8sync

8sync, asynchronous actors for Guile

Christopher Allan Webber

Table of Contents

1 Preface

Welcome to 8sync's documentation! 8sync is an asynchronous programming environment for GNU Guile. (Get it? 8sync? Async??? Quiet your groans, it's a great name!)

8sync has some nice properties:

- 8sync uses the actor model as its fundamental concurrency synchronization mechanism. Since the actor model is a "shared nothing" asynchronous environment, you don't need to worry about deadlocks or other tricky problems common to other asynchronous models. Actors are modular units of code and state which communicate by sending messages to each other.
- If you've done enough asynchronous programming, you're probably familiar with the dreaded term "callback hell". Getting around callback hell usually involves a tradeoff of other, still rather difficult to wrap your brain around programming patterns. 8sync uses some clever tricks involving "delimited continuations" under the hood to make the code you write look familiar and straightforward. When you need to send a request to another actor and get some information back from it without blocking, there's no need to write a separate procedure... 8sync's scheduler will suspend your procedure and wake it back up when a response is ready.
- Even nonblocking I/O code is straightforward to write. Thanks to the "suspendable ports" code introduced in Guile 2.2, writing asynchronous, nonblocking networked code looks mostly like writing the same synchronous code. 8sync's scheduler handles suspending and resuming networked code that would otherwise block.
- 8sync aims to be "batteries included". Useful subsystems for IRC bots, HTTP servers, and so on are included out of the box.
- 8sync prioritizes live hacking. If using an editor like Emacs with a nice mode like Geiser, an 8sync-using developer can change and fine-tune the behavior of code *while it runs*. This makes both debugging and development much more natural, allowing the right designs to evolve under your fingertips. A productive hacker is a happy hacker, after all!

In the future, 8sync will also provide the ability to spawn and communicate with actors on different threads, processes, and machines, with most code running the same as if actors were running in the same execution environment.

But as a caution, 8sync is still very young. The API is stabilizing, but not yet stable, and it is not yet well "battle-tested". Hacker beware! But, consider this as much an opportunity as a warning. 8sync is in a state where there is much room for feedback and contributions. Your help wanted!

And now, into the wild, beautiful frontier. Onward!

2 Tutorial

2.1 A silly little IRC bot

IRC! Internet Relay Chat! The classic chat protocol of the Internet. And it turns out, one of the best places to learn about networked programming.[1] We ourselves are going to explore chat bots as a basis for getting our feet wet in 8sync.

First of all, we're going to need to import some modules. Put this at the top of your file:

```
(use-modules (8sync)               ; 8sync's agenda and actors
             (8sync systems irc)   ; the irc bot subsystem
             (oop goops)           ; 8sync's actors use GOOPS
             (ice-9 format)        ; basic string formatting
             (ice-9 match))        ; pattern matching
```

Now we need to add our bot. Initially, it won't do much.

```
(define-class <my-irc-bot> (<irc-bot>))

(define-method (handle-line (irc-bot <my-irc-bot>) message
                            speaker channel line emote?)
  (if emote?
      (format #t "~a emoted ~s in channel ~a\n"
              speaker line channel)
      (format #t "~a said ~s in channel ~a\n"
              speaker line channel)))
```

We've just defined our own IRC bot! This is an 8sync actor. (8sync uses GOOPS to define actors.) We extended the handle-line generic method, so this is the code that will be called whenever the IRC bot "hears" anything. This method is itself an action handler, hence the second argument for `message`, which we can ignore for now. Pleasantly, the message's argument body is passed in as the rest of the arguments.

For now the code is pretty basic: it just outputs whatever it "hears" from a user in a channel to the current output port. Pretty boring! But it should help us make sure we have things working when we kick things off.

Speaking of, even though we've defined our actor, it's not running yet. Time to fix that!

```
(define* (run-bot #:key (username "examplebot")
                  (server "irc.freenode.net")
                  (channels '("##botchat")))
  (define hive (make-hive))
```

[1] In the 1990s I remember stumbling into some funky IRC chat rooms and being astounded that people there had what they called "bots" hanging around. From then until now, I've always enjoyed encountering bots whose range of functionality has spanned from saying absurd things, to taking messages when their "owners" were offline, to reporting the weather, to logging meetings for participants. And it turns out, IRC bots are a great way to cut your teeth on networked programming; since IRC is a fairly simple line-delineated protocol, it's a great way to learn to interact with sockets. (My first IRC bot helped my team pick a place to go to lunch, previously a source of significant dispute!) At the time of writing, venture capital awash startups are trying to turn chatbots into "big business". . . a strange (and perhaps absurd) thing given chat bots being a fairly mundane novelty amongst hackers and teenagers everywhere a few decades ago.

```
    (define irc-bot
      (bootstrap-actor hive <my-irc-bot>
                       #:username username
                       #:server server
                       #:channels channels))
    (run-hive hive '()))
```

Actors are connected to something called a "hive", which is a special kind of actor that runs and manages all the other actors. Actors can spawn other actors, but before we start the hive we use this special `bootstrap-actor` method. It takes the hive as its first argument, the actor class as the second argument, and the rest are initialization arguments to the actor. `bootstrap-actor` passes back not the actor itself (we don't get access to that usually) but the **id** of the actor. (More on this later.) Finally we run the hive with run-hive and pass it a list of "bootstrapped" messages. Normally actors send messages to each other (and sometimes themselves), but we need to send a message or messages to start things or else nothing is going to happen.

We can run it like:

```
(run-bot #:username "some-bot-name") ; be creative!
```

Assuming all the tubes on the internet are properly connected, you should be able to join the "##botchat" channel on irc.freenode.net and see your bot join as well. Now, as you probably guessed, you can't really *do* much yet. If you talk to the bot, it'll send messages to the terminal informing you as such, but it's hardly a chat bot if it's not chatting yet.

So let's do the most boring (and annoying) thing possible. Let's get it to echo whatever we say back to us. Change handle-line to this:

```
(define-method (handle-line (irc-bot <my-irc-bot>) message
                            speaker channel line emote?)
  (<- (actor-id irc-bot) 'send-line channel
      (format #f "Bawwwwk! ~a says: ~a" speaker line)))
```

This will do exactly what it looks like: repeat back whatever anyone says like an obnoxious parrot. Give it a try, but don't keep it running for too long... this bot is so annoying it's likely to get banned from whatever channel you put it in.

This method handler does have the advantage of being simple though. It introduces a new concept simply... sending a message! Whenever you see "<-", you can think of that as saying "send this message". The arguments to "<-" are as follows: the actor sending the message, the id of the actor the message is being sent to, the "action" we want to invoke (a symbol), and the rest are arguments to the "action handler" which is in this case send-line (with itself takes two arguments: the channel our bot should send a message to, and the line we want it to spit out to the channel).[2]

Normally in the actor model, we don't have direct references to an actor, only an identifier. This is for two reasons: to quasi-enforce the "shared nothing" environment (actors

[2] 8sync's name for sending a message, "<-", comes from older, early lisp object oriented systems which were, as it turned out, inspired by the actor model! Eventually message passing was dropped in favor of something called "generic functions" or "generic methods" (you may observe we made use of such a thing in extending handle-line). Many lispers believe that there is no need for message passing with generic methods and some advanced functional techniques, but in a concurrent environment message passing becomes useful again, especially when the communicating objects / actors are not in the same address space.

absolutely control their own resources, and "all you can do is send a message" to request that they modify them) and because... well, you don't even know where that actor is! Actors can be anything, and anywhere. It's possible in 8sync to have an actor on a remote hive, which means the actor could be on a remote process or even remote machine, and in most cases message passing will look exactly the same. (There are some exceptions; it's possible for two actors on the same hive to "hand off" some special types of data that can't be serialized across processes or the network, eg a socket or a closure, perhaps even one with mutable state. This must be done with care, and the actors should be careful both to ensure that they are both local and that the actor handing things off no longer accesses that value to preserve the actor model. But this is an advanced topic, and we are getting ahead of ourselves.) We have to supply the id of the receiving actor, and usually we'd have only the identifier. But since in this case, since the actor we're sending this to is ourselves, we have to pass in our identifier, since the Hive won't deliver to anything other than an address.

Astute readers may observe, since this is a case where we are just referencing our own object, couldn't we just call "sending a line" as a method of our own object without all the message passing? Indeed, we do have such a method, so we *could* rewrite handle-line like so:

```
(define-method (handle-line (irc-bot <my-irc-bot>) message
                            speaker channel line emote?)
  (irc-bot-send-line irc-bot channel
                     (format #f "Bawwwwk! ~a says: ~a" speaker line)))
```

... but we want to get you comfortable and familiar with message passing, and we'll be making use of this same message passing shortly so that *other* actors may participate in communicating with IRC through our IRC bot.

Anyway, our current message handler is simply too annoying. What we would really like to do is have our bot respond to individual "commands" like this:

```
<foo-user> examplebot: hi!
<examplebot> Oh hi foo-user!
<foo-user> examplebot: botsnack
<examplebot> Yippie! *does a dance!*
<foo-user> examplebot: echo I'm a very silly bot
<examplebot> I'm a very silly bot
```

Whee, that looks like fun! To implement it, we're going to pull out Guile's pattern matcher.

```
(define-method (handle-line (irc-bot <my-irc-bot>) message
                            speaker channel line emote?)
  (define my-name (irc-bot-username irc-bot))
  (define (looks-like-me? str)
    (or (equal? str my-name)
        (equal? str (string-concatenate (list my-name ":")))))
  (match (string-split line #\space)
    (((? looks-like-me? _) action action-args ...)
     (match action
       ;; The classic botsnack!
```

```
           ("botsnack"
            (<- (actor-id irc-bot) 'send-line channel
                "Yippie! *does a dance!*"))
           ;; Return greeting
           ((or "hello" "hello!" "hello." "greetings" "greetings." "greetings!"
                "hei" "hei." "hei!" "hi" "hi!")
            (<- (actor-id irc-bot) 'send-line channel
                (format #f "Oh hi ~a!" speaker)))
           ("echo"
            (<- (actor-id irc-bot) 'send-line channel
                (string-join action-args " ")))

           ;; --->  Add yours here <---

           ;; Default
           (_
            (<- (actor-id irc-bot) 'send-line channel
                "*stupid puppy look*"))))))
```

Parsing the pattern matcher syntax is left as an exercise for the reader.

If you're getting the sense that we could make this a bit less wordy, you're right:

```
(define-method (handle-line (irc-bot <my-irc-bot>) message
                            speaker channel line emote?)
  (define my-name (irc-bot-username irc-bot))
  (define (looks-like-me? str)
    (or (equal? str my-name)
        (equal? str (string-concatenate (list my-name ":")))))
  (define (respond respond-line)
    (<- (actor-id irc-bot) 'send-line channel
        respond-line))
  (match (string-split line #\space)
    (((? looks-like-me? _) action action-args ...)
     (match action
       ;; The classic botsnack!
       ("botsnack"
        (respond "Yippie! *does a dance!*"))
       ;; Return greeting
       ((or "hello" "hello!" "hello." "greetings" "greetings." "greetings!"
            "hei" "hei." "hei!" "hi" "hi." "hi!")
        (respond (format #f "Oh hi ~a!" speaker)))
       ("echo"
        (respond (string-join action-args " ")))

       ;; --->  Add yours here <---

       ;; Default
       (_
            (respond "*stupid puppy look*"))))))
```

Okay, that looks pretty good! Now we have enough information to build an IRC bot that can do a lot of things. Take some time to experiment with extending the bot a bit before moving on to the next section! What cool commands can you add?

2.2 Writing our own actors

Let's write the most basic, boring actor possible. How about an actor that start sleeping, and keeps sleeping?

```
(use-modules (oop goops)
             (8sync))

(define-class <sleeper> (<actor>)
  (actions #:allocation #:each-subclass
           #:init-thunk (build-actions
                         (*init* sleeper-loop))))

(define (sleeper-loop actor message)
  (while (actor-alive? actor)
    (display "Zzzzzzzz....\n")
    ;; Sleep for one second
    (8sleep (sleeper-sleep-secs actor))))

(let* ((hive (make-hive))
       (sleeper (bootstrap-actor hive <sleeper>)))
  (run-hive hive '()))
```

We see some particular things in this example. One thing is that our `<sleeper>` actor has an actions slot. This is used to look up what the "action handler" for a message is. We have to set the #:allocation to either `#:each-subclass` and use `#:init-thunk`.[3]

The only action handler we've added is for `*init*`, which is called implicitly when the actor first starts up. (This will be true whether we bootstrap the actor before the hive starts or create it during the hive's execution.)

In our sleeper-loop we also see a call to "8sleep". "8sleep" is like Guile's "sleep" method, except it is non-blocking and will always yield to the scheduler.

Our while loop also checks "actor-alive?" to see whether or not it is still registered. In general, if you keep a loop in your actor that regularly yields to the scheduler, you should check this.[4] (An alternate way to handle it would be to not use a while loop at all but simply send a message to ourselves with "<-" to call the sleeper-loop handler again. If the actor was dead, the message simply would not be delivered and thus the loop would stop.)

It turns out we could have written the class for the actor much more simply:

```
;; You could do this instead of the define-class above.
```

[3] `build-subclass` returns a thunk to be called later so that each subclass may correctly build its own instance. This is important because the structure returned contains a cache, which may vary from subclass to subclass based on its inheritance structure.

[4] Or rather, for now you should call `actor-alive?` if your code is looping like this. In the future, after an actor dies, its coroutines will automatically be "canceled".

```
(define-actor <sleeper> (<actor>)
  ((*init* sleeper-loop)))
```

This is sugar, and expands into exactly the same thing as the define-class above. The third argument is an argument list, the same as what's passed into build-actions. Everything after that is a slot. So for example, if we had added an optional slot to specify how many seconds to sleep, we could have done it like so:

```
(define-actor <sleeper> (<actor>)
  ((*init* sleeper-loop))
  (sleep-secs #:init-value 1
              #:getter sleeper-sleep-secs))
```

This actor is pretty lazy though. Time to get back to work! Let's build a worker / manager type system.

```
(use-modules (8sync)
             (oop goops))

(define-actor <manager> (<actor>)
  ((assign-task manager-assign-task))
  (direct-report #:init-keyword #:direct-report
                 #:getter manager-direct-report))

(define (manager-assign-task manager message difficulty)
  "Delegate a task to our direct report"
  (display "manager> Work on this task for me!\n")
  (<- (manager-direct-report manager)
      'work-on-this difficulty))
```

This manager keeps track of a direct report and tells them to start working on a task... simple delegation. Nothing here is really new, but note that our friend "<-" (which means "send message") is back. There's one difference this time... the first time we saw "<-" was in the handle-line procedure of the irc-bot, and in that case we explicitly pulled the actor-id after the actor we were sending the message to (ourselves), which we aren't doing here. But that was an unusual case, because the actor was ourself. In this case, and in general, actors don't have direct references to other actors; instead, all they have is access to identifiers which reference other actors.

```
(define-actor <worker> (<actor>)
  ((work-on-this worker-work-on-this))
  (task-left #:init-keyword #:task-left
             #:accessor worker-task-left))

(define (worker-work-on-this worker message difficulty)
  "Work on one task until done."
  (set! (worker-task-left worker) difficulty)
  (display "worker> Whatever you say, boss!\n")
  (while (and (actor-alive? worker)
              (> (worker-task-left worker) 0))
    (display "worker> *huff puff*\n")
```

```
        (set! (worker-task-left worker)
              (- (worker-task-left worker) 1))
        (8sleep (/ 1 3))))
```

The worker also contains familiar code, but we now see that we can call 8sleep with non-integer real numbers.

Looks like there's nothing left to do but run it.

```
(let* ((hive (make-hive))
       (worker (bootstrap-actor hive <worker>))
       (manager (bootstrap-actor hive <manager>
                                 #:direct-report worker)))
  (run-hive hive (list (bootstrap-message hive manager 'assign-task 5))))
```

Unlike the `<sleeper>`, our `<manager>` doesn't have an implicit `*init*` method, so we've bootstrapped the calling `assign-task` action.

```
manager> Work on this task for me!
worker> Whatever you say, boss!
worker> *huff puff*
worker> *huff puff*
worker> *huff puff*
worker> *huff puff*
worker> *huff puff*
```

"<-" pays no attention to what happens with the messages it has sent off. This is useful in many cases. . . we can blast off many messages and continue along without holding anything back.

But sometimes we want to make sure that something completes before we do something else, or we want to send a message and get some sort of information back. Luckily 8sync comes with an answer to that with "<-wait", which will suspend the caller until the callee gives some sort of response, but which does not block the rest of the program from running. Let's try applying that to our own code by turning our manager into a micromanager.

```
;;; Update this method
(define (manager-assign-task manager message difficulty)
  "Delegate a task to our direct report"
  (display "manager> Work on this task for me!\n")
  (<- (manager-direct-report manager)
      'work-on-this difficulty)

  ;; Wait a moment, then call the micromanagement loop
  (8sleep (/ 1 2))
  (manager-micromanage-loop manager))

;;; And add the following
;;;   (... Note: do not model actual employee management off this)
(define (manager-micromanage-loop manager)
  "Pester direct report until they're done with their task."
  (display "manager> Are you done yet???\n")
  (let ((worker-is-done
         (mbody-val (<-wait (manager-direct-report manager)
                            'done-yet?))))
    (if worker-is-done
        (begin (display "manager> Oh!  I guess you can go home then.\n")
               (<- (manager-direct-report manager) 'go-home))
        (begin (display "manager> Harumph!\n")
               (8sleep (/ 1 2))
               (when (actor-alive? manager)
                 (manager-micromanage-loop manager))))))
```

We've appended a micromanagement loop here... but what's going on? "<-wait", as it sounds, waits for a reply, and returns a reply message. In this case there's a value in the body of the message we want, so we pull it out with mbody-val. (It's possible for a remote actor to return multiple values, in which case we'd want to use mbody-receive, but that's a bit more complicated.)

Of course, we need to update our worker accordingly as well.

```
;;; Update the worker to add the following new actions:
(define-actor <worker> (<actor>)
  ((work-on-this worker-work-on-this)
   ;; Add these:
   (done-yet? worker-done-yet?)
   (go-home worker-go-home))
  (task-left #:init-keyword #:task-left
             #:accessor worker-task-left))

;;; New procedures:
(define (worker-done-yet? worker message)
  "Reply with whether or not we're done yet."
  (let ((am-i-done? (= (worker-task-left worker) 0)))
    (if am-i-done?
        (display "worker> Yes, I finished up!\n")
        (display "worker> No... I'm still working on it...\n"))
    (<-reply message am-i-done?)))

(define (worker-go-home worker message)
  "It's off of work for us!"
  (display "worker> Whew!  Free at last.\n")
  (self-destruct worker))
```

(As you've probably guessed, you wouldn't normally call `display` everywhere as we are in this program... that's just to make the examples more illustrative.)

"<-reply" is what actually returns the information to the actor waiting on the reply. It takes as an argument the actor sending the message, the message it is in reply to, and the rest of the arguments are the "body" of the message. (If an actor handles a message that is being "waited on" but does not explicitly reply to it, an auto-reply with an empty body will be triggered so that the waiting actor is not left waiting around.)

The last thing to note is the call to "self-destruct". This does what you might expect: it removes the actor from the hive. No new messages will be sent to it. Ka-poof!

Running it is the same as before:

```
(let* ((hive (make-hive))
       (worker (bootstrap-actor hive <worker>))
       (manager (bootstrap-actor hive <manager>
                                 #:direct-report worker)))
  (run-hive hive (list (bootstrap-message hive manager 'assign-task 5))))
```

But the output is a bit different:

```
manager> Work on this task for me!
worker> Whatever you say, boss!
worker> *huff puff*
worker> *huff puff*
manager> Are you done yet???
worker> No... I'm still working on it...
manager> Harumph!
worker> *huff puff*
manager> Are you done yet???
worker> *huff puff*
worker> No... I'm still working on it...
manager> Harumph!
worker> *huff puff*
manager> Are you done yet???
worker> Yes, I finished up!
manager> Oh!  I guess you can go home then.
worker> Whew!  Free at last.
```

2.3 Writing our own network-enabled actor

So, you want to write a networked actor! Well, luckily that's pretty easy, especially with all you know so far.

```
(use-modules (oop goops)
             (8sync)
             (ice-9 rdelim)  ; line delineated i/o
             (ice-9 match))  ; pattern matching

(define-actor <telcmd> (<actor>)
  ((*init* telcmd-init)
   (*cleanup* telcmd-cleanup)
   (new-client telcmd-new-client)
   (handle-line telcmd-handle-line))
  (socket #:accessor telcmd-socket
          #:init-value #f))
```

Nothing surprising about the actor definition, though we do see that it has a slot for a socket. Unsurprisingly, that will be set up in the `*init*` handler.

```
(define (set-port-nonblocking! port)
  (let ((flags (fcntl port F_GETFL)))
    (fcntl port F_SETFL (logior O_NONBLOCK flags))))
```

```
(define (setup-socket)
  ;; our socket
  (define s
    (socket PF_INET SOCK_STREAM 0))
  ;; reuse port even if busy
  (setsockopt s SOL_SOCKET SO_REUSEADDR 1)
  ;; connect to port 8889 on localhost
  (bind s AF_INET INADDR_LOOPBACK 8889)
  ;; make it nonblocking and start listening
  (set-port-nonblocking! s)
  (listen s 5)
  s)

(define (telcmd-init telcmd message)
  (set! (telcmd-socket telcmd) (setup-socket))
  (display "Connect like: telnet localhost 8889\n")
  (while (actor-alive? telcmd)
    (let ((client-connection (accept (telcmd-socket telcmd))))
      (<- (actor-id telcmd) 'new-client client-connection))))

(define (telcmd-cleanup telcmd message)
  (display "Closing socket!\n")
  (when (telcmd-socket telcmd)
    (close (telcmd-socket telcmd))))
```

That `setup-socket` code looks pretty hard to read! But that's pretty standard code for setting up a socket. One special thing is done though... the call to `set-port-nonblocking!` sets flags on the socket port so that, you guessed it, will be a nonblocking port.

This is put to immediate use in the telcmd-init method. This code looks suspiciously like it *should* block... after all, it just keeps looping forever. But since 8sync is using Guile's suspendable ports code feature, so every time this loop hits the `accept` call, if that call *would have* blocked, instead this whole procedure suspends to the scheduler... automatically!... allowing other code to run.

So, as soon as we do accept a connection, we send a message to ourselves with the `new-client` action. But wait! Aren't actors only supposed to handle one message at a time? If the telcmd-init loop just keeps on looping and looping, when will the `new-client` message ever be handled? 8sync actors only receive one message at a time, but by default if an actor's message handler suspends to the agenda for some reason (such as to send a message or on handling I/O), that actor may continue to accept other messages, but always in the same thread.[5]

We also see that we've established a `*cleanup*` handler. This is run any time either the actor dies, either through self destructing, because the hive completes its work, or because

[5] This is customizable: an actor can be set up to queue messages so that absolutely no messages are handled until the actor completely finishes handling one message. Our loop couldn't look quite like this though!

a signal was sent to interrupt or terminate our program. In our case, we politely close the socket when `<telcmd>` dies.

```
(define (telcmd-new-client telcmd message client-connection)
  (define client (car client-connection))
  (set-port-nonblocking! client)
  (let loop ()
    (let ((line (read-line client)))
      (cond ((eof-object? line)
             (close client))
            (else
             (<- (actor-id telcmd) 'handle-line
                 client (string-trim-right line #\return))
             (when (actor-alive? telcmd)
               (loop)))))))

(define (telcmd-handle-line telcmd message client line)
  (match (string-split line #\space)
    (("") #f)  ; ignore empty lines
    (("time" _ ...)
     (display
      (strftime "The time is: %c\n" (localtime (current-time)))
      client))
    (("echo" rest ...)
     (format client "~a\n" (string-join rest " ")))
    ;; default
    (_ (display "Sorry, I don't know that command.\n" client))))
```

Okay, we have a client, so we handle it! And once again... we see this goes off on a loop of its own! (Also once again, we have to do the `set-port-nonblocking!` song and dance.) This loop also automatically suspends when it would otherwise block... as long as read-line has information to process, it'll keep going, but if it would have blocked waiting for input, then it would suspend the agenda.[6]

The actual method called whenever we have a "line" of input is pretty straightforward... in fact it looks an awful lot like the IRC bot handle-line procedure we used earlier. No surprises there![7]

Now let's run it:

```
(let* ((hive (make-hive))
       (telcmd (bootstrap-actor hive <telcmd>)))
```

[6] If there's a lot of data coming in and you don't want your I/O loop to become too "greedy", take a look at `setvbuf`.

[7] Well, there may be one surprise to a careful observer. Why are we sending a message to ourselves? Couldn't we have just dropped the argument of "message" to telcmd-handle-line and just called it like any other procedure? Indeed, we *could* do that, but sending a message to ourself has an added advantage: if we accidentally "break" the telcmd-handle-line procedure in some way (say we add a fun new command we're playing with it), raising an exception won't break and disconnect the client's main loop, it'll just break the message handler for that one line, and our telcmd will happily chug along accepting another command from the user while we try to figure out what happened to the last one.

```
  (run-hive hive '()))
```

Open up another terminal... you can connect via telnet:

```
$ telnet localhost 8889
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
time
The time is: Thu Jan  5 03:20:17 2017
echo this is an echo
this is an echo
shmmmmmmorp
Sorry, I don't know that command.
```

Horray, it works! Type `Ctrl+] Ctrl+d` to exit telnet.

Not so bad! There's more that could be optimized, but we'll consider that to be advanced topics of discussion.

So that's a pretty solid intro to how 8sync works! Now that you've gone through this introduction, we hope you'll have fun writing and hooking together your own actors. Since actors are so modular, it's easy to have a program that has multiple subystems working together. You could build a worker queue system that displayed a web interface and spat out notifications about when tasks finish to IRC, and making all those actors talk to each other should be a piece of cake. The sky's the limit!

Happy hacking!

2.4 An intermission on live hacking

This section is optional, but highly recommended. It requires that you're a user of GNU Emacs. If you aren't, don't worry... you can forge ahead and come back in case you ever do become an Emacs user. (If you're more familiar with Vi/Vim style editing, I hear good things about Spacemacs...)

Remember all the way back when we were working on the IRC bot? So you may have noticed while updating that section that the start/stop cycle of hacking isn't really ideal. You might either edit a file in your editor, then run it, or type the whole program into the REPL, but then you'll have to spend extra time copying it to a file. Wouldn't it be nice if it were possible to both write code in a file and try it as you go? And wouldn't it be even better if you could live edit a program while it's running?

Luckily, there's a great Emacs mode called Geiser which makes editing and hacking and experimenting all happen in harmony. And even better, 8sync is optimized for this experience. 8sync provides easy drop-in "cooperative REPL" support, and most code can be simply redefined on the fly in 8sync through Geiser and actors will immediately update their behavior, so you can test and tweak things as you go.

Okay, enough talking. Let's add it! Redefine run-bot like so:

```
(define* (run-bot #:key (username "examplebot")
                  (server "irc.freenode.net")
                  (channels '("##botchat"))
                  (repl-path "/tmp/8sync-repl"))
```

```
  (define hive (make-hive))
  (define irc-bot
    (bootstrap-actor hive <my-irc-bot>
                     #:username username
                     #:server server
                     #:channels channels))
  (define repl-manager
    (bootstrap-actor hive <repl-manager>
                     #:path repl-path))

  (run-hive hive '()))
```

If we put a call to run-bot at the bottom of our file we can call it, and the repl-manager will start something we can connect to automatically. Horray! Now when we run this it'll start up a REPL with a unix domain socket at the repl-path. We can connect to it in emacs like so:

```
M-x geiser-connect-local <RET> guile <RET> /tmp/8sync-repl <RET>
```

Okay, so what does this get us? Well, we can now live edit our program. Let's change how our bot behaves a bit. Let's change handle-line and tweak how the bot responds to a botsnack. Change this part:

```
;; From this:
("botsnack"
 (respond "Yippie! *does a dance!*"))

;; To this:
("botsnack"
 (respond "Yippie! *catches botsnack in midair!*"))
```

Okay, now let's evaluate the change of the definition. You can hit "C-M-x" anywhere in the definition to re-evaluate. (You can also position your cursor at the end of the definition and press "C-x C-e", but I've come to like "C-M-x" better because I can evaluate as soon as I'm done writing.) Now, on IRC, ask your bot for a botsnack. The bot should give the new message. . . with no need to stop and start the program!

Let's fix a bug live. Our current program works great if you talk to your bot in the same IRC channel, but what if you try to talk to them over private message?

```
IRC> /query examplebot
<foo-user> examplebot: hi!
```

Hm, we aren't seeing any response on IRC! Huh? What's going on? It's time to do some debugging. There are plenty of debugging tools in Guile, but sometimes the simplest is the nicest, and the simplest debugging route around is good old fashioned print debugging.

It turns out Guile has an under-advertised feature which makes print debugging really easy called "pk", pronounced "peek". What pk accepts a list of arguments, prints out the whole thing, but returns the last argument. This makes wrapping bits of our code pretty easy to see what's going on. So let's peek into our program with pk. Edit the respond section to see what channel it's really sending things to:

```
(define-method (handle-line (irc-bot <my-irc-bot>) message
                             speaker channel line emote?)
  ;; [... snip ...]
  (define (respond respond-line)
    (<- (actor-id irc-bot) 'send-line (pk 'channel channel)
        respond-line))
  ;; [... snip ...]
  )
```

Re-evaluate. Now let's ping our bot in both the channel and over PM.

```
;;; (channel "##botchat")

;;; (channel "sinkbot")
```

Oh okay, this makes sense. When we're talking in a normal multi-user channel, the channel we see the message coming from is the same one we send to. But over PM, the channel is a username, and in this case the username we're sending our line of text to is ourselves. That isn't what we want. Let's edit our code so that if we see that the channel we're sending to looks like our own username that we respond back to the sender. (We can remove the pk now that we know what's going on.)

```
(define-method (handle-line (irc-bot <my-irc-bot>) message
                            speaker channel line emote?)
  ;; [... snip ...]
  (define (respond respond-line)
    (<- (actor-id irc-bot) 'send-line
        (if (looks-like-me? channel)
            speaker    ; PM session
            channel)   ; normal IRC channel
        respond-line))
  ;; [... snip ...]
  )
```

Re-evaluate and test.

```
IRC> /query examplebot
<foo-user> examplebot: hi!
<examplebot> Oh hi foo-user!
```

Horray!

3 API reference

4 Systems reference

4.1 IRC

4.2 Web / HTTP

5 Addendum

5.1 Recommended emacs additions

In order for `mbody-receive` to indent properly, put this in your .emacs:

```
(put 'mbody-receive 'scheme-indent-function 2)
```

5.2 8sync and Fibers

One other major library for asynchronous communication in Guile-land is Fibers (`https://github.com/wingo/fibers/`). There's a lot of overlap:

- Both use Guile's suspendable-ports facility
- Both communicate between asynchronous processes using message passing; you don't have to squint hard to see the relationship between Fibers' channels and 8sync's actor inboxes.

However, there are clearly differences too. There's a one to one relationship between 8sync actors and an actor inbox, whereas each Fibers fiber may read from multiple channels, for example.

Luckily, it turns out there's a clear relationship, based on real, actual theory! 8sync is based on the actor model (`https://en.wikipedia.org/wiki/Actor_model`) whereas fibers follows Communicating Sequential Processes (CSP) (`http://usingcsp.com/`), which is a form of process calculi (`https://en.wikipedia.org/wiki/Process_calculus`). And it turns out, the relationship between the actor model and process calculi (`https://en.wikipedia.org/wiki/Actor_model_and_process_calculi`) is well documented, and even more precisely, the relationship between CSP and the actor model (`https://en.wikipedia.org/wiki/Communicating_sequential_processes#Comparison_with_the_Actor_Model`) is well understood too.

So, 8sync and Fibers do take somewhat different approaches, but both have a solid theoretical backing... and their theories are well understood in terms of each other. Good news for theory nerds!

(Since the actors and CSP are dual (`https://en.wikipedia.org/wiki/Dual_%28mathematics%29`), maybe eventually 8sync will be implemented on top of Fibers... that remains to be seen!)

5.3 8sync's license and general comments on copyleft

8sync is released under the GNU LGPL (Lesser General Public License), version 3 or later, as published by the Free Software Foundation. The short version of this is that if you distribute a modifications to 8sync, whether alone or in some larger combination, must release the corresponding source code. A program which uses this library, if distributed without source code, must also allow relinking with a modified version of this library. In general, it is best to contribute them back to 8sync under the same terms; we'd appreciate any enhancements or fixes to be contributed upstream to 8sync itself. (This is an intentional oversimplification for brevity, please read the LGPL for the precise terms.)

This usage of the LGPL helps us ensure that 8sync and derivatives of 8sync as a library will remain free. Though it is not a requirement, we request you use 8sync to build free software rather than use it to contribute to the growing world of proprietary software.

The choice of the LGPL for 8sync was a strategic one. This is not a general recommendation to use the LGPL instead of the GPL for all libraries. In general, we encourage stronger copyleft. (For more thinking on this position, see Why you shouldn't use the Lesser GPL for your next library (`https://www.gnu.org/licenses/why-not-lgpl.html`).)

Although 8sync provides some unique features, its main functionality is as an asynchronous programming environment, and there are many other asynchronous programming environments out there such as Node.js for Javascript and Asyncio for Python (there are others as well). It is popular in some of these communities to hold anti-copyleft positions, which is unfortunate, and many community members seem to be adopting these positions because other developers they look up to are holding them. If you have come from one of these communities and are exploring 8sync, we hope reading this will help you reconsider your position.

In particular, if you are building a library or application that uses 8sync in some useful way, consider releasing your program under the GNU GPL or GNU AGPL! In a world where more and more software is locked down, where software is used to restrict users, we could use every chance we can get to provide protections so that software which is free remains free, and encourages even more software freedom to be built upon it.

So to answer the question, "Can I build a proprietary program on top of 8sync?" our response is "Yes, but please don't. Choose to release your software under a freedom-respecting license. And help us turn the tide towards greater software freedom... consider a strong copyleft license!"

5.4 Acknowledgements

8sync has a number of inspirations:

- asyncio (`https://docs.python.org/3.5/library/asyncio.html`) for Python provides a nice asynchronous programming environment, and makes great use of generator-style coroutines. It's a bit more difficult to work with than 8sync (or so thinks the author) because you have to "line up" the coroutines.
- Sly (`http://dthompson.us/pages/software/sly.html`) by David Thompson is an awesome functional reactive game programming library for Guile. If you want to write graphical games, Sly is almost certainly a better choice than 8sync. Thanks to David for being very patient in explaining tough concepts; experience on hacking Sly greatly informed 8sync's development. (Check out Sly, it rocks!)
- Reading SICP (`https://mitpress.mit.edu/sicp/`), particularly Chapter 3's writings on concurrent systems (`https://mitpress.mit.edu/sicp/full-text/book/book-Z-H-19.html#%_chap_3`), greatly informed 8sync's design.
- Finally, XUDD (`http://xudd.readthedocs.io/en/latest/`) was an earlier "research project" that preceeded 8sync. It attempted to bring an actor model system to Python. However, the author eventually grew frustrated with some of Python's limitations, fell in love with Guile, and well... now we have 8sync. Much of 8sync's actor model design came from experiments in developing XUDD.

The motivation to build 8sync came out of a conversation (`https://lists.gnu.org/archive/html/guile-devel/2015-10/msg00015.html`) at the FSF 30th party between Mark Weaver, David Thompson, Andrew Engelbrecht, and Christopher Allan Webber over how to build an asynchronous event loop for Guile and just what would be needed.

A little over a month after that, hacking on 8sync began!

Appendix A Copying This Manual

This manual is licensed under the GNU Free Documentation License, with no invariant sections. At your option, it is also available under the GNU Lesser General Public License, as published by the Free Software Foundation, version 3 or any later version.

A.1 GNU Free Documentation License

textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

ADDENDUM: How to use this License for your documents

Index

(Index is nonexistent)