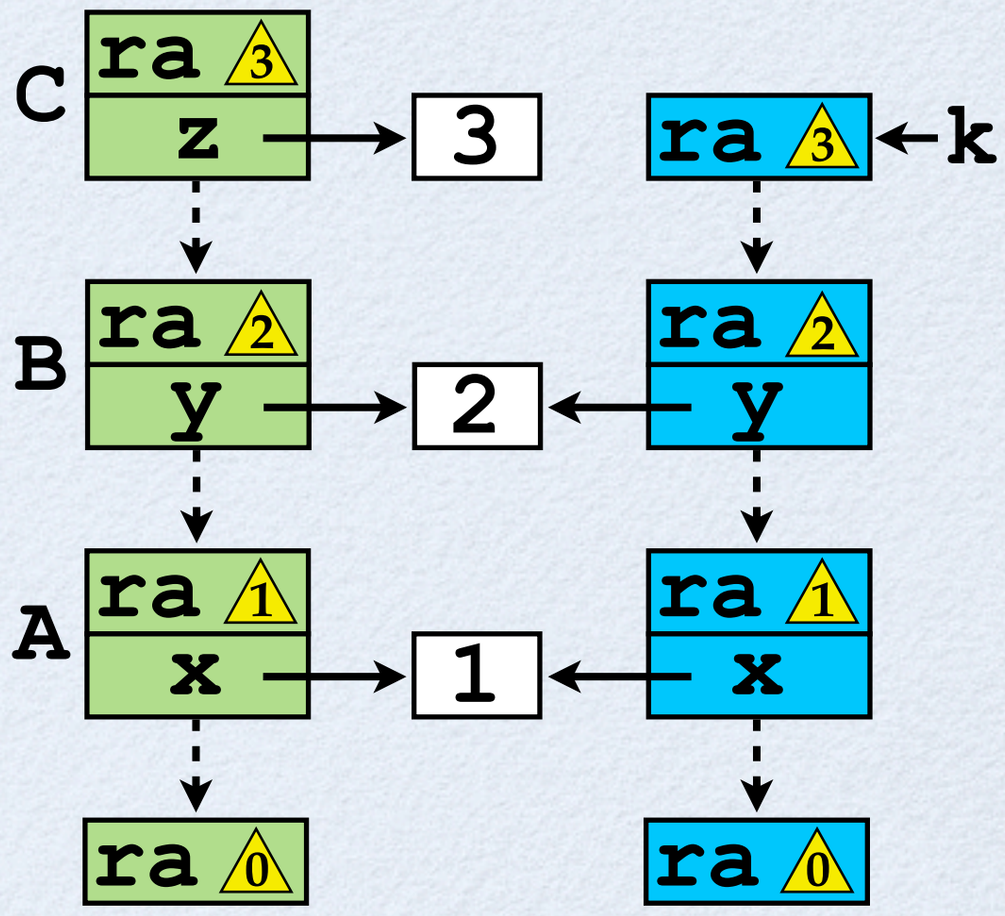
Runtime stack



Cont. Example (part 2)

```
(define (C z)
  (continuation-capture
    (lambda (k)
      (continuation-return
        k
        (* z z)))))

(define (B y)
  (* (C 3) y))
             3

(define (A x)
  (+ (B 2) x))
             2

(A 1)
     1
```

Runtime stack



Continuation Serialization

- The implementation of the **`rpc`** macro must ***serialize*** the continuation **`k`** to send it to the destination (where it will be ***deserialized***)

```
(rpc destination
     (begin
       (spawn (continuation-return k #t))
       'done)
     5)
```

- For this to be possible, all the objects reachable from the continuation must be serializable

Continuation Serialization

- Unfortunately, continuations may contain objects that are not serializable:
  - Foreign data (such as ptrs to C structures)
  - I/O ports (if they refer to an OS resource)
  - Threads (they indirectly refer to all other threads)
- In the teleportation demo: **current input/ output ports** and **nodes** (which are threads)

Object Serialization

- Serialization/deserialization is done with
  - **`(object->u8vector`** *obj* [*encoder*]**`)`**
  - **`(u8vector->object`** *u8vect* [*decoder*]**`)`**

- The optional *encoder*/*decoder* functions allow giving a serialization semantics to objects which would not be otherwise serializable

- These functions are called on every sub-object during the DFS traversal

```
> (define (pprinter port)
    (lambda (x) (pp x port)))

> (define p
    (pprinter (current-output-port)))

> (p '(a b c))
(a b c)

> (define v (object->u8vector p))
*** ERROR IN ##object->u8vector
  -- can't serialize #<mutex #2 #f>
```

```
;; Map current input/output ports to those of the
;; destination.

(define-type ser-stdin
  id: FA2D330A-F35B-4B81-A5AD-D75B742D687D)

(define-type ser-stdout
  id: CB3FD64E-2734-4D9E-BAB5-463029CC9F40)

(define (encode obj)
  (object->u8vector
   obj
   (lambda (x)
     (cond ((eq? x (current-input-port))   (make-ser-stdin))
           ((eq? x (current-output-port)) (make-ser-stdout))
           (else                          x)))))

(define (decode u8v)
  (u8vector->object
   u8v
   (lambda (x)
     (cond ((ser-stdin? x)   (current-input-port))
           ((ser-stdout? x) (current-output-port))
           (else            x)))))
```

```
> (define (pprinter port)
    (lambda (x) (pp x port)))

> (define p
    (pprinter (current-output-port)))

> (define v (encode p))

> (define q (decode v))

> (q '(a b c))
(a b c)

> (u8vector-length v)
4359
```

Node Discovery

- Each node has a manager thread which listens for incoming TCP connections on port 12345
- Each node periodically searches the LAN for other nodes with an open port 12345
- For efficiency, the search is done concurrently (100 concurrent connections)
- A broadcast would be more efficient, but it is not supported by Gambit

```
;; Search LAN concurrently for DCS servers.

(define (discover-local-DCS-servers ip found)
  (let* ((nm #xffffff00)
         (throttle (make-throttle 100)))
    (pfor 1
          (- #xffffffff nm)
          (lambda (i)
            (throttle
             (lambda ()
               (check-for-DCS-server
                (num->ip
                  (+ i (bitwise-and nm (ip->num ip))))
                found)))))))

(define (check-for-DCS-server ip found)
  (let ((conn (DCS-connect ip)))
    (if conn
        (found conn))
    (sleep 0.5)))
```

RPC

- RPC call sends a closure and receives a closure
- The source node contacts the destination node and sends it a (serialized) closure which is called at the destination in a new thread
- The destination returns a closure, which, when called, returns the computed result or raises the exception that was raised at the destination

```
;; RPC examples.

> (nodes)
(#<thread #3> #<thread #4> #<thread #2>)

> (table->list ip2node)
((#u8(192 168 0 100) . #<thread #2>)
 (#u8(192 168 0 102) . #<thread #3>)
 (#u8(192 168 0 103) . #<thread #4>))

> (map (lambda (n) (rpc n (host-name)))
       (nodes))
("macro.local" "Marc-Feeleys-iPod" "mega.local")

> (rpc #4 (host-name))
"Marc-Feeleys-iPod"

> (rpc #4 (/ 1 0))
*** ERROR IN (console)@19.1 -- Divide by zero
(/ 1 0)
1>
```

Questions?