

NetWorkSpaces for R

Scientific Computing Associates, Inc.

Contents

1	Introduction	7
2	Getting Started	11
2.1	Prerequisites	11
2.2	NetWorkSpaces Server	11
2.2.1	Starting the Server	11
2.2.2	Stopping the Server	12
2.3	NetWorkSpaces Client	13
2.3.1	Installing the Client	13
2.3.2	Start NetWorkSpaces Client	13
2.4	Starting Sleigh	14
2.4.1	Local Launch Mechanism	14
2.4.2	SSH Launch Mechanism	15
2.4.3	RSH Launch Mechanism on Windows	16
2.4.4	Web Launch Mechanism	17
3	Tutorials	19
3.1	NetWorkSpaces Tutorial	19
3.2	Sleigh Tutorial	22
4	Reference	25
4.1	netWorkSpace class	25
	netWorkSpace	25
	nwsFetch	27
	nwsFetchTry	28
	nwsFind	29
	nwsFindTry	30
	nwsStore	31
	nwsFetchFile	32
	nwsFetchTryFile	33
	nwsFindFile	34
	nwsFindTryFile	35

nwsStoreFile	36
nwsIFetch	37
nwsIFetchTry	38
nwsIFind	39
nwsIFindTry	40
nwsClose	41
nwsDeclare	42
nwsDeleteVar	43
nwsListVars	43
nwsServerObject	44
nwsVariable	45
nwsWsName	46
4.2 nwsServer class	48
nwsServer	48
nwsDeleteWs	49
nwsListWss	50
nwsMktempWs-methods	51
nwsOpenWs	52
nwsUseWs	53
close	54
4.3 sleigh class	55
sleigh	55
eachElem	57
eachWorker	62
rankCount	64
status	64
stopSleigh	65
workerCount	66
cmdLaunch	67
defaultSleighOptions	67
envcmd	68
isClosure	68
launch	69
lsfcmd	69
lsfSleigh	70
rshcmd	70
scriptcmd	71
sshcmd	71
workerLoop	72
4.4 sleighPending class	73
sleighPending	73
checkSleigh	74

CONTENTS

5

waitSleigh 75

Chapter 1

Introduction

NetWorkSpaces (NWS) provides a framework to coordinate programs written in scripting languages; NWS currently supports the languages Python, Matlab, and R. This User Guide is for the R system, and it assumes that the reader is familiar with R.

An R program uses variables to communicate data from one part of the program to another. For example, `'x <- 123'` assigns the value 123 to the variable named x. Later portions of the program can reference “x” to use the value 123. This mechanism is generally known as *binding*. In this case, the binding associates the value 123 with the name “x”. The collection of name-value bindings in use by a program is often referred to as its “workspace.”

Two or more R programs use NWS to communicate data by means of name-value bindings stored in a network-based workspace (a NetWorkSpace, which in R is an instance of a NetWorkSpace object). One program creates a binding, while another program reads the value the binding associated with the name. This is clearly quite similar to a traditional workspace; however, a NetWorkSpace differs from a traditional workspace in two important ways.

First, in a setting in which two or more R programs are interacting, it would not be unusual for one to attempt to “read” the value of a name before that name has been bound to a value. Rather than receiving an “unbound variable” error, the reading program (by default) simply waits (or “blocks”) until the binding occurs. Second, a common usage paradigm involves processing a sequence of values for a given name. One R program carries out a computation based on the first value, while another might carry out a computation on the second, and so on. To facilitate this paradigm, more than one value may be bound to a name in a workspace and values may be “removed” (fetch) as opposed to read (find). By default, values bound to a name are consumed in first-in-first-out (FIFO) order, but other modes are supported: last-in-first-out (LIFO), multiset (no ordering implied) and single (only the last value bound is retained). Since all its values could be removed, a name can, in fact, have no values associated with it.

A NetWorkSpace provides five basic operations: `nwsStore()`, `nwsFetch()`, `nwsFetchTry()`,

`nwsFind()`, and `nwsFindTry()`:

`nwsStore()` introduces a new binding for a specific name in a given workspace.

`nwsFetch()` fetches (removes) a value associated with a name.

`nwsFind()` reads a value without removing it.

Note that `nwsFetch()` and `nwsFind()` block if no value is bound to the name. `nwsFetchTry()` and `nwsFindTry()` are non-blocking; they return an empty value or user-supplied default if no value is available.

There are several additional `NetWorkspace` operations:

`nwsClose()` closes the connection to a workspace. Depending on the ownership, closing a connection to a workspace can result in removing the workspace.

`nwsDeclare()` declares a variable name with a specific mode.

`nwsDeleteVar()` deletes a name from a workspace.

`nwsListVars()` provides a list of variables (bindings) in a workspace.

`nwsWsName()` returns the name of the specified workspace.

In addition to a `NetWorkspace`, an R client of NWS also uses an `NWSServer` object. This object is created automatically when a new `NetWorkspace` object is created, so you don't need to interact directly with it. However, `server` object is an attribute of the `NetWorkspace`, and you can access it using the R syntax `netWorkspace_object@server`.

A `NWSServer` object supports the following actions:

`nwsOpenWs()` connects to a workspace or creates one if the specified workspace does not exist.

`nwsUseWs()` uses a `NetWorkspace` without claiming ownership.

`nwsDeleteWs()` explicitly deletes a workspace.

`nwsListWs()` provides a list of workspaces in the server.

The operations above enable coordination of different programs using NWS. There is

also a mechanism, built on top of NWS, called Sleigh (inspired by R's SNOW package) to enable parallel function evaluation. Sleigh is especially useful for running “embarrassingly parallel” programs on multiple networked computers. Once a sleigh is created by specifying the nodes that are participating in computations, you can use:

eachElem() to invoke a function on each element of a vector, with the invocations being evaluated concurrently by the sleigh participants.

eachWorker() to have each participant in a sleigh invoke a function. You can use this operation, for example, to build up requisite state prior to an `eachElem()` invocation.

Various data structures (workspaces, name-value bindings, etc.) in a NWS server can be monitored and even modified using a Web interface. In distributed programming, even among cooperating programs, the state of shared data can be hard to understand. The Web interface presents a simple way of checking current state remotely. This tool can also be used for learning and debugging purposes, and for monitoring “normal” R programs as well.

Chapter 2

Getting Started

This chapter provides step-by-step procedures for setting up NetWorkSpaces and Sleigh for R.

2.1 Prerequisites

NetWorkSpaces and Sleigh run on all Linux and Windows platforms. To use them, you must install the following software:

1. R 2.1.1 or above
2. NetWorkSpaces server <http://nws-r.sourceforge.net>
The NetWorkSpaces server also requires the following:
 - (a) Python 2.4 <http://www.python.org>
(We recommend ActiveState Python <http://www.activestate.com> for Windows).
 - (b) Twisted Framework <http://twistedmatrix.com>

2.2 NetWorkSpaces Server

2.2.1 Starting the Server

There are three ways to start a NetWorkSpaces server:

- Use the `twistd` command:
Open up a shell in UNIX, or open up a twisted command prompt on Windows, and type the following:

```
% twistd -y nws.tac
```

Note: `nws.tac` can reside in different directories, depending on the platform and the type of installation (root versus non-root). For root installation, `nws.tac` is located in `/etc` on UNIX, and in the `PYTHON24` directory on Windows.

- execute the `nws` script (UNIX only):

```
% nws start
```

- start Windows services (Windows only):

1. Open up a command prompt.
2. Install `NwsService` by executing `NwsService.py`, which is located in Python's `scripts` directory.

```
% python NwsService.py install
```

3. Start `NwsService`

```
% python NwsService.py start
```

This starts a `NetWorkSpaces` server on localhost at port 8765.

2.2.2 Stopping the Server

There are also three ways to stop a `NetWorkSpaces` server:

- If you used the `twistd` command to start the server:

```
% kill `cat twistd.pid`
```

- If you used the `nws` script to start the server:

```
% nws stop
```

- If you used Windows service to start the server:

```
% python NwsService.py stop
```

2.3 NetWorkSpaces Client

2.3.1 Installing the Client

To install NetWorkSpaces source distribution on UNIX:

1. R CMD INSTALL *nws_version.tar.gz*

To install NetWorkSpaces on Windows XP:

- For binary distribution (in a zip file format),
 1. Start RGui.
 2. Select Packages Menu.
 3. Install package(s) from local zip file.
 4. Find *nws_version.zip* you have obtained.
- For source distribution,
 1. Follow instructions on this website, <http://www.murdoch-sutherland.com/Rtools/>
 2. R CMD INSTALL *nws*

2.3.2 Start NetWorkSpaces Client

Once you've got a NetWorkSpace server up and running, you're ready to use NetWorkSpaces.

1. Start up an R session.
2. Type the following:

```
> ws = netWorkSpace('R space')
> nwsStore(ws, 'x', 1)
```

This step creates a workspace named 'R space' and stores a variable x with value 1 to the workspace.

You can also view what's in the workspace using a web interface. To do this, you point your browser to http://server_host_name:8766, where *server_host_name* is the machine that a NetWorkSpaces server resides on.

To examine values that you've created in a workspace using the server's web interface, you also need a *babelfish*. The *babelfish* translates values into a human readable format so they can be displayed in a web browser. If a value is a string, then the web interface simply displays the contents of the string, without any help from the *babelfish*. But, if the value is any other type of R object, it needs help from the R *babelfish*. To start up *babelfish*, execute the following command in another terminal:

```
% R CMD BATCH babelfish.R
```

Note: this function will not return until you exit out of R.

For Windows, user can also start babelfish as Windows service by following these steps:

1. Open up a command prompt
2. cd to the directory where NWS client package is installed.
3. cd to bin directory
4. Install R Babelfish Service

```
python RBabelfishService.py install
```

5. Start R Babelfish Service

```
python RBabelfishService.py start
```

For more examples on using NetWorkSpaces, see the Tutorials chapter.

2.4 Starting Sleigh

Sleigh is a R class, built on top of the NetWorkSpaces, that makes it very easy to write simple parallel programs. Sleigh has concept of one master and multiple workers. The master sends jobs to workers who may or may not be on the same machine as the master. To enable the master to communicate with workers, Sleigh supports several mechanisms to launch workers to run jobs. For remote launch mechanism, make sure R is in the PATH of worker machines.

2.4.1 Local Launch Mechanism

Local launch mechanism is the default option to start up workers. It starts three workers by default on local machine. Users may choose to change the number of workers by setting workerCount variable in the sleigh constructor. Local launch is useful for SMP machine, where multiple cores/processors are available. It is also useful in debugging parallel programs locally before running over a large cluster.

To create a sleigh object, simply load the nws package and type the following:

```
> s = sleigh()
```

This is equivalent to:

```
> s = sleigh(launch='local')
```

2.4.2 SSH Launch Mechanism

To start up workers on different machines, a remote login mechanism, such as SSH client, is needed for the sleigh master. Remote workers need to run SSH server in order to accept requests.

We don't recommend running sleigh workers on Windows, as it often requires UNIX-like SSH server running on the Windows machine. This can lead to problems when different quoting style was used by the SSH server and the Windows. If users insist on running workers on Windows, we recommend Cygwin's ssh server or copSSH from ITef!x, <http://www.itefix.no/phpws/>. To setup Cygwin ssh server, <http://ncyoung.com/entry/389> provides a nice tutorial. Installation of copSSH is straight out of box. After installation, remember to activate users.

Next, we need to setup password-less ssh login.

Setting Up a Password-less SSH Login

- To generate public and private keys, follow the steps below.
 1. Open a terminal (shell on UNIX and DOS prompt on Windows).
 2. `ssh-keygen -t rsa` (assume `ssh-keygen` is in your `PATH`)
 3. `cd .ssh` (`.ssh` directory is located in your `HOME` directory, `/home/user` on UNIX or Cygwin and `C:/Program Files/copssh/home/user` on Windows)
 4. `cp id_rsa.pub authorized_keys` This step allows password-less login to local machine.
 5. For all remote machines that you want password-less login, append the content of `id_rsa.pub` to their `authorized_keys` file.
- To test the password-less login, type the following command:

```
% ssh hostname date
```

If everything is setup correctly, you should not be asked for password and the current date on remote machine will be returned.

To start up sleigh workers using SSH, simply load the `nws` package and type the following:

```
> s = sleigh(launch=sshcmd)
```

This creates three workers on local machine.

To start up sleigh workers on remote machines,

```
> s = sleigh(nodeList=c('node1', 'node2'), launch=sshcmd)
```

2.4.3 RSH Launch Mechanism on Windows

On Windows, where SSH is not available, users can use RSH instead. Windows 2000/XP comes with RSH client, but in order to communicate with other Windows machines, user must have an RSH server running on the machine. To do so, user must first download and install a copy of Windows Services for UNIX (SFU), which is available for free at Microsoft website. SFU allows users to start up RSH server as Windows service or as an UNIX daemon. In this section, we will focus on how to start up RSH server as Windows service.

1. Install SFU to local machine.
2. Log in to the machine as an Administrator.
3. Open up a command prompt.
4. Go to SFU's common directory.
5. Install Windows Remote Shell Service.

```
rshsvc.exe -install
```

6. Start Windows Remote Shell Service.

```
rshsvc.exe -start
```

Users can also enable automatic or manual start of Windows Remote Shell Service through Services console in Administrative Tools.

7. Add .rhosts file.

The .rhosts file resides in the location specified by the registry entry:

HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\RshSvc\RhostsPath

This value usually is C:\Windows\System32\Drivers\etc

The format of the .rhosts file is:

```
machine1 user1          # user1 can log in from machine1
+ user2                # user2 can log in from any machine
machine2 user1 user2   # user1 and user2 can log in from machine2
machine3 +             # all users on machine3 can log in
+ +                    # any user from any machine can log in
```

8. Test RSH from a command prompt

```
rsh localhost set
```

If everything setup correctly, this command should return a list of environment variables set locally.

To start up sleigh workers using RSH, simply load the nws package and type the following:

```
> s = sleigh(launch=rshcmd)
```

2.4.4 Web Launch Mechanism

Web launch mechanism allows user to start up workers on different machines in an ad-hoc way, without setting up remote login mechanisms, such as SSH and RSH.

To use the web launch option, follow the steps below.

1. Create an instance of Sleigh:

```
> s = sleigh(launch='web')
```

The Sleigh constructor does not return until it gets a signal that all workers have started and are ready to accept jobs.

2. Log in to a remote machine.
3. Start a R session.
4. Open a web browser and point to `http://server_host_name:8766`
5. Click on the newly created Sleigh workspace, and read the value from variable 'runMe'. It usually has value similar to:
`webLaunch('sleigh_ride_0000000004_tmp1a6c0h', 'mercury', 8765);`
6. Copy the 'runMe' value to the R session.
7. Repeat steps 2-6 for each worker that needs to be started.
8. Once all workers have started, delete the 'DeleteMeWhenAllWorkersStarted' variable from the Sleigh workspace. This signals Sleigh master that the workers have started and are ready to accept work.

Now you're ready to send jobs to remote workers. See the Sleigh for R Tutorial section in the Tutorials chapter for more information.

Chapter 3

Tutorials

3.1 NetWorkSpaces Tutorial

NetWorkSpaces (NWS) is an R package that makes it very easy for different R programs running on (potentially) different machines to communicate and coordinate with one another.

First, an NWS server must be started, as described in the Getting Started chapter. Second, start `babelfish.R`:

```
% R CMD BATCH babelfish.R
```

A `babelfish` is useful if you're using the web interface to examine your workspaces.

Now start an R session, load the `nws` package, and create an NWS workspace:

```
% R -q  
> library(nws)  
> ws = netWorkSpace('bayport')
```

The workspace is called `bayport`, and it is using the NWS server on the local machine. Additional arguments can be used to specify the hostname and port used by the NWS server if necessary. Once we have a workspace, we can write data into a variable using the `nwsStore` function:

```
> nwsStore(ws, 'joe', 17)
```

The variable `joe` now has a value of 17 in our workspace. To read that variable we use the `nwsFind` function:

```
> age = nwsFind(ws, 'joe')
```

which sets age to 17.

Note that the `nwsFind` function will block until the variable `joe` has a value. That is important when we're trying to read that variable from a different machine. If it didn't block, you might have to repeatedly try to read the variable until it succeeded. Of course, there are times when you don't want to block, but just want to see if some variable has a value. That is done with the `nwsFindTry` function. Let's try reading a variable that doesn't exist using `nwsFindTry`:

```
> age = nwsFindTry(ws, 'chet')
```

That assigns a `NULL` to `age`, since we haven't stored any value to that variable. If you'd rather have `nwsFindTry` return some other value when the variable doesn't exist (or has no value), you can use the command:

```
> age = nwsFindTry(ws, 'chet', 0)
```

which assigns 0 to `age`.

So far, we've been using variables in workspaces in much the same way that global variables are used within a single program. This is certainly useful, but NWS workspaces can also be used to send a sequence of messages from one program to another. If the `nwsStore` function is executed multiple times, the new values don't overwrite the previous values, they are all saved. Now the `nwsFind` function will only be able to read the first value that was written, but this is where another function, called `nwsFetch` is useful. The `nwsFetch` function works the same as `nwsFind`, but in addition, it removes the value.

Let's try to write multiple values to a variable:

```
> n = c(16, 19, 25, 22)
> for (x in n) {
+   nwsStore(ws, 'biff', x)
+ }
```

To read the values, we just call `nwsFetch` repeatedly:

```
> n = vector()
> for (i in 1:4) {
+   n[i] = nwsFetch(ws, 'biff')
+ }
> n
[1] 16 19 25 22
```

If we didn't know how many values were stored in a variable, we could have done the following:

```
> n = vector()
> i = 1
> while (!is.null(tmp <- nwsFetchTry(ws, 'biff'))) {
+   n[i] = tmp
+   i = i + 1
+ }
> n
[1] 16 19 25 22
```

This uses `nwsFetchTry`, which works like `nwsFetch`, except that it is non-blocking, just like `nwsFindTry`.

These are the basic operations provided by NWS. It's a good idea to play around with these operations using two R sessions. That way, you can really transfer data between two different programs. Also, you can see how the blocking operations work. We were pretty careful never to block in any of the examples above because we were only using one R session. To use two R sessions, just execute R in another window, load the `nws` package, and then open the bayport workspace. This can be done with the same commands that we used previously, but this time, the command:

```
> ws = netWorkspace('bayport')
```

won't create the bayport workspace, since it already exists.

Now you can execute an operation such as:

```
> x = nwsFetch(ws, 'frank')
```

in one session, watch it block for a minute, and then execute `nwsStore` in the other session:

```
> nwsStore(ws, 'frank', 18)
```

and see that the `nwsFetch` in the first session completes.

While you're experimenting with these operations, it can be very helpful to use the Net-WorkSpace server's web interface to see what's going on in your workspaces. Just point a web browser to the URL <http://localhost:8766>. If you're using a browser on a different machine from the NWS server you'll have to use the appropriate hostname, rather than `localhost`.

3.2 Sleigh Tutorial

Sleigh is the part of the NWS package that makes it easy to write parallel programs. It provides two basic functions for executing tasks in parallel: *eachElem* and *eachWorker*.

eachElem is used to execute a specified function multiple times in parallel with a varying set of arguments.

eachWorker is used to execute a function exactly once on every worker in the sleigh with a fixed set of arguments.

eachElem is all that is needed for many basic programs, so that is what we will start with.

First, you need to start up your sleigh, so after loading the NWS package, you type:

```
> s = sleigh()
```

This starts three sleigh workers locally, but an *workerCount* argument can specify different number of workers to be started. Let's shut down the sleigh so we can start different number of workers. Here's how to shut down the current sleigh:

```
> stopSleigh(s)
```

This deletes the sleigh's workspace, and shuts down all of the sleigh worker processes.

Now we'll make a new sleigh with two workers, and a NWSServer that's running on node10:

```
> s = sleigh(workerCount=2, nwsHost='node10')
```

Now we're ready to run a parallel program. Here it is:

```
> result = eachElem(s, function(x) x + 1, 1:10)
```

In that simple command, we have defined a function and a set of data that is processed by multiple workers in parallel, and returned each of the results in a list.

This *eachElem* command puts 10 tasks into the sleigh workspace. Each task contains one value from 1 to 10. This value is passed to the function as the value of *x*. The return value of the function is put into the sleigh workspace. The *eachElem* command waits for all of the results to be put into the workspace, and returns them as a list of numbers from 2 to 11.

As a second example, let's define a function that adds two numbers together:

```
> add2 = function(x, y) x + y
```

Now we'll use that function to add two vectors:

```
> result = eachElem(s, add2, list(0:9, 10:1))
```

This is the parallel equivalent to the R expression `0:9 + 10:1`.

We can keep adding more vector arguments in this way, but there is also a way to add arguments that are the same for every task, which we call fixed arguments:

```
> result = eachElem(s, add2, list(0:9), list(20))
```

This is equivalent to the R expression `0:9 + 20`.

The order of the arguments passed to the function are normally in the specified order, which means that the fixed arguments always come after the varying arguments. To change this order, a permutation vector can be specified. The permutation vector is specified using the `eo` parameter, which can specify extra, optional arguments. We create this argument as follows:

```
> eo = list(argPermute=2:1)
```

For example, to perform the parallel equivalent of the R expression `20 - 0:19`, we do the following:

```
> result = eachElem(s, function(x,y) x - y, list(0:19), list(20), eo=eo)
```

This permutation vector says to first use the second argument, and then use the first, thereby reversing the order of the two arguments.

Actually, we can simplify this by taking advantage of recycling:

```
> result = eachElem(s, '-', list(20, 0:19))
```

The `20` is recycled 20 times, just as in the R expression `20 - 0:19`. This also demonstrates the direct use of the R subtraction function.

The `eo` argument can also be used to make `eachElem` return immediately after submitting the tasks, thus making it non-blocking. A “pending object” is returned, which can be used to check how many of the tasks are complete, and to wait until all tasks are finished. Here’s a quick example:

```
> eo = list(blocking=0)
> p = eachElem(s, '+', list(0:19, 20:1), eo=eo)
> while (checkSleigh(p) > 0) {
+   # Do something useful for a little while
+ }
> result = waitSleigh(p)
```

The `eo` argument can also be used to enable watermarking by specifying a “load factor.” This limits the number of tasks that are put into the workspace at the same time. That could be important if you’re executing a lot of tasks. Setting the load factor to 3 limits the number of tasks in the workspace to 3 times the number of workers in the sleigh.

Here’s how to do it:

```
> eo = list(loadFactor=3)
> result = eachElem(s, '+', list(0:1000, 0:1000), eo=eo)
```

The results are exactly the same as not using a load factor. Setting this option only changes the way that tasks are submitted by the `eachElem` command.

Chapter 4

Reference

4.1 netWorkspace class

netWorkspace	Class “netWorkspace”
--------------	----------------------

Description

Class representing netWorkspace.

Objects from the Class

Objects can be created by calls of the form
`new("netWorkspace", wsName, serverHost, port, useUse, serverWrap, ...)`.

`wsName`: name of the netWorkspace to be created.

`serverHost`: host name of the server this netWorkspace will be connected to. By default, local machine is used.

`port`: port number of the server this netWorkspace will be connected to. Default port number is 8765.

`useUse`: a boolean value indicating whether ownership will be claimed for this netWorkspace. By default, `useUse=FALSE`, which means ownership will be claimed.

`serverWrap`: a netWorkSpaces server object. Reuse an existing server connection, instead of creating a new server connection.

Slots

server: Object of class "nwsServer" representation of the server that this netWorkspace connects to.

wsName: Object of class "character" representation of this netWorkspace's name.

Methods

initialize signature(.Object = "netWorkspace"): netWorkspace class constructor.

nwsFetch signature(.Object = "netWorkspace"): fetch a value of a workspace variable.

nwsFetchTry signature(.Object = "netWorkspace"): try to fetch a value of a workspace variable.

nwsFind signature(.Object = "netWorkspace"): find a value of a workspace variable.

nwsFindTry signature(.Object = "netWorkspace"): try to find a value of a workspace variable.

nwsStore signature(.Object = "netWorkspace"): store a value into a workspace variable.

nwsFetchFile signature(.Object = "netWorkspace"): fetch a value of a workspace variable and write it to a file.

nwsFetchTryFile signature(.Object = "netWorkspace"): try to fetch a value of a workspace variable and write it to a file.

nwsFindFile signature(.Object = "netWorkspace"): find a value of a workspace variable and write it to a file.

nwsFindTryFile signature(.Object = "netWorkspace"): try to find a value of a workspace variable and write it to a file.

nwsStoreFile signature(.Object = "netWorkspace"): store data from a file into a workspace variable.

nwsIFetch signature(.Object = "netWorkspace"): create a function that acts as a destructive iterator over the values of the specified variable.

nwsIFetchTry signature(.Object = "netWorkspace"): create a function that acts as a destructive iterator over the values of the specified variable.

nwsIFind signature(.Object = "netWorkspace"): create a function that acts as a non-destructive iterator over the values of the specified variable.

nwsIFindTry signature(.Object = "netWorkspace"): create a function that acts as a non-destructive iterator over the values of the specified variable.

nwsClose signature(.Object = "netWorkspace"): close the connection to the NWS server.

nwsDeclare signature(.Object = "netWorkspace"): declare the mode of a workspace variable.

nwsDeleteVar signature(.Object = "netWorkspace"): delete a variable from a workspace.

nwsListVars signature(.Object = "netWorkspace"): list all variables in a workspace.

ServerObject signature(.Object = "netWorkspace"): return the associated Nws-Server object.

nwsVariable signature(.Object = "netWorkspace"): create an Active Binding for a NetWorkspace Variable

nwsWsName signature(.Object = "netWorkspace"): return the name of the workspace.

Examples

```
## Not run:
# To create a new workspace with the name "my space" use:
ws = netWorkspace('my space')

# To create a new workspace called "my space2" on nws server
# running on port 8245 on machine zeus:
ws2 = netWorkspace(wsName='my space2', serverHost='zeus', port=8245)
## End(Not run)
```

nwsFetch

netWorkspace Class Method

Description

Fetch value associates with a variable from the shared netWorkspace, .Object.

Usage

```
## S4 method for signature 'netWorkspace':
nwsFetch(.Object, xName)
```

Arguments

.Object	a netWorkspace class object
xName	name of the variable to be fetched

Details

Fetch method blocks until a value for xName is found in the shared netWorkspace, .Object. Once found, remove a value associated with xName from the shared netWorkspace. This operation is atomic. If there are multiple NetWorkSpaces clients nwsFetch or nwsFetchTry a given xName, any given value from the set of values associated with xName will be returned to just one client session. If there is more than one value associated with xName, the particular value removed depends on xName's behavior. See nwsDeclare for details.

See Also

nwsDeclare, nwsFetchTry

Examples

```
## Not run:
ws <- netWorkspace('nws example')
nwsStore(ws, 'x', 10)
nwsFetch(ws, 'x')
nwsFetch(ws, 'x') # no value for x; therefore block on fetch
## End(Not run)
```

nwsFetchTry	<i>netWorkspace Class Method</i>
-------------	----------------------------------

Description

Attempt to fetch value associates with a variable from the shared netWorkspace; a non-blocking version of nwsFetch.

Usage

```
## S4 method for signature 'netWorkspace':
nwsFetchTry(.Object, xName, defaultVal=NULL)
```

Arguments

.Object	a netWorkspace class object
xName	name of the variable to be fetched
defaultVal	value to return, if xName is not found in the netWorkspace

Details

Look in the shared netWorkspace for a value bound to xName. If found, remove a value associated with xName from the shared netWorkspace. This operation is atomic. If there are multiple NetWorkSpaces clients nwsFetch or nwsFetchTry a given xName, any given value from the set of values associated with xName will be returned to just one client session.

If variable is not found, return immediately rather than block on the operation (as in the case of nwsFetch). If variable is not found, the value of argument defaultVal is returned. By default, defaultVal is NULL.

See Also

nwsDeclare, nwsFetch

Examples

```
## Not run:
ws <- netWorkspace('nws example')
# If variable 'x' is not found in the shared netWorkspace,
# return default value, NULL.
nwsFetchTry(ws, 'x')
# If variable 'x' is not found in the shared netWorkspace, return 10.
nwsFetchTry(ws, 'x', 10)
## End(Not run)
```

nwsFind

netWorkspace Class Method

Description

Find value associates with a variable in the shared netWorkspace, .Object.

Usage

```
## S4 method for signature 'netWorkspace':
nwsFind(.Object, xName)
```

Arguments

.Object	a netWorkspace class object
xName	name of the variable to be found

Details

Find method blocks until a value for xName is found in the shared netWorkspace .Object. Once found, return the value associated with xName, but the value is not removed from the shared netWorkspace (as in the case of nwsFetch). If there is more than one value associated with xName, the particular value returned depends on xName's behavior. See nwsDeclare for details.

See Also

nwsDeclare, nwsFetch

Examples

```
## Not run:
ws <- netWorkspace('nws example')
nwsStore(ws, 'x', 1)
x <- nwsFind(ws, 'x')
## End(Not run)
```

nwsFindTry

netWorkspace Class Method

Description

Attempt to find value associates with a variable from the shared netWorkspace; a non-blocking version of nwsFind.

Usage

```
## S4 method for signature 'netWorkspace':
nwsFindTry(.Object, xName, defaultVal=NULL)
```

Arguments

.Object	a netWorkspace class object
xName	name of variable to be found
defaultVal	value to return if xName is not found

Details

Look in the shared netWorkspace for a value bound to xName. Once found, return the value associated with xName, but the value is not removed from the shared netWorkspace. If there is more than one value associated with xName, the particular value returned depends on varName's behavior. See `nwsDeclare` for details.

If variable is not found, return immediately rather than block on the operation (as in the case of `nwsFind`), and the value of argument defaultVal is returned. By default, defaultVal is NULL.

See Also

`nwsDeclare`, `nwsFind`

Examples

```
## Not run:
ws <- netWorkspace('nws example')
x <- nwsFindTry(ws, 'abc', -1)
## End(Not run)
```

nwsStore

netWorkspace Class Method

Description

Store value associates with a variable to the shared netWorkspace.

Usage

```
## S4 method for signature 'netWorkspace':
nwsStore(.Object, xName, xVal)
```

Arguments

.Object	a netWorkspace class object
xName	name of the variable to be stored
xVal	value to be stored

Details

nwsStore associates the value xVal with the variable xName in the shared netWorkSpace corresponding to .Object, thereby making the value available to all the distributed R processes. If a mode has not already been set for xName, 'fifo' will be used (see nwsDeclare).

Note that, by default ('fifo' mode), nwsStore is not idempotent: repeating nwsStore (nws, xName, xVal) will add additional values to the set of values associated with the variable named xName. See the examples below for details.

See Also

nwsDeclare

Examples

```
## Not run:
ws <- netWorkSpace('nws example')

# To store value 5 bound to variable 'x' on the netWorkSpace 'ws'
# (If 'x' was declared, then its mode is inherited,
# otherwise 'x' uses the default mode 'fifo')
nwsStore(ws, 'x', 5)

# store 10 values associate with variable y to the netWorkSpace
for (i in 1:10)
  nwsStore(ws, 'y', i)

# retrieve 10 values associate with variable y from the netWorkSpace
for (i in 1:10)
  print(nwsFetch(ws, 'y'))
## End(Not run)
```

nwsFetchFile

netWorkSpace Class Method

Description

Fetch a value of a workspace variable and write it to a file.

Usage

```
## S4 method for signature 'netWorkSpace':
nwsFetchFile(.Object, xName, fObj)
```

Arguments

.Object	a netWorkspace object
xName	name of the variable to find
fObj	File to write data to

Details

The nwsFetchFile method blocks until a value in the variable specified by 'xName' is found. Once found, it writes the value to the file object, and the value is removed from the variable.

See Also

nwsFindFile, nwsFetch

Examples

```
## Not run:  
ws <- netWorkspace('nws example')  
nwsStore(ws, 'x', 'Hello, world\n')  
nwsFetchFile(ws, 'x', 'hello.txt')  
## End(Not run)
```

nwsFetchTryFile	<i>netWorkspace Class Method</i>
-----------------	----------------------------------

Description

Fetch a value of a workspace variable and write it to a file.

Usage

```
## S4 method for signature 'netWorkspace':  
nwsFetchTryFile(.Object, xName, fObj)
```

Arguments

.Object	a netWorkspace object
xName	name of the variable to find
fObj	File to write data to

Details

The `nwsFetchTryFile` method tries to find a value in the variable specified by `'xName'`. If found, the value is removed from the variable, it writes the value to the file object, and the method returns `TRUE`. If it is not found, it simply returns `FALSE`.

See Also

`nwsFindTryFile`, `nwsFetchTry`

Examples

```
## Not run:
ws <- netWorkspace('nws example')
nwsStore(ws, 'x', 'Hello, world\n')
if (nwsFetchTryFile(ws, 'x', 'hello.txt')) {
  cat('success\n')
} else {
  cat('failure\n')
}
## End(Not run)
```

<code>nwsFindFile</code>	<i>netWorkspace Class Method</i>
--------------------------	----------------------------------

Description

Find a value of a workspace variable and write it to a file.

Usage

```
## S4 method for signature 'netWorkspace':
nwsFindFile(.Object, xName, fObj)
```

Arguments

<code>.Object</code>	a <code>netWorkspace</code> object
<code>xName</code>	name of the variable to find
<code>fObj</code>	File to write data to

Details

The nwsFindFile method blocks until a value in the variable specified by 'xName' is found. Once found, it writes the value to the file object, but the value is not removed from the variable (as in the case of nwsFetchFile).

See Also

nwsFetchFile, nwsFind

Examples

```
## Not run:  
ws <- netWorkspace('nws example')  
nwsStore(ws, 'x', 'Hello, world\n')  
nwsFindFile(ws, 'x', 'hello.txt')  
## End(Not run)
```

nwsFindTryFile	<i>netWorkspace Class Method</i>
----------------	----------------------------------

Description

Find a value of a workspace variable and write it to a file.

Usage

```
## S4 method for signature 'netWorkspace':  
nwsFindTryFile(.Object, xName, fObj)
```

Arguments

.Object	a netWorkspace object
xName	name of the variable to find
fObj	File to write data to

Details

The nwsFindTryFile method tries to find a value in the variable specified by 'xName'. If found, it writes the value to the file object, and the method returns TRUE. If it is not found, it simply returns FALSE.

See Also

nwsFetchTryFile, nwsFindTry

Examples

```
## Not run:
ws <- netWorkspace('nws example')
nwsStore(ws, 'x', 'Hello, world\n')
if (nwsFindTryFile(ws, 'x', 'hello.txt')) {
  cat('success\n')
} else {
  cat('failure\n')
}
## End(Not run)
```

nwsStoreFile	<i>netWorkspace Class Method</i>
--------------	----------------------------------

Description

Store a new value into a variable in the workspace from a file.

Usage

```
## S4 method for signature 'netWorkspace':
nwsStoreFile(.Object, xName, fObj, n=0)
```

Arguments

.Object	a netWorkspace class object
xName	name of the variable to be stored
fObj	file to read data from to store in the variable.
n	Number of bytes to write. A value of zero means to write all the data in the file.

Details

nwsStoreFile works like nwsStore, except that the value to store in the workspace variable comes from a file. If 'fObj' is a character string, nwsStoreFile calls 'file' to obtain a file connection which is opened for the duration of the method.

See Also

nwsStore

Examples

```
## Not run:  
ws <- netWorkspace('nws example')  
nwsStoreFile(ws, 'x', '/etc/printcap')  
## End(Not run)
```

nwsIFetch	<i>netWorkspace Class Method</i>
-----------	----------------------------------

Description

Create a function that acts as a destructive iterator over the values of the specified variable.

Usage

```
## S4 method for signature 'netWorkspace':  
nwsIFetch(.Object, xName)
```

Arguments

.Object	a netWorkspace class object
xName	name of the variable to be fetched

Details

The iterator function returned by the nwsIFetch method takes no arguments, and works just like the nwsFetch method, specified with the same arguments that were passed to nwsIFetch method. Note that the nwsIFind and nwsIFindTry methods are much more useful, since they provide the only way to iterate over values of a variable non-destructively. nwsIFetch and nwsIFetchTry are provided for completeness.

See Also

nwsFetch, nwsIFetchTry

Examples

```
## Not run:
ws <- netWorkspace('nws example')
nwsStore(ws, 'x', 10)
it <- nwsIFetch(ws, 'x')
it() # returns the value 10
it() # blocks until another process stores a value in the variable
## End(Not run)
```

nwsIFetchTry	<i>netWorkspace Class Method</i>
--------------	----------------------------------

Description

Create a function that acts as a destructive iterator over the values of the specified variable.

Usage

```
## S4 method for signature 'netWorkspace':
nwsIFetchTry(.Object, xName, defaultVal=NULL)
```

Arguments

.Object	a netWorkspace class object
xName	name of the variable to be fetched
defaultVal	value to return, if xName is not found in the netWorkspace

Details

The iterator function returned by the nwsIFetchTry method takes no arguments, and works just like the nwsFetchTry method, specified with the same arguments that were passed to nwsIFetchTry method. Note that the nwsIFind and nwsIFindTry methods are much more useful, since they provide the only way to iterate over values of a variable non-destructively. nwsIFetch and nwsIFetchTry are provided for completeness.

See Also

nwsFetchTry, nwsIFetch

Examples

```
## Not run:
ws <- netWorkspace('nws example')
nwsStore(ws, 'x', 10)
it <- nwsIFetchTry(ws, 'x', NA)
it() # returns the value 10
it() # returns NA
## End(Not run)
```

nwsIFind

netWorkspace Class Method

Description

Create a function that acts as a non-destructive iterator over the values of the specified variable.

Usage

```
## S4 method for signature 'netWorkspace':
nwsIFind(.Object, xName)
```

Arguments

.Object	a netWorkspace class object
xName	name of the variable to be fetched

Details

The iterator function returned by the nwsIFind method takes no arguments, and works somewhat like the nwsFind method, specified with the same arguments that were passed to nwsIFind. The difference is that the nwsFind method cannot iterate through the values of a variable; it always returns the same value until the variable is modified. The iterator function, however, maintains some state that allows it to see subsequent values. Each time the iterator function is called, it returns the next value in the variable. Once all values in the variable have been returned, the iterator function will block, waiting for a new value to be stored in the variable.

See Also

nwsFind, nwsIFindTry

Examples

```
## Not run:
ws <- netWorkspace('nws example')
nwsStore(ws, 'x', 1)
nwsStore(ws, 'x', 2)
it <- nwsIFind(ws, 'x')
it() # returns the value 1
it() # returns the value 2
it() # blocks until another process stores a value in the variable
## End(Not run)
```

nwsIFindTry	<i>netWorkspace Class Method</i>
-------------	----------------------------------

Description

Create a function that acts as a non-destructive iterator over the values of the specified variable.

Usage

```
## S4 method for signature 'netWorkspace':
nwsIFindTry(.Object, xName, defaultVal=NULL)
```

Arguments

<code>.Object</code>	a <code>netWorkspace</code> class object
<code>xName</code>	name of the variable to be fetched
<code>defaultVal</code>	value to return, if <code>xName</code> is not found in the <code>netWorkspace</code>

Details

The iterator function returned by the `nwsIFindTry` method takes no arguments, and works somewhat like the `nwsFindTry` method, specified with the same arguments that were passed to `nwsIFindTry`. The difference is that the `nwsFindTry` method cannot iterate through the values of a variable; it always returns the same value until the variable is modified. The iterator function, however, maintains some state that allows it to see subsequent values. Each time the iterator function is called, it returns the next value in the variable. Once all values in the variable have been returned, the iterator function will return `defaultVal`. However, when new values are stored into the variable, the iterator function will return them, picking right up where it left off.

See Also

nwsFindTry, nwsIFind

Examples

```
## Not run:
ws <- netWorkspace('nws example')
nwsStore(ws, 'x', 1)
nwsStore(ws, 'x', 2)
it <- nwsIFindTry(ws, 'x', NA)
it() # returns the value 1
it() # returns the value 2
it() # returns the value NA
nwsStore(ws, 'x', 3)
it() # returns the value 3
it() # returns the value NA
## End(Not run)
```

nwsClose

netWorkspace Class Method

Description

Close connection of a shared netWorkspace to the netWorkspace server.

Usage

```
## S4 method for signature 'netWorkspace':
nwsClose(.Object)
```

Arguments

.Object a netWorkspace class object

Examples

```
## Not run:
ws <- netWorkspace('nws example')
# do some works
# ...
nwsClose(ws)
## End(Not run)
```

nwsDeclare *netWorkspace Class Method*

Description

Declare a variable with particular mode in a shared netWorkspace.

Usage

```
nwsDeclare(.Object, xName, mode)
```

Arguments

<code>.Object</code>	a netWorkspace class object
<code>xName</code>	name of variable to be declared
<code>mode</code>	mode of the variable, see details

Details

If xName has not already been declared in the netWorkspace, the behavior of xName will be determined by mode. Mode can be 'fifo', 'lifo', 'multi', or 'single'. In the first three cases, multiple values can be associated with xName. When a value is retrieved for xName, the oldest value stored will be used in 'fifo' mode, the youngest in 'lifo' mode, and a nondeterministic choice will be made in 'multi' mode. In 'single' mode, only the most recent value is retained.

Examples

```
## Not run:
ws <- netWorkspace('nws example')
nwsDeclare(ws, 'pi', 'single')
nwsStore(ws, 'pi', 2.171828182)
nwsStore(ws, 'pi', 3.141592654)

# nwsListVars(ws) will show that only the most recent value of pi is retained
## End(Not run)
```

nwsDeleteVar	<i>netWorkspace Class Method</i>
--------------	----------------------------------

Description

Delete a variable from the shared netWorkspace.

Usage

```
## S4 method for signature 'netWorkspace':  
nwsDeleteVar(.Object, xName)
```

Arguments

.Object	a netWorkspace class object
xName	name of the variable to be deleted

Examples

```
## Not run:  
ws <- netWorkspace('nws example')  
nwsStore(ws, 'x', 10)  
nwsDeleteVar(ws, 'x')  
## End(Not run)
```

nwsListVars	<i>netWorkspace Class Method</i>
-------------	----------------------------------

Description

List variables in a netWorkspace.

Usage

```
## S4 method for signature 'netWorkspace':  
nwsListVars(.Object, wsName='', showDataFrame=FALSE)
```

Arguments

`.Object` a netWorkspace class object
`wsName` name of the netWorkspace
`showDataFrame`
 show result in data frame or string

Details

Return listing of variables in the netWorkspace with the name passed by wsName argument. If wsName is empty, then return variables in the netWorkspace represented by .Object.

The return values from nwsListVars can be represented in a string or a data frame. If showDataFrame is set to FALSE (default value), then the listing is returned in a string. To see list output clearly, use: write(nwsListVars(.Object), stdout()). If showDataFrame is set to TRUE, then the listing is returned in a data frame with these fields: Variables, NumValues, NumFetchers, NumFinders, and Mode.

Examples

```
## Not run:
# example 1
ws <- netWorkspace('nws example')
write(nwsListVars(ws), stdout())
## End(Not run)
```

nwsServerObject *netWorkspace Class Method*

Description

Return the nwsServer object associated with a netWorkspace.

Usage

```
## S4 method for signature 'netWorkspace':
nwsServerObject(.Object)
```

Arguments

`.Object` a netWorkspace class object

Examples

```
## Not run:
ws = netWorkspace('nws example')
nwsServerObject(ws)
## End(Not run)
```

nwsVariable	<i>Create an Active Binding for a NetWorkspace Variable</i>
-------------	---

Description

`nwsVariable` creates a variable in an R workspace that mirrors a variable in a `netWorkspace`. This allows standard R operations (`get`, `assign`) to be used to share data between R programs running on different machines.

Usage

```
## S4 method for signature 'netWorkspace':
nwsVariable(.Object, xName, mode=c('fifo', 'lifo', 'multi', 'single'), env=parent.frame)
```

Arguments

<code>.Object</code>	a <code>netWorkspace</code> class object.
<code>xName</code>	name of variable to be declared.
<code>mode</code>	mode of the variable, see details.
<code>env</code>	environment in which to define active binding.
<code>force</code>	logical; if TRUE, an existing binding will be overwritten.
<code>quietly</code>	logical; if TRUE, no warnings are issued.

Details

`nwsVariable` is built on top of the R `makeActiveBinding` function. It is experimental, but we have found that it is very useful for introducing people to the concept of `netWorkspace` variables. It's not clear that this API is ever preferable to `nwsStore`, `nwsFetch`, `nwsFind` for real programs, however.

The `mode` of the variable controls what happens when a variable is accessed. If the `mode` is `'single'`, then all accesses use the `nwsFind` operation. If the `mode` is `'fifo'`, `'lifo'`, or `'multi'`, then all accesses use the `nwsFetch` operation. Assigning a value to an `nwsVariable` always uses the `nwsStore` operation.

Examples

```
## Not run:
# create a netWorkspace
ws = netWorkspace('nws example')

# create a variable in the local R workspace that is linked to
# a netWorkspace variable
nwsVariable(ws, 'x', 'single')

x <- 0
x <- 999 # overwrites the 0
x <- 3.14159 # overwrites the 999
x # returns 3.14159
x # returns 3.14159
x # returns 3.14159

# create a 'fifo' mode variable
nwsVariable(ws, 'message', 'fifo')

message <- 1
message <- 2
message <- 3
message # returns 1
message # returns 2
message # returns 3
## End(Not run)
```

nwsWsName	<i>netWorkspace Class Method</i>
-----------	----------------------------------

Description

Return name of a netWorkspace.

Usage

```
## S4 method for signature 'netWorkspace':
nwsWsName(.Object)
```

Arguments

.Object a netWorkspace class object

Examples

```
## Not run:  
ws = netWorkspace('nws example')  
nwsWsName(ws)  
## End(Not run)
```

4.2 nwsServer class

<code>nwsServer</code>	<i>Class "nwsServer"</i>
------------------------	--------------------------

Description

Class representing nwsServer.

Objects from the Class

Objects can be created by calls of the form `new("nwsServer", serverHost, port)`.

`serverHost`: server host name. Default value is local machine.

`port`: server port number. Default value is 8765.

Slots

`nwsSocket`: Object of class "ANY" representation of the socket connection to the server.

`port`: Object of class "numeric" representation of the server port number.

`serverHost`: Object of class "character" representation of the server host name.

Methods

`initialize` signature(`.Object = "nwsServer"`): nwsServer class constructor.

`nwsDeleteWs` signature(`.Object = "nwsServer"`): delete a netWorkspace from the server.

`nwsListWss` signature(`.Object = "nwsServer"`): list all netWorkSpaces in the server.

`nwsMktempWs` signature(`.Object = "nwsServer"`): create a unique temporary workspace using the default or specified template.

`nwsOpenWs` signature(`.Object = "nwsServer"`): create and owned a netWorkSpace.

`nwsUseWs` signature(`.Object = "nwsServer"`): connect to a netWorkspace but does not claim ownership.

Examples

```
## Not run:
# example 1
nwss = nwsServer()
# Or,
nwss = new("nwsServer")

# example 2
nwss = nwsServer(serverHost="node1", port=5555)
## End(Not run)
```

nwsDeleteWs	<i>nwsServer Class Method</i>
-------------	-------------------------------

Description

Delete a shared netWorkspace from the netWorkSpaces server.

Usage

```
## S4 method for signature 'nwsServer':
nwsDeleteWs(.Object, wsName)
```

Arguments

.Object	a nwsServer class object
wsName	name of the netWorkspace to be deleted

Examples

```
## Not run:
# example 1
nwss <- nwsServer()
ws <- nwsOpenWs(nwss, "nws example")
# do some works
# ...
nwsDeleteWs(nwss, "nws example")

# example 2 illustrates accessing a server object
# from the netWorkspace class object
ws <- netWorkspace("nws example 2")
# do some works
# ...
nwsDeleteWs(ws@server, "nws example 2")
## End(Not run)
```

nwsListWss	<i>nwsServer Class Method</i>
------------	-------------------------------

Description

List all netWorkSpaces in the netWorkSpaces server.

Usage

```
## S4 method for signature 'nwsServer':  
nwsListWss(.Object, showDataFrame=FALSE)
```

Arguments

<code>.Object</code>	a nwsServer class object
<code>showDataFrame</code>	show result in data frame or string

Details

By default, `showDataFrame` is set to `FALSE`, which means the return value is a text string containing a list of workspaces in the netWorkSpaces server. To see list output clearly, use: `write(nwsListWss(.Object), stdout())`. If `showDataFrame` is set to `TRUE`, then return value is a data frame with these fields: Owned, Name, Owner, Persistent, NumVariables, and Variables.

Examples

```
## Not run:  
# example 1  
nwss <- nwsServer()  
ws1 <- nwsOpenWs(nwss, 'my space')  
ws2 <- nwsOpenWs(nwss, 'other space')  
write(nwsListWss(nwss), stdout())  
  
# example 2  
# retrieve all workspace names  
df <- nwsListWss(nwss), showDataFrame=TRUE)  
df$Name  
$"1"  
[1] "__default"  
  
$"2"  
[1] "my space"
```

```
 $"3"  
 [1] "other space"  
 ## End(Not run)
```

nwsMktempWs-methods

nwsServer class method

Description

Create a unique temporary netWorkspace using the template string.

Usage

```
## S4 method for signature 'nwsServer':  
nwsMktempWs(.Object, wsNameTemplate)
```

Arguments

`.Object` a nwsServer class object
`wsNameTemplate` template for the netWorkspace name

Details

`nwsMktempWs(nwss, wsNameTemplate)` returns the name of a temporary space created on the netWorkSpaces server. The template should contain a '%d'-like construct which will be replaced by a serial counter maintained by the server to generate a unique new netWorkspace name. The user must then invoke `nwsOpenWs()` or `nwsUseWs()` with this name to create an object to access this workspace. `WsNameTemplate` defaults to `'_Rws_%010d'`

Examples

```
## Not run:  
s <- nwsServer()  
tempWsName <- nwsMktempWs(s, 'temp_%d')  
ws <- nwsOpenWs(s, tempWsName)  
## End(Not run)
```

nwsOpenWs	<i>nwsServer Class Method</i>
-----------	-------------------------------

Description

Create and owned a netWorkspace, if it does not already exist. If the netWorkspace already exists, but no one owns it, then caller will claim the ownership of the netWorkspace. If the netWorkspace already exists and someone already claimed the ownership of it, then caller simply make connection to the netWorkspace.

Usage

```
## S4 method for signature 'nwsServer':
nwsOpenWs(.Object, wsName, space=NULL, ...)
```

Arguments

.Object	a nwsServer class object
wsName	name of the netWorkspace to be created
space	a netWorkspace class object
...	optional arguments related to the persistent state of a netWorkspace

Details

If space argument is not provided, then create a shared netWorkspace with the name wsName to the server represented by .Object. Otherwise, use the existing netWorkspace object provided by argument space.

Optional argument, 'persistent', is a boolean value indicating whether the shared netWorkspace is persistent or not. If the netWorkspace is persistent (TRUE or 1), then the netWorkspace is not purged when the owner of the shared netWorkspace exited. Otherwise, the netWorkspace will be purged.

See Also

nwsUseWs

Examples

```
## Not run:
# example 1
nwss <- nwsServer()
ws <- nwsOpenWs(nwss, "nws example")

# example 2
xs <- nwsOpenWs(nwss, wsName='nws example', space=ws)

# example 3
ys <- nwsOpenWs(nwss, "persistent space", persistent=TRUE)
## End(Not run)
```

nwsUseWs	<i>nwsServer Class Method</i>
----------	-------------------------------

Description

Connect to a netWorkspace but does not claim ownership. If the netWorkspace does not exist, then it will be created, but no ownership will be claimed.

Usage

```
## S4 method for signature 'nwsServer':
nwsUseWs(.Object, wsName, space=NULL)
```

Arguments

.Object	a nwsServer class object
wsName	name of the netWorkspace to open
space	a netWorkspace class object

See Also

nwsOpenWs

Examples

```
## Not run:
nwss <- nwsServer()
ws <- nwsUseWs(nwss, "nws example")
## End(Not run)
```

close	<i>nwsServer, sleigh Class Method</i>
-------	---------------------------------------

Description

Depends on the first argument type passed to the close function, it behaves differently. If the first argument is a sleigh object, then close function calls stopSleigh to shutdown sleigh workers and delete sleigh workspace. If the first argument is a nwsServer object, then the connection to the netWorkSpace server is closed.

Usage

```
close(con, ...)
```

Arguments

con	a sleigh class object or a nwsServer class object
...	optional fields

Details

The optional fields are not passed to stopSleigh method. They are defined to be compatible with the default, non-generic close method.

Examples

```
## Not run:  
s = sleigh()  
close(s)  
  
wss = nwsServer()  
close(wss)  
## End(Not run)
```

4.3 sleigh class

sleigh	<i>Class "sleigh"</i>
--------	-----------------------

Description

Class representing sleigh.

Objects from the Class

Objects can be created by calls of the form `new("sleigh", ...)`, where ‘...’ can be one or more of the following named arguments.

`nodeList`: a list of hosts that workers will be created. This argument is irrelevant when `launch` equals ‘local’. Default is to start up three workers on local machine.

`workerCount`: number of workers that will be created. This argument is only relevant when `launch` equals ‘local’. Default is three workers.

`launch`: method to launch remote workers. Default is ‘local’.

`nwsHost`: host name of the `netWorkSpaces` server. Default is the machine where sleigh starts up.

`nwsPort`: port number of the `netWorkSpaces` server. Default is 8765.

`scriptExec`: command to execute worker script. Default uses `scriptcmd` function on Windows, and uses `envcmd` function on other platforms.

`scriptDir`: location of the sleigh worker script. Default is the `bin` directory under where `nws` library is installed on the system. If library cannot be found, then use current working directory.

`scriptName`: worker script file name. Default is `RNWSSleighWorker.py` on Windows, and `RNWSSleighWorker.sh` on other platforms.

`workingDir`: worker’s working directory. Default to master’s current working directory.

`logDir`: location where log files will be stored. Default is `NULL`.

`outfile`: remote workers’ standard errors will be redirected to this file. Default is `NULL`.

`wsNameTemplate`: template name to create sleigh workspace. Default is ‘`sleigh_ride_%010d`’.

user: user name. Default is the value returned from `Sys.info()['user']`.

verbose: a boolean value indicating whether to print out debug messages. Default is `FALSE`.

Slots

nodeList: Object of class `"character"` representation of a list of host names where workers are created.

nws: Object of class `"netWorkspace"` representation of the sleigh workspace.

nwsName: Object of class `"character"` representation of the sleigh workspace name.

nwss: Object of class `"nwsServer"` representation of the `netWorkSpaces` server that this sleigh workspace connects to.

state: Object of class `"environment"` representation of the sleigh state.

Methods

initialize signature(`.Object = "sleigh"`): sleigh class constructor.

eachElem signature(`.Object = "sleigh"`): evaluate the given function with multiple argument sets using the workers in sleigh.

eachWorker signature(`.Object = "sleigh"`): evaluate the given function exactly once for each worker in sleigh.

rankCount signature(`.Object = "sleigh"`): get sleigh's `rankCount`.

status signature(`.Object = "sleigh"`): return the status of the sleigh.

stopSleigh signature(`.Object = "sleigh"`): shutdown workers and remove sleigh workspace.

workerCount signature(`.Object = "sleigh"`): get number of workers started in sleigh.

Details

There are five different launch types (`'local'`, `sshcmd`, `rshcmd`, `lsfcmd`, and `'web'`) to tailor client's working environment. This is done by setting launch variable to a function (`sshcmd`, `rshcmd`, or `lsfcmd`) or a string (`'local'` and `'web'`). see examples section.

Examples

```
## Not run:
# Default option: create three sleigh workers on local host
s <- sleigh()
# which is equivalent to:
s <- sleigh(launch='local')
```

```

# Create sleigh workers on multiple machines using SSH
s <- sleigh(c('n1', 'n2', 'n3'), launch=sshcmd)

# Use LSF to remote login to remote machines.
s <- sleigh(launch=lsfcmd)

# Use web launch
s <- sleigh(launch='web')

## End(Not run)

```

eachElem

Apply a Function in Parallel over a Set of Lists and Vectors

Description

`eachElem` executes function `fun` multiple times in parallel with a varying set of arguments, and returns the results in a list. It is functionally similar to the standard R `lapply` function, but is more flexible in the way that the function arguments can be specified.

Usage

```

## S4 method for signature 'sleigh':
eachElem(.Object, fun, elementArgs=list(), fixedArgs=list(), eo=NULL, DEBUG=FALSE)

```

Arguments

<code>.Object</code>	sleigh class object.
<code>fun</code>	the function to be evaluated by the sleigh. In the case of functions like <code>+</code> , <code>%*%</code> , etc., the function name must be quoted.
<code>elementArgs</code>	list of vectors, lists, matrices, and data frames that specify (some of) the arguments to be passed to <code>fun</code> .
<code>fixedArgs</code>	list of additional arguments to be passed to <code>fun</code> .
<code>eo</code>	list specifying special options.
<code>DEBUG</code>	logical; should browser function be called upon entry to <code>eachElem</code> ?

Details

`eachElem` forms argument sets from objects passed in via `elementArgs` and `fixedArgs`. Both `elementArgs` and `fixedArgs` should be lists, with each element of these lists corresponding to an argument of `fun`. The elements of `elementArgs` are used to specify the arguments that are changing, or varying, from task to task, while the elements of `fixedArgs` are used to specify the arguments that do not vary from task to task. The number of tasks that are executed by a call to `eachElem` is basically equal to the length of the longest vector (or list, etc) in `elementArgs`. If any elements of `elementArgs` are shorter, then their values are recycled, using the standard R rules.

The elements of `elementArgs` may be vectors, lists, matrices, or data frames. The vectors and lists are always iterated over by element, or "cell", but matrices and data frames can also be iterated over by row or column. This is controlled by the `by` option, specified via the `eo` argument. See below for more information.

For example:

```
eachElem(s, '+', elementArgs=list(1:4), fixedArgs=list(100))
```

This will submit four tasks, since the length of `1:4` is four. The four tasks will be to add the arguments 1 and 100, 2 and 100, 3 and 100, and 4 and 100. The result is a list containing the four values 101, 102, 103, and 104.

Another way to do the same thing is with:

```
eachElem(s, '+', elementArgs=list(1:4, 100))
```

Since the second element of `elementArgs` is length one, its value is recycled four times, thus specifying the same set of tasks as in the previous example. This method also has the advantage of making it easy to put fixed values before varying values, without the need for the `eo$argPermute` option, discussed later. For example:

```
eachElem(s, '-', elementArgs=list(100, 1:4))
```

is similar to the R statement:

```
100 - 1:4
```

Note that in simple examples like these, where the results are numeric values, the standard R `unlist` function can be very useful for converting the resulting list into a vector.

The `eo` argument is a list that can be used to specify various options. The current options are: `eo$elementFunc`, `eo$accumulator`, `eo$by`, `eo$chunkSize`, `eo$loadFactor`, `eo$blocking`, and `eo$argPermute`.

The `eo$elementFunc` option can be used to specify a callback function that provides the varying arguments for `fun` in place of `elementArgs` (ie. you can't specify both `eo$elementFunc` and `elementArgs`). `eachElem` calls the `eo$elementFunc` function to get a list of arguments for one invocation of `fun`, and will keep calling it until `eo$elementFunc` signals that there are no more tasks to execute by calling the `stop` function with no arguments. `eachElem` appends any values specified by `fixedArgs` to the list returned by `eo$elementFunc` just as if `elementArgs` had been specified.

`eachElem` passes the number of the desired task (starting from 1) as the first argument to `eo$elementFunc`, and the value of the `eo$by` option as the second argument. Note that the use of the `eo$elementFunc` function is an advanced feature, but is very useful when executing a large number of tasks, or when the arguments are coming from a database query, for example. For that reason, the `eo$loadFactor` option should usually be used in conjunction with `eo$elementFunc` (see description below).

The `eo$accumulator` option can be used to specify a callback function that will receive the results of the task execution as soon as they are complete, rather than returning all of the task results as a list when `eachElem` completes. In other words, `eachElem` will call the `eo$accumulator` function with task results as soon as it receives them from the sleigh workers, rather than saving them in memory until all the tasks are complete. Note that if the tasks are "chunked" (using the `eo$chunkSize` option described below), then the `eo$accumulator` function will receive multiple task results, which is why the task results are always passed to the `eo$accumulator` function in a list.

The first argument to the `eo$accumulator` function is a list of results, where the length of the list is equal to `eo$chunkSize`. The second argument is a vector of task numbers, starting from 1, where the length of the vector is also equal to `eo$chunkSize`. The task numbers are very important, because the results are not guaranteed to be returned in order. `eo$accumulator` is another advanced feature, and like `eo$elementFunc`, is very useful when executing a large number of tasks. It allows you to process each result as they finish, rather than forcing you to wait until all of the tasks are complete. In conjunction with `eo$elementFunc` and `eo$loadFactor`, you can set up a pipeline, allowing you to process an unlimited number of tasks efficiently. Note that when `eo$accumulator` is specified, `eachElem` returns `NULL`, not the list of results, since `eachElem` doesn't save any of the results after passing them to the `eo$accumulator` function.

The `eo$by` option specifies the iteration scheme to use for matrix and data frame elements in `elementArgs`. The default value is "row", but it can also be set to "column" or "cell". Vectors and lists in `elementArgs` are not effected by this option.

The `eo$chunkSize` option is a tuning parameter that specifies the number of tasks that sleigh workers should allocate at a time. The default value is 1, but if the tasks are small, performance can be improved by specifying a larger value, which decreases the overhead per task.

The `eo$loadFactor` option is a tuning parameter that specifies the maximum number of tasks per worker that are submitted to the sleigh at the same time. If set, no more than $(\text{loadFactor} * \text{workerCount})$ tasks will be submitted at the same time. This helps to control the resource demands that are made on the NetWorkSpaces server, which is especially important if there are a large number of tasks. Note that this option is ignored if `blocking` is set to `TRUE`, since the two options are incompatible with each other.

The `eo$blocking` option is used to indicate whether to wait for the results, or to return as soon as the tasks have been submitted. If set to `FALSE`, `eachElem` will return a `sleighPending` object that is used to monitor the status of the tasks, and to eventually retrieve the results. You must wait for the results to be complete before executing any further tasks on the sleigh, or an exception will be raised. The default value is `TRUE`.

The `eo$argPermute` option is used to reorder the arguments passed to `fun`. It is generally only useful if the `fixedArgs` argument has been specified, and some of those arguments need to precede the arguments specified via `elementArgs`. Note that by using recycling of elements in `elementArgs`, the use of `fixedArgs` and `argPermute` can often be avoided entirely.

The `DEBUG` argument is used call the `browser` function upon entering `eachElem`. The default value is `FALSE`.

Note

If `elementArgs` or `fixedArgs` isn't a list, `eachElem` will automatically wrap it in a list. This is a convenience that only works for passing in a single vector and matrix, however.

If `elementArgs` or `fixedArgs` are named lists, then the names are used to map the values to the appropriate argument of `fun`. This can be used as another technique to avoid the use of `eo$argPermute`.

The `elementArgs` argument can be specified as a data frame. This works just like a named list, and therefore, the column names of the data frame must all correspond to arguments of `fun`. Note that if the data frame has many rows, the performance may not be good due to the overhead of subsetting data frames in R.

If the `fun` function executes very quickly, you may not be able to keep your workers busy, giving you poor performance. In that case, consider setting the `eo$chunkSize` option to a large enough number to increase the effective task execution time.

If you have a huge number of tasks, consider using the `eo$elementFunc`, `eo$accumulator`, and `eo$loadFactor` options.

If in doubt, set the `eo$loadFactor` option to 10. That will almost certainly avoid putting a big on the NetWorkSpaces server, and if that isn't enough to keep your

workers busy, then you should be really be using the `eo$chunkSize` option to give the workers more to do.

If `eo$elementFunc` returns a value that isn't a list, `eachElem` will automatically wrap that value in a list.

The `eo$elementFunc` function doesn't have to define a second formal argument (the `by` argument) if it's not needed.

The `eo$accumulator` function doesn't have to define a second formal argument (the `taskVector` argument) if it's not needed. Just remember that the results are not guaranteed to come back in order.

See Also

`eachWorker`, `sleighPending`

Examples

```
## Not run:
# create a sleigh
s <- sleigh()

# compute the list mean for each list element
x <- list(a=1:10, beta=exp(-3:3), logic=c(TRUE,FALSE,FALSE,TRUE))
eachElem(s, mean, list(x))

# median and quartiles for each list element
eachElem(s, quantile, elementArgs=list(x), fixedArgs=list(probs=1:3/4))

# use eo$elementFunc to supply 100 random values and eo$accumulator to
# receive the results
elementFunc <- function(i, by) {
  if (i <= 100) list(i=i, x=runif(1)) else stop()
}
accumulator <- function(resultList, taskVector) {
  if (resultList[[1]][[1]] != taskVector[1]) stop('assertion failure')
  cat(paste(resultList[[1]], collapse=' '), '\n')
}
eo <- list(elementFunc=elementFunc, accumulator=accumulator)
eachElem(s, function(i, x) list(i=i, x=x, xsq=x*x), eo=eo)
## End(Not run)
```

<code>eachWorker</code>	<i>Execute a Function in Parallel on all Workers of a Sleigh</i>
-------------------------	--

Description

`eachWorker` executes function `fun` once on each worker in the specified sleigh, and returns the results in a list. This can be useful for initializing each of the workers before starting to execute tasks using `eachElem`. Loading packages using the `library` function, loading data sets using the `data` function, and assigning variables in the global environment are common tasks for `eachWorker`.

Usage

```
## S4 method for signature 'sleigh':
eachWorker(.Object, fun, ..., eo=NULL, DEBUG=FALSE)
```

Arguments

<code>.Object</code>	sleigh class object.
<code>fun</code>	the function to be evaluated by each of the sleigh workers. In the case of functions like <code>+</code> , <code>%*%</code> , etc., the function name must be quoted.
<code>...</code>	optional arguments to <code>fun</code> .
<code>eo</code>	additional options, see details
<code>DEBUG</code>	logical; should <code>browser</code> function be called upon entry to <code>eachWorker</code> ?

Details

The `eo` argument is a list that can be used to specify various options. The current options are: `eo$blocking`, and `eo$accumulator`.

The `eo$blocking` option is used to indicate whether to wait for the results, or to return as soon as the tasks have been submitted. If set to `FALSE`, `eachWorker` will return a `sleighPending` object that is used to monitor the status of the tasks, and to eventually retrieve the results. You must wait for the results to be complete before executing any further tasks on the sleigh, or an exception will be raised. The default value is `TRUE`.

The `eo$accumulator` option can be used to specify a function that will receive the results of the task execution. Note that while this can be a very useful feature with `eachElem`, it's not commonly used with `eachWorker`, but is included for consistency. The first argument to `eo$accumulator` function is a list of results, where the length of the list is always equal to 1 (because there is no `eo$chunkSize` option in `eachWorker`).

The second argument is a vector of task numbers, starting from 1, where the length of the vector is also always equal to 1. The task numbers are not very important when used with `eachWorker`, because the order of tasks isn't specified, as it is with `eachElem`. Note that when `eo$accumulator` is specified, `eachWorker` returns `NULL`, not the list of results, since `eachWorker` doesn't save any of the results after passing them to the `eo$accumulator` function.

The `DEBUG` argument is used call the `browser` function upon entering `eachWorker`. The default value is `FALSE`.

Note

The `eo$blocking` option can be very useful for starting a function on each of the workers, and then allowing the master process to interact with the workers via `Net-Workspace` operations in order to implement sophisticated parallel applications.

See Also

`eachElem`, `sleighPending`

Examples

```
## Not run:
# create a sleigh
s <- sleigh()

# assign to global variable x on each worker
eachWorker(s, function() x <<- 1)

# get a listing of each worker's global environment
eachWorker(s, function() ls(globalenv()))

# get system info from each worker
eachWorker(s, Sys.info)

# load MASS package on each worker
eachWorker(s, function() library(MASS))

# non-blocking example using simple NWS operations
sp <- eachWorker(s, function() nwsFind(SleighNws, 'hello'), eo=list(blocking=FALSE))
nwsStore(s@nws, 'hello', 'world')
waitSleigh(sp)
## End(Not run)
```

rankCount	<i>sleigh Class Method</i>
-----------	----------------------------

Description

Return rankCount in sleigh. If rankCount equals -1, then the worker group has closed, and no more workers may join the group.

Usage

```
## S4 method for signature 'sleigh':  
rankCount(.Object)
```

Arguments

.Object a sleigh class object

Examples

```
## Not run:  
# simple hello world  
s = sleigh()  
rankCount(s)  
## End(Not run)
```

status	<i>sleigh Class Method</i>
--------	----------------------------

Description

Check status of sleigh workers and return a list that contains two values: numWorkers and closed. If status is zero, then some workers failed to start. If status is one, then either all workers have started or closeGroup is set to TRUE. numWorkers indicates the number of workers started.

Usage

```
## S4 method for signature 'sleigh':  
status(.Object, closeGroup=FALSE, timeout=0)
```

Arguments

`.Object` a sleigh class object
`closeGroup` a boolean value indicating whether to close the worker group
`timeout` timeout value measure in seconds

Details

`closeGroup` is set to `FALSE` by default. If `closeGroup` is set to `FALSE`, then all workers must wait for each other to start before they can start working on tasks. If `closeGroup` is set to `TRUE`, no new workers may join the group after `timeout` value has expired. Once the group is closed, all launched workers can start working on tasks.

`timeout` indicates how long to wait and check on the status of workers.

Examples

```
## Not run:  
# example 1  
# one available machine and one non-existent machine  
s <- sleigh(c('localhost', 'noname'))  
slist <- status(s)  
# slist$numWorkers = the number of worker started  
# slist$closed = whether the worker group is closed or not.  
  
# example 2  
# check the status of worker group after 20 seconds  
slist <- status(s, timeout=20)  
  
# example 3  
# close the group after 10 seconds, regardless of whether  
# all workers have started up successfully.  
slist <- status(s, closeGroup=TRUE, timeout=10)  
## End(Not run)
```

stopSleigh

sleigh Class Method

Description

Shutdown sleigh workers and delete sleigh workspace.

Usage

```
## S4 method for signature 'sleigh':  
stopSleigh(.Object)
```

Arguments

`.Object` a sleigh class object

Details

This function is called by `close` method.

See Also

`close`

Examples

```
## Not run:  
s = sleigh()  
stopSleigh(s)  
## End(Not run)
```

`workerCount`

sleigh Class Method

Description

Return number of Sleigh workers that have successfully started.

Usage

```
## S4 method for signature 'sleigh':  
workerCount(.Object)
```

Arguments

`.Object` a sleigh class object

Examples

```
## Not run:
# simple hello world
s = sleigh()
workerCount(s)
## End(Not run)
```

cmdLaunch

sleigh Auxiliary Function

Description

This assumes the sleigh object on the master uses a worker script to launch remote workers. This function is intended to be called within the worker script, which automates the process of launching remote workers. See RNWSSleighWorker.sh and RNWSleighWorker.py for example.

defaultSleighOptions

Default Sleigh Options Environment

Description

This environment specifies the default options that are used when creating a new sleigh object.

Details

The default options can be modified by assigning new values to this environment. Those new values will be used when constructing all new sleigh objects. You can also define new entries, which can be used from custom launch functions.

Note that the defaultSleighOptions environment is used to check for illegal options to the sleigh initializer. Therefore, to allow new options to be passed to a custom launch function, you must first define a default value for that option in defaultSleighOptions.

See Also

sleigh

Examples

```
defaultSleighOptions$user
```

envcmd	<i>Sleigh Auxiliary Function</i>
--------	----------------------------------

Description

This function is set through optional argument 'scriptExec' in the sleigh constructor.

Details

This function indicates that configuration variables such as RSleighName and RSleighNwsName are exported to the environment using env command.

envcmd is the default choice to execute worker scripts on UNIX.

Examples

```
## Not run:
s = sleigh(scriptExec=envcmd)
## End(Not run)
```

isClosure	<i>isClosure Function</i>
-----------	---------------------------

Description

This is a heuristic function that is used by eachWorker and eachElem to guess if the worker function is a closure.

Details

If the "closure" option wasn't specified via the "eo" argument to eachWorker or eachElem, then this function is used to guess if the worker function is a closure. It can be very useful to use a closures with eachWorker and eachElem, but if not used properly, you could accidentally include a lot of unnecessary data in the tasks, thus hurting your performance.

This function is included as a development tool so you can manually test your worker functions. This could be useful if you are getting a warning from eachWorker or eachElem, and are trying to determine how to modify the function or set the "closure" option.

Examples

```
# this should return FALSE
isClosure(sqrt)
f <- function(x) function(y) x + y
g <- f(1)
# this should return TRUE
isClosure(g)
```

launch	<i>Sleigh Auxiliary Function</i>
--------	----------------------------------

Description

Setup appropriate worker accounting information, and start the worker loop.

Details

The only time clients ever need to invoke this function directly is when web launch is used. The 'runMe' variable contains value, which is to execute launch function. For other launch types such as SSH and local launch, launch function is invoked within cmdLaunch.

See Also

cmdLaunch

lsfcmd	<i>Sleigh Auxiliary Function</i>
--------	----------------------------------

Description

This function indicates that LSF will be used to launch sleigh workers.

Details

This function is only meant to be passed through optional argument, 'launch', in the sleigh constructor.

See Also

sleigh

Examples

```
## Not run:  
s = sleigh(launch=lsfcmd)  
## End(Not run)
```

lsfSleigh	<i>sleigh Auxiliary Function</i>
-----------	----------------------------------

Description

This function will properly check to see if the R script has been launched in a parallel fashion. If so, it will start jobs on the provided nodes using SSH. If not, it expects that the parameter 'N' contains the number of nodes to start via 'bsub'. Extra arguments to bsub can be provided using the lsfoptions parameter.

rshcmd	<i>Sleigh Auxiliary Function</i>
--------	----------------------------------

Description

This function indicates that RSH will be used to launch sleigh workers.

Details

This function is only meant to be passed through optional argument 'launch' in the sleigh constructor.

See Also

sleigh

Examples

```
## Not run:  
s = sleigh(launch=rshcmd)  
## End(Not run)
```

scriptcmd	<i>Sleigh Auxiliary Function</i>
-----------	----------------------------------

Description

This function is set through optional argument 'scriptExec' in the sleigh constructor. It uses Python to execute worker script.

Details

This function indicates that configuration variables such as RSleighName and RSleighNwsName are passed as part of the worker script arguments rather than through env command. This is often useful on systems that do not support env command.

scriptcmd is the default choice to execute worker scripts on Windows.

Examples

```
## Not run:  
s = sleigh(scriptExec=scriptcmd)  
## End(Not run)
```

sshcmd	<i>sleigh Auxiliary Function</i>
--------	----------------------------------

Description

This function indicates that SSH will be used to launch sleigh workers.

Details

This function is only meant to be passed through optional argument, 'launch' in the sleigh constructor.

See Also

sleigh

Examples

```
## Not run:
# Both master and sleigh workers run on Linux.
s = sleigh(launch=sshcmd)

# master runs on Windows and sleigh workers run on Linux.
# Since Windows and Linux have different file structure,
# we need to specify the location of worker scripts.
s = sleigh(launch=sshcmd, scriptDir='/usr/local/lib/R/library/nws/bin')
## End(Not run)
```

workerLoop

sleigh Auxiliary Function

Description

Worker waits and listens for tasks in the sleigh workspace. Worker then works on the retrieved tasks and return results back to the sleigh workspace. This loop ends when stopSleigh is invoked.

Details

This function is invoked by launch function. It is not meant to be called directly by clients.

See Also

launch

4.4 sleighPending class

sleighPending *Class "sleighPending"*

Description

Class representing sleighPending.

Details

This class object is usually obtained from non-blocking eachElem or non-blocking eachWorker.

Objects from the Class

Objects can be created by calls of the form `new("sleighPending", nws, numTasks, bn, ss)`.

nws: netWorkspace class object.

numTasks: number of submitted tasks.

bn: barrier names.

ss: sleigh state.

Slots

nws: Object of class "netWorkspace" representation of the netWorkspace class.

numTasks: Object of class "numeric" representation of the number of pending tasks in sleigh.

barrierName: Object of class "character" representation of the barrier name.

sleighState: Object of class "environment" representation of the sleigh state.

state: Object of class "environment" representation of the sleighPending state.

Methods

initialize signature(.Object = "sleighPending"): sleighPending class constructor.

checkSleigh signature(.Object = "sleighPending"): return the number of results yet to be generated for the pending sleigh job.

waitSleigh signature(.Object = "sleighPending"): wait and block for the results to be generated for the pending sleigh job.

See Also

eachWorker, eachElem

checkSleigh *sleighPending Class Method*

Description

Return the number of results yet to be generated for the pending sleigh job, .Object.

Usage

```
## S4 method for signature 'sleighPending':  
checkSleigh(.Object)
```

Arguments

.Object a sleighPending class object

Details

The sleighPending class object, .Object, is usually obtained through non-blocking eachElem or non-blocking eachWorker. If the pending job is finished, i.e. all results are generated, then 0 is returned.

See Also

eachWorker, eachElem

Examples

```
## Not run:  
eo = list(blocking=0)  
s = sleigh()  
sp = eachElem(s, function(x) {Sys.sleep(100); x}, list(1:10), eo=eo)  
checkSleigh(sp)  
## End(Not run)
```

waitSleigh	<i>sleighPending Class Method</i>
------------	-----------------------------------

Description

Wait and block for the results to be generated for the pending sleigh job, .Object. If the job has completed, then waitSleigh returns immediately with the generated results.

Usage

```
## S4 method for signature 'sleighPending':  
waitSleigh(.Object)
```

Arguments

.Object a sleighPending class object

Details

The sleighPending class object, .Object, is usually obtained from the return value of non-blocking eachElem or non-blocking eachWorker.

See Also

eachWorker, eachElem

Examples

```
## Not run:  
eo = list(blocking=0)  
sp = eachWorker(s, function() {Sys.sleep(100)}, eo=eo)  
waitSleigh(sp) # wait on workers. Each worker sleeps for 100 seconds  
## End(Not run)
```