Perl 6

&

The Zen of Erlang

Ben Tyler
booking.com
The Perl Conference, 2016

Context

Context

Not a "I’ve done this thing in production" talk

Context

Not a "I’ve done this thing in production" talk

(…yet! :D)

Context

Not a "I’ve done this thing in production" talk

(…yet! :D)

Casing the joint

Context

Not a "I've done this thing in production" talk

(...yet! :D)

Casing the joint

Signal boosting

Context

Not a "I've done this thing in production" talk

(...yet! :D)

Casing the joint

Signal boosting

Let's chat about this!

Overview

Overview

unified theory of (un)reliability

Overview

unified theory of (un)reliability

juggling raptors

Overview

unified theory of (un)reliability

juggling raptors

mecha-concurrency

Overview

unified theory of (un)reliability

juggling raptors

mecha-concurrency

the power of power cycling

Overview

unified theory of (un)reliability

juggling raptors

mecha-concurrency

the power of power cycling

The Zen of Erlang

Overview

unified theory of (un)reliability

juggling raptors

mecha-concurrency

the power of power cycling

The Zen of Erlang

Perl 6

Overview

unified theory of (un)reliability

juggling raptors

mecha-concurrency

the power of power cycling

The Zen of Erlang

Perl 6

Theory of (un)reliability

Theory of (un)reliability

1) don’t write bugs

Theory of (un)reliability

1) don’t write bugs

OR

Theory of (un)reliability

1) don’t write bugs

OR

2a) write *fewer* bugs

Theory of (un)reliability

1) don't write bugs

OR

2a) write *fewer* bugs

2b) limit bug impact

Theory of (un)reliability

1) don't write bugs

OR

2a) write *fewer* bugs <— do this

2b) limit bug impact

Theory of (un)reliability

1) don’t write bugs

OR

2a) write *fewer* bugs <— do this

2b) limit bug impact <— and this

a fever dream

a fever dream

"the sync script is broken"

a fever dream

"the sync script is broken"

`/usr/local/bin/sync.pl`

a fever dream

"the sync script is broken"

`/usr/local/bin/sync.pl`

```
$ less sync.pl
```

a fever dream

a fever dream

no shebang

a fever dream

no shebang

no strict

a fever dream

no shebang

no strict

```
`ssh $ENV{REMOTE} perl /usr/local/bin/sync.pl`
```

a fever dream

no shebang

no strict

```
`ssh $ENV{REMOTE} perl /usr/local/bin/sync.pl`
```

no subs

a fever dream

no shebang

no strict

```
`ssh $ENV{REMOTE} perl /usr/local/bin/sync.pl`
```

no subs

```
$host =~ s/[a-z]{5}//; # TODO substr??
```

a fever dream

no shebang

no strict

```
`ssh $ENV{REMOTE} perl /usr/local/bin/sync.pl`
```

no subs

```
$host =~ s/[a-z]{5}//; # TODO substr??
```

~1200 lines

talk about your feelings

Why?

Why?

epic baby perl?

Why?

epic baby perl?

unmaintainable?

Why?

epic baby perl?

unmaintainable?

unreliable?

IT WORKS

SHIP IT

coupling -> reliability

coupling -> reliability

'the right amount'

coupling -> reliability

'the right amount'

abstraction

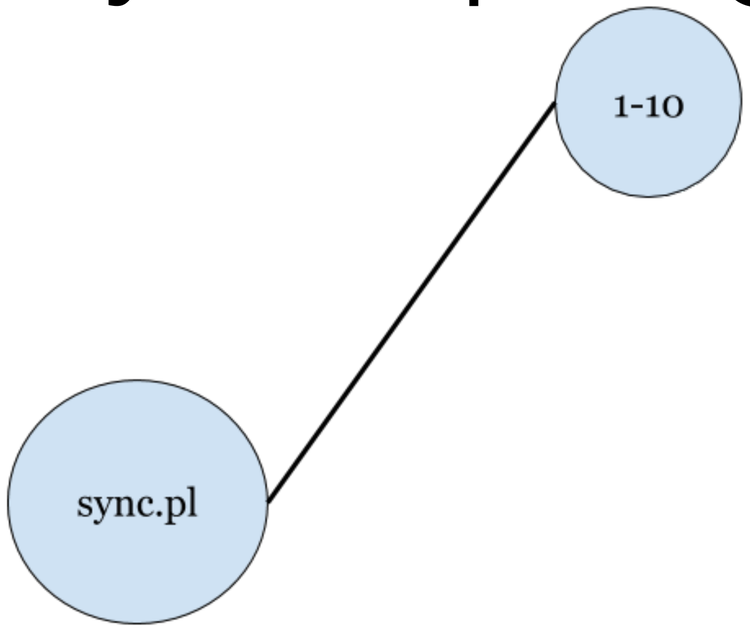
coupling -> reliability

'the right amount'

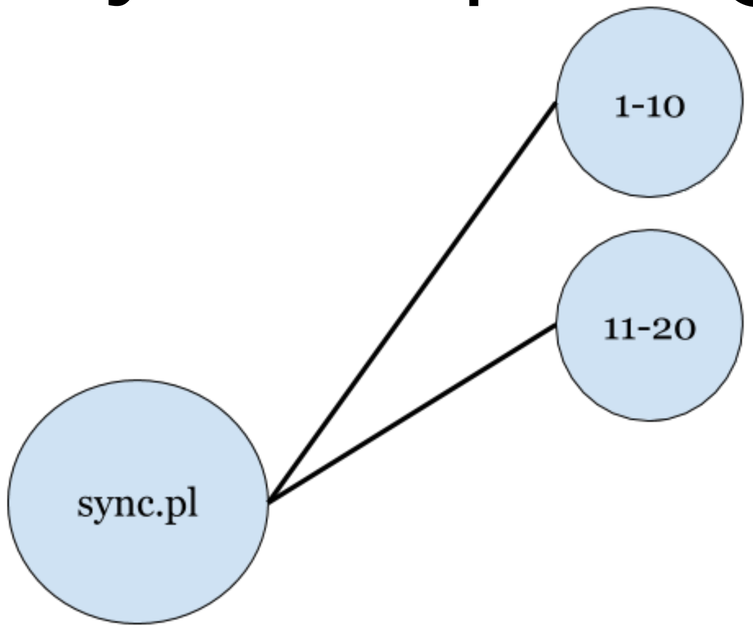
abstraction

scope

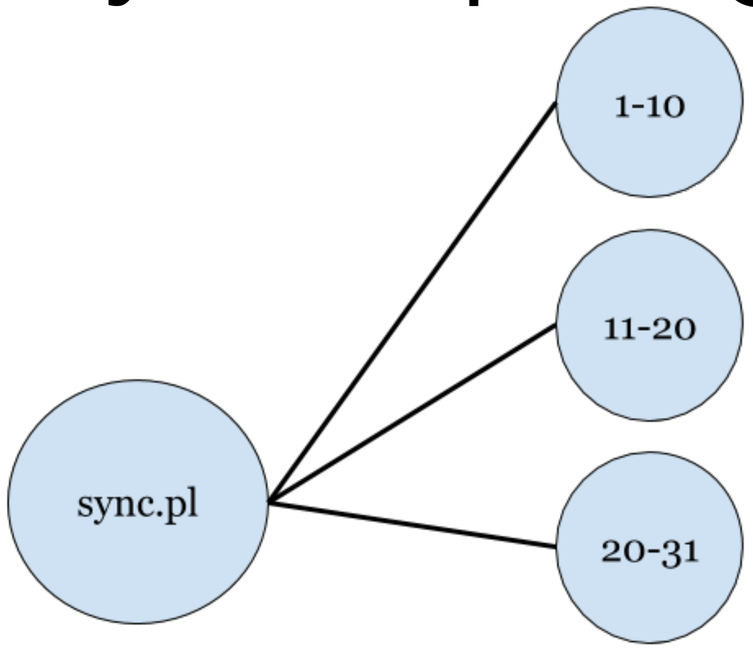
reliability scope: global



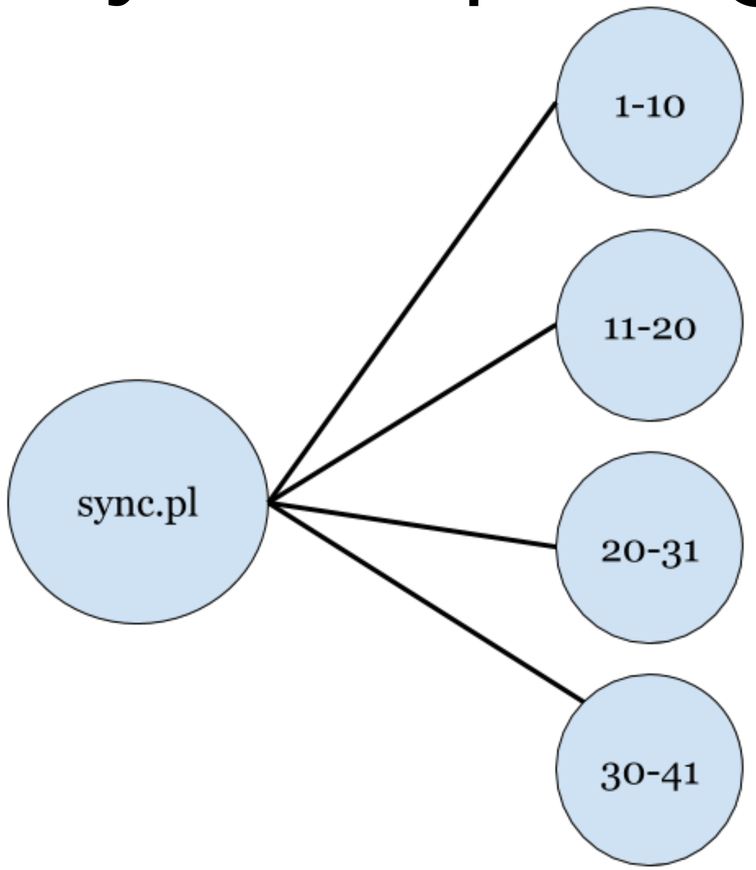
reliability scope: global



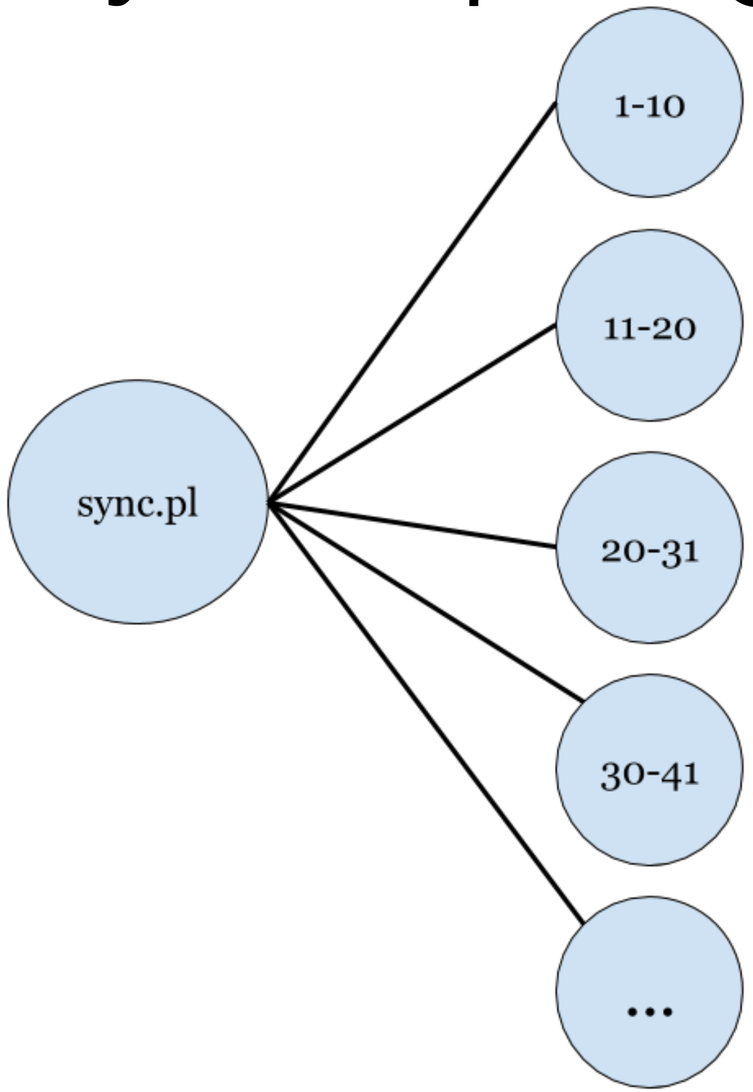
reliability scope: global



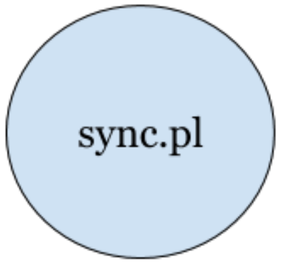
reliability scope: global



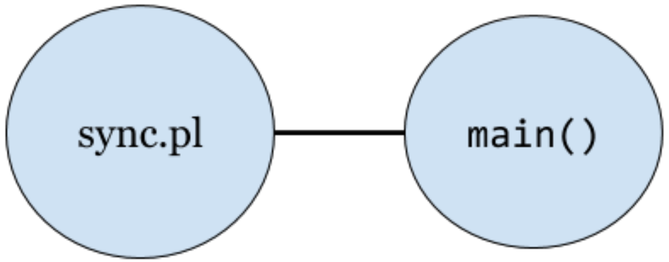
reliability scope: global



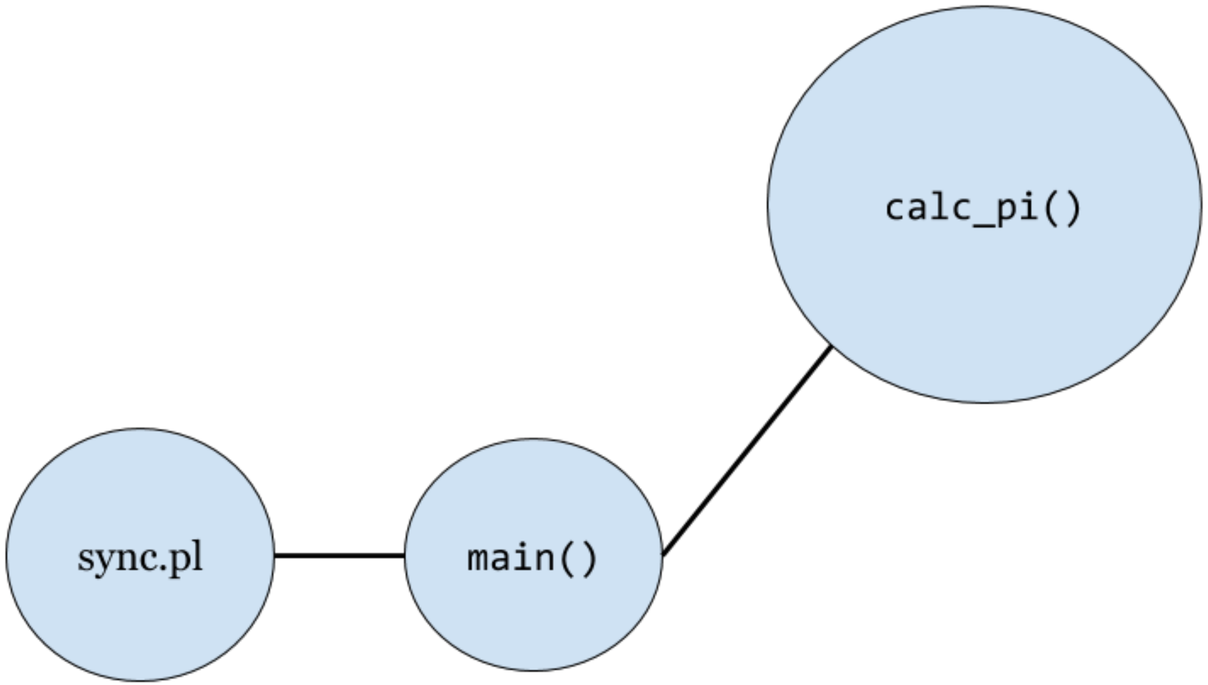
reliability scope: local



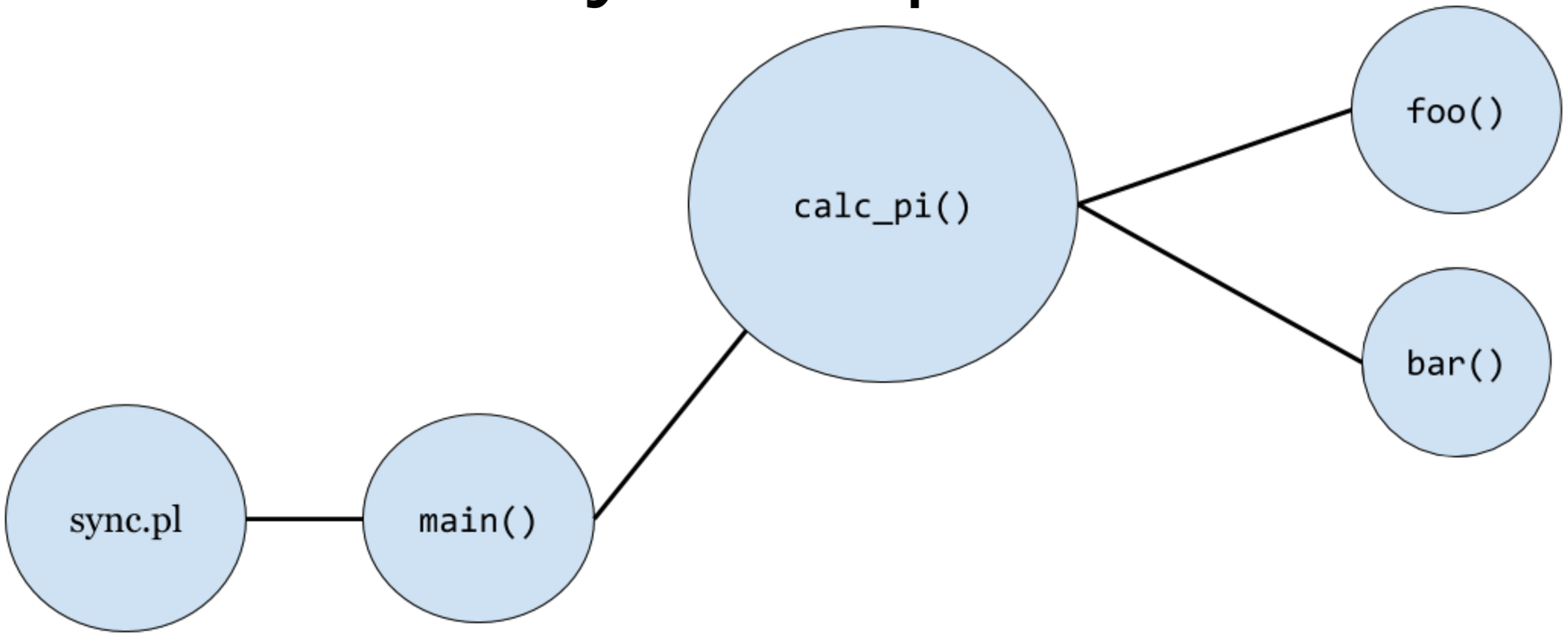
reliability scope: local



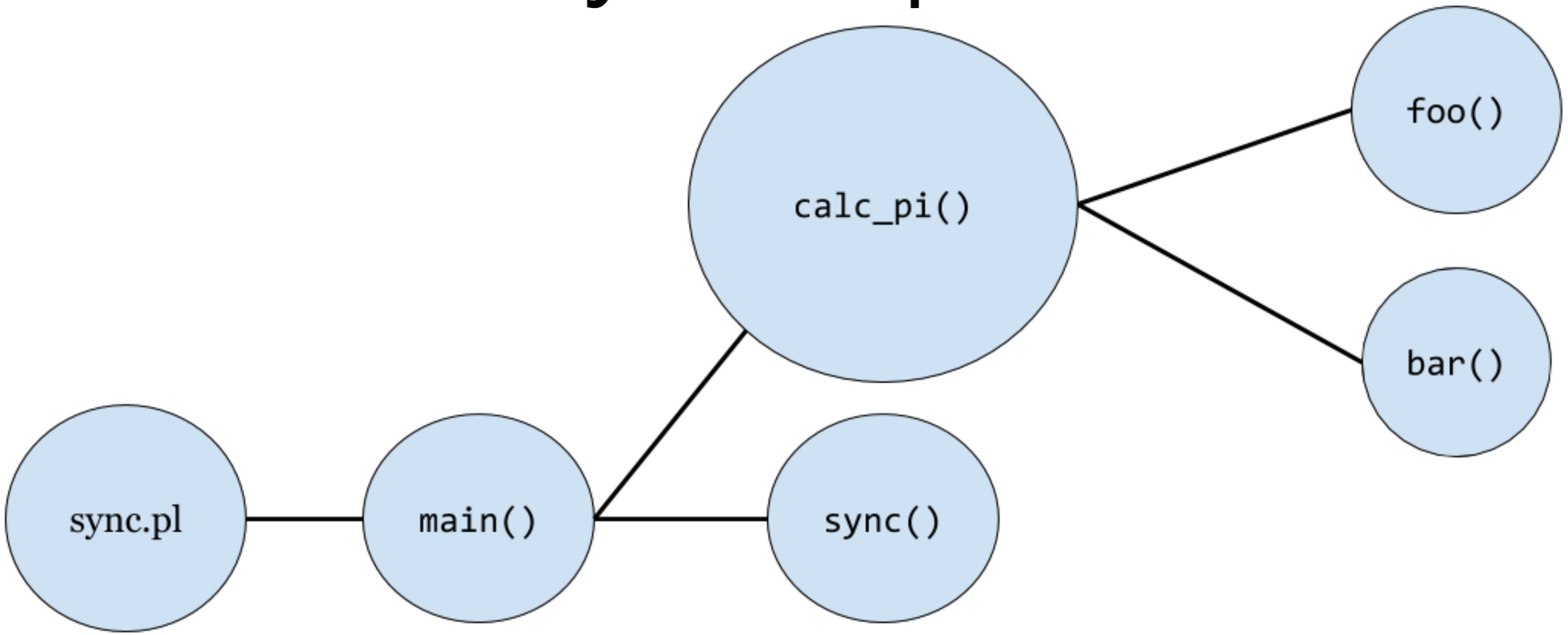
reliability scope: local



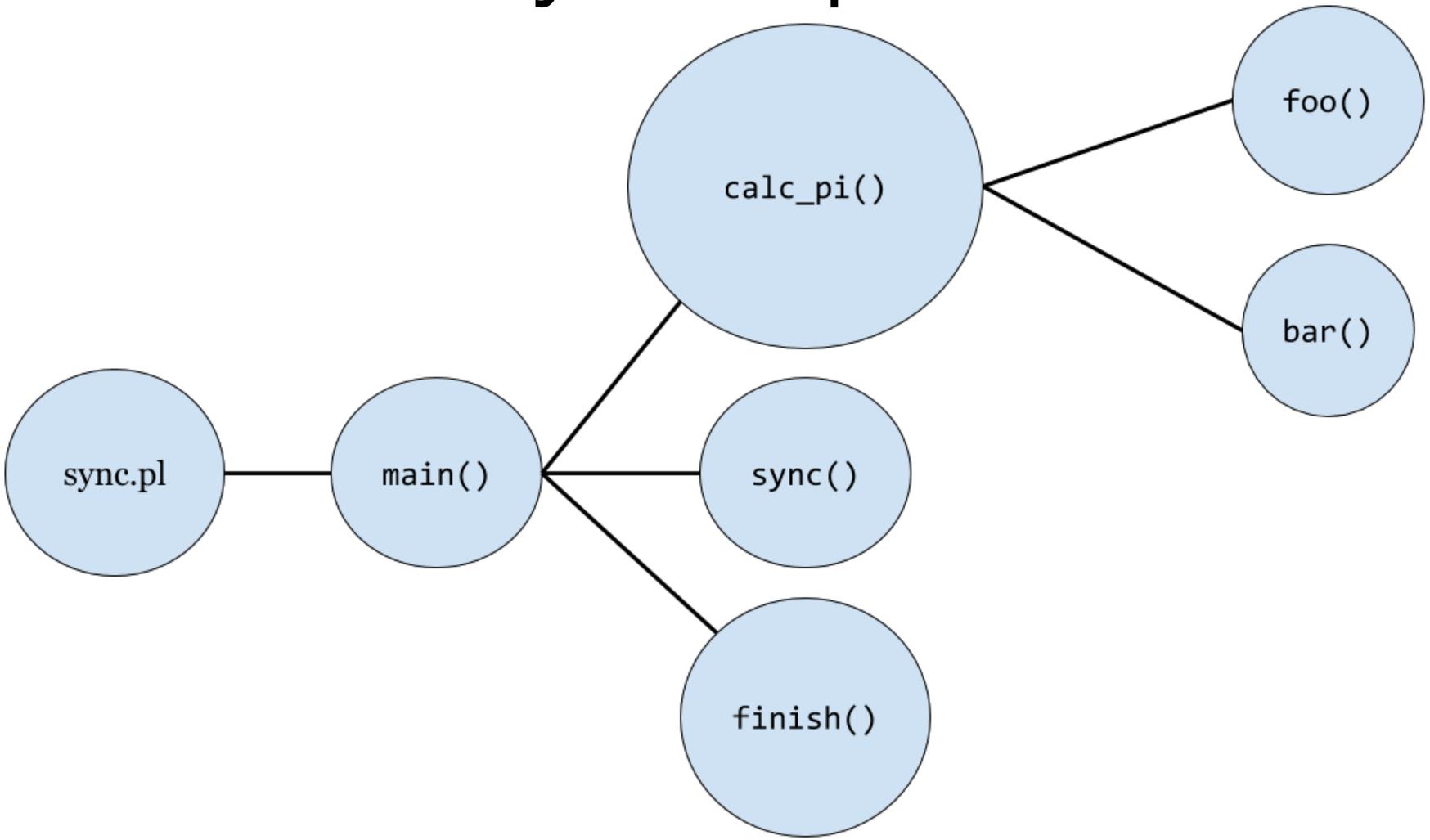
reliability scope: local



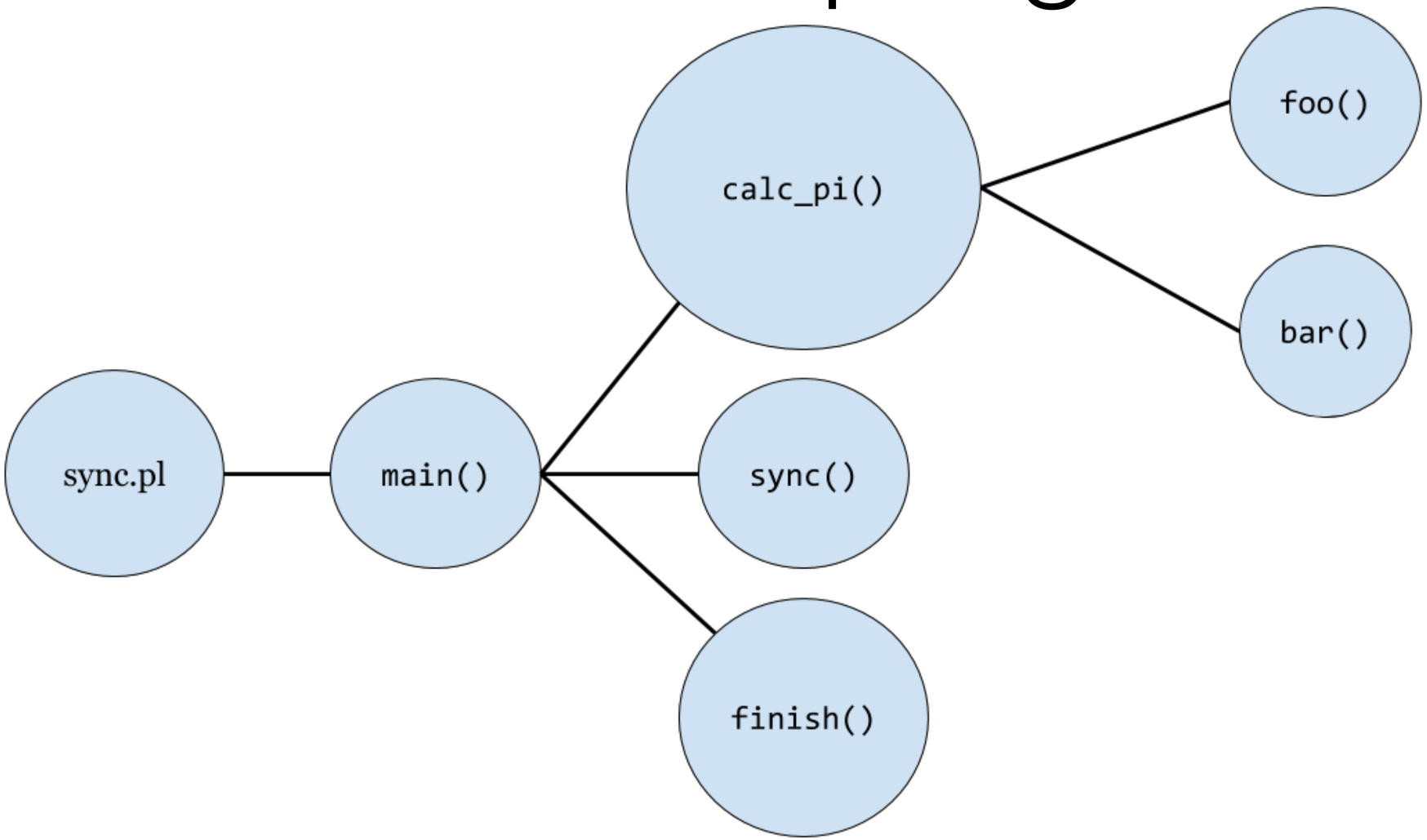
reliability scope: local



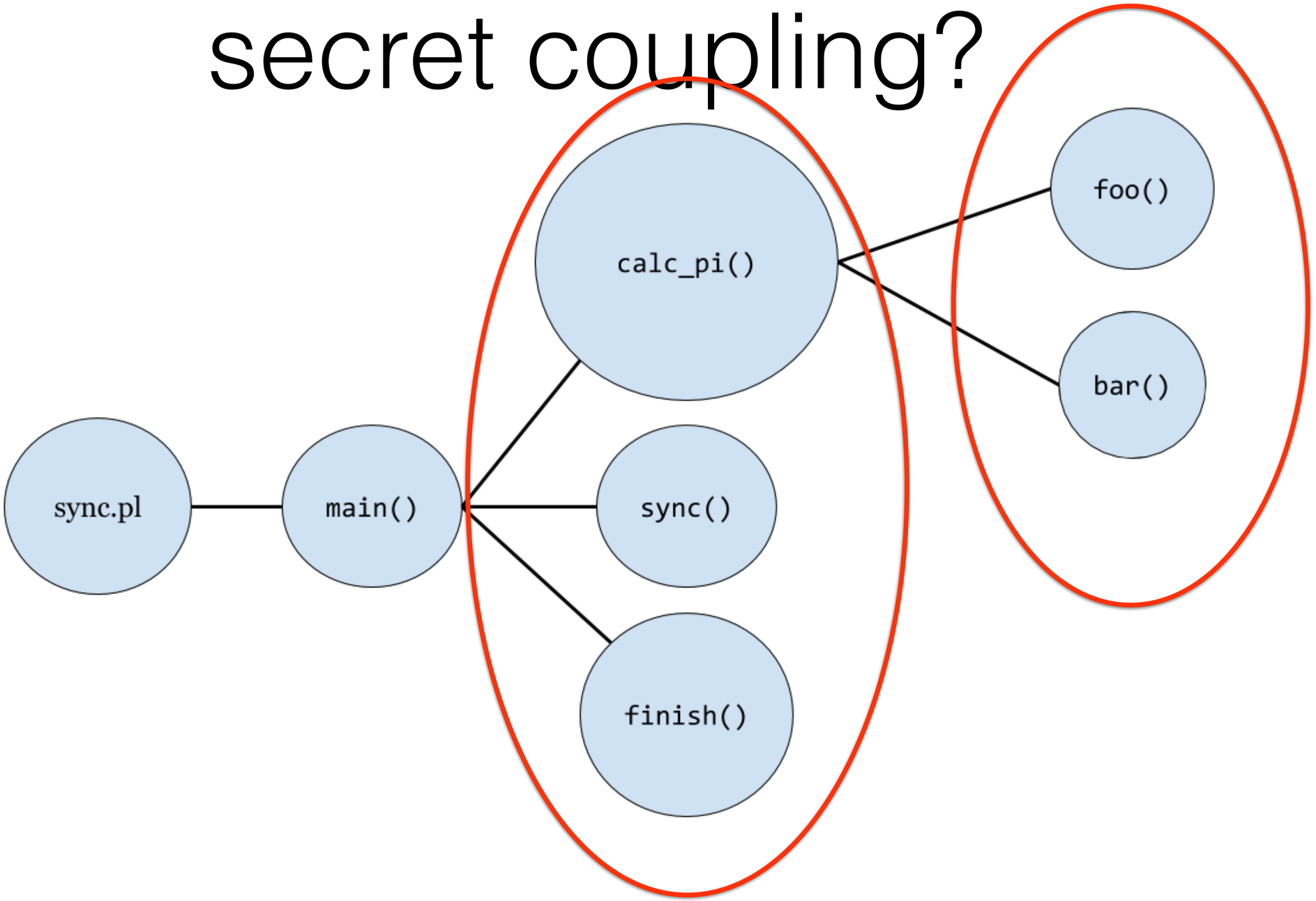
reliability scope: local



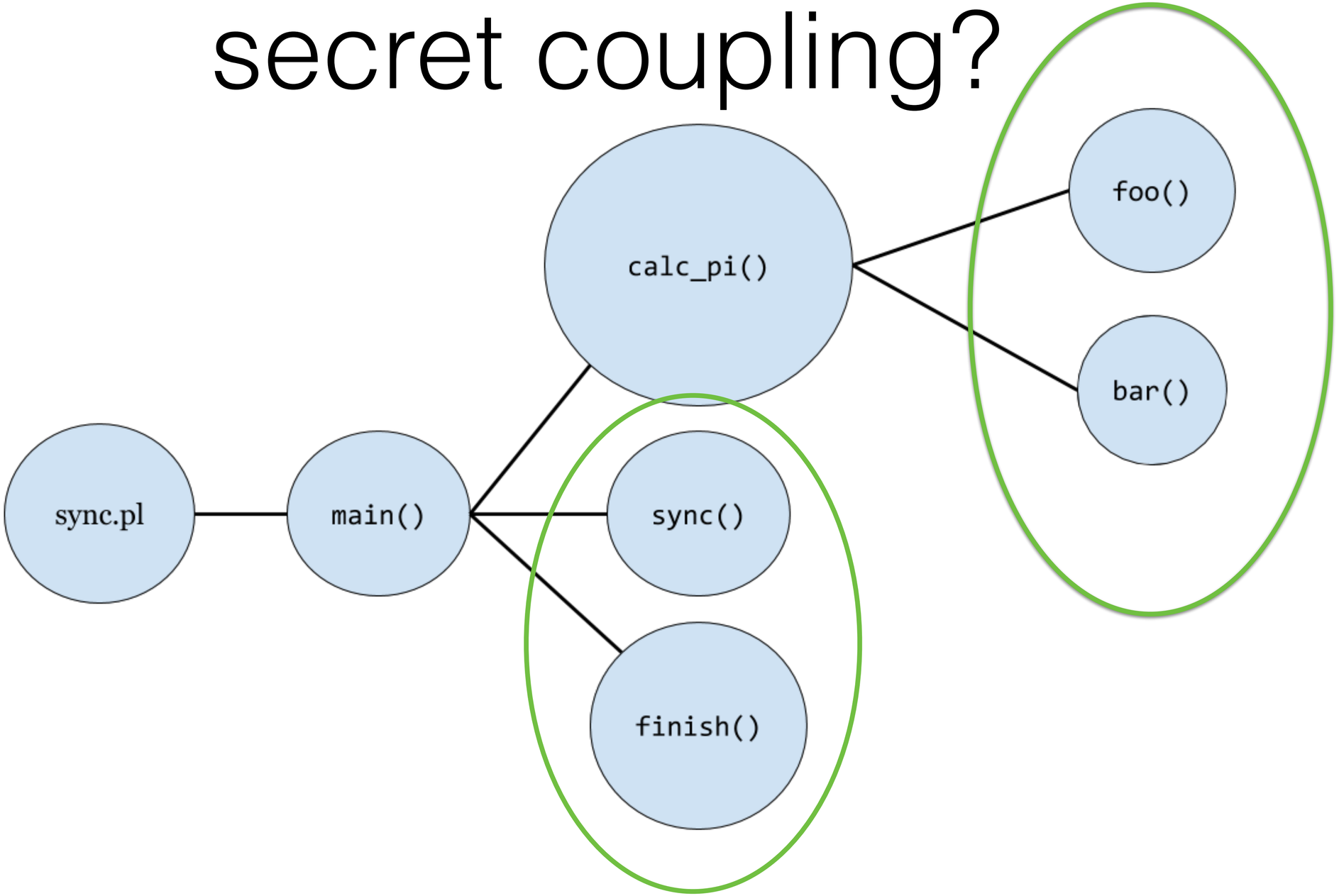
secret coupling?



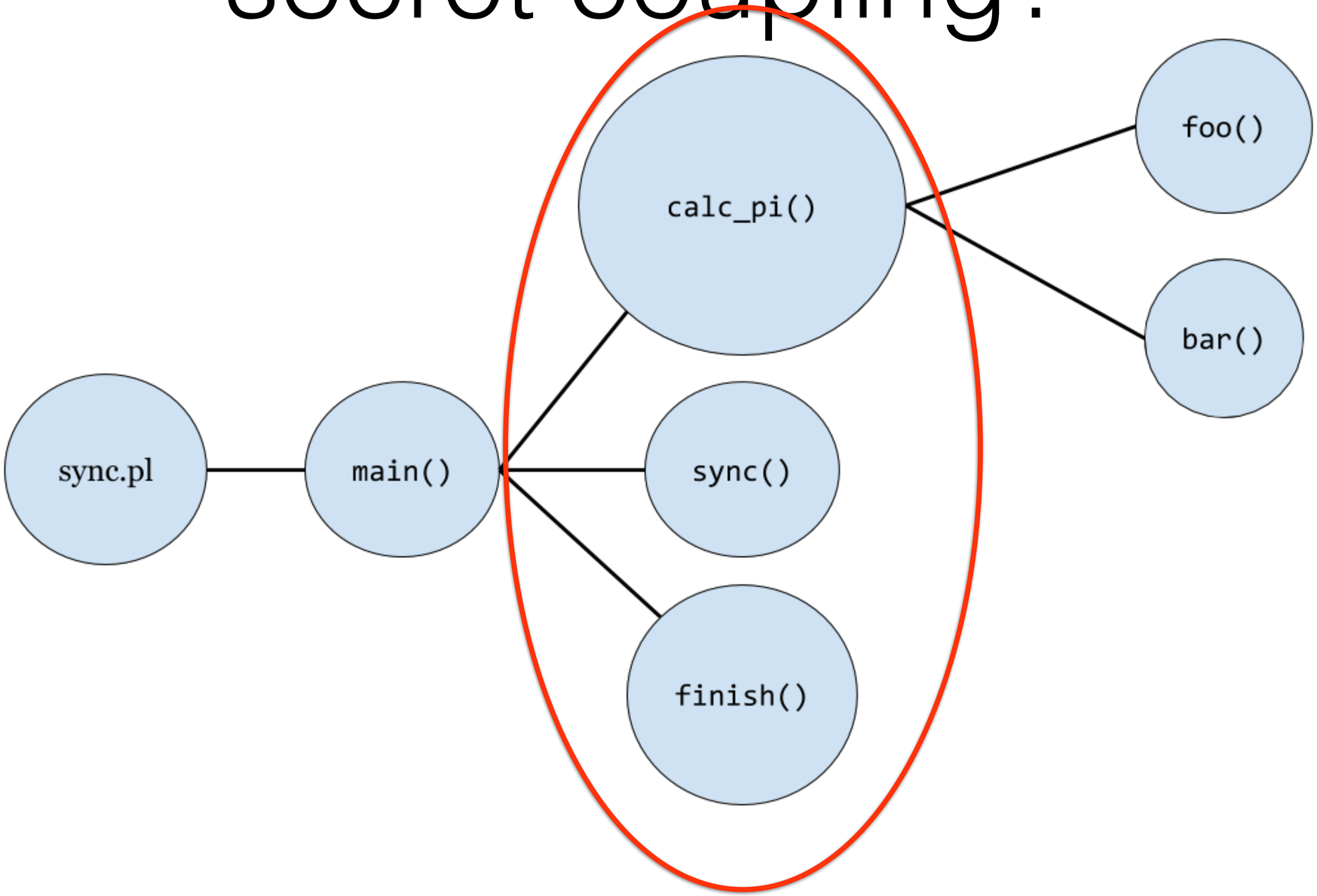
secret coupling?

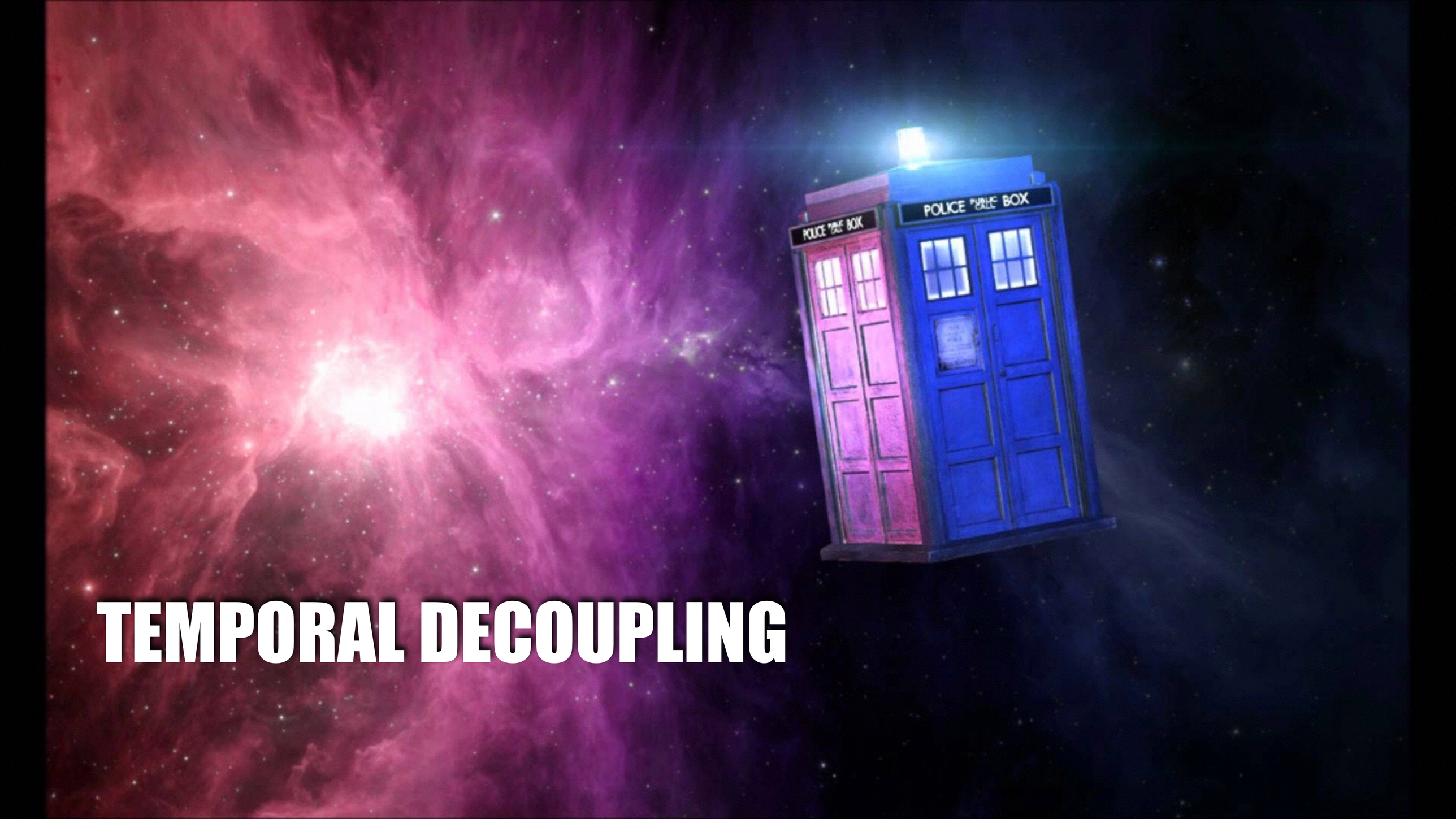


secret coupling?



secret coupling?





TEMPORAL DECOUPLING

Temporal Decoupling - why?

Temporal Decoupling - why?

component simplicity

Temporal Decoupling - why?

component simplicity

enforce interface between components

Temporal Decoupling - why?

component simplicity

enforce interface between components

fault tolerance/reliability

Temporal Decoupling - how?

Temporal Decoupling - how?

sounds a lot like...concurrency

Temporal Decoupling - how?

sounds a lot like...concurrency

"I don’t care what order it runs in"

Temporal Decoupling - how?

sounds a lot like...concurrency

"I don’t care what order it runs in"

concurrency + Perl 5 == event loops

Temporal Decoupling - how?

sounds a lot like...concurrency

"I don’t care what order it runs in"

concurrency + Perl 5 == event loops

POE, AnyEvent, Mojo::IOLoop, IO::Async, etc.

Temporal Decoupling - how?

sounds a lot like...concurrency

"I don’t care what order it runs in"

concurrency + Perl 5 == event loops

POE, AnyEvent, Mojo::IOLoop, IO::Async, etc.

let’s write one!

Overview

unified theory of (un)reliability

juggling raptors

mecha-concurrency

the power of power cycling

The Zen of Erlang

Perl 6

Overview

unified theory of (un)reliability - ✓

juggling raptors

mecha-concurrency

the power of power cycling

The Zen of Erlang

Perl 6

Event Loop

Event Loop

```
while (1) {

}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {









  }
  if (my @writable = $select->can_write(1)) {

  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {









  }
  if (my @writable = $select->can_write(1)) {

  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {









  }
  if (my @writable = $select->can_write(1)) {

  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
  }
  if (my @writable = $select->can_write(1)) {
                                  for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {


      } else {









      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {


      } else {









      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;

      } else {









      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(                    )) {

          } else {

          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {

          } else {

          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              printf "%s data: %s", $sock->fileno, $buffer;
          } else {

          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              printf "%s data: %s", $sock->fileno, $buffer;
          } else {

          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              printf "%s data: %s", $sock->fileno, $buffer;
          } else {
              $select->remove($sock) and $sock->close;
          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              printf "%s data: %s", $sock->fileno, $buffer;
          } else {
              $select->remove($sock) and $sock->close;
          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              # this would be an 'on_read' callback
          } else {
              $select->remove($sock) and $sock->close;
          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              # this would be an 'on_read' callback
          } else {
              # this would be an 'on_close' callback
          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      $_->print("hi there!\n") for @writable;
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              # this would be an 'on_read' callback
          } else {
              # this would be an 'on_close' callback
          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      # on_writable callback
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              # this would be an 'on_read' callback
          } else {
              # this would be an 'on_close' callback
          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      # on_writable callback
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              sleep(5);
          } else {
              # this would be an 'on_close' callback
          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      # on_writable callback
  }
}
```

Event Loop

```
while (1) {
  for my $sock ($select->can_read(1)) {
      if ($sock == $listen) {
          my $new = $listen->accept;
          $select->add($new);
      } else {
          if ($sock->sysread(my $buffer, 4096, 0)) {
              sleep(5); # or DBI fetch
          } else {
              # this would be an 'on_close' callback
          }
      }
  }
  if (my @writable = $select->can_write(1)) {
      # on_writable callback
  }
}
```

Event Loop … Poll

Event Loop … Poll

simpler

Event Loop … Poll

simpler

faster

Event Loop … Poll

simpler

faster

more reliable

Event Loop … Poll

simpler

faster

more reliable

more scalable

Event Loops & Decoupling

Event Loops & Decoupling

minimal decoupling

Event Loops & Decoupling

minimal decoupling

a leaky abstraction

Event Loops & Decoupling

minimal decoupling

a leaky abstraction

even with nicer APIs (promises, futures, etc.)

Event Loops & Decoupling

minimal decoupling

a leaky abstraction

even with nicer APIs (promises, futures, etc.)

why?

Overview

unified theory of (un)reliability - ✓

juggling raptors

mecha-concurrency

the power of power cycling

The Zen of Erlang

Perl 6

Overview

unified theory of (un)reliability - ✓

juggling raptors - ✓

mecha-concurrency

the power of power cycling

The Zen of Erlang

Perl 6

a shrewdness of schedulers

cooperative and preemptive

a shrewdness of schedulers

cooperative

a shrewdness of schedulers

cooperative

event loops

a shrewdness of schedulers

cooperative

event loops

coroutines - Coro, Python generators, Lua

a shrewdness of schedulers

cooperative

event loops

coroutines - Coro, Python generators, Lua

programmer must explicitly 'yield'

a shrewdness of schedulers

cooperative

event loops

coroutines - Coro, Python generators, Lua

programmer must explicitly 'yield'

(call a callback, `->promise` a promise, `->done` a future)

a shrewdness of schedulers

cooperative

event loops

coroutines - Coro, Python generators, Lua

programmer must explicitly 'yield'

(call a callback, `->promise` a promise, `->done` a future)

bad actor can wreck the system

a shrewdness of schedulers

preemptive

a shrewdness of schedulers

preemptive

operating system processes

a shrewdness of schedulers

preemptive

operating system processes

(except Windows before 95, Mac before OS X)

a shrewdness of schedulers

preemptive

operating system processes

(except Windows before 95, Mac before OS X)

threads

a shrewdness of schedulers

preemptive

operating system processes

(except Windows before 95, Mac before OS X)

threads

user code has no say in the matter

a shrewdness of schedulers

preemptive

operating system processes

(except Windows before 95, Mac before OS X)

threads

user code has no say in the matter

scheduler is the boss

a shrewdness of schedulers

cooperative vs. preemptive

a shrewdness of schedulers

cooperative

malloc/free

vs.

preemptive

a shrewdness of schedulers

cooperative

malloc/free

vs.

preemptive

garbage collection

a shrewdness of schedulers

cooperative

malloc/free

vs.

preemptive

garbage collection

a shrewdness of schedulers

cooperative

malloc/free

vs.

preemptive

garbage collection

- fast (when correct)

a shrewdness of schedulers

cooperative

malloc/free

vs.

preemptive

garbage collection

- fast (when correct)
- simple to implement

a shrewdness of schedulers

cooperative

malloc/free

vs.

preemptive

garbage collection

- fast (when correct)
- simple to implement
- fragile

a shrewdness of schedulers

cooperative

malloc/free

vs.

preemptive

garbage collection

- fast (when correct)
- simple to implement
- fragile

a shrewdness of schedulers

cooperative

malloc/free

- fast (when correct)
- simple to implement
- fragile

vs.

preemptive

garbage collection

- slower

a shrewdness of schedulers

cooperative

malloc/free

- fast (when correct)
- simple to implement
- fragile

vs.

preemptive

garbage collection

- slower
- complex implementation

a shrewdness of schedulers

cooperative

malloc/free

- fast (when correct)
- simple to implement
- fragile

vs.

preemptive

garbage collection

- slower
- complex implementation
- resilient to programmer error

a shrewdness of schedulers

cooperative

malloc/free

- fast (when correct)
- simple to implement
- fragile

vs.

preemptive

garbage collection

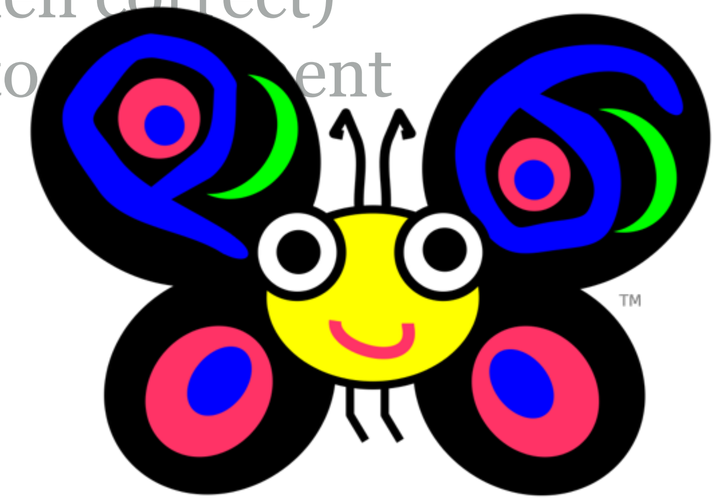
- slower
- complex implementation
- resilient to programmer error

a shrewdness of schedulers

cooperative vs. preemptive

malloc/free vs. garbage collection

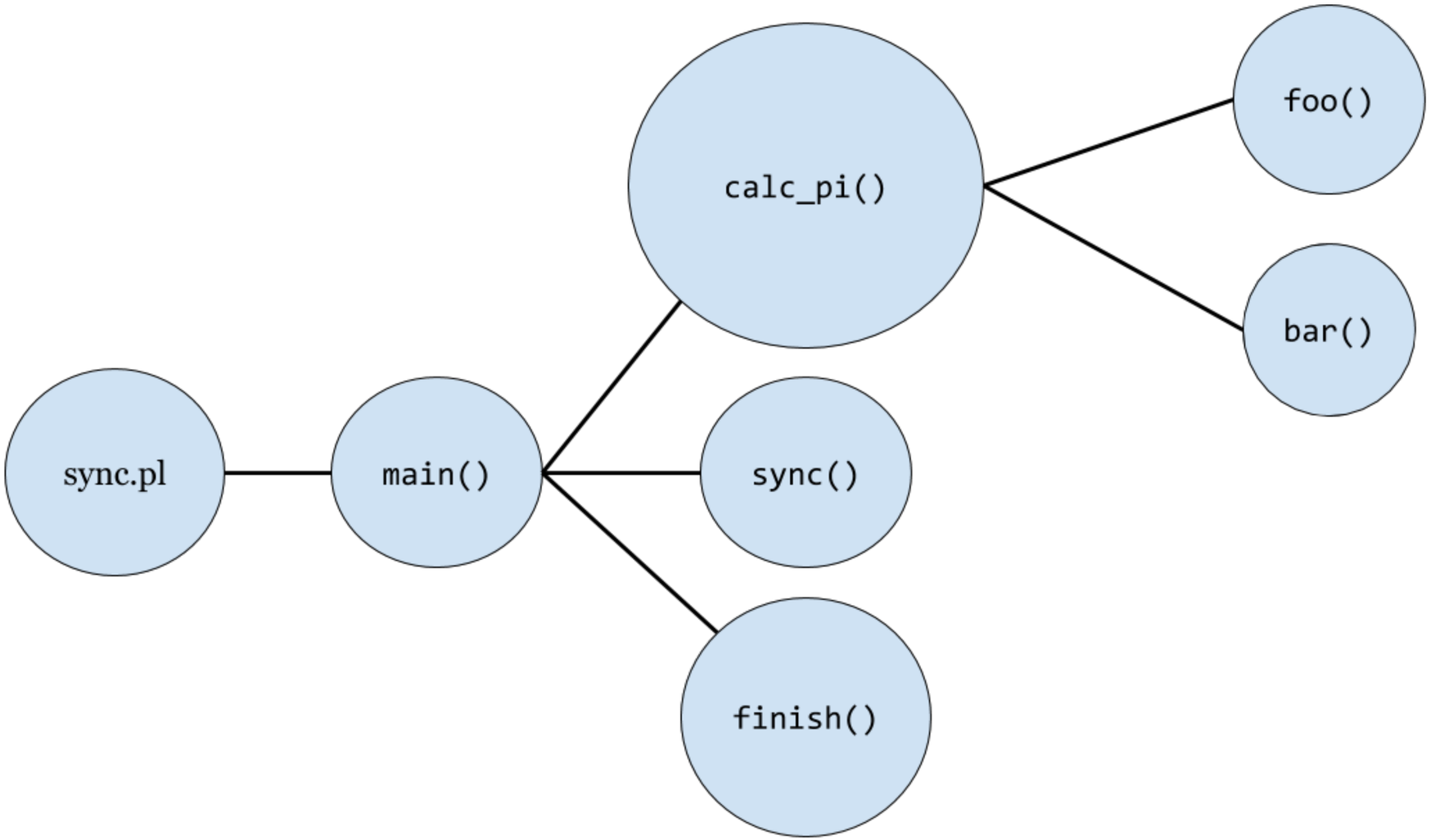
- fast (when correct)
- simple to [illegible]ent
- fragile



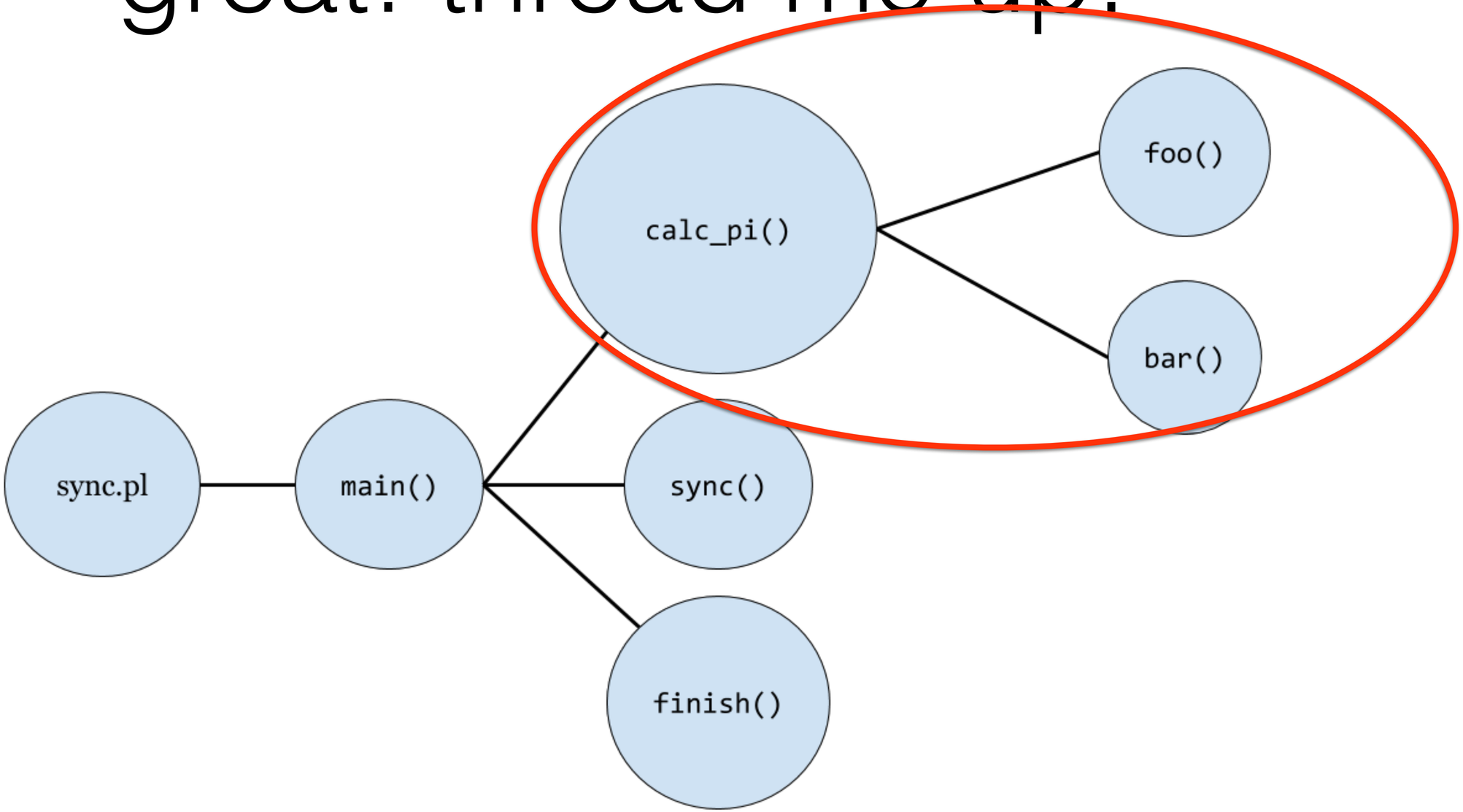
- slower
- complex implementation
- resilient to programmer error

great! thread me up!

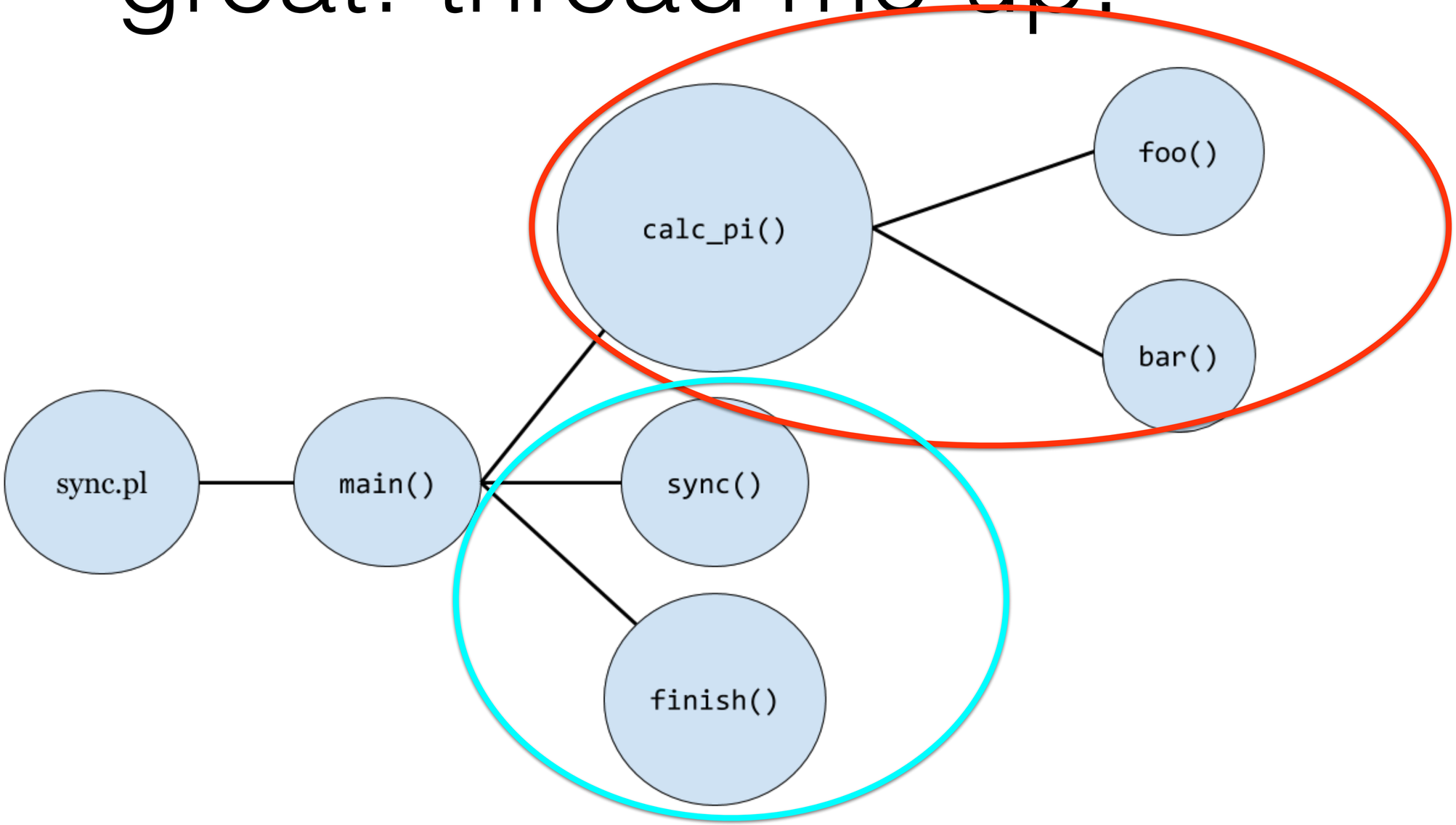
great! thread me up!



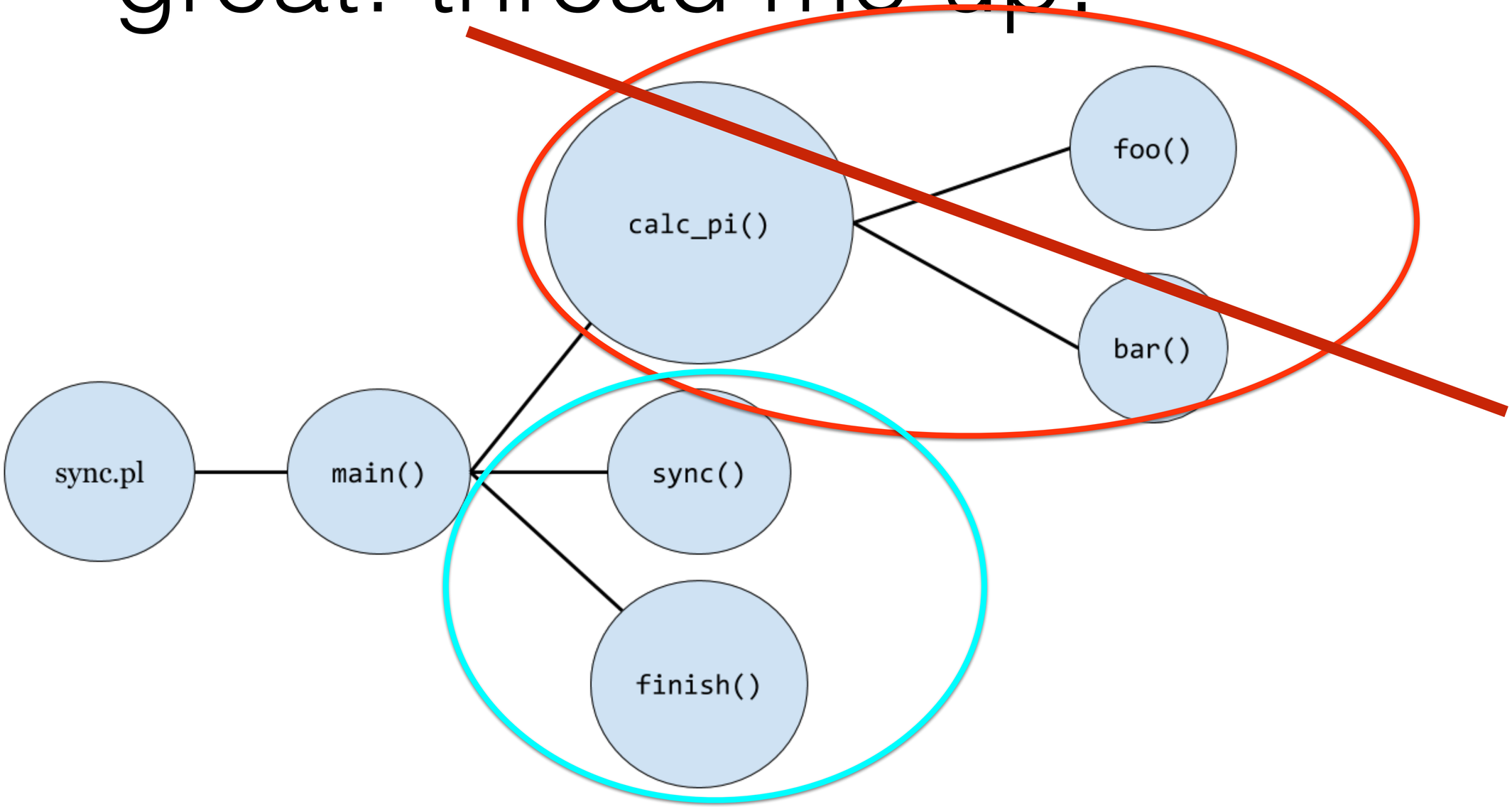
great! thread me up!



great! thread me up!



great! thread me up!




THREADS FOR RELIABILITY?
YOUR OPTIMISM IS ADORABLE

The trouble with threads

The trouble with threads

goal: 'temporally decouple' parts of our program

The trouble with threads

goal: 'temporally decouple' parts of our program

for component simplicity

The trouble with threads

goal: 'temporally decouple' parts of our program

for component simplicity

for enforcement of interfaces

The trouble with threads

goal: 'temporally decouple' parts of our program

for component simplicity

for enforcement of interfaces

for reliability

The trouble with threads: limits

The trouble with threads: limits

```
/proc/sys/kernel/threads-max (since Linux 2.3.11)
       This file specifies the system-wide limit on the number of
       threads (tasks) that can be created on the system.
```

The trouble with threads: limits

```
/proc/sys/kernel/threads-max (since Linux 2.3.11)
       This file specifies the system-wide limit on the number of
       threads (tasks) that can be created on the system.
```

```
$ cat /proc/sys/kernel/threads-max
7744
```

The trouble with threads: limits

```
/proc/sys/kernel/threads-max (since Linux 2.3.11)
       This file specifies the system-wide limit on the number of
       threads (tasks) that can be created on the system.
```

```
$ cat /proc/sys/kernel/threads-max
7744
```

"you are allowed to have 7744 objects, and no more!"

The trouble with threads: coupling

do they help?

The trouble with threads: coupling

do they help?

simplicity? locks, atomics, thread-safe queues…

The trouble with threads: coupling

do they help?

simplicity? locks, atomics, thread-safe queues…

interfaces? same memory space

The trouble with threads: coupling

do they help?

simplicity? locks, atomics, thread-safe queues…

interfaces? same memory space

reliability? unsafe termination/restart

Termination/restart?

Overview

unified theory of (un)reliability - ✓

juggling raptors - ✓

mecha-concurrency

the power of power cycling

The Zen of Erlang

Perl 6

Overview

unified theory of (un)reliability - ✓

juggling raptors - ✓

mecha-concurrency - ✓

the power of power cycling

The Zen of Erlang

Perl 6

Turning it off and back on again

the magic words in software reliability

Why is restarting *so good?*

Why is restarting *so good?*

some bugs happen every time some code runs (*bohrbugs*)

Why is restarting *so good?*

some bugs happen every time some code runs (*bohrbugs*)

these are easy to reproduce, and thus to fix

Why is restarting *so good?*

some bugs happen every time some code runs (*bohrbugs*)

these are easy to reproduce, and thus to fix

some bugs happen almost never (*heisenbugs*)

Why is restarting *so good?*

some bugs happen every time some code runs (*bohrbugs*)

these are easy to reproduce, and thus to fix

some bugs happen almost never (*heisenbugs*)

these are impossible* to reproduce, and thus to fix

Why is restarting *so good?*

Why is restarting *so good?*

bohrbugs happen every time, so restarting doesn't help

Why is restarting *so good?*

bohrbugs happen every time, so restarting doesn't help

(but you should have caught it in dev, it happens every time!)

Why is restarting *so good?*

bohrbugs happen every time, so restarting doesn't help

(but you should have caught it in dev, it happens every time!)

heisenbugs happen almost never, so restarting does the trick!

Why is restarting *so good?*

bohrbugs happen every time, so restarting doesn't help

(but you should have caught it in dev, it happens every time!)

heisenbugs happen almost never, so restarting does the trick!

(and 'almost never' at 10000 requests/second -> every minute)

Why is restarting *so good?*

Why is restarting *so good?*

reliable systems do not need to run error-free!

Why is restarting *so good?*

reliable systems do not need to run error-free!

can we use this?

Why is restarting *so good?*

reliable systems do not need to run error-free!

can we use this?

(we do, all the time: operating systems)

Operating systems are the bomb

Operating systems are the bomb

preemptively scheduled no-shared-state isolated safe-to-kill safe-to-restart units of concurrency

Operating systems are the bomb

preemptively scheduled no-shared-state isolated safe-to-kill safe-to-restart units of concurrency

(aka "processes")

Operating systems are the bomb

preemptively scheduled no-shared-state isolated safe-to-kill safe-to-restart units of concurrency

(aka "processes")

tree of reliability: init or supervisord above things that do actual work (uwsgi, starman, etc.)

Operating systems are the bomb

preemptively scheduled no-shared-state isolated safe-to-kill safe-to-restart units of concurrency

(aka "processes")

tree of reliability: init or supervisord above things that do actual work (uwsgi, starman, etc.)

it’s ok if the web worker crashes, less so if init goes down

Operating systems are the bomb

wouldn’t it be nice to use this at a finer granularity? like a single function?

Overview

unified theory of (un)reliability - ✓

juggling raptors - ✓

mecha-concurrency - ✓

the power of power cycling

The Zen of Erlang

Perl 6

Overview

unified theory of (un)reliability - ✓

juggling raptors - ✓

mecha-concurrency - ✓

the power of power cycling - ✓

The Zen of Erlang

Perl 6

Erlang (the blurb)

"... a general-purpose, concurrent, functional programming language"

Erlang (the hype)

Erlang (the hype)

(probably) runs your telephone!

Erlang (the hype)

(probably) runs your telephone!

many nines!

Erlang (the hype)

(probably) runs your telephone!

many nines!

millions of sockets on a single system!

Erlang (the hype)

(probably) runs your telephone!

many nines!

millions of sockets on a single system!

built-in cross-server coordination!

Erlang (the hype)

(probably) runs your telephone!

many nines!

millions of sockets on a single system!

built-in cross-server coordination!

as mature as Perl!

Erlang (the insight)

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads
forked child processes

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads
forked child processes
processes on the OS

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads
forked child processes
processes on the OS
OS virtualization

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads
forked child processes
processes on the OS
OS virtualization
services with a network API

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads
forked child processes
processes on the OS
OS virtualization
services with a network API
systems

Erlang (the insight)

Erlang (the insight)

(true) concurrency and reliability are two sides of the same coin.

Erlang (the insight)

(true) concurrency and reliability are two sides of the same coin.

pieces need to fail and be restarted without breaking other stuff.

Erlang (the insight)

(true) concurrency and reliability are two sides of the same coin.

pieces need to fail and be restarted without breaking other stuff.

operating systems get this.

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads
forked child processes
processes on the OS
OS virtualization
services with a network API
systems

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads
forked child processes
processes on the OS
OS virtualization
services with a network API
systems

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads
forked child processes
processes on the OS
OS virtualization
services with a network API
systems

Erlang takes this

Erlang (the insight)

a hierarchy of coupled-ness
(from most to least)

lines of code
functions/classes
threads
forked child processes
processes on the OS
OS virtualization
services with a network API
systems

... and applies it here

Erlang takes this

Erlang (the OS)

Erlang (the OS)

"processes"

Erlang (the OS)

"processes"

'green processes' - not OS processes!

Erlang (the OS)

"processes"

'green processes' - not OS processes!

isolated memory

Erlang (the OS)

"processes"

'green processes' - not OS processes!

isolated memory

preemptively scheduled

Erlang (the OS)

"processes"

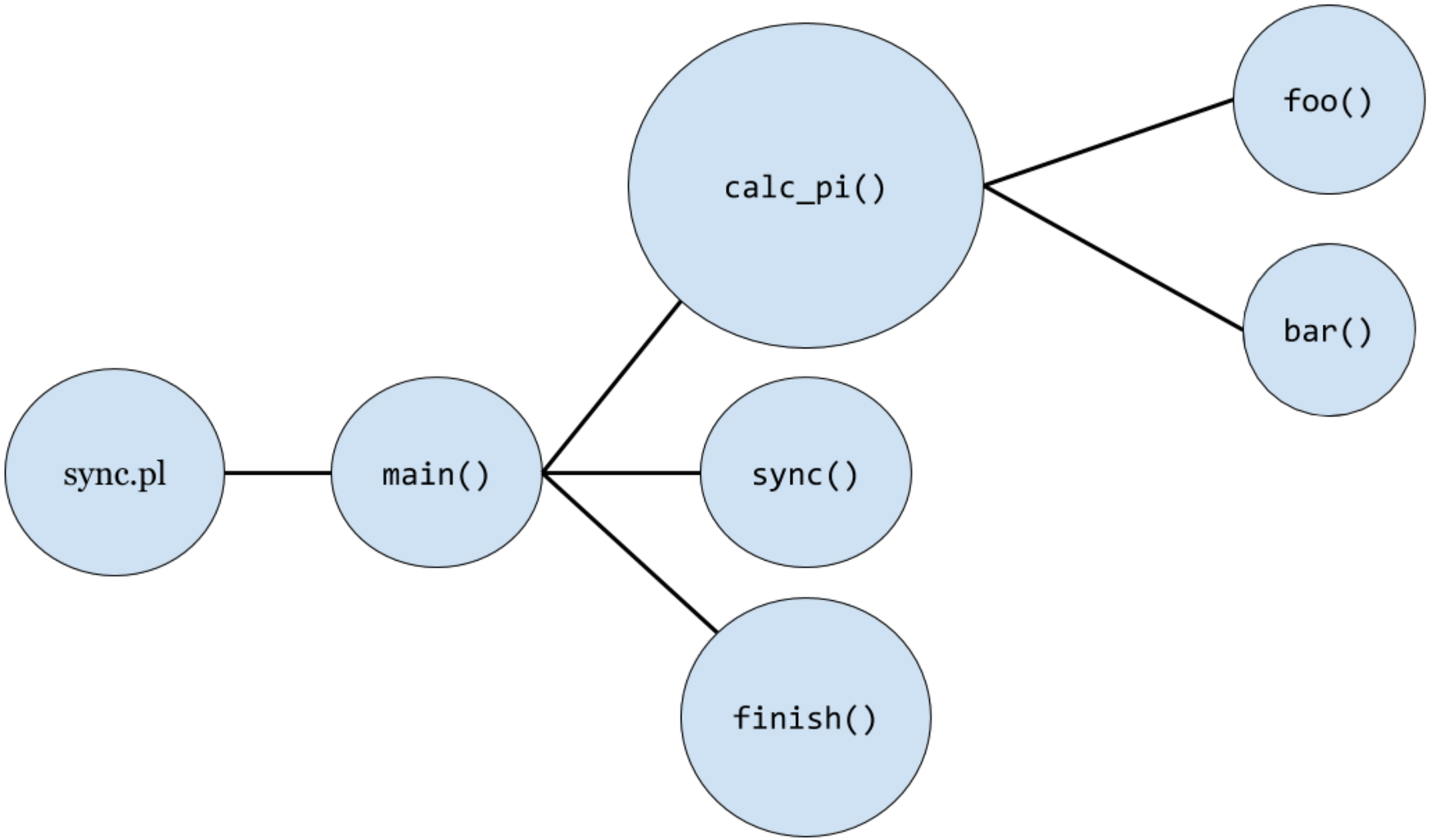
'green processes' - not OS processes!

isolated memory

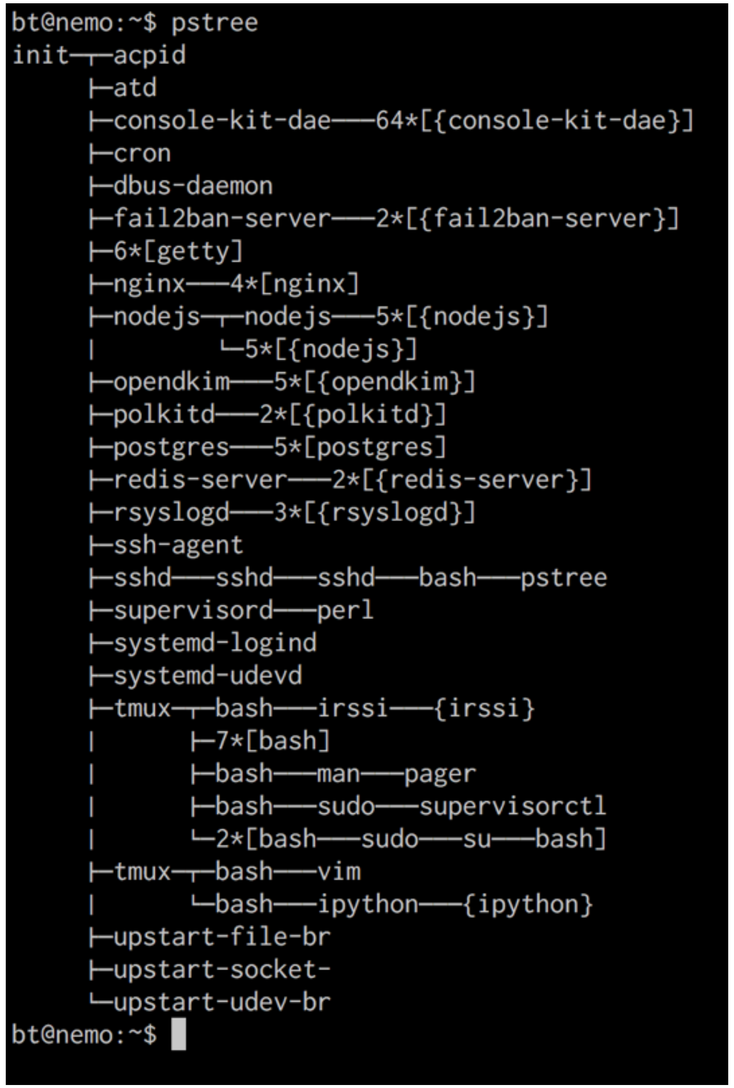
preemptively scheduled

safe to kill, safe to restart

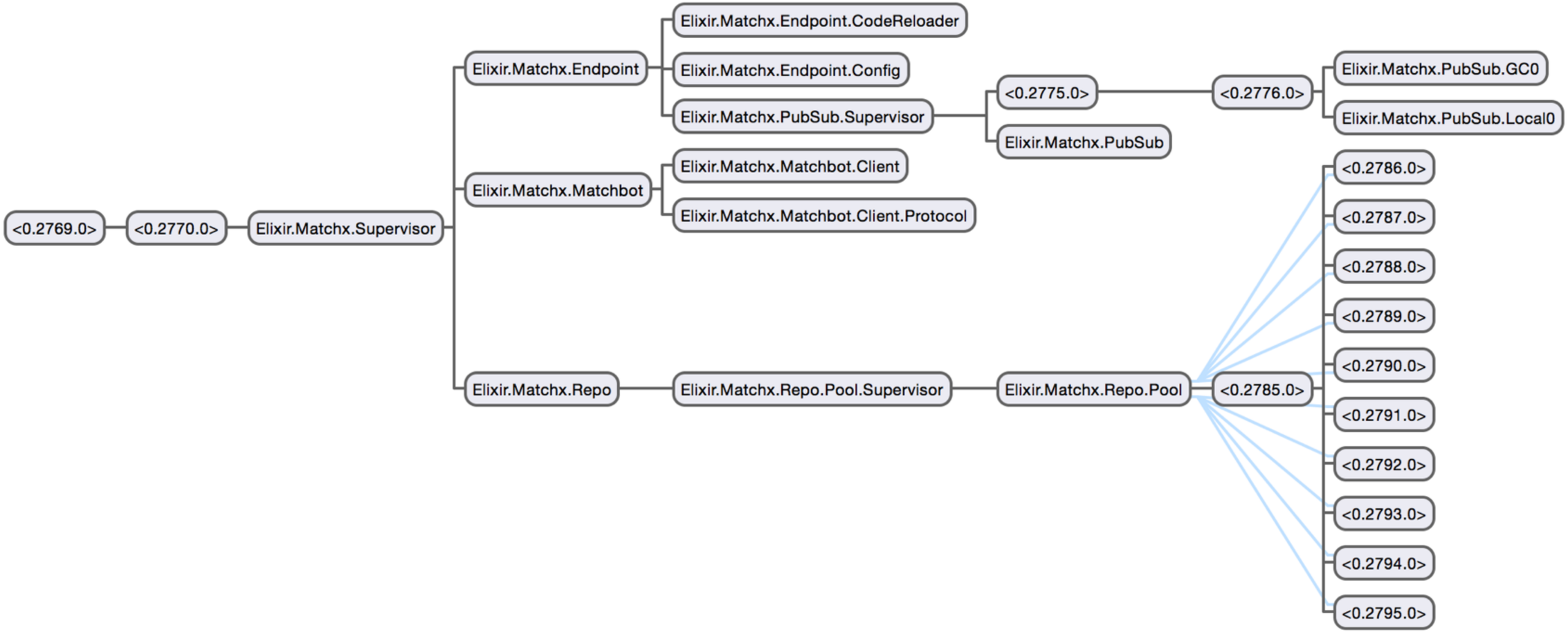
program structure



program structure

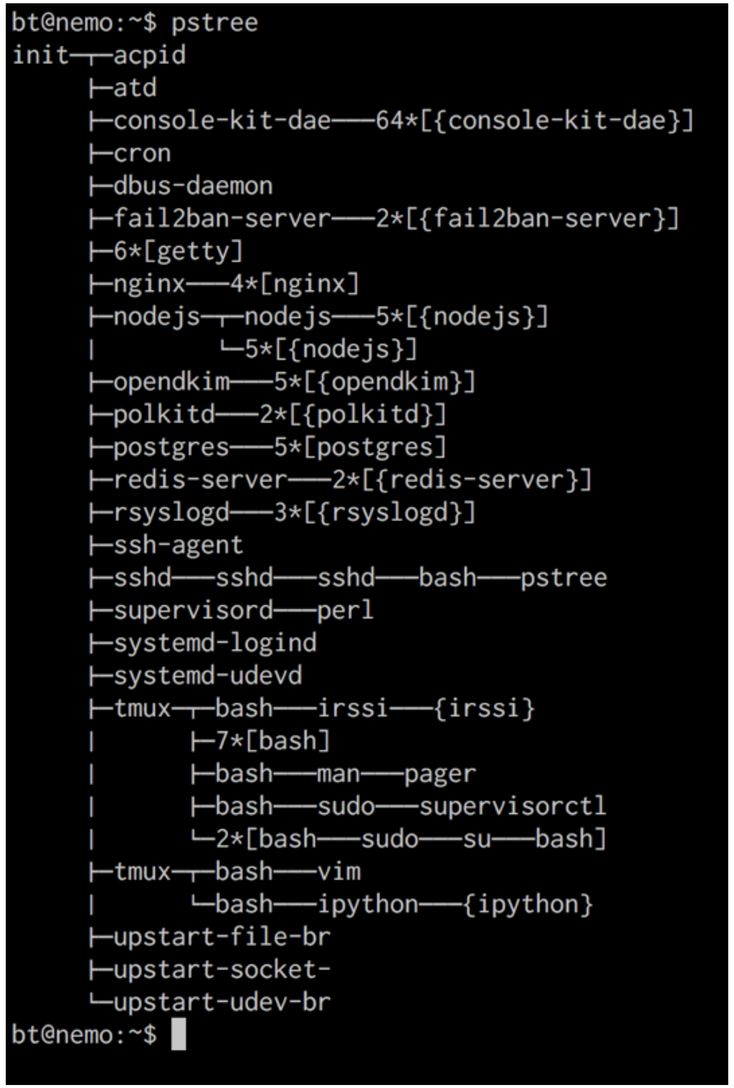


program structure



"let it crash"

"let it crash"


bt@nemo:~$ pstree
init─┬─acpid
├─atd
├─console-kit-dae───64*[{console-kit-dae}]
├─cron
├─dbus-daemon
├─fail2ban-server───2*[{fail2ban-server}]
├─6*[getty]
├─nginx───4*[nginx]
├─nodejs─┬─nodejs───5*[{nodejs}]
│ └─5*[{nodejs}]
├─opendkim───5*[{opendkim}]
├─polkitd───2*[{polkitd}]
├─postgres───5*[postgres]
├─redis-server───2*[{redis-server}]
├─rsyslogd───3*[{rsyslogd}]
├─ssh-agent
├─sshd───sshd───sshd───bash───pstree
├─supervisord───perl
├─systemd-logind
├─systemd-udevd
├─tmux─┬─bash───irssi───{irssi}
│ ├─7*[bash]
│ ├─bash───man───pager
│ ├─bash───sudo───supervisorctl
│ └─2*[bash───sudo───su───bash]
├─tmux─┬─bash───vim
│ └─bash───ipython───{ipython}
├─upstart-file-br
├─upstart-socket-
└─upstart-udev-br
bt@nemo:~$

"let it crash"

sshd

"let it crash"

init

sshd

"let it crash"

init

sshd

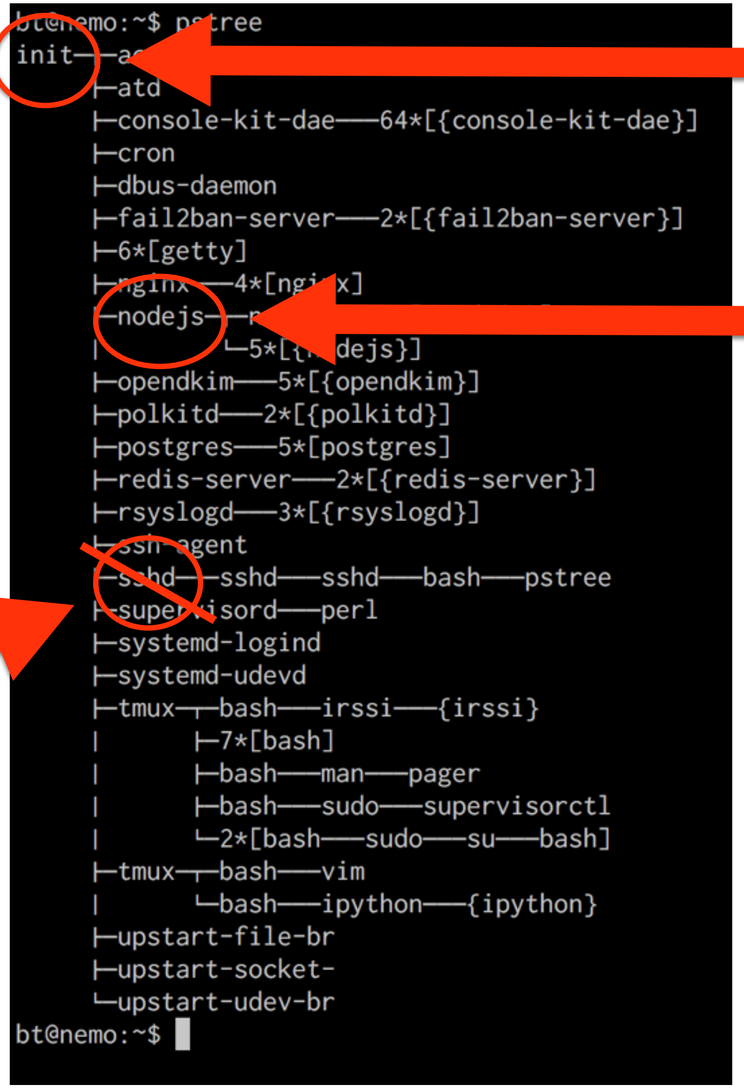
"let it crash"

sshd

```
bt@nemo:~$ pstree
init─┬─ac
     ├─atd
     ├─console-kit-dae───64*[{console-kit-dae}]
     ├─cron
     ├─dbus-daemon
     ├─fail2ban-server───2*[{fail2ban-server}]
     ├─6*[getty]
     ├─nginx───4*[nginx]
     ├─nodejs─┬─nodejs───5*[{nodejs}]
     │        └─5*[{nodejs}]
     ├─opendkim───5*[{opendkim}]
     ├─polkitd───2*[{polkitd}]
     ├─postgres───5*[postgres]
     ├─redis-server───2*[{redis-server}]
     ├─rsyslogd───3*[{rsyslogd}]
     ├─ssh-agent
     ├─sshd─┬─sshd───sshd───bash───pstree
     ├─supervisord───perl
     ├─systemd-logind
     ├─systemd-udevd
     ├─tmux─┬─bash───irssi───{irssi}
     │      ├─7*[bash]
     │      ├─bash───man───pager
     │      ├─bash───sudo───supervisorctl
     │      └─2*[bash───sudo───su───bash]
     ├─tmux─┬─bash───vim
     │      └─bash───ipython───{ipython}
     ├─upstart-file-br
     ├─upstart-socket-
     └─upstart-udev-br
bt@nemo:~$
```

init

"let it crash"

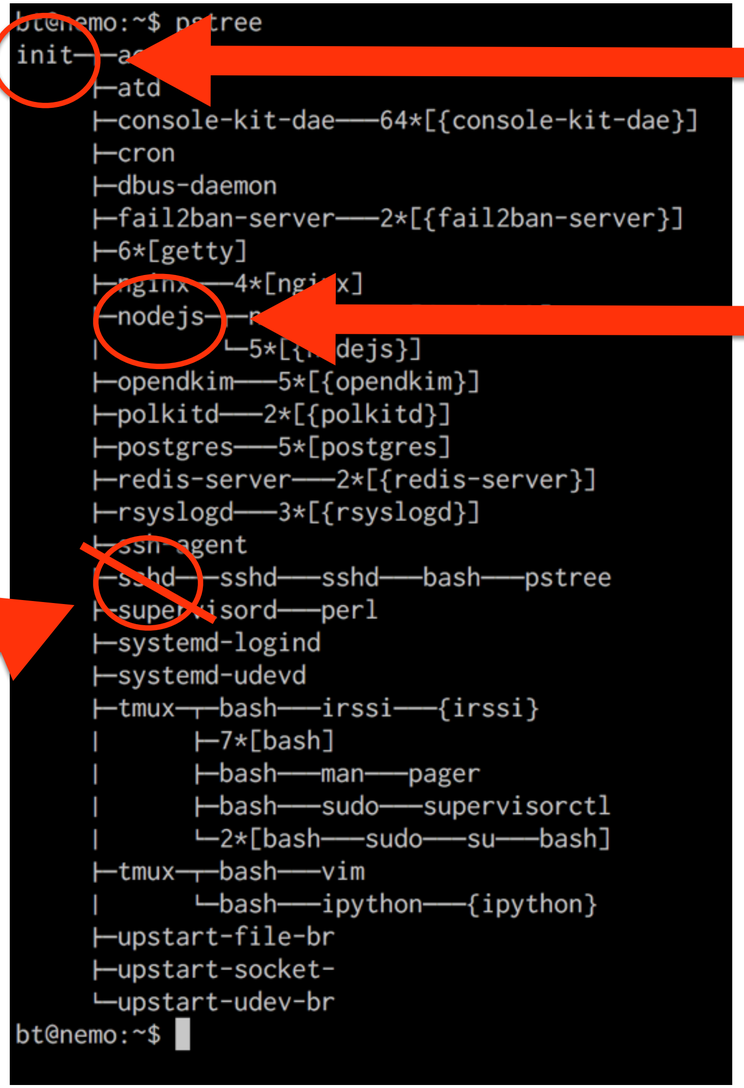


init

node.js

sshd

"let it crash"



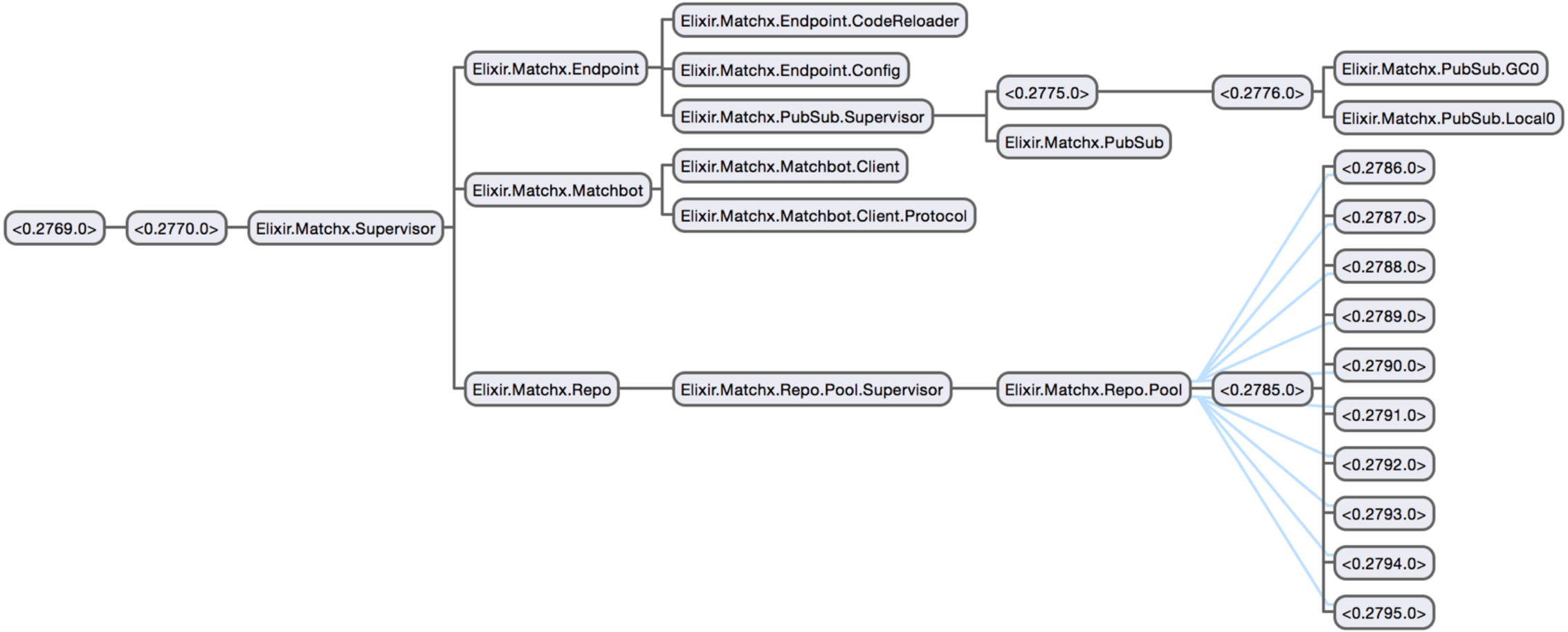
`sshd`

`init`

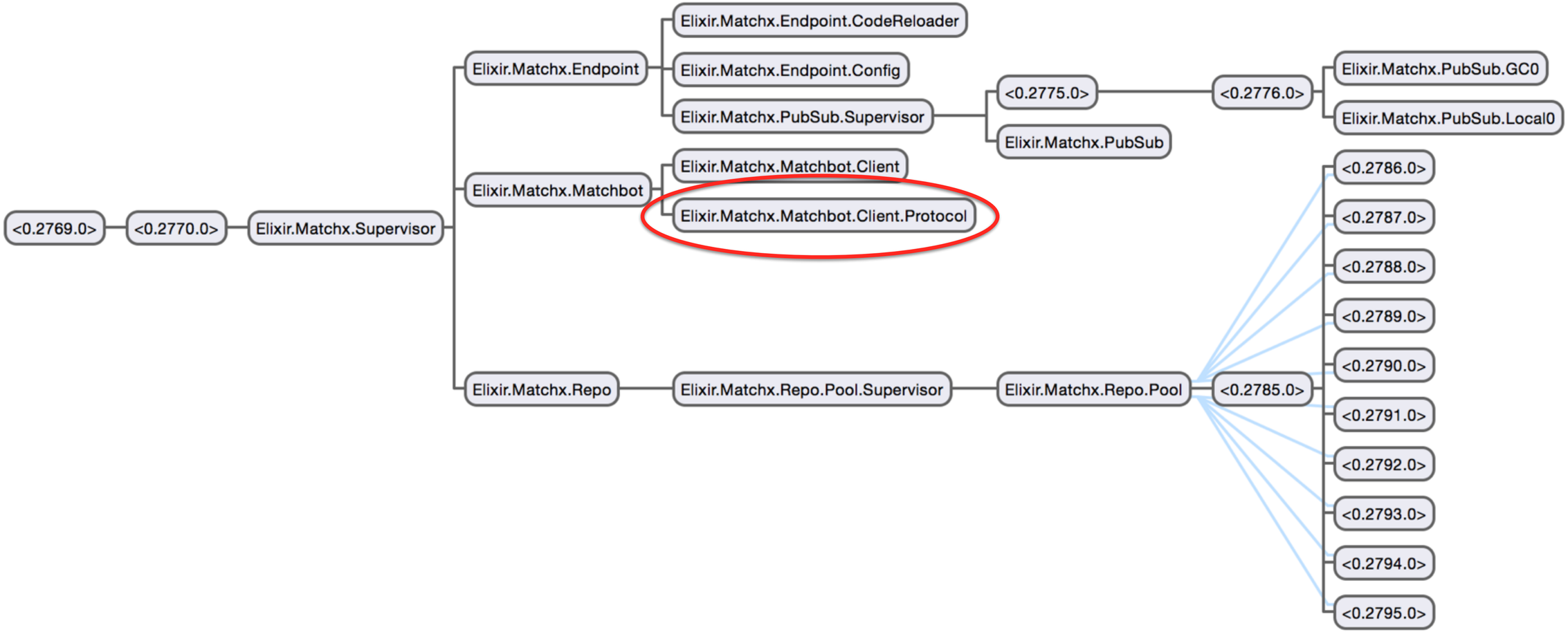
`node.js`

… completely ignorant of sshd's troubles

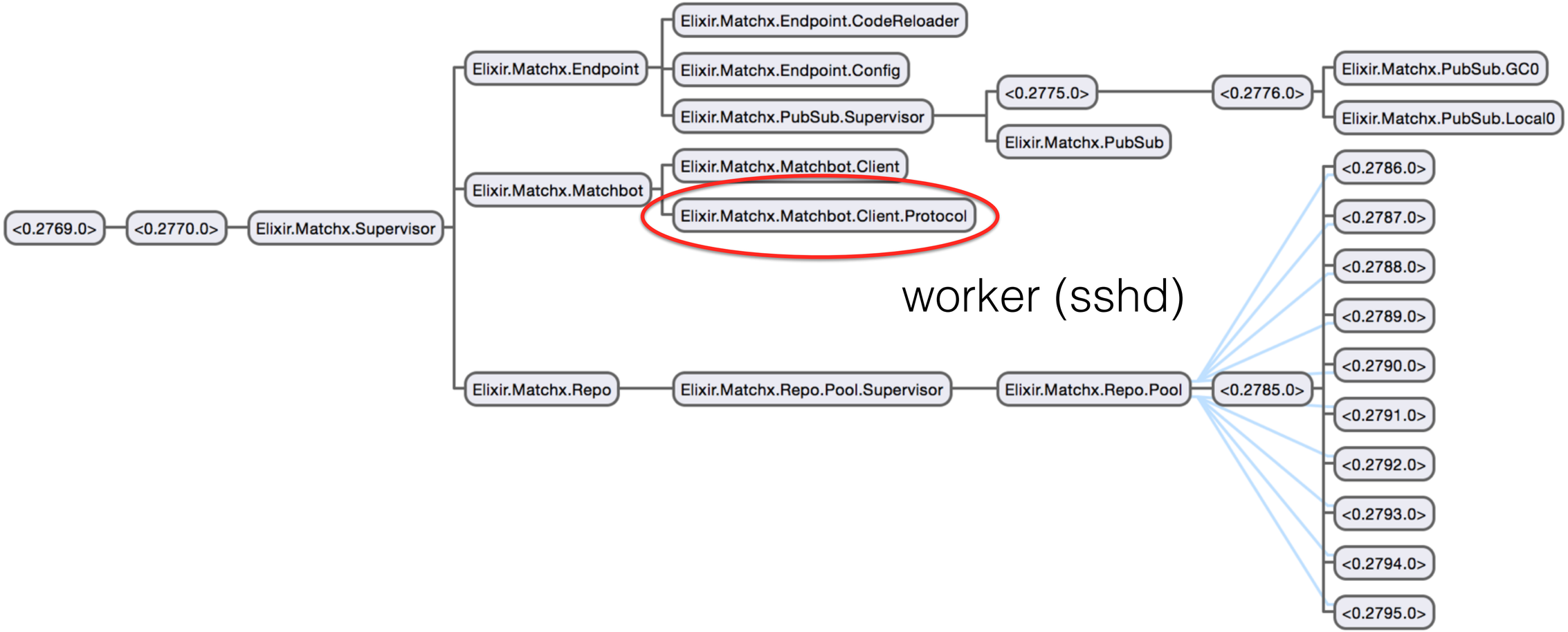
"let it crash"


<0.2769.0>
<0.2770.0>
Elixir.Matchx.Supervisor
Elixir.Matchx.Endpoint
Elixir.Matchx.Endpoint.CodeReloader
Elixir.Matchx.Endpoint.Config
Elixir.Matchx.PubSub.Supervisor
<0.2775.0>
<0.2776.0>
Elixir.Matchx.PubSub.GC0
Elixir.Matchx.PubSub.Local0
Elixir.Matchx.PubSub
Elixir.Matchx.Matchbot
Elixir.Matchx.Matchbot.Client
Elixir.Matchx.Matchbot.Client.Protocol
Elixir.Matchx.Repo
Elixir.Matchx.Repo.Pool.Supervisor
Elixir.Matchx.Repo.Pool
<0.2785.0>
<0.2786.0>
<0.2787.0>
<0.2788.0>
<0.2789.0>
<0.2790.0>
<0.2791.0>
<0.2792.0>
<0.2793.0>
<0.2794.0>
<0.2795.0>

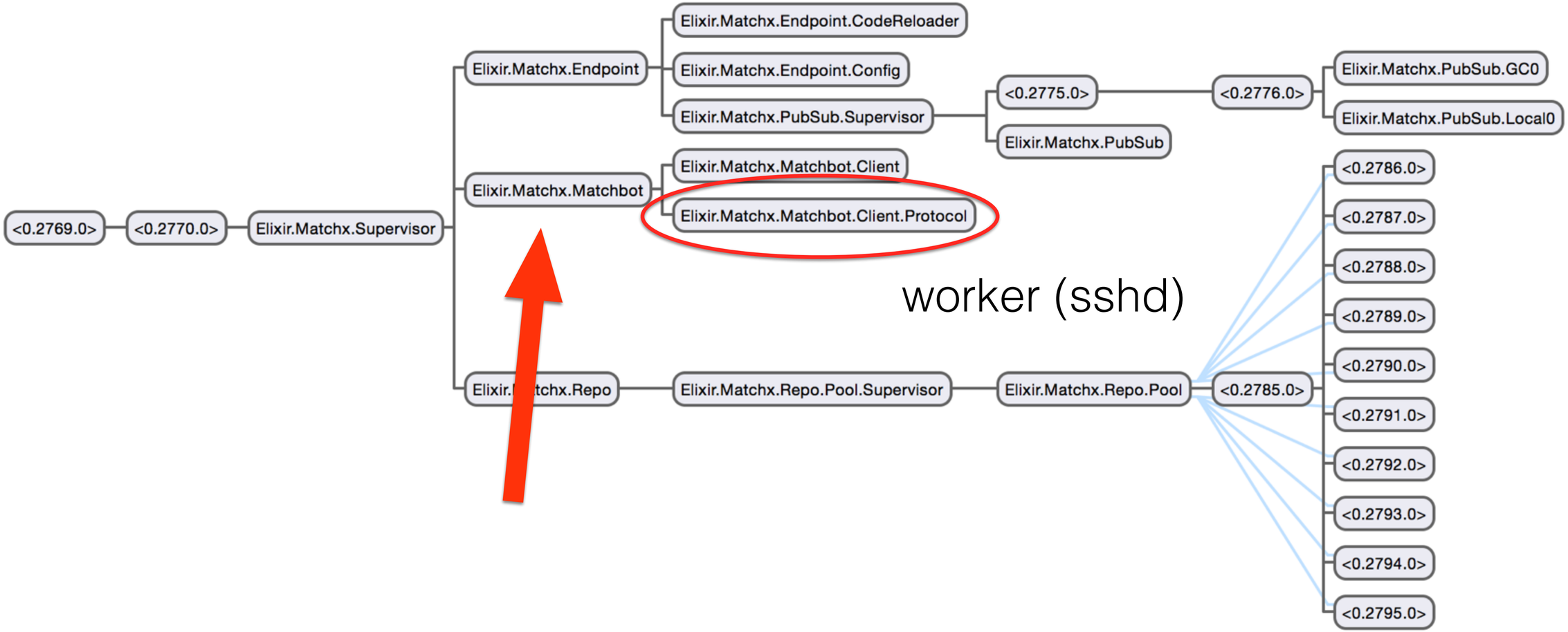
"let it crash"


<0.2769.0>
<0.2770.0>
Elixir.Matchx.Supervisor
Elixir.Matchx.Endpoint
Elixir.Matchx.Endpoint.CodeReloader
Elixir.Matchx.Endpoint.Config
Elixir.Matchx.PubSub.Supervisor
<0.2775.0>
<0.2776.0>
Elixir.Matchx.PubSub.GC0
Elixir.Matchx.PubSub.Local0
Elixir.Matchx.PubSub
Elixir.Matchx.Matchbot
Elixir.Matchx.Matchbot.Client
Elixir.Matchx.Matchbot.Client.Protocol
Elixir.Matchx.Repo
Elixir.Matchx.Repo.Pool.Supervisor
Elixir.Matchx.Repo.Pool
<0.2785.0>
<0.2786.0>
<0.2787.0>
<0.2788.0>
<0.2789.0>
<0.2790.0>
<0.2791.0>
<0.2792.0>
<0.2793.0>
<0.2794.0>
<0.2795.0>

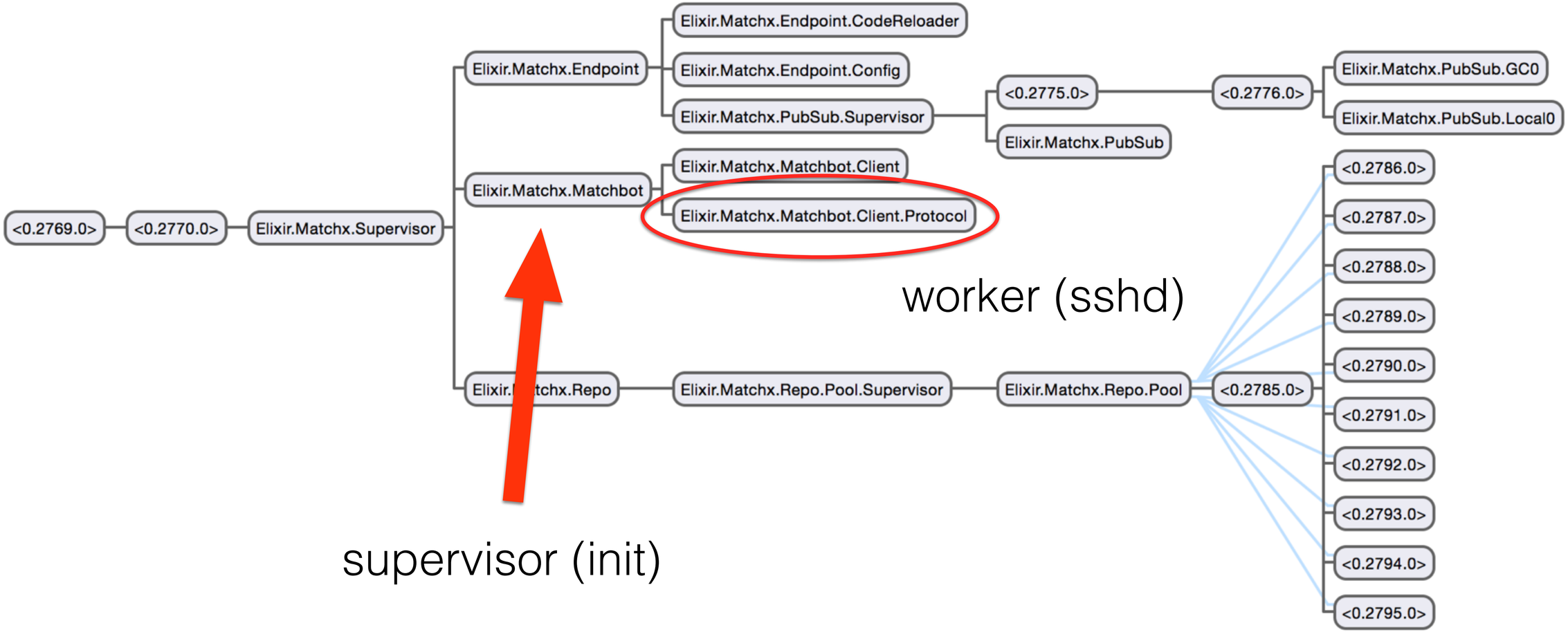
"let it crash"


<0.2769.0>
<0.2770.0>
Elixir.Matchx.Supervisor
Elixir.Matchx.Endpoint
Elixir.Matchx.Endpoint.CodeReloader
Elixir.Matchx.Endpoint.Config
Elixir.Matchx.PubSub.Supervisor
<0.2775.0>
Elixir.Matchx.PubSub
<0.2776.0>
Elixir.Matchx.PubSub.GC0
Elixir.Matchx.PubSub.Local0
Elixir.Matchx.Matchbot
Elixir.Matchx.Matchbot.Client
Elixir.Matchx.Matchbot.Client.Protocol
worker (sshd)
Elixir.Matchx.Repo
Elixir.Matchx.Repo.Pool.Supervisor
Elixir.Matchx.Repo.Pool
<0.2785.0>
<0.2786.0>
<0.2787.0>
<0.2788.0>
<0.2789.0>
<0.2790.0>
<0.2791.0>
<0.2792.0>
<0.2793.0>
<0.2794.0>
<0.2795.0>

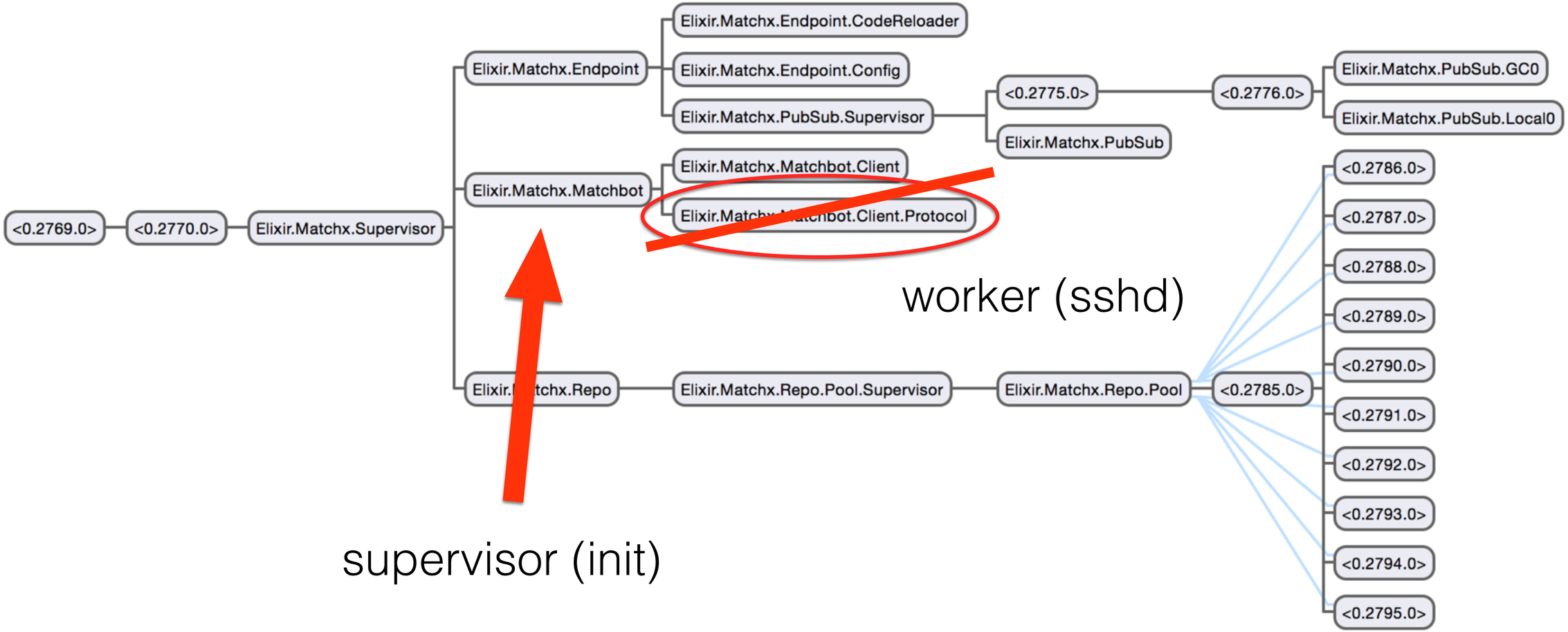
"let it crash"


<0.2769.0>
<0.2770.0>
Elixir.Matchx.Supervisor
Elixir.Matchx.Endpoint
Elixir.Matchx.Endpoint.CodeReloader
Elixir.Matchx.Endpoint.Config
Elixir.Matchx.PubSub.Supervisor
<0.2775.0>
Elixir.Matchx.PubSub
<0.2776.0>
Elixir.Matchx.PubSub.GC0
Elixir.Matchx.PubSub.Local0
Elixir.Matchx.Matchbot
Elixir.Matchx.Matchbot.Client
Elixir.Matchx.Matchbot.Client.Protocol
worker (sshd)
Elixir.Matchx.Repo
Elixir.Matchx.Repo.Pool.Supervisor
Elixir.Matchx.Repo.Pool
<0.2785.0>
<0.2786.0>
<0.2787.0>
<0.2788.0>
<0.2789.0>
<0.2790.0>
<0.2791.0>
<0.2792.0>
<0.2793.0>
<0.2794.0>
<0.2795.0>

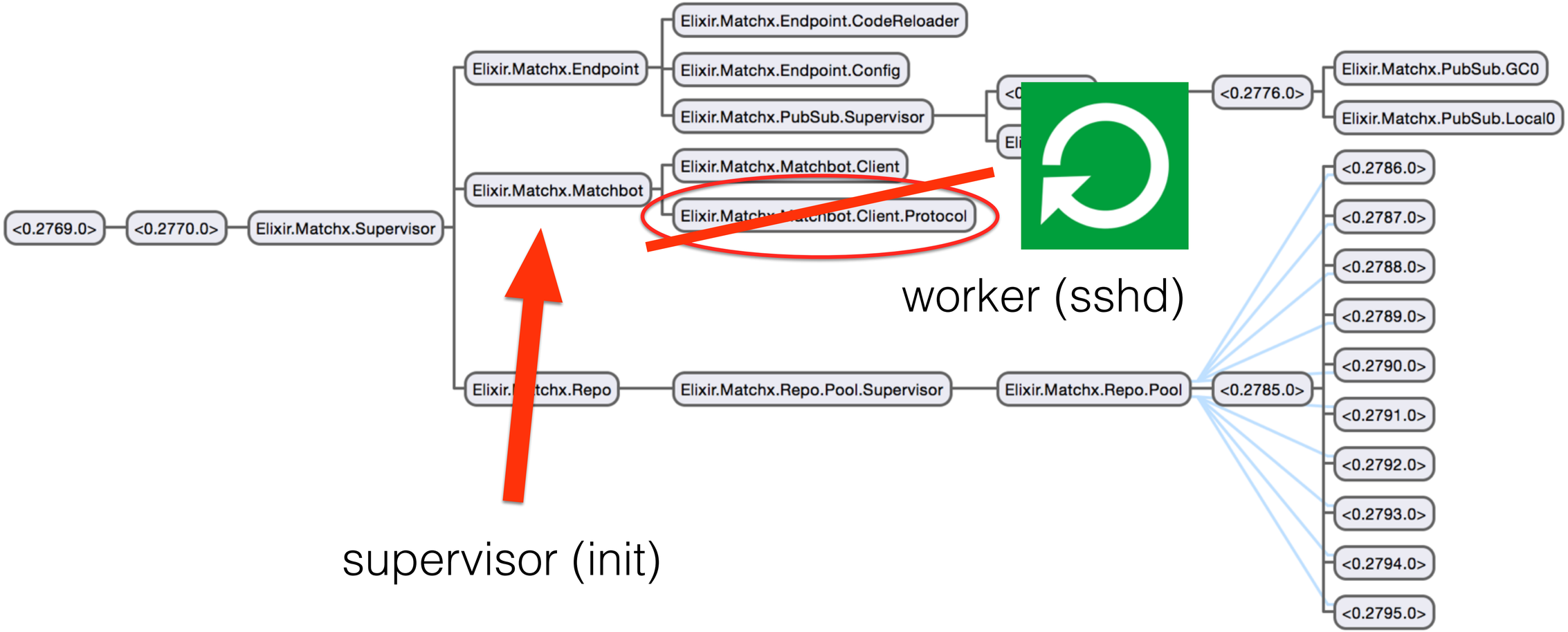
"let it crash"


<0.2769.0>
<0.2770.0>
Elixir.Matchx.Supervisor
Elixir.Matchx.Endpoint
Elixir.Matchx.Endpoint.CodeReloader
Elixir.Matchx.Endpoint.Config
Elixir.Matchx.PubSub.Supervisor
<0.2775.0>
Elixir.Matchx.PubSub
<0.2776.0>
Elixir.Matchx.PubSub.GC0
Elixir.Matchx.PubSub.Local0
Elixir.Matchx.Matchbot
Elixir.Matchx.Matchbot.Client
Elixir.Matchx.Matchbot.Client.Protocol
worker (sshd)
Elixir.Matchx.Repo
Elixir.Matchx.Repo.Pool.Supervisor
Elixir.Matchx.Repo.Pool
<0.2785.0>
<0.2786.0>
<0.2787.0>
<0.2788.0>
<0.2789.0>
<0.2790.0>
<0.2791.0>
<0.2792.0>
<0.2793.0>
<0.2794.0>
<0.2795.0>
supervisor (init)

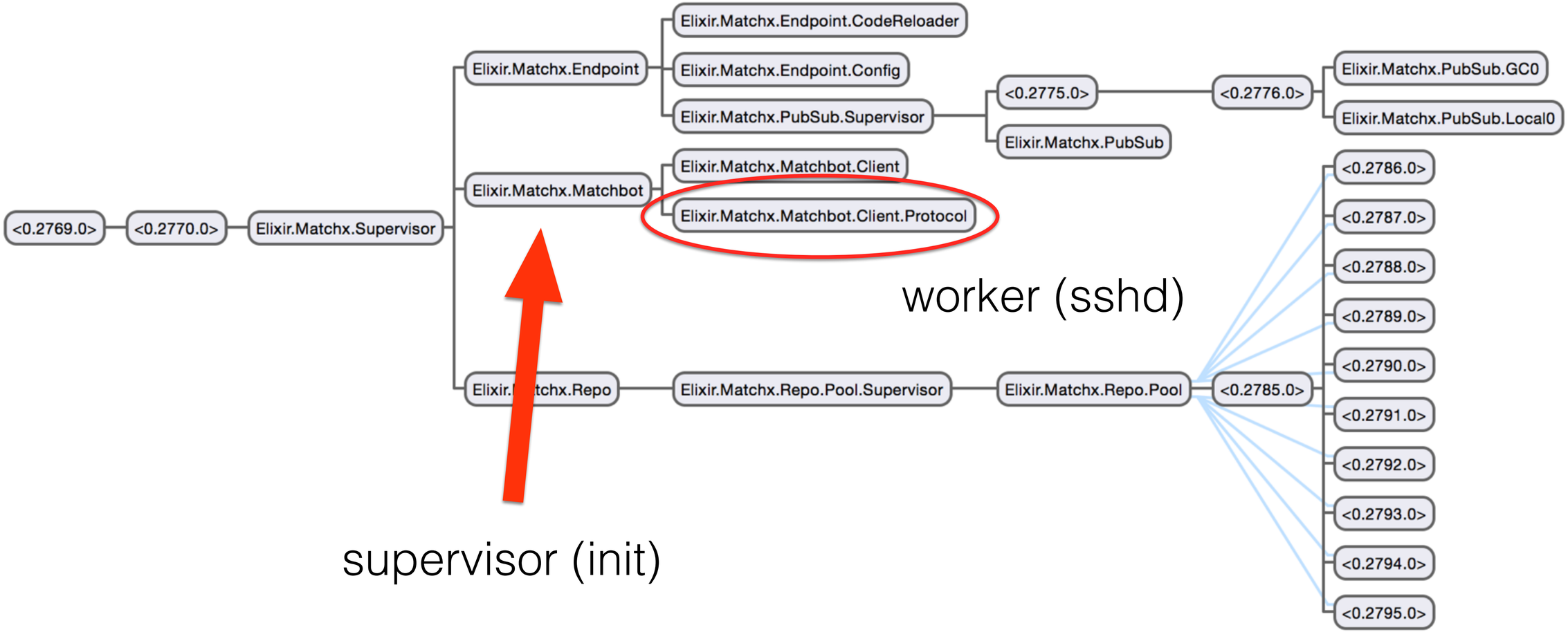
"let it crash"


Elixir.Matchx.Endpoint.CodeReloader
Elixir.Matchx.Endpoint
Elixir.Matchx.Endpoint.Config
Elixir.Matchx.PubSub.Supervisor
<0.2775.0>
<0.2776.0>
Elixir.Matchx.PubSub.GC0
Elixir.Matchx.PubSub.Local0
Elixir.Matchx.PubSub
Elixir.Matchx.Matchbot.Client
Elixir.Matchx.Matchbot
Elixir.Matchx.Matchbot.Client.Protocol
<0.2769.0>
<0.2770.0>
Elixir.Matchx.Supervisor
worker (sshd)
Elixir.Matchx.Repo
Elixir.Matchx.Repo.Pool.Supervisor
Elixir.Matchx.Repo.Pool
<0.2785.0>
<0.2786.0>
<0.2787.0>
<0.2788.0>
<0.2789.0>
<0.2790.0>
<0.2791.0>
<0.2792.0>
<0.2793.0>
<0.2794.0>
<0.2795.0>
supervisor (init)

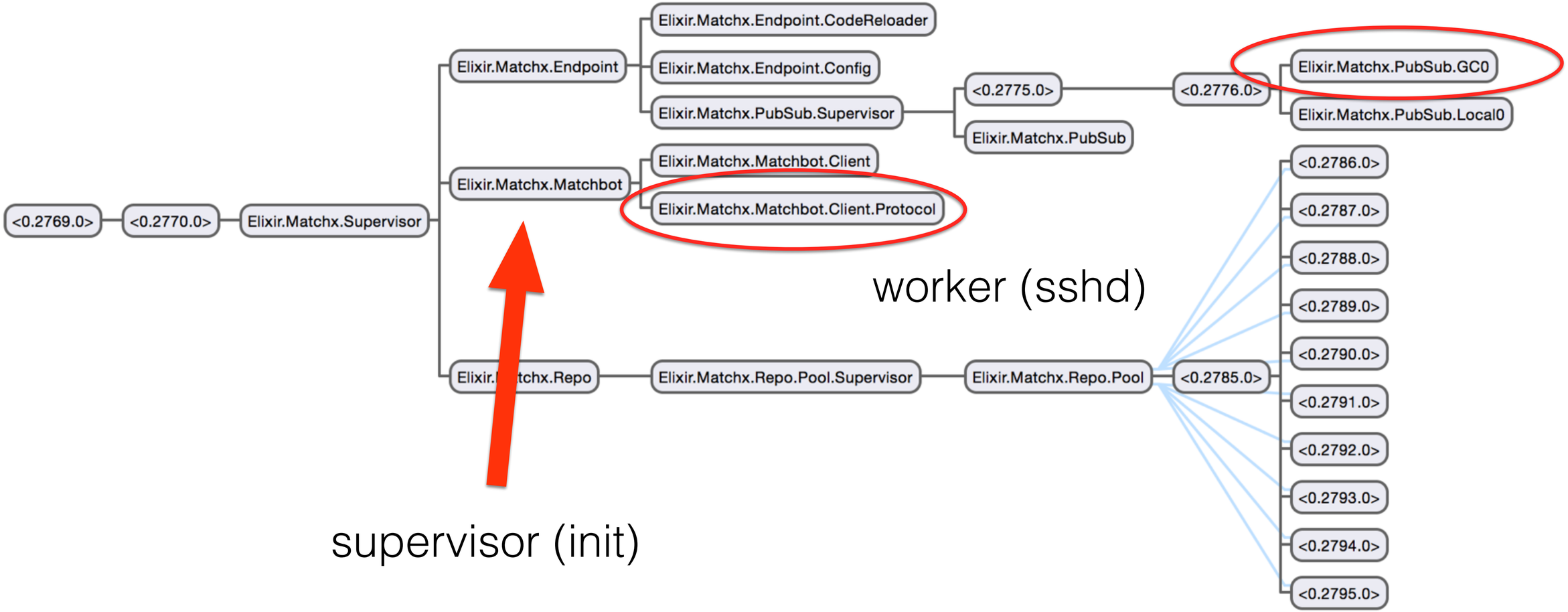
"let it crash"


<0.2769.0>
<0.2770.0>
Elixir.Matchx.Supervisor
Elixir.Matchx.Endpoint
Elixir.Matchx.Endpoint.CodeReloader
Elixir.Matchx.Endpoint.Config
Elixir.Matchx.PubSub.Supervisor
Elixir.Matchx.Matchbot
Elixir.Matchx.Matchbot.Client
Elixir.Matchx.Matchbot.Client.Protocol
Elixir.Matchx.Repo
Elixir.Matchx.Repo.Pool.Supervisor
Elixir.Matchx.Repo.Pool
<0.2776.0>
Elixir.Matchx.PubSub.GC0
Elixir.Matchx.PubSub.Local0
<0.2785.0>
<0.2786.0>
<0.2787.0>
<0.2788.0>
<0.2789.0>
<0.2790.0>
<0.2791.0>
<0.2792.0>
<0.2793.0>
<0.2794.0>
<0.2795.0>
worker (sshd)
supervisor (init)

"let it crash"

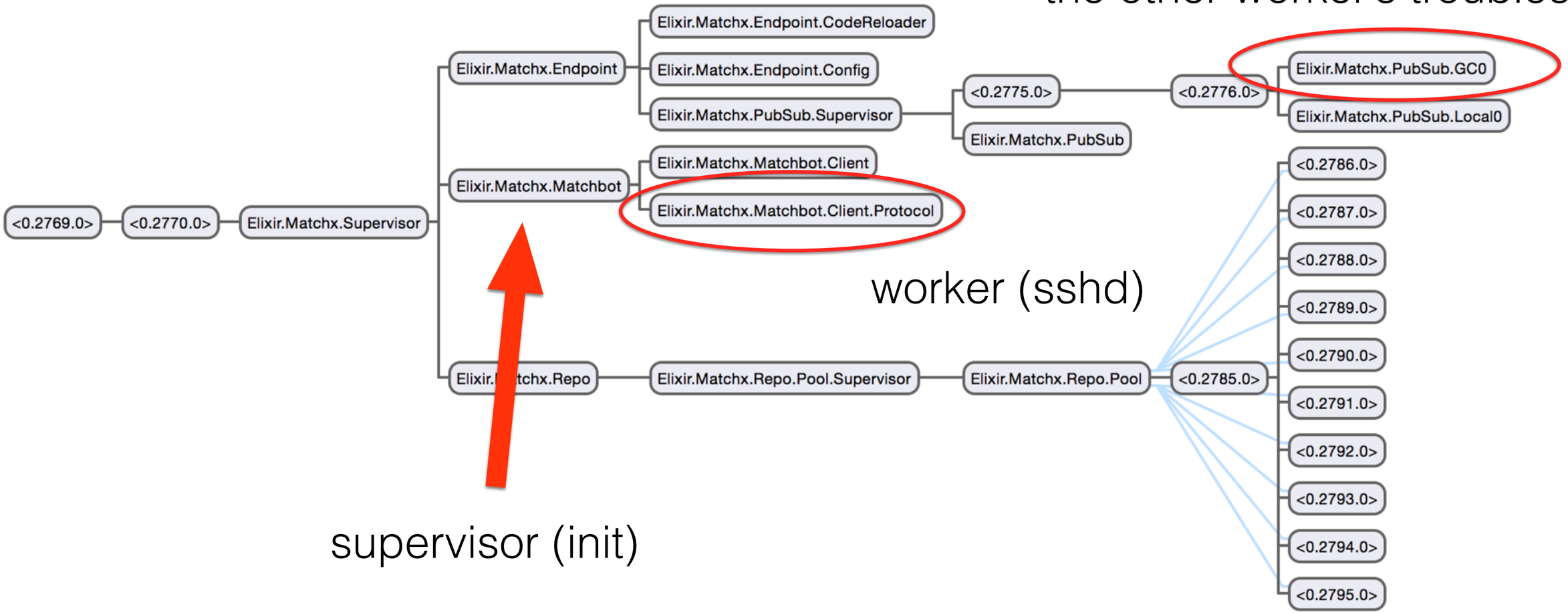

<0.2769.0>
<0.2770.0>
Elixir.Matchx.Supervisor
Elixir.Matchx.Endpoint
Elixir.Matchx.Endpoint.CodeReloader
Elixir.Matchx.Endpoint.Config
Elixir.Matchx.PubSub.Supervisor
<0.2775.0>
Elixir.Matchx.PubSub
<0.2776.0>
Elixir.Matchx.PubSub.GC0
Elixir.Matchx.PubSub.Local0
Elixir.Matchx.Matchbot
Elixir.Matchx.Matchbot.Client
Elixir.Matchx.Matchbot.Client.Protocol
worker (sshd)
Elixir.Matchx.Repo
Elixir.Matchx.Repo.Pool.Supervisor
Elixir.Matchx.Repo.Pool
<0.2785.0>
<0.2786.0>
<0.2787.0>
<0.2788.0>
<0.2789.0>
<0.2790.0>
<0.2791.0>
<0.2792.0>
<0.2793.0>
<0.2794.0>
<0.2795.0>
supervisor (init)

"let it crash"


<0.2769.0>
<0.2770.0>
Elixir.Matchx.Supervisor
Elixir.Matchx.Endpoint
Elixir.Matchx.Endpoint.CodeReloader
Elixir.Matchx.Endpoint.Config
Elixir.Matchx.PubSub.Supervisor
<0.2775.0>
Elixir.Matchx.PubSub
<0.2776.0>
Elixir.Matchx.PubSub.GC0
Elixir.Matchx.PubSub.Local0
Elixir.Matchx.Matchbot
Elixir.Matchx.Matchbot.Client
Elixir.Matchx.Matchbot.Client.Protocol
Elixir.Matchx.Repo
Elixir.Matchx.Repo.Pool.Supervisor
Elixir.Matchx.Repo.Pool
<0.2785.0>
<0.2786.0>
<0.2787.0>
<0.2788.0>
<0.2789.0>
<0.2790.0>
<0.2791.0>
<0.2792.0>
<0.2793.0>
<0.2794.0>
<0.2795.0>
worker (sshd)
supervisor (init)

"let it crash"

… completely ignorant of the other worker's troubles



Overview

unified theory of (un)reliability - ✓

juggling raptors - ✓

mecha-concurrency - ✓

the power of power cycling - ✓

The Zen of Erlang

Perl 6

Overview

unified theory of (un)reliability - ✓

juggling raptors - ✓

mecha-concurrency - ✓

the power of power cycling - ✓

The Zen of Erlang - ✓

Perl 6

Perl 6 & Concurrency

Perl 6 & Concurrency

It has threads! Real ones!

Perl 6 & Concurrency

It has threads! Real ones!

But threads are hard to build on, so the primitives are higher-level

Perl 6 & Concurrency

Perl 6 & Concurrency

Supplies (asynchronous events)

Perl 6 & Concurrency

Supplies (asynchronous events)

Channels (thread-safe blocking queue)

Perl 6 & Concurrency

Supplies (asynchronous events)

Channels (thread-safe blocking queue)

Promises (asynchronous computation)

Perl 6 Promises

Perl 6 Promises

Encapsulates some chunk of code

Perl 6 Promises

Encapsulates some chunk of code

Failures are isolated (part of the Promise's result)

Perl 6 Promises

Encapsulates some chunk of code

Failures are isolated (part of the Promise's result)

I can do blocking operations inside them?

Perl 6 Promises

example: concurrent sleep

Perl 6 Promises

example: concurrent sleep (but more)

Perl 6 Promises: a thread pool

Perl 6 Promises: a thread pool

promises are cooperatively scheduled on 'return' from the function

Perl 6 Promises: a thread pool

promises are cooperatively scheduled on 'return' from the function

temporal decoupling ≈ size of your thread pool: you can block 'some'

Perl 6 Promises: a thread pool

promises are cooperatively scheduled on 'return' from the function

temporal decoupling ≈ size of your thread pool: you can block 'some'

similar caveats as threads: shared memory access, no way to know how to 'restart' (unrestricted access to shared state)

Perl 6 && Erlang

Perl 6 && Erlang

few of the reliability ideas from Erlang/operating systems are applicable

Perl 6 && Erlang

few of the reliability ideas from Erlang/operating systems are applicable

as long as they're closures, this situation is unlikely to change dramatically

Perl 6 && Golang?

Perl 6 && Golang?

Goroutines have almost identical semantics to Perl 6 Promises

Perl 6 && Golang?

Goroutines have almost identical semantics to Perl 6 Promises

(but preemptively scheduled, instead of on a 1:1 thread pool)

Perl 6 && Golang?

Goroutines have almost identical semantics to Perl 6 Promises

(but preemptively scheduled, instead of on a 1:1 thread pool)

more resilient to programmer error (accidental blocking)

Perl 6 && Golang?

Goroutines have almost identical semantics to Perl 6 Promises

(but preemptively scheduled, instead of on a 1:1 thread pool)

more resilient to programmer error (accidental blocking)

... but this is 'cheap threads', not fault isolation

Perl 6 && Golang?

Goroutines have almost identical semantics to Perl 6 Promises

(but preemptively scheduled, instead of on a 1:1 thread pool)

more resilient to programmer error (accidental blocking)

... but this is 'cheap threads', not fault isolation

mega engineering...

Perl 6 && Golang?

Goroutines have almost identical semantics to Perl 6 Promises

(but preemptively scheduled, instead of on a 1:1 thread pool)

more resilient to programmer error (accidental blocking)

... but this is 'cheap threads', not fault isolation

mega engineering...

but consistent semantics

Perl 6 && ???

Perl 6 && ???

this sounds negative

Perl 6 && ???

this sounds negative

not the talk I expected to give!

Perl 6 && ???

this sounds negative

not the talk I expected to give!

Erlang is a much more specialised language than Perl 6

Perl 6 && ???

this sounds negative

not the talk I expected to give!

Erlang is a much more specialised language than Perl 6

not everyone is writing network services

Perl 6 && ???

this sounds negative

not the talk I expected to give!

Erlang is a much more specialised language than Perl 6

not everyone is writing network services

(although lots of us are)

Perl 6 && ???

Perl 6 && ???

Perl 6's concurrency is still *great*

Perl 6 && ???

Perl 6's concurrency is still *great*

a big improvement from Perl 5

Perl 6 && ???

Perl 6's concurrency is still *great*

a big improvement from Perl 5

but oriented differently from Erlang

Thanks

#perl6

Fred Herbert — http://ferd.ca/the-zen-of-erlang.html

booking.com

Questions?