

# Distributed Places

Version 8.13.0.9

Kevin Tew

June 25, 2024

```
(require racket/place/distributed)
package: distributed-places-lib
```

See also §20.3  
“Distributed  
Places” in *The  
Racket Guide*.

Distributed places support programs whose computation may span physical machines. The design relies on machine *nodes* that perform computation. The programmer configures a new distributed system using a declarative syntax and callbacks. A node begins life with one initial place: the *message router*. After a node has been configured, its message router is activated by calling the `message-router` function. The message router listens on a TCP port for incoming connections from other nodes in the distributed system. Places can be spawned within the node by sending place-spawn request messages to the node’s message router.

The distributed places implementation relies on two assumptions:

- The user’s `.ssh/config` and `.ssh/authorized_keys` files are configured correctly to allow passwordless connection to remote hosts via public key authentication.
- Distributed places does not support the specification of ssh usernames. If a non-default ssh username is required the `.ssh/config` file should be used to specify the username.
- All machines run the same version of Racket. Futures versions of distributed places may use the zo binary data format for serialization.

The following example illustrates a configuration and use of distributed places that starts a new node on the current machine and passes it a `"Hello World"` string:

Example:

```
> (module hello-world-example racket/base
  (require racket/place/distributed
```

```

        racket/place)

(provide hello-world)

(define (hello-world)
  (place ch
    (printf "hello-world received: ~a\n" (place-channel-
get ch))
    (place-channel-put ch "Hello World\n")
    (printf "hello-world sent: Hello World\n")))

(module+ main
  ; 1) spawns a node running at "localhost" and listening on
port
  ; 6344 for incoming connections.
  ; 2) connects to the node running at localhost:6344 and cre-
ates a
  ; place on that node by calling the hello-world procedure
from
  ; the current module.
  ; 3) returns a remote-node% instance (node) and a
  ; remote-connection% instance (pl) for communicating with
the
  ; new node and place
  (define-values (node pl)
    (spawn-node-supervise-place-at "localhost"
      #:listen-port 6344
      #:thunk #t
      (quote-module-path "..")
      'hello-world))

  ; starts a message router which adds three event-
container<%>s to
  ; its list of events to handle: the node and two after-
seconds
  ; event containers . Two seconds after the launch of the
message-router, a
  ; message will be sent to the pl place. After six seconds,
the
  ; program and all spawned nodes and places will terminate.
  (message-router
    node
    (after-seconds 2
      (*channel-put pl "Hello")
      (printf "message-router received: ~a\n" (*channel-

```

```

get pl)))

      (after-seconds 6
        (exit 0))))))

```

```

(message-router ec ...+) → void?
ec : (is-a?/c event-container<%>)

```

Waits in an endless loop for one of many events to become ready. The `message-router` procedure constructs a `node%` instance to serve as the message router for the node. The `message-router` procedure then adds all the declared `event-container<%>`s to the `node%` and finally calls the never ending loop `sync-events` method, which handles events for the node.

```

(spawn-node-with-place-at
 hostname
 instance-module-path
 instance-place-function-name
 [#:listen-port port
 #:initial-message initial-message
 #:racket-path racket-path
 #:ssh-bin-path ssh-path
 #:distributed-launch-path launcher-path
 #:restart-on-exit restart-on-exit
 #:named place-name
 #:thunk thunk])
→ (is-a?/c remote-connection%)
hostname : string?
instance-module-path : module-path?
instance-place-function-name : symbol?
port : port-no? = DEFAULT-ROUTER-PORT
initial-message : any = #f
racket-path : string-path? = (racket-path)
ssh-path : string-path? = (ssh-bin-path)
launcher-path : string-path?
              = (path->string distributed-launch-path)
restart-on-exit : any/c = #f
place-name : (or/c #f symbol?) = #f
thunk : (or/c #f #t) = #f

```

Spawns a new remote node at `hostname` with one instance place specified by the `instance-module-path` and `instance-place-function-name`.

When `thunk` is `#f`, the place is created as the result of the framework calling (`dynamic-place instance-module-path instance-place-function-name`). in the new node.

When *thunk* is `#t` the *instance-place-function-name* function should use *dynamic-place* or *place* to create and return an initial place in the new node.

When the *place-name* symbol is present a named place is created. The *place-name* symbol is used to establish later connections to the named place.

The result is a *remote-node%* instance, not a *remote-connection%*. Use *get-first-place* on the result to obtain a *remote-connection%*.

The *restart-on-exit* argument can be `#t` to instruct the *remote-connection%* instance to respawn the place on the remote node should it exit or terminate at any time. It can also be a procedure of zero arguments to implement the restart procedure, or it can be an object that support a *restart* method that takes a place argument.)

```
(spawn-node-supervise-place-at
 hostname
 instance-module-path
 instance-place-function-name
 [#:listen-port port
 #:initial-message initial-message
 #:racket-path racket-path
 #:ssh-bin-path ssh-path
 #:distributed-launch-path launcher-path
 #:restart-on-exit restart-on-exit
 #:named named
 #:thunk thunk])
→ (is-a?/c remote-node%)
   (is-a?/c remote-connection%)
hostname : string?
instance-module-path : module-path?
instance-place-function-name : symbol?
port : port-no? = DEFAULT-ROUTER-PORT
initial-message : any = #f
racket-path : string-path? = (racket-path)
ssh-path : string-path? = (ssh-bin-path)
launcher-path : string-path?
              = (path->string distributed-launch-path)
restart-on-exit : any/c = #f
named : (or/c #f string?) = #f
thunk : (or/c #f #t) = #f
```

Like *spawn-node-with-dynamic-place-at*, but the result is two values: the new *remote-node%* and its *remote-connection%* instance.

```

(spawn-remote-racket-node
 hostname
 [#:listen-port port
 #:racket-path racket-path
 #:ssh-bin-path ssh-path
 #:distributed-launch-path launcher-path]
 #:use-current-ports use-current-ports)
→ (is-a?/c remote-node%)
hostname : string?
port : port-no? = DEFAULT-ROUTER-PORT
racket-path : string-path? = (racket-path)
ssh-path : string-path? = (ssh-bin-path)
launcher-path : string-path?
                = (path->string distributed-launch-path)
use-current-ports : #f

```

Spawns a new remote node at *hostname* and returns a `remote-node%` handle.

```

(create-place-node hostname
 [#:listen-port port
 #:racket-path racket-path
 #:ssh-bin-path ssh-path
 #:distributed-launch-path launcher-path
 #:use-current-ports use-current-ports])
→ (is-a?/c remote-node%)
hostname : string?
port : port-no? = DEFAULT-ROUTER-PORT
racket-path : string-path? = (racket-path)
ssh-path : string-path? = (ssh-bin-path)
launcher-path : string-path?
                = (path->string distributed-launch-path)
use-current-ports : boolean? = #t

```

Like `spawn-remote-racket-node`, but the `current-output-port` and `current-error-port` are used as the standard ports for the spawned process instead of new pipe ports.

```

(supervise-place-at remote-node
 instance-module-path
 instance-place-function-name
 [#:restart-on-exit restart-on-exit
 #:named named
 #:thunk thunk])
→ (is-a?/c remote-connection%)
remote-node : (is-a?/c remote-node%)
instance-module-path : module-path?

```

```

instance-place-function-name : symbol?
restart-on-exit : any/c = #f
named : (or/c #f symbol?) = #f
thunk : (or/c #f #t) = #f

```

When *thunk* is *#f*, creates a new place on *remote-node* by using *dynamic-place* to invoke *instance-place-function-name* from the module *instance-module-path*.

When *thunk* is *#t*, creates a new place at *remote-node* by executing the thunk exported as *instance-place-function-name* from the module *instance-module-path*. The function should use *dynamic-place* or *place* to create and return a place in the new node.

When the *place-name* symbol is present a named place is created. The *place-name* symbol is used to establish later connections to the named place.

```

(supervise-process-at hostname
  cmdline-argument ...+
  [#:listen-port port])
→ (is-a?/c remote-process%)
hostname : string?
cmdline-argument : string?
port : port-no? = DEFAULT-ROUTER-PORT

```

Spawns an attached external process at host *hostname*.

```

(supervise-thread-at remote-node
  instance-module-path
  instance-thunk-function-name
  [#:restart-on-exit restart-on-exit])
→ (is-a?/c remote-connection%)
remote-node : (is-a?/c remote-node%)
instance-module-path : module-path?
instance-thunk-function-name : symbol?
restart-on-exit : any/c = #f

```

Creates a new thread on the *remote-node* by using *dynamic-require* to invoke *instance-place-function-name* from the module *instance-module-path*.

```

(restart-every seconds
  [#:retry retry
   #:on-final-fail on-final-fail])
→ (is-a/c respawn-and-fire%)
seconds : (number?)
retry : (or/c number? #f) = #f
on-final-fail : (or/c #f (-> any/c)) = #f

```

Returns a `restarter%` instance that should be supplied to a `#:restart-on-exit` argument.

```
(every-seconds seconds-expr body ....)
```

Returns a `respawn-and-fire%` instance that should be supplied to a `message-router`. The `respawn-and-fire%` instance executes `bodys` once every  $N$  seconds, where  $N$  is the result of `seconds-expr`.

```
(after-seconds seconds-expr body ....)
```

Returns a `after-seconds%` instance that should be supplied to a `message-router`. The `after-seconds%` instance executes the `bodys` after a delay of  $N$  seconds from the start of the event loop, where  $N$  is the result of `seconds-expr`.

```
(connect-to-named-place node name)  
→ (is-a?/c remote-connection%)  
  node : (is-a?/c remote-node%)  
  name : symbol?
```

Connects to a named place on the `node` named `name` and returns a `remote-connection%` object.

```
(log-message severity msg) → void?  
  severity : (or/c 'fatal 'error 'warning 'info 'debug)  
  msg : string?
```

Logs a message at the root node.

```
event-container<%> : interface?
```

All objects that are supplied to the `message-router` must implement the `event-container<%>` interface. The `message-router` calls the `register` method on each supplied `event-container<%>` to obtain a list of events on which the event loop should wait.

```
(send an-event-container register events) → (listof events?)  
  events : (listof events?)
```

Returns the list of events inside the `event-container<%>` that should be waited on by the `message-router`.

The following classes all implement `event-container<%>` and can be supplied to a `message-router`: `spawned-process%`, `place-socket-bridge%`, `node%`, `remote-node%`, `remote-connection%`, `place% connection%`, `respawn-and-fire%`, and `after-seconds%`.

```
spawned-process% : class?
  superclass: object%
  extends: event-container<%>
```

```
(send a-spawned-process get-pid) → exact-positive-integer?
```

```
(new spawned-process%
  [cmdline-list cmdline-list]
  [[parent parent]])
→ (is-a?/c spawned-process%)
cmdline-list : (listof (or/c string? path?))
parent : (is-a?/c remote-node%) = #f
```

The *cmdline-list* is a list of command line arguments of type `string` and/or `path`.

The *parent* argument is a `remote-node%` instance that will be notified when the process dies via a `(send parent process-died this)` call.

```
place-socket-bridge% : class?
  superclass: object%
  extends: event-container<%>
```

```
(send a-place-socket-bridge get-sc-id)
→ exact-positive-integer?
```

```
(new place-socket-bridge%
  [pch pch]
  [sch sch]
  [id id])
→ (is-a?/c place-socket-bridge%)
pch : place-channel?
sch : (is-a?/c socket-connection%)
id : exact-positive-integer?
```

The *pch* argument is a `place-channel`. Messages received on *pch* are forwarded to the `socket-connection%` *sch* via a `dcmg` message. e.g. `(sconn-write-flush sch (dcmg DCGM-TYPE-INTER-DCHANNEL id id msg))` The *id* is a `exact-positive-integer` that identifies the socket-connection subchannel for this inter-node place connection.



```
socket-connection% : class?
  superclass: object%
  extends: event-container<%>
```

```
(new socket-connection%
  [[host host]
   [port port]
   [retry-times retry-times]
   [delay delay]
   [background-connect? background-connect?]
   [in in]
   [out out]
   [remote-node remote-node]])
→ (is-a?/c socket-connection%)
  host : (or/c string? #f) = #f
  port : (or/c port-no? #f) = #f
  retry-times : exact-nonnegative-integer? = 30
  delay : number? = 1
  background-connect? : any/c = #f
  in : (or/c input-port? #f) = #f
  out : (or/c output-port #f) = #f
  remote-node : (or/c (is-a?/c remote-node%) #f) = #f
```

When a *host* and *port* are supplied a new tcp connection is established. If a *input-port?* and *output-port?* are supplied as *in* and *out*, the ports are used as a connection to the remote host. The *retry-times* argument specifies how many times to retry the connection attempt should it fail to connect and defaults to 30 retry attempts. Often a remote node is still booting up when a connection is attempted and the connection needs to be retried several times. The *delay* argument specifies how many seconds to wait between retry attempts. The *background-connect?* argument defaults to *#t* and specifies that the constructor should retry immediately and that connection establishment should occur in the background. Finally, the *remote-node* argument specifies the *remote-node%* instance that should be notified should the connection fail.

```
node% : class?
  superclass: object%
  extends: event-container<%>
```

The *node%* instance controls a distributed places node. It launches places and routes inter-node place messages in the distributed system. The *message-router* form constructs a *node%* instance under the hood. Newly spawned nodes also have a *node%* instance in their initial place that serves as the node's message router.

```
(new node% [[listen-port listen-port]]) → (is-a?/c node%)
listen-port : listen-port-number? = #f
```

Constructs a `node%` that will listen on `listen-port` for inter-node connections.

```
(send a-node sync-events) → void?
```

Starts the never ending event loop for this distributed places node.

```
remote-node% : class?
superclass: object%
extends: event-container<?>
```

Like `node%`, but for the remote API to a distributed places node. Instances of `remote-node%` are returned by `create-place-node`, `spawn-remote-racket-node`, and `spawn-node-supervise-place-at`.

A `remote-node%` is a place location in the sense of `place-location?`, which means that it can be supplied as the `#:at` argument to `dynamic-place`.

```
(new remote-node%
  [[listen-port listen-port]
  [restart-on-exit restart-on-exit]])
→ (is-a?/c remote-node%)
listen-port : listen-port-number? = #f
restart-on-exit : any/c = #f
```

Constructs a `node%` that will listen on `listen-port` for inter-node connections.

When set to true the `restart-on-exit` parameter causes the specified node to be restarted when the ssh session spawning the node dies.

```
(send a-remote-node get-first-place)
→ (is-a?/c remote-connection%)
```

Returns the `remote-connection%` object instance for the first place spawned on this node.

```
(send a-remote-node get-first-place-channel) → place-
channel?
```

Returns the communication channel for the first place spawned on this node.

```
(send a-remote-node get-log-prefix) → string?
```

Returns (format "PLACE ~a:~a" host-name listen-port)

```
(send a-remote-node launch-place
  place-exec
  [#:restart-on-exit restart-on-exit
   #:one-sided-place? one-sided-place?])
→ (is-a?/c remote-connection%)
place-exec : list?
restart-on-exit : any/c = #f
one-sided-place? : any/c = #f
```

Launches a place on the remote node represented by this `remote-node%` instance.

The `place-exec` argument describes how the remote place should be launched, and it should have one of the following shapes:

- (list 'place place-module-path place-thunk)
- (list 'dynamic-place place-module-path place-func)

The difference between these two launching methods is that the 'place version of `place-exec` expects a thunk to be exported by the module `place-module-path`. Executing the thunk is expected to create a new place and return a place descriptor to the newly created place. The 'dynamic-place version of `place-exec` expects `place-func` to be a function taking a single argument, the initial channel argument, and calls `dynamic-place` on behalf of the user and creates the new place from the `place-module-path` and `place-func`.

The `restart-on-exit` argument is treated in the same way as for `spawn-node-with-dynamic-place-at`.

The `one-sided-place?` argument is an internal use argument for launching remote places from within a place using the old design pattern.

```
(send a-remote-node remote-connect name) → remote-
connection%
name : string?
```

Connects to a named place on the remote node represented by this `remote-node%` instance.

```
(send a-remote-node send-exit) → void?
```

Sends a message instructing the remote node represented by this `remote-node%` instance to exit immediately

```
(node-send-exit remote-node%) → void?
remote-node% : node
```

Sends `node` a message telling it to exit immediately.

```
(node-get-first-place remote-node%)  
→ (is-a?/c remote-connection%)  
remote-node% : node
```

Returns the `remote-connection%` instance of the first place spawned at this node

```
(distributed-place-wait remote-connection%) → void?  
remote-connection% : place
```

Waits for place to terminate.

```
remote-connection% : class?  
superclass: object%  
extends: event-container<%>
```

The `remote-connection%` instance provides a remote api to a place running on a remote distributed places node. It launches a places or connects to a named place and routes inter-node place messages to the remote place.

```
(new remote-connection%  
 [node node]  
 [place-exec place-exec]  
 [name name]  
 [restart-on-exit restart-on-exit]  
 [one-sided-place? one-sided-place?]  
 [on-channel on-channel])  
→ (is-a?/c remote-connection%)  
node : (is-a?/c remote-node%)  
place-exec : list?  
name : string?  
restart-on-exit : #f  
one-sided-place? : #f  
on-channel : #f
```

Constructs a `remote-connection%` instance.

The `place-exec` argument describes how the remote place should be launched in the same way as for `launch-place` in `remote-node%`.

The `restart-on-exit` argument is treated in the same way as for `spawn-node-with-dynamic-place-at`.

The `one-sided-place?` argument is an internal use argument for launching remote places from within a place using the old design pattern.

See `set-on-channel!` for description of `on-channel` argument.

```
(send a-remote-connection set-on-channel! callback) → void?  
callback : (-> channel msg void?)
```

Installs a handler function that handles messages from the remote place. The `setup/distributed-docs` module uses this callback to handle job completion messages.

```
place% : class?  
superclass: object%  
extends: event-container<%>
```

The `place%` instance represents a place launched on a distributed places node at that node. It launches a compute places and routes inter-node place messages to the place.

```
(new place%  
  [node node]  
  [place-exec place-exec]  
  [ch-id ch-id]  
  [sc sc]  
  [[on-place-dead on-place-dead]]) → (is-a?/c place%)  
node : (is-a?/c remote-connection%)  
place-exec : list?  
ch-id : exact-positive-integer?  
sc : (is-a?/c socket-connection%)  
on-place-dead : (-> event void?) = default-on-place-dead
```

Constructs a `remote-connection%` instance. The `place-exec` argument describes how the remote place should be launched in the same way as for `launch-place` in `remote-node%`. The `ch-id` and `sc` arguments are internally used to establish routing between the remote node spawning this place and the place itself. The `on-place-dead` callback handles the event when the newly spawned place terminates.

```
(send a-place wait-for-die) → void?
```

Blocks and waits for the subprocess representing the `remote-node%` to exit.

```
connection% : class?  
superclass: object%  
extends: event-container<%>
```

The `connection%` instance represents a connection to a named-place instance running on the current node. It routes inter-node place messages to the named place.

```
(new connection%
  [node node]
  [name name]
  [ch-id ch-id]
  [sc sc]) → (is-a?/c connection%)
node : (is-a?/c remote-node%)
name : string?
ch-id : exact-positive-integer?
sc : (is-a?/c socket-connection%)
```

Constructs a `remote-connection%` instance. The `place-exec` argument describes how the remote place should be launched in the same way as for `launch-place` in `remote-node%`. The `ch-id` and `sc` arguments are internally used to establish routing between the remote node and this named-place.

```
respawn-and-fire% : class?
superclass: object%
extends: event-container<%>
```

The `respawn-and-fire%` instance represents a thunk that should execute every `n` seconds.

```
(new respawn-and-fire%
  [seconds seconds]
  [thunk thunk])
→ (is-a?/c respawn-and-fire%)
seconds : (and/c real? (not/c negative?))
thunk : (-> void?)
```

Constructs a `respawn-and-fire%` instance that when placed inside a `message-router` construct causes the supplied thunk to execute every `n` seconds.

```
after-seconds% : class?
superclass: object%
extends: event-container<%>
```

The `after-seconds%` instance represents a thunk that should execute after `n` seconds.

```
(new after-seconds%
  [seconds seconds]
  [thunk thunk])
→ (is-a?/c after-seconds%)
```

```
seconds : (and/c real? (not/c negative?))
thunk : (-> void?)
```

Constructs an `after-seconds%` instance that when placed inside a `message-router` construct causes the supplied `thunk` to execute after `n` seconds.

```
restarter% : class?
superclass: after-seconds%
extends: event-container<%>
```

The `restarter%` instance represents a restart strategy.

```
(new restarter%
  [seconds seconds]
  [[retry retry]
   [on-final-fail on-final-fail]])
→ (is-a?/c restarter%)
seconds : number?
retry : (or/c number? #f) = #f
on-final-fail : (or/c #f (-> any/c)) = #f
```

Constructs an `restarter%` instance that when supplied to a `#:restart-on-exit` argument, attempts to restart the process every `seconds`. The `retry` argument specifies how many time to attempt to restart the process before giving up. If the process stays alive for `(* 2 seconds)` the attempted retries count is reset to 0. The `on-final-fail` thunk is called when the number of retries is exceeded

```
distributed-launch-path : path?
```

Contains the local path to the distributed places launcher. The distributed places launcher is the bootstrap file that launches the message router on a new node.

```
(ssh-bin-path) → string?
```

Returns the path to the ssh binary on the local system in string form.

Example:

```
> (ssh-bin-path)
#<path:/usr/bin/ssh>
```

```
(racket-path) → path?
```

Returns the path to the currently executing Racket binary on the local system.

```
(build-distributed-launch-path collects-path) → string?  
  collects-path : path-string?
```

Returns the path to the distributed places launch file. The function can take an optional argument specifying the path to the collects directory.

```
(spawn-node-at hostname  
  [#:listen-port port  
   #:racket-path racket-path  
   #:ssh-bin-path ssh-path  
   #:distributed-launch-path launcher-path])  
→ channel?  
hostname : string?  
port : port-no? = DEFAULT-ROUTER-PORT  
racket-path : string-path? = (racket-path)  
ssh-path : string-path? = (ssh-bin-path)  
launcher-path : string-path?  
              = (path->string distributed-launch-path)
```

Spawns a node in the background using a Racket thread and returns a channel that becomes ready with a `remote-node%` once the node has spawned successfully

```
(spawn-nodes/join nodes-descs) → void?  
  nodes-descs : list?
```

Spawns a list of nodes by calling `(lambda (x) (apply keyword-apply spawn-node-at x))` for each node description in `nodes-descs` and then waits for each node to spawn.

```
(*channel-put ch msg) → void?  
  ch : (or/c place-channel? async-bi-channel?  
         channel? (is-a?/c remote-connection%))  
  msg : any
```

Sends `msg` over `ch` channel.

```
(*channel-get ch) → any  
  ch : (or/c place-channel? async-bi-channel?  
         channel? (is-a?/c remote-connection%))
```

Returns a message received on `ch` channel.

```
(*channel? v) → boolean?  
  v : any/c
```



Returns `#t` if `v` is one of `place-channel?`, `async-bi-channel?`, `channel?`, or `(isa?/c remote-connection%)`.

```
(send-new-place-channel-to-named-dest ch
                                     src-id
                                     dest-list)
→ place-channel?
ch : *channel?
src-id : any
dest-list : (listof string? port-no? string?)
```

Creates and returns a new place channel connection to a named place at `dest-list`. The `dest-list` argument is a list of a remote-hostname remote-port and named-place name. The channel `ch` should be a connection to a `message-router`.

```
(mr-spawn-remote-node mrch
                    host
                    [#:listen-port listen-port
                    #:solo solo]) → void?
mrch : *channel?
host : string?
listen-port : port-no? = DEFAULT-ROUTER-PORT
solo : boolean? = #f
```

Sends a message to a message router over `mrch` channel asking the message router to spawn a new node at `host` listening on port `listen-port`. If the `#:solo` keyword argument is supplied the new node is not folded into the complete network with other nodes in the distributed system.

```
(mr-supervise-named-dynamic-place-at mrch
                                     dest
                                     name
                                     path
                                     func) → void?
mrch : *channel?
dest : (listof string? port-no?)
name : string?
path : string?
func : symbol?
```

Sends a message to a message router over `mrch` channel asking the message router to spawn a named place at `dest` named `name`. The place is spawned at the remote node by calling dynamic place with module-path `path` and function `func`. The `dest` parameter should be a list of remote-hostname and remote-port.

```
(mr-connect-to mrch dest name) → void?
  mrch : *channel?
  dest : (listof string? port-no?)
  name : string?
```

Sends a message to a message router over *mrch* channel asking the message router to create a new connection to the named place named *name* at *dest*. The *dest* parameter should be a list of remote-hostname and remote-port.

```
(start-message-router/thread [#:listen-port listen-port
                             #:nodes nodes])
→ thread? channel?
  listen-port : port-no? = DEFAULT-ROUTER-PORT
  nodes : list? = null
```

Starts a message router in a Racket thread connected to *nodes*, listening on port *listen-port*, and returns a *channel?* connection to the message router.

```
(port-no? no) → boolean?
  no : (and/c exact-nonnegative-integer? (integer-in 0 65535))
```

Returns *#t* if *no* is a *exact-nonnegative-integer?* between 0 and 65535.

```
DEFAULT-ROUTER-PORT : port-no?
```

The default port for distributed places message router.

```
named-place-typed-channel% : class?
  superclass: object%
```

```
(new named-place-typed-channel% [ch ch])
→ (is-a?/c named-place-typed-channel%)
  ch : place-channel?
```

The *ch* argument is a *place-channel*.

```
(send a-named-place-typed-channel get type) → any
  type : symbol?
```

Returns the first message received on *ch* that has the type *type*. Messages are lists and their type is the first item of the list which should be a *symbol?*. Messages of other types that are received are queued for later *get* requests.

```
(tc-get type ch) → void?  
  type : symbol?  
  ch : place-channel?
```

Gets a message of type *type* from the `named-place-typed-channel%` *ch*.

```
(write-flush datum port) → void?  
  datum : any  
  port : port?
```

Writes *datum* to *port* and then flushes *port*.

```
(printf/f format args ...) → void?  
  format : string?  
  args : any
```

Calls `printf` followed by a call to `flush-output`.

```
(displayln/f item) → void?  
  item : any
```

Calls `displayln` followed by a call to `flush-output`.

Example:

```
> (write-flush "Hello World" (current-output-port))  
"Hello World"
```

# 1 Define Remote Server

```
(require racket/place/define-remote-server)
package: distributed-places-lib
```

```
(define-remote-server [name identifier?] rpc-forms ...+)
(define-named-remote-server [name identifier?] rpc-forms ...+)
```

The `define-remote-server` and `define-named-remote-server` forms are nearly identical. The `define-remote-server` form should be used with `supervise-dynamic-place-at` to build a private rpc server, while the `define-named-remote-server` form should be used with `supervise-named-dynamic-place-at` to build a rpc server inside a named place.

The `define-named-remote-server` form takes an identifier and a list of custom expressions as its arguments. From the identifier a function is created by prepending the `make-` prefix. This procedure takes a single argument a `place-channel`. In the example below, the `make-tuple-server` identifier is the `place-function-name` given to the `supervise-named-dynamic-place-at` form to spawn an rpc server. The server created by the `make-tuple-server` procedure sits in a loop waiting for rpc requests from the `define-rpc` functions documented below.

```
(define-state id value)
```

Expands to a `define`, which is closed over by the `define-rpc` functions to form local state.

```
(define-rpc (id args ...) body ...)
```

Expands to a client rpc function `name-id` which sends `id` and `args` ... to the rpc server `rpc-place` and waits for a response. `(define (name-id rpc-place args ...) body)`

```
(define-cast (id args ...) body ...)
```

Expands to a client rpc function `name-id` which sends `id` and `args` ... to the rpc server `rpc-place` but does not receive any response. A cast is a one-way communication technique. `(define (name-id rpc-place args ...) body)`

The `define-state` custom form translates into a simple `define` form, which is closed over by the `define-rpc` forms.

The `define-rpc` form is expanded into two parts. The first part is the client stubs that call the rpc functions. The client function name is formed by concatenating the `define-named-remote-server` identifier, `tuple-server`, with the RPC function name set to form `tuple-server-set`. The RPC client functions take a destination argument which

is a `remote-connection%` descriptor and then the RPC function arguments. The RPC client function sends the RPC function name, `set`, and the RPC arguments to the destination by calling an internal function `named-place-channel-put`. The RPC client then calls `named-place-channel-get` to wait for the RPC response.

The second expansion part of `define-rpc` is the server implementation of the RPC call. The server is implemented by a match expression inside the `make-tuple-server` function. The match clause for `tuple-server-set` matches on messages beginning with the `'set` symbol. The server executes the RPC call with the communicated arguments and sends the result back to the RPC client.

The `define-cast` form is similar to the `define-rpc` form except there is no reply message from the server to client

Example:

```
> (module tuple-server-example racket/base
  (require racket/match
           racket/place/define-remote-server)

  (define-named-remote-server tuple-server
    (define-state h (make-hash))
    (define-rpc (set k v)
      (hash-set! h k v)
      v)
    (define-rpc (get k)
      (hash-ref h k #f))
    (define-cast (hello)
      (printf "Hello from define-cast\n")
      (flush-output))))
```

Example:

```
> (module bank-server-example racket/base
  (require racket/match
           racket/place/define-remote-server)

  (define-remote-server bank
    (define-state accounts (make-hash))
    (define-rpc (new-account who)
      (match (hash-has-key? accounts who)
        [#t '(already-exists)]
        [else
         (hash-set! accounts who 0)
         (list 'created who)]))
    (define-rpc (remove who amount)
```

```

(cond
  [(hash-ref accounts who (lambda () #f)) =>
   (lambda (balance)
     (cond [(<= amount balance)
            (define new-balance (- balance amount))
            (hash-set! accounts who new-balance)
            (list 'ok new-balance)]
          [else
           (list 'insufficient-funds balance)]))]
  [else
   (list 'invalid-account who)]))
(define-rpc (add who amount)
  (cond
    [(hash-ref accounts who (lambda () #f)) =>
     (lambda (balance)
       (define new-balance (+ balance amount))
       (hash-set! accounts who new-balance)
       (list 'ok new-balance))]
    [else
     (list 'invalid-account who)])))

```

```

(log-to-parent msg [#:severity severity]) → void?
  msg : string?
  severity : symbol? = 'info

```

The `log-to-parent` procedure can be used inside a `define-remote-server` or `define-named-remote-server` form to send a logging message to the remote owner of the rpc server.

## 2 Async Bidirectional Channels

```
(require racket/place/private/async-bi-channel)
package: base
```

```
(make-async-bi-channel) → async-bi-channel?
```

Creates and returns an opaque structure, which is the async bidirectional channel.

```
(async-bi-channel? ch) → boolean?
  ch : any
```

A predicate that returns `#t` for async bidirectional channels.

```
(async-bi-channel-get ch) → any
  ch : async-bi-channel?
```

Returns the next available message from the async bidirectional channel `ch`.

```
(async-bi-channel-put ch msg) → void?
  ch : async-bi-channel?
  msg : any
```

Sends message `msg` to the remote end of the async bidirectional channel `ch`.

### 3 Distributed Places MPI

```
(require racket/place/distributed/rmpi)
package: distributed-places-lib
```

```
(struct rmpi-comm (id cnt channels))
  id : exact-nonnegative-integer?
  cnt : exact-positive-integer?
  channels : vector?
```

The communicator struct `rmpi-comm` contains the rmpi process rank `id`, the quantity of processes in the communicator group, `cnt`, and a vector of place-channels, one for each process in the group.

```
(rmpi-id comm) → exact-nonnegative-integer?
  comm : rmpi-comm?
```

Takes a rmpi communicator structure, `comm`, and returns the node id of the RMPI process.

```
(rmpi-cnt comm) → exact-positive-integer?
  comm : rmpi-comm?
```

Takes a rmpi communicator structure, `comm`, and returns the count of the RMPI processes in the communicator group.

```
(rmpi-send comm dest val) → void?
  comm : rmpi-comm?
  dest : exact-nonnegative-integer?
  val : any
```

Sends `val` to destination rmpi process number `dest` using the RMPI communicator structure `comm`.

```
(rmpi-recv comm src) → any
  comm : rmpi-comm?
  src : exact-nonnegative-integer?
```

Receives a message from source rmpi process number `src` using the RMPI communicator structure `comm`.

```
(rmpi-init ch) → rmpi-comm?
  (listof any)
  (is-a?/c named-place-typed-channel%)
  ch : place-channel?
```



Creates the `rmpi-comm` structure instance using the named place's original place-channel `ch`. In addition to the communicator structure, `rmpi-init` returns a list of initial arguments and the original place-channel `ch` wrapped in a `named-place-typed-channel%`. The `named-place-typed-channel%` wrapper allows for the reception of list messages typed by an initial symbol.

```
(rmpi-broadcast comm src) → any
  comm : rmpi-comm?
  src : exact-nonnegative-integer?
(rmpi-broadcast comm src val) → any
  comm : rmpi-comm?
  src : exact-nonnegative-integer?
  val : any
```

Broadcasts `val` from `src` to all `rmpi` processes in the communication group using a hypercube algorithm. Receiving processes call `(rmpi-broadcast comm src)`.

```
(rmpi-reduce comm dest op val) → any
  comm : rmpi-comm?
  dest : exact-nonnegative-integer?
  op : procedure?
  val : any
```

Reduces `val` using the `op` operator to `dest` `rmpi` node using a hypercube algorithm.

```
(rmpi-barrier comm) → void?
  comm : rmpi-comm?
```

Introduces a synchronization barrier for all `rmpi` processes in the communication group `comm`.

```
(rmpi-allreduce comm op val) → any
  comm : rmpi-comm?
  op : procedure?
  val : any
```

Reduces `val` using the `op` operator to `rmpi` node 0 and then broadcasts the reduced value to all nodes in the communication group.

```
(rmpi-partition comm num) → exact-nonnegative-integer?
                                exact-nonnegative-integer?
  comm : rmpi-comm?
  num : exact-nonnegative-integer?
```

Partitions `num` into `rmpi-cnt` equal pieces and returns the offset and length for the `rmpi-idth` piece.

```

(rmpi-build-default-config
 #:racket-path racket-path
 #:distributed-launch-path distributed-launch-path
 #:mpi-module mpi-module
 #:mpi-func mpi-func
 #:mpi-args mpi-args)
→ hash?
racket-path : string?
distributed-launch-path : string?
mpi-module : string?
mpi-func : symbol?
mpi-args : (listof any)

```

Builds a hash from keywords to keyword arguments for use with the `rmpi-launch` function.

```

(rmpi-launch default-node-config config) → void?
default-node-config : hash?
config : (listof (list/c string? port-no? symbol?
                    exact-nonnegative-integer?))

```

Launches distributed places nodes running `#:mpi-func` in `#:mpi-module` with `#:mpi-args`. The config is a list of node configs, where each node config consists of a host-name, port, named place symbol and rmpi id number, followed by an optional hash of keyword `#:racket-path`, `#:distributed-launch-path`, `#:mpi-module`, `#:mpi-func`, and `#:mpi-args` to keyword arguments. Missing optional keyword arguments will be taken from the `default-node-config` hash of keyword arguments.

```

(rmpi-finish comm tc) → void?
comm : rmpi-comm?
tc : (is-a?/c named-place-typed-channel%)

```

Rendezvous with the `rmpi-launch`, using the `tc` returned by `rmpi-launch`, to indicate that the RMPI module is done executing and that `rmpi-launch` can return control to its caller.

Example:

```

> (rmpi-launch
   (rmpi-build-default-config
    #:racket-path "/tmp/mplt/bin/racket"
    #:distributed-launch-path (build-distributed-launch-path
                              "/tmp/mplt/collects")
    #:mpi-module "/tmp/mplt/kmeans.rkt"
    #:mpi-func 'kmeans-place

```

```
#:mpi-args (list "/tmp/mpl/color100.bin" #t 100
                9 10 1e-7))

(list (list "nodea.example.com" 6340 'kmeans_0 0)
      (list "nodeb.example.com" 6340 'kmeans_1 1)
      (list "nodec.example.com" 6340 'kmeans_2 2)
      (list "noded.example.com" 6340 'kmeans_3 3))
```