Distributed Computing

`Distributed.addprocs` — Function

```
addprocs(manager::ClusterManager; kwargs...) -> List of process identifiers
```

Launches worker processes via the specified cluster manager.

For example, Beowulf clusters are supported via a custom cluster manager implemented in the package `ClusterManagers.jl`.

The number of seconds a newly launched worker waits for connection establishment from the master can be specified via variable `JULIA_WORKER_TIMEOUT` in the worker process's environment. Relevant only when using TCP/IP as transport.

To launch workers without blocking the REPL, or the containing function if launching workers programmatically, execute `addprocs` in its own task.

Examples

```
# On busy clusters, call `addprocs` asynchronously
t = @async addprocs(...)
```

```
# Utilize workers as and when they come online
if nprocs() > 1   # Ensure at least one new worker is available
   ....   # perform distributed execution
end
```

```
# Retrieve newly launched worker IDs, or any error messages
if istaskdone(t)   # Check if `addprocs` has completed to ensure `fetch` doesn't
    if nworkers() == N
        new_pids = fetch(t)
    else
        fetch(t)
    end
  end
```

```
addprocs(machines; tunnel=false, sshflags=``, max_parallel=10, kwargs...) -> Lis
```

Add processes on remote machines via SSH. Requires `julia` to be installed in the same location on each node, or to be available via a shared file system.

`machines` is a vector of machine specifications. Workers are started for each specification.

A machine specification is either a string `machine_spec` or a tuple - `(machine_spec, count)`.

`machine_spec` is a string of the form `[user@]host[:port] [bind_addr[:port]]`. `user` defaults to current user, `port` to the standard ssh port. If `[bind_addr[:port]]` is specified, other workers will connect to this worker at the specified `bind_addr` and `port`.

`count` is the number of workers to be launched on the specified host. If specified as `:auto` it will launch as many workers as the number of CPU threads on the specific host.

Keyword arguments:

- `tunnel`: if `true` then SSH tunneling will be used to connect to the worker from the master process. Default is `false`.
- `multiplex`: if `true` then SSH multiplexing is used for SSH tunneling. Default is `false`.
- `sshflags`: specifies additional ssh options, e.g. `sshflags=`-i /home/foo/bar.pem`
- `max_parallel`: specifies the maximum number of workers connected to in parallel at a host. Defaults to 10.
- `dir`: specifies the working directory on the workers. Defaults to the host's current directory (as found by `pwd()`)
- `enable_threaded_blas`: if `true` then BLAS will run on multiple threads in added processes. Default is `false`.
- `exename`: name of the `julia` executable. Defaults to `"$(Sys.BINDIR)/julia"` or `"$(Sys.BINDIR)/julia-debug"` as the case may be.
- `exeflags`: additional flags passed to the worker processes.
- `topology`: Specifies how the workers connect to each other. Sending a message between unconnected workers results in an error.
    - `topology=:all_to_all`: All processes are connected to each other. The default.
    - `topology=:master_worker`: Only the driver process, i.e. `pid` 1 connects to the workers. The workers do not connect to each other.
    - `topology=:custom`: The `launch` method of the cluster manager specifies the connection topology via fields `ident` and `connect_idents` in `WorkerConfig`. A worker with a cluster manager identity `ident` will connect to all workers specified in `connect_idents`.
- `lazy`: Applicable only with `topology=:all_to_all`. If `true`, worker-worker connections are setup lazily, i.e. they are setup at the first instance of a remote call between workers. Default is true.

Environment variables :

If the master process fails to establish a connection with a newly launched worker within 60.0

seconds, the worker treats it as a fatal situation and terminates. This timeout can be controlled via environment variable `JULIA_WORKER_TIMEOUT`. The value of `JULIA_WORKER_TIMEOUT` on the master process specifies the number of seconds a newly launched worker waits for connection establishment.

```
addprocs(; kwargs...) -> List of process identifiers
```

Equivalent to `addprocs(Sys.CPU_THREADS; kwargs...)`

Note that workers do not run a `.julia/config/startup.jl` startup script, nor do they synchronize their global state (such as global variables, new method definitions, and loaded modules) with any of the other running processes.

```
addprocs(np::Integer; restrict=true, kwargs...) -> List of process identifiers
```

Launches workers using the in-built `LocalManager` which only launches workers on the local host. This can be used to take advantage of multiple cores. `addprocs(4)` will add 4 processes on the local machine. If `restrict` is `true`, binding is restricted to `127.0.0.1`. Keyword args `dir`, `exename`, `exeflags`, `topology`, `lazy` and `enable_threaded_blas` have the same effect as documented for `addprocs(machines)`.

`Distributed.nprocs` — Function

```
nprocs()
```

Get the number of available processes.

Examples

```
julia> nprocs()
3

julia> workers()
5-element Array{Int64,1}:
 2
 3
```

`Distributed.nworkers` — Function

```
nworkers()
```

Get the number of available worker processes. This is one less than `nprocs()`. Equal to `nprocs()` if `nprocs() == 1`.

Examples

```
$ julia -p 5

julia> nprocs()
6

julia> nworkers()
5
```

`Distributed.procs` — Method

```
procs()
```

Return a list of all process identifiers, including pid 1 (which is not included by `workers()`).

Examples

```
$ julia -p 5

julia> procs()
3-element Array{Int64,1}:
 1
 2
 3
```

`Distributed.procs` — Method

```
procs(pid::Integer)
```

Return a list of all process identifiers on the same physical node. Specifically all workers bound to the same ip-address as `pid` are returned.

`Distributed.workers` — Function

```
workers()
```

Return a list of all worker process identifiers.

Examples

```
$ julia -p 5

julia> workers()
2-element Array{Int64,1}:
 2
 3
```

`Distributed.rmprocs` — Function

```
rmprocs(pids...; waitfor=typemax(Int))
```

Remove the specified workers. Note that only process 1 can add or remove workers.

Argument `waitfor` specifies how long to wait for the workers to shut down:

- If unspecified, `rmprocs` will wait until all requested `pids` are removed.
- An `ErrorException` is raised if all workers cannot be terminated before the requested `waitfor` seconds.
- With a `waitfor` value of 0, the call returns immediately with the workers scheduled for removal in a different task. The scheduled `Task` object is returned. The user should call `wait` on the task before invoking any other parallel calls.

Examples

```
$ julia -p 5

julia> t = rmprocs(2, 3, waitfor=0)
```

```
Task (runnable) @0x0000000107c718d0

julia> wait(t)

julia> workers()
3-element Array{Int64,1}:
 4
 5
 6
```

Distributed.interrupt — Function

```
interrupt(pids::Integer...)
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

```
interrupt(pids::AbstractVector=workers())
```

Interrupt the current executing task on the specified workers. This is equivalent to pressing Ctrl-C on the local machine. If no arguments are given, all workers are interrupted.

Distributed.myid — Function

```
myid()
```

Get the id of the current process.

Examples

```
julia> myid()
1

julia> remotecall_fetch(() -> myid(), 4)
4
```

`Distributed.pmap` — Function

```
pmap(f, [::AbstractWorkerPool], c...; distributed=true, batch_size=1, on_error=
```

Transform collection `c` by applying `f` to each element using available workers and tasks.

For multiple collection arguments, apply `f` elementwise.

Note that `f` must be made available to all worker processes; see Code Availability and Loading Packages for details.

If a worker pool is not specified, all available workers, i.e., the default worker pool is used.

By default, `pmap` distributes the computation over all specified workers. To use only the local process and distribute over tasks, specify `distributed=false`. This is equivalent to using `asyncmap`. For example, `pmap(f, c; distributed=false)` is equivalent to `asyncmap(f,c; ntasks=()->nworkers())`

`pmap` can also use a mix of processes and tasks via the `batch_size` argument. For batch sizes greater than 1, the collection is processed in multiple batches, each of length `batch_size` or less. A batch is sent as a single request to a free worker, where a local `asyncmap` processes elements from the batch using multiple concurrent tasks.

Any error stops `pmap` from processing the remainder of the collection. To override this behavior you can specify an error handling function via argument `on_error` which takes in a single argument, i.e., the exception. The function can stop the processing by rethrowing the error, or, to continue, return any value which is then returned inline with the results to the caller.

Consider the following two examples. The first one returns the exception object inline, the second a 0 in place of any exception:

```
julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=identity)
4-element Array{Any,1}:
 1
  ErrorException("foo")
 3
  ErrorException("foo")

julia> pmap(x->iseven(x) ? error("foo") : x, 1:4; on_error=ex->0)
4-element Array{Int64,1}:
 1
 0
```

```
3
0
```

Errors can also be handled by retrying failed computations. Keyword arguments `retry_delays` and `retry_check` are passed through to `retry` as keyword arguments `delays` and `check` respectively. If batching is specified, and an entire batch fails, all items in the batch are retried.

Note that if both `on_error` and `retry_delays` are specified, the `on_error` hook is called before retrying. If `on_error` does not throw (or rethrow) an exception, the element will not be retried.

Example: On errors, retry `f` on an element a maximum of 3 times without any delay between retries.

```
pmap(f, c; retry_delays = zeros(3))
```

Example: Retry `f` only if the exception is not of type `InexactError`, with exponentially increasing delays up to 3 times. Return a `NaN` in place for all `InexactError` occurrences.

```
pmap(f, c; on_error = e->(isa(e, InexactError) ? NaN : rethrow()), retry_delays
```

`Distributed.RemoteException` — Type

```
RemoteException(captured)
```

Exceptions on remote computations are captured and rethrown locally. A `RemoteException` wraps the `pid` of the worker and a captured exception. A `CapturedException` captures the remote exception and a serializable form of the call stack when the exception was raised.

`Distributed.Future` — Type

```
Future(w::Int, rrid::RRID, v::Union{Some, Nothing}=nothing)
```

A `Future` is a placeholder for a single computation of unknown termination status and time. For multiple potential computations, see `RemoteChannel`. See `remoteref_id` for identifying an `AbstractRemoteRef`.

`Distributed.RemoteChannel` — Type

```
RemoteChannel(pid::Integer=myid())
```

Make a reference to a `Channel{Any}(1)` on process `pid`. The default `pid` is the current process.

```
RemoteChannel(f::Function, pid::Integer=myid())
```

Create references to remote channels of a specific size and type. `f` is a function that when executed on `pid` must return an implementation of an `AbstractChannel`.

For example, `RemoteChannel(()->Channel{Int}(10), pid)`, will return a reference to a channel of type `Int` and size 10 on `pid`.

The default `pid` is the current process.

`Base.fetch` — Method

```
fetch(x::Future)
```

Wait for and get the value of a `Future`. The fetched value is cached locally. Further calls to `fetch` on the same reference return the cached value. If the remote value is an exception, throws a `RemoteException` which captures the remote exception and backtrace.

`Base.fetch` — Method

```
fetch(c::RemoteChannel)
```

Wait for and get a value from a `RemoteChannel`. Exceptions raised are the same as for a `Future`. Does not remove the item fetched.

`Distributed.remotecall` — Method

```
remotecall(f, id::Integer, args...; kwargs...) -> Future
```

Call a function `f` asynchronously on the given arguments on the specified process. Return a `Future`. Keyword arguments, if any, are passed through to `f`.

`Distributed.remotecall_wait` — Method

```
remotecall_wait(f, id::Integer, args...; kwargs...)
```

Perform a faster `wait(remotecall(...))` in one message on the `Worker` specified by worker id `id`. Keyword arguments, if any, are passed through to `f`.

See also `wait` and `remotecall`.

`Distributed.remotecall_fetch` — Method

```
remotecall_fetch(f, id::Integer, args...; kwargs...)
```

Perform `fetch(remotecall(...))` in one message. Keyword arguments, if any, are passed through to `f`. Any remote exceptions are captured in a `RemoteException` and thrown.

See also `fetch` and `remotecall`.

Examples

```
$ julia -p 2

julia> remotecall_fetch(sqrt, 2, 4)
2.0

julia> remotecall_fetch(sqrt, 2, -4)
ERROR: On worker 2:
DomainError with -4.0:
sqrt will only return a complex result if called with a complex argument. Try s
...
```

`Distributed.remote_do` — Method

```
remote_do(f, id::Integer, args...; kwargs...) -> nothing
```

Executes `f` on worker `id` asynchronously. Unlike `remotecall`, it does not store the result of computation, nor is there a way to wait for its completion.

A successful invocation indicates that the request has been accepted for execution on the remote node.

While consecutive `remotecalls` to the same worker are serialized in the order they are invoked, the order of executions on the remote worker is undetermined. For example, `remote_do(f1, 2); remotecall(f2, 2); remote_do(f3, 2)` will serialize the call to `f1`, followed by `f2` and `f3` in that order. However, it is not guaranteed that `f1` is executed before `f3` on worker 2.

Any exceptions thrown by `f` are printed to `stderr` on the remote worker.

Keyword arguments, if any, are passed through to `f`.

`Base.put!` — Method

```
put!(rr::RemoteChannel, args...)
```

Store a set of values to the `RemoteChannel`. If the channel is full, blocks until space is available. Return the first argument.

`Base.put!` — Method

```
put!(rr::Future, v)
```

Store a value to a `Future` `rr`. `Futures` are write-once remote references. A `put!` on an already set `Future` throws an `Exception`. All asynchronous remote calls return `Futures` and set the value to the return value of the call upon completion.

`Base.take!` — Method

```
take!(rr::RemoteChannel, args...)
```

Fetch value(s) from a `RemoteChannel` `rr`, removing the value(s) in the process.

`Base.isready` — Method

```
isready(rr::RemoteChannel, args...)
```

Determine whether a `RemoteChannel` has a value stored to it. Note that this function can cause race conditions, since by the time you receive its result it may no longer be true. However, it can be safely used on a `Future` since they are assigned only once.

`Base.isready` — Method

```
isready(rr::Future)
```

Determine whether a `Future` has a value stored to it.

If the argument `Future` is owned by a different node, this call will block to wait for the answer. It is recommended to wait for `rr` in a separate task instead or to use a local `Channel` as a proxy:

```
p = 1
f = Future(p)
@async put!(f, remotecall_fetch(long_computation, p))
isready(f)  # will not block
```

`Distributed.AbstractWorkerPool` — Type

```
AbstractWorkerPool
```

Supertype for worker pools such as `WorkerPool` and `CachingPool`. An `AbstractWorkerPool` should implement:

- `push!` - add a new worker to the overall pool (available + busy)
- `put!` - put back a worker to the available pool
- `take!` - take a worker from the available pool (to be used for remote function execution)
- `length` - number of workers available in the overall pool

- `isready` - return false if a `take!` on the pool would block, else true

The default implementations of the above (on a `AbstractWorkerPool`) require fields

- `channel::Channel{Int}`
- `workers::Set{Int}`

where `channel` contains free worker pids and `workers` is the set of all workers associated with this pool.

`Distributed.WorkerPool` — Type

```
WorkerPool(workers::Vector{Int})
```

Create a `WorkerPool` from a vector of worker ids.

Examples

```
$ julia -p 3

julia> WorkerPool([2, 3])
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:2), Set([2, 3]), R
```

`Distributed.CachingPool` — Type

```
CachingPool(workers::Vector{Int})
```

An implementation of an `AbstractWorkerPool`. `remote`, `remotecall_fetch`, `pmap` (and other remote calls which execute functions remotely) benefit from caching the serialized/deserialized functions on the worker nodes, especially closures (which may capture large amounts of data).

The remote cache is maintained for the lifetime of the returned `CachingPool` object. To clear the cache earlier, use `clear!(pool)`.

For global variables, only the bindings are captured in a closure, not the data. `let` blocks can be used to capture global data.

Examples

```
const foo = rand(10^8);
wp = CachingPool(workers())
let foo = foo
    pmap(wp, i -> sum(foo) + i, 1:100);
end
```

The above would transfer `foo` only once to each worker.

`Distributed.default_worker_pool` — Function

```
default_worker_pool()
```

`WorkerPool` containing idle `workers` - used by `remote(f)` and `pmap` (by default).

Examples

```
$ julia -p 3

julia> default_worker_pool()
WorkerPool(Channel{Int64}(sz_max:9223372036854775807,sz_curr:3), Set([4, 2, 3]))
```

`Distributed.clear!` — Method

```
clear!(pool::CachingPool) -> pool
```

Removes all cached functions from all participating workers.

`Distributed.remote` — Function

```
remote([p::AbstractWorkerPool], f) -> Function
```

Return an anonymous function that executes function `f` on an available worker (drawn from `WorkerPool` `p` if provided) using `remotecall_fetch`.

Distributed.remotecall — Method

```
remotecall(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remotecall` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> f = remotecall(maximum, wp, A)
Future(2, 1, 6, nothing)
```

In this example, the task ran on pid 2, called from pid 1.

Distributed.remotecall_wait — Method

```
remotecall_wait(f, pool::AbstractWorkerPool, args...; kwargs...) -> Future
```

`WorkerPool` variant of `remotecall_wait(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remotecall_wait` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> f = remotecall_wait(maximum, wp, A)
Future(3, 1, 9, nothing)

julia> fetch(f)
```

```
0.9995177101692958
```

Distributed.remotecall_fetch — Method

```
remotecall_fetch(f, pool::AbstractWorkerPool, args...; kwargs...) -> result
```

`WorkerPool` variant of `remotecall_fetch(f, pid, ....)`. Waits for and takes a free worker from `pool` and performs a `remotecall_fetch` on it.

Examples

```
$ julia -p 3

julia> wp = WorkerPool([2, 3]);

julia> A = rand(3000);

julia> remotecall_fetch(maximum, wp, A)
0.9995177101692958
```

Distributed.remote_do — Method

```
remote_do(f, pool::AbstractWorkerPool, args...; kwargs...) -> nothing
```

`WorkerPool` variant of `remote_do(f, pid, ....)`. Wait for and take a free worker from `pool` and perform a `remote_do` on it.

Distributed.@spawnat — Macro

```
@spawnat p expr
```

Create a closure around an expression and run the closure asynchronously on process `p`. Return a `Future` to the result. If `p` is the quoted literal symbol `:any`, then the system will pick a processor to use automatically.

Examples

```
julia> addprocs(3);

julia> f = @spawnat 2 myid()
Future(2, 1, 3, nothing)

julia> fetch(f)
2

julia> f = @spawnat :any myid()
Future(3, 1, 7, nothing)

julia> fetch(f)
3
```

Julia 1.3

The `:any` argument is available as of Julia 1.3.

`Distributed.@fetch` — Macro

```
@fetch expr
```

Equivalent to `fetch(@spawnat :any expr)`. See `fetch` and `@spawnat`.

Examples

```
julia> addprocs(3);

julia> @fetch myid()
2

julia> @fetch myid()
3

julia> @fetch myid()
4

julia> @fetch myid()
2
```

`Distributed.@fetchfrom` — Macro

```
@fetchfrom
```

Equivalent to `fetch(@spawnat p expr)`. See `fetch` and `@spawnat`.

Examples

```
julia> addprocs(3);

julia> @fetchfrom 2 myid()
2

julia> @fetchfrom 4 myid()
4
```

`Distributed.@distributed` — Macro

```
@distributed
```

A distributed memory, parallel for loop of the form :

```
@distributed [reducer] for var = range
    body
end
```

The specified range is partitioned and locally executed across all workers. In case an optional reducer function is specified, `@distributed` performs local reductions on each worker with a final reduction on the calling process.

Note that without a reducer function, `@distributed` executes asynchronously, i.e. it spawns independent tasks on all available workers and returns immediately without waiting for completion. To wait for completion, prefix the call with `@sync`, like :

```
@sync @distributed for var = range
    body
end
```

`Distributed.@everywhere` — Macro

```
@everywhere [procs()] expr
```

Execute an expression under `Main` on all `procs`. Errors on any of the processes are collected into a `CompositeException` and thrown. For example:

```
@everywhere bar = 1
```

will define `Main.bar` on all current processes. Any processes added later (say with `addprocs()`) will not have the expression defined.

Unlike `@spawnat`, `@everywhere` does not capture any local variables. Instead, local variables can be broadcast using interpolation:

```
foo = 1
@everywhere bar = $foo
```

The optional argument `procs` allows specifying a subset of all processes to have execute the expression.

Equivalent to calling `remotecall_eval(Main, procs, expr)`.

`Distributed.clear!` — Method

```
clear!(syms, pids=workers(); mod=Main)
```

Clears global bindings in modules by initializing them to `nothing`. `syms` should be of type `Symbol` or a collection of `Symbols`. `pids` and `mod` identify the processes and the module in which global variables are to be reinitialized. Only those names found to be defined under `mod` are cleared.

An exception is raised if a global constant is requested to be cleared.

`Distributed.remoteref_id` — Function

```
remoteref_id(r::AbstractRemoteRef) -> RRID
```

`Futures` and `RemoteChannels` are identified by fields:

- `where` - refers to the node where the underlying object/storage referred to by the reference actually exists.
- `whence` - refers to the node the remote reference was created from. Note that this is different from the node where the underlying object referred to actually exists. For example calling `RemoteChannel(2)` from the master process would result in a `where` value of 2 and a `whence` value of 1.
- `id` is unique across all references created from the worker specified by `whence`.

Taken together, `whence` and `id` uniquely identify a reference across all workers.

`remoteref_id` is a low-level API which returns a `RRID` object that wraps `whence` and `id` values of a remote reference.

`Distributed.channel_from_id` — Function

```
channel_from_id(id) -> c
```

A low-level API which returns the backing `AbstractChannel` for an `id` returned by `remoteref_id`. The call is valid only on the node where the backing channel exists.

`Distributed.worker_id_from_socket` — Function

```
worker_id_from_socket(s) -> pid
```

A low-level API which, given a `IO` connection or a `Worker`, returns the `pid` of the worker it is connected to. This is useful when writing custom `serialize` methods for a type, which optimizes the data written out depending on the receiving process id.

`Distributed.cluster_cookie` — Method

```
cluster_cookie() -> cookie
```

Return the cluster cookie.

`Distributed.cluster_cookie` — Method

```
cluster_cookie(cookie) -> cookie
```

Set the passed cookie as the cluster cookie, then returns it.

Cluster Manager Interface

This interface provides a mechanism to launch and manage Julia workers on different cluster environments. There are two types of managers present in Base: `LocalManager`, for launching additional workers on the same host, and `SSHManager`, for launching on remote hosts via `ssh`. TCP/IP sockets are used to connect and transport messages between processes. It is possible for Cluster Managers to provide a different transport.

`Distributed.ClusterManager` — Type

```
ClusterManager
```

Supertype for cluster managers, which control workers processes as a cluster. Cluster managers implement how workers can be added, removed and communicated with. `SSHManager` and `LocalManager` are subtypes of this.

`Distributed.WorkerConfig` — Type

```
WorkerConfig
```

Type used by `ClusterManager`s to control workers added to their clusters. Some fields are used by all cluster managers to access a host:

- `io` – the connection used to access the worker (a subtype of `IO` or `Nothing`)
- `host` – the host address (either an `AbstractString` or `Nothing`)
- `port` – the port on the host used to connect to the worker (either an `Int` or `Nothing`)

Some are used by the cluster manager to add workers to an already-initialized host:

- `count` – the number of workers to be launched on the host
- `exename` – the path to the Julia executable on the host, defaults to `"$(Sys.BINDIR)/julia"` or `"$(Sys.BINDIR)/julia-debug"`
- `exeflags` – flags to use when lauching Julia remotely

The `userdata` field is used to store information for each worker by external managers.

Some fields are used by `SSHManager` and similar managers:

- `tunnel` – `true` (use tunneling), `false` (do not use tunneling), or `nothing` (use default for the manager)
- `multiplex` – `true` (use SSH multiplexing for tunneling) or `false`
- `forward` – the forwarding option used for `-L` option of ssh
- `bind_addr` – the address on the remote host to bind to
- `sshflags` – flags to use in establishing the SSH connection
- `max_parallel` – the maximum number of workers to connect to in parallel on the host

Some fields are used by both `LocalManager`s and `SSHManager`s:

- `connect_at` – determines whether this is a worker-to-worker or driver-to-worker setup call
- `process` – the process which will be connected (usually the manager will assign this during `addprocs`)
- `ospid` – the process ID according to the host OS, used to interrupt worker processes
- `environ` – private dictionary used to store temporary information by Local/SSH managers
- `ident` – worker as identified by the `ClusterManager`
- `connect_idents` – list of worker ids the worker must connect to if using a custom topology
- `enable_threaded_blas` – `true`, `false`, or `nothing`, whether to use threaded BLAS or not on the workers

`Distributed.launch` — Function

```
launch(manager::ClusterManager, params::Dict, launched::Array, launch_ntfy::Con
```

Implemented by cluster managers. For every Julia worker launched by this function, it should append a `WorkerConfig` entry to `launched` and notify `launch_ntfy`. The function MUST exit once all workers, requested by `manager` have been launched. `params` is a dictionary of all keyword arguments `addprocs` was called with.

`Distributed.manage` — Function

```
manage(manager::ClusterManager, id::Integer, config::WorkerConfig. op::Symbol)
```

Implemented by cluster managers. It is called on the master process, during a worker's lifetime, with appropriate `op` values:

- with `:register`/`:deregister` when a worker is added / removed from the Julia worker pool.
- with `:interrupt` when `interrupt(workers)` is called. The `ClusterManager` should signal the appropriate worker with an interrupt signal.
- with `:finalize` for cleanup purposes.

`Base.kill` — Method

```
kill(manager::ClusterManager, pid::Int, config::WorkerConfig)
```

Implemented by cluster managers. It is called on the master process, by `rmprocs`. It should cause the remote worker specified by `pid` to exit. `kill(manager::ClusterManager.....)` executes a remote `exit()` on `pid`.

`Sockets.connect` — Method

```
connect(manager::ClusterManager, pid::Int, config::WorkerConfig) -> (instrm::IO
```

Implemented by cluster managers using custom transports. It should establish a logical connection to worker with id `pid`, specified by `config` and return a pair of `IO` objects. Messages from `pid` to current process will be read off `instrm`, while messages to be sent to `pid` will be written to `outstrm`. The custom transport implementation must ensure that messages are delivered and received completely and in order. `connect(manager::ClusterManager.....)` sets up TCP/IP socket connections in-between workers.

`Distributed.init_worker` — Function

```
init_worker(cookie::AbstractString, manager::ClusterManager=DefaultClusterManag
```

Called by cluster managers implementing custom transports. It initializes a newly launched process as a worker. Command line argument `--worker[=<cookie>]` has the effect of initializing a process as a worker using TCP/IP sockets for transport. `cookie` is a `cluster_cookie`.

`Distributed.start_worker` — Function

```
start_worker([out::IO=stdout], cookie::AbstractString=readline(stdin); close_st
```

`start_worker` is an internal function which is the default entry point for worker processes connecting via TCP/IP. It sets up the process as a Julia cluster worker.

host:port information is written to stream `out` (defaults to stdout).

The function reads the cookie from stdin if required, and listens on a free port (or if specified, the port in the `--bind-to` command line option) and schedules tasks to process incoming TCP connections and requests. It also (optionally) closes stdin and redirects stderr to stdout.

It does not return.

`Distributed.process_messages` — Function

```
process_messages(r_stream::IO, w_stream::IO, incoming::Bool=true)
```

Called by cluster managers using custom transports. It should be called when the custom transport implementation receives the first message from a remote worker. The custom transport must manage a logical connection to the remote worker and provide two `IO` objects, one for incoming messages and the other for messages addressed to the remote worker. If `incoming` is `true`, the remote peer initiated the connection. Whichever of the pair initiates the connection sends the cluster cookie and its Julia version number to perform the authentication handshake.

See also `cluster_cookie`.

« Delimited Files File Events »