

SCAI

NetWorkSpaces[®] for R

User Guide

SCIENTIFIC
COMPUTING ASSOCIATES
INC.

One Century Tower
265 Church Street, New Haven, CT 06510-7010
Tel: (203) 777-7442 • Fax (203) 776-4074

www.LindaSpaces.com

Acknowledgements

D. Atapattu, N. Carriero, J. Lai, S. Weston
Scientific Computing Associates, Inc.

and

G. Warnes
Pfizer

© Copyright 2005 Scientific Computing Associates, Inc. All rights reserved. SCAI, NetWorkSpaces, and Sleigh are trademarks of Scientific Computing Associates, Inc. All other trademarks and registered trademarks are property of their respective holders.

Manual version 1.0 December 2005. Printed in the USA.

Contents

Introduction	1
Getting Started	5
Example Program	5
Required Software	5
Setup	5
Tutorials	7
R NetWorkSpaces Tutorial.....	7
R Sleigh Tutorial.....	10
Appendix	13
NetWorkSpaces for R Vignette	13
checkSleigh	13
eachElem	13
eachWorker	16
netWorkSpace	17
nwsClose.....	18
nwsDeclare	18
nwsDeleteVar.....	19
nwsDeleteWs	19
nwsFetch.....	20
nwsFetchTry	21
nwsFind	22
nwsFindTry	22
nwsListVars	23
nwsListWss.....	23
nwsMktempWs	24
nwsOpenWs	24
nwsServer.....	25
nwsStore	25
nwsUseWs	26
nwsWsName	26
sleighPending	27
sleigh.....	27
stopSleigh.....	29
waitSleigh.....	29

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.



2 Getting Started

Example Program

Required Software

Python - <http://www.python.org/>
twisted - <http://twistedmatrix.com/>
R - <http://www.r-project.org/>
ssh - <http://www.openssh.com/>

Setup

Start `nws` using the command:

```
% nws start
```

Example

The simplest way to get started is to look at a short example. The complete source code for a sample program is shown below.

```
sum.R
```

```
-----  
1 library(nws)  
2  
3 nws = new('netWorkspace', 'sum test')  
4  
5 count = 10  
6 cat('sum to:', count, '\n')  
7 nwsStore(nws, 'summingTo', count)  
8 j = 0  
9 for (i in 1:count) {  
10     nwsStore(nws, 'i', i)  
11     j = j + nwsFetch(nws, 'i')  
12 }  
13 nwsStore(nws, 'sum', j)  
-----
```

A detailed explanation of the code is shown below.

```
1 library(nws)
```

In line 1, we set load the **NWS** system.

```
3 nws = new('netWorkspace', 'sum test')
```

In line 3, we create a NetWorkspace. It has the name `sum test`; any other R program that knows this name can access it. A default name applies if you don't specify a name.

The workspace is created and maintained by an NWSServer, therefore you must supply the hostname of the node where you started the server. The default hostname is `localhost` and the default port number is `8765`, both the defaults are used in this example. (If your NWSServer is not running on the local host, you will need to make appropriate changes for this example to work.) You can point a web browser to port `8766` (which is a configurable default) of the NWSServer to view the workspaces held by the server.

```
5 count = 10
6 cat('sum to:', count, '\n')
7 nwsStore(nws, 'summingTo', count)
```

In line 7, we use `nwsStore` to bind the name `summingTo` to the value `10` in the workspace `sum test`. You can point a browser to the URL `http://localhost:8766` and browse through the workspaces to see this variable and its associated value.

```
9 for (i in 1:count) {
10     nwsStore(nws, 'i', i)
11     j = nwsFetch(nws, 'i')
12 }
13 nwsStore(nws, 'sum', j)
```

In line 10, we bind the name `i` to the value `i` and in line 11, we remove the value from the binding and add it to the local variable `j`. When the loop completes, we bind the result to the name `sum` in our NetWorkspace, which we can view using the web browser.

R NetWorkSpaces Tutorial

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

First, an NWS server must be started. This can be done from a shell using the `nws` script command, as follows:

```
% nws start
```

This starts the server in daemon mode, with log messages going to the file `twistd.log` in the current directory. Second, start `babelfish.R`:

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

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

```
% R -q  
> library(nws)  
> ws = new('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 (i in 1:length(n)) {
+   nwsStore(ws, 'biff', n[i])
+ }
```

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 (1) {
+   tmp = nwsFetchTry(ws, 'biff')
+   if (is.null(tmp)) {
+     break
+   }
+   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 = new('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 **NWS'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 `NWSServer` you'll have to use the appropriate `hostname`, rather than `localhost`. That's all you need to get started. Have fun!

R Sleigh Tutorial

Sleigh is an R package, built on top of *NWS*, that makes it very easy to write simple 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 most 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 on the local host, but a vector argument can be used to specify the machines to start the workers on. Let's shut down the sleigh so we can start workers on some other machines. 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 workers on `node1` and `node2`, and we'll use an `NWSServer` that's running on `node10`:

```
> s = sleigh(nodeList=c('node1', 'node2'), nwsHost='node10')
```

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

```
> result = eachElem(s, function(x) {x + 1}, list(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 of numbers:

```
> 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.

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, add2, 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, add2, 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.

NetWorkSpaces for R Vignette

checkSleigh

Check for number of outstanding tasks.

Description

Check for number of outstanding tasks.

Usage

```
checkSleigh(.Object)
```

Arguments

.Object a `sleighPending` object returned from non-blocking `eachElem` or non-blocking `eachWorker`.

Details

This method is only for non-blocking version of `eachWorker` and `eachElem`.

See Also

`eachWorker`, `eachElem`

Examples

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

eachElem

Apply a given function with arguments supplied to workers.

Description

Evaluate the given function with multiple argument sets using the workers in sleigh.

Usage

```
eachElem(.Object, fun, elementArgs=list(), fixedArgs=list(),  
eo=NULL)
```

Arguments

`.Object` a sleigh object

`fun` function to be evaluated by workers

`elementArgs` variable arguments

`fixedArgs` fixed arguments

`eo` optional arguments, see *Details*

Details

All vector lengths in `elementArgs` have to be equal. For example:

```
eachElem(s, function(a, b, c) {a+b+c}, elementArgs=list(1:5,  
11:15, 21:25))
```

This invocation of `eachElem` add five tasks into the sleigh workspace. Each task consists of one value from `1:5`, one value from `11:15`, and one value from `21:25`. Values are paired up based on their indices in the vector. For example, first pair would be `(1, 11, 21)`, second pair would be `(2, 12, 22)`, and so on. When a sleigh worker grabs a task from the workspace, it will evaluate the function `fun` with the supplied value mapping to three arguments, `a`, `b`, and `c`.

`fixedArgs` can be used to pass fixed values to the function. For example:

```
eachElem(s, function(a, b, c) {a+b+c}, elementArgs=list(1:5,  
11:15), fixedArgs=list(1111))
```

This invocation of `eachElem` also adds five tasks to the workspace. This time, instead of having a varying value for the third argument, we have fixed value for the third argument. The ordering of arguments is variable arguments first followed by fixed arguments. If you want to change such ordering, see the optional argument, `eo`, for more details.

`eo` in `eachElem` argument can have these fields: `argPermute`, `blocking`, and `loadFactor`.

`argPermute`: a vector of integers used to represent the ordering of arguments. For example:

```
eo = list(argPermute=c(3, 2, 1))  
elemList=list(1:5, 11:15)  
fixedArgs=list(1111)
```

```
reordering = function(x, y, z) {paste('(x, y, z) =', x, y, z)}
eachElem(s, reordering, elementArgs=elementList,
fixedArgs=fixedArgs, eo=eo)
```

The original argument pairs without the permute argument are (1, 11, 1111), (2, 12, 1111), (3, 13, 1111), (4, 14, 1111), and (5, 15, 1111). With permute argument, the pairs would become (1111, 11, 1), (1111, 12, 2), (1111, 13, 3), (1111, 14, 4), and (1111, 15, 5).

blocking: set blocking mode to 0 or 1. By default, blocking mode is set to 1, which means `eachElem` does not return until all tasks are finished. If blocking mode is 0, then `eachElem` returns immediately with a `SleighPending` object. You can then use this object to invoke `checkSleigh` to check how many outstanding tasks there are, or invoke `waitSleigh` to wait for results. See `eachWorker` documentation for example on how to set blocking mode.

loadFactor: to restrict the number of tasks put out to the workspace.

The number of tasks in a workspace has to be less than or equal to `loadFactor` multiplied by the number of sleigh workers.

Examples

```
## Not run:
# simple hello world
s = sleigh()
hello = function(x) {paste("hello world", x, "from worker",
SleighRank)}
eachElem(s, hello, list(1:10))

# matrix addition
A<-matrix(1:6, 3, 3)
B<-matrix(1:6, 3, 3)
elemList = list(1:3)
fixedArgs = list(A, B)
mat_add = function(index, A, B) {
  row = A[index,]+B[index,]
  list(row)
}

result=eachElem(s, mat_add, elementArgs=elemList,
fixedArgs=fixedArgs)
print(result)

# clean up sleigh
stopSleigh(s)
## End(Not run)
```

eachWorker

`eachWorker` ensures that each worker executes a function once.

Description

`eachWorker` ensures that each worker executes the function `fun` exactly once.

Usage

```
eachWorker(.Object, fun, ..., eo=NULL)
```

Arguments

<code>.Object</code>	A sleigh object
<code>fun</code>	function to run on remote nodes
<code>...</code>	optional fixed arguments to be passed on to function <code>fun</code>
<code>eo</code>	additional options, see details

Details

The `eo` argument is used to set blocking or non-blocking for `eachWorker`. By default, blocking mode is set `1`, which means `eachWorker` does not return until every worker finishes executing the function `fun`. If blocking mode is `0`, then `eachWorker` returns immediately with a `SleighPending` object. Users can then use this object to invoke `checkSleigh` to check the number of outstanding tasks or invoke `waitSleigh` to wait for the results.

Examples

```
## Not run:
# example 1
s = sleigh()
eachWorker(s, function() {x<-1})
# eachElem can use global variable x initialized by eachWorker.
eachElem(s, function(y) {x+y}, list(1:10))
# example 2
options = list(blocking=0)
sp = eachWorker(s, function(z) {Sys.sleep(100)}, eo=options)
checkSleigh(sp)
waitSleigh(sp)

# example 3
# pass in fixed arguments to eachWorker
eachWorker(s, function(x, y) {x+y}, 10, 5)
## End(Not run)
```

netWorkspace

Class `netWorkspace`.

Description

Class representing `netWorkspace`.

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

Methods

```
initialize signature(.Object = "netWorkspace"): ...
nwsClose signature(.Object = "netWorkspace"): ...
nwsDeclare signature(.Object = "netWorkspace"): ...
nwsDeleteVar signature(.Object = "netWorkspace"): ...
nwsFetch signature(.Object = "netWorkspace"): ...
nwsFetchTry signature(.Object = "netWorkspace"): ...
nwsFind signature(.Object = "netWorkspace"): ...
nwsFindTry signature(.Object = "netWorkspace"): ...
nwsListVars signature(.Object = "netWorkspace"): ...
nwsStore signature(.Object = "netWorkspace"): ...
nwsWsName signature(.Object = "netWorkspace"): ...
```

Note

`netWorkspace` creates a `netWorkspace` object referencing the workspace in outer space (and creating the workspace if necessary). More generally, `netWorkspace` may include arguments to specify the host on which the `NetWorkspace` server runs, the port for that server, and optional attributes for the workspace.

Examples

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

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

nwsClose

Close connection to shared netWorkspace server.

Description

Close connection to shared netWorkspace server.

Usage

```
nwsClose(.Object)
```

Arguments

`.Object` A netWorkspace object

Examples

```
## Not run:  
ws = new("netWorkspace", "my Space")  
# ... (use ws)  
nwsClose(ws)  
## End(Not run)
```

nwsDeclare

Declare a mode for variable in a shared netWorkspace.

Description

Declare a mode for variable in a shared netWorkspace.

Usage

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

Arguments

`.Object` a netWorkspace object

`xName` name of variable to be stored.

`mode` mode of the variable, see *Details*.

Details

If `xName` has not already been declared in `nws`, the behavior of `xName` is 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 is used in `FIFO` mode, the youngest in `LIFO` mode, and a non-deterministic choice is made in `MULTI` mode. In `SINGLE` mode, only the most recent value is retained.

Examples

```
## Not run:  
ws = new("netWorkspace", "my space")  
nwsDeclare(ws, 'x', "LIFO")  
## End(Not run)
```

nwsDeleteVar

Delete a variable from a shared netWorkspace.

Description

Delete a variable from a shared netWorkspace.

Usage

```
nwsDeleteVar(.Object, xName)
```

Arguments

`.Object` a netWorkspace object

`xName` name of variable to be stored

Examples

```
## Not run:  
ws = new("netWorkspace", "my space")  
nwsDeleteVar(ws, 'x')  
## End(Not run)
```

nwsDeleteWs

Delete a workspace form server.

Description

Delete a workspace form server.

Usage

```
nwsDeleteWs(.Object, 'wsName')
```

Arguments

`.Object` a nwsServer object

`wsName` name of workSpace to be deleted

Examples

```
## Not run:
# example 1
s = new("nwsServer")
ws = openWs(s, "my space")
# ...
nwsDeleteWs(s, "my space")

# example 2
ws = new("netWorkspace", "my space")
# ...
nwsDeleteWs(ws@server, "my space")
## End(Not run)
```

nwsFetch

Fetch something from the shared workspace (blocking).

Description

Fetch something from the shared workspace (blocking).

Usage

```
nwsFetch(.Object, xName)
```

Arguments

`.Object` a netWorkspace object

`xName` name of variable to be fetched

Details

Block until a value for `xName` is found in the shared workspace `.Object`. If found, remove a value associated with `xName` from the workspace corresponding to `.Object` and return it in `xName`. Block if not found. This operation is atomic; if there is more than one value associated with `xName`, the particular value removed depends on `xName`'s behavior.

See Also

[nwsStore](#), [nwsFetchTry](#)

Examples

```
## Not run:
ws = new(netWorkspace, 'my space')
nwsFetch(ws, 'x')
## End(Not run)
```

nwsFetchTry

Fetch something from the shared workspace (non-blocking).

Description

Fetch something from the shared workspace (non-blocking).

Usage

```
nwsFetchTry(.Object, xName, defaultVal=NULL)
```

Arguments

`.Object` a netWorkspace object

`xName` name of variable to be fetched

`defaultVal` default value to return if `xName` is not found

Details

Look in the shared workspace for a value bound to `xName`. If found, remove a value associated with `xName` from `nws` and return it in `xName`. The third optional argument `defaultVal` is returned if `xName` is not found. If the third argument is missing, failure to fetch will return NULL. This operation is atomic. If there is more than one value associated with `xName`, the particular value removed depends on `xName`'s behavior.

See Also

[nwsStore](#), [nwsFetch](#)

Examples

```
## Not run:
ws = new(netWorkspace, 'my space')
# To fetch value of variable 'x' from workspace 'ws, and to
return NULL if not found:
nwsFetch(ws, 'xName')
# To fetch value of variable 'x' from workspace 'ws, and to
return 10 if not found:
nwsFetch(ws, 'xName', 10)
## End(Not run)
```

nwsFind

Find something from the shared netWorkspace (blocking).

Description

Find something from the shared netWorkspace (blocking).

Usage

```
nwsFind(.Object, xName)
```

Arguments

`.Object` a netWorkspace object.

`xName` name of variable to be fetched.

Details

Block until a value for `varName` is found in the shared workspace `ws`. Once found, return in `X` a value associated with `varName`. If there is more than one value associated with `varName`, the particular value returned depends on `varName`'s behavior. See [nwsStore](#) for details.

Examples

```
## Not run:  
x = nwsFind(ws, 'x')  
## End(Not run)
```

nwsFindTry

Find something from the shared workspace (non-blocking).

Description

Find something from the shared netWorkspace (non-blocking).

Usage

```
nwsFindTry(.Object, xName, defaultVal=NULL)
```

Arguments

`.Object` a netWorkspace object.

`xName` name of variable to be found.

`defaultVal` default value to return if `xName` is not found.

Details

Look in the shared workspace for a value bound to `xName`. If found, return in `xName` a value associated with `xName`. The third optional argument, `defaultVal`, is returned if `xName` is not found. If the third argument is missing, failure to find returns NULL. If there is more than one value associated with `xName`, the particular value returned depends on `varName`'s behavior. See [nwsStore](#) for details.

Examples

```
## Not run:  
x = nwsFindTry(ws, 'x', 0)  
## End(Not run)
```

nwsListVars

List variables in a netWorkspace.

Description

List variables in a netWorkspace.

Usage

```
nwsListVars(.Object, wsName="")
```

Arguments

`.Object` a netWorkspace object

`wsName` name of netWorkspace

Details

Return listing of variables in the named netWorkspace. To see list output clearly use:
`write(nwsListVars(), stdout())`.

nwsListWss

Returns a text string containing a list of workspaces in a server object.

Description

Returns a text string containing a list of workspaces in a server object.

Usage

```
nwsListWss(.Object)
```

Arguments

`.Object` a `nwsServer` object

nwsMktempWs

Returns a text string containing a list of workspaces in a server object.

Description

Returns a text string containing a list of workspaces in a server object.

Usage

```
nwsMktempWs(.Object, wsNameTemplate)
```

Arguments

`.Object` a `nwsServer` object

`wsNameTemplate` text string

nwsOpenWs

Return a workspace object, claim ownership if not owned.

Description

Return a workspace object. Claim ownership if not owned.

Usage

```
nwsOpenWs(.Object, wsName, space=NULL, ...)
```

Arguments

`.Object` a `nwsServer` object.

`wsName` name of workspace to be deleted.

`space` a `netWorkSpace` object.

`...` optional arguments related to persistence.

Examples

```
## Not run:  
s = new("nwsServer")  
ws = openWs(s, "my space")  
## End(Not run)
```

nwsServer

Class `nwsServer`

Description

Class representing connection to a netWorkspace server. Objects can be created by calls in the form `new("nwsServer")`.

Methods

```
initialize signature(.Object = "nwsServer"): ...
nwsDeleteWs signature(.Object = "nwsServer"): ...
nwsListWss signature(.Object = "nwsServer"): ...
nwsMktempWs signature(.Object = "nwsServer"): ...
nwsOpenWs signature(.Object = "nwsServer"): ...
nwsUseWs signature(.Object = "nwsServer"): ...
```

Examples

```
## Not run:
s = new("nwsServer")
## End(Not run)
```

nwsStore

Store something in a workspace.

Description

Store something in a workspace.

Usage

```
nwsStore(.Object, xName, xVal)
```

Arguments

<code>.Object</code>	a netWorkspace object
<code>xName</code>	name of variable to be stored
<code>xVal</code>	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 no mode is given and a mode has not already been set for `VarName`, `FIFO` is

used. The other options for mode are `LIFO`, `MULTI`, and `SINGLE` (see [nwsDeclare](#)). If no variable name is given, then `xName` must itself be a variable, and the value `xName` is associated with the variables name in the `WorkSpace`.

nwsUseWs

Return a workspace object, but do not claim ownership.

Description

Return a workspace object, but do not claim ownership..

Usage

```
nwsUseWs(.Object, wsName, space=NULL)
```

Arguments

<code>.Object</code>	a <code>nwsServer</code> object
<code>wsName</code>	name of <code>workSpace</code>
<code>space</code>	a <code>netWorkSpace</code> object

Examples

```
## Not run:  
s = new("nwsServer")  
ws = nwsUseWs(s, "my space")  
## End(Not run)
```

nwsWsName

Returns name of a `netWorkSpace`.

Description

Returns name of a `netWorkSpace`.

Usage

```
nwsWsName(.Object)
```

Arguments

<code>.Object</code>	a <code>netWorkSpace</code> object
----------------------	------------------------------------

Details

Returns the name of an object `netWorkSpace` as a string.

sleighPending

Class representing `sleighPending`.

Description

Class representing `sleighPending`.

Objects can be created by calls in the form `new("sleighPending", nws, numTasks, bn, ss)`. However, this class is often used in conjunction with non-blocking `eachWorker` or non-blocking `eachElem`.

Slots

```
nws: Object of class "netWorkspace" representation of
netWorkspace object.
numTasks: Object of class "numeric" represents the number of
tasks.
barrierName: Object of class "character" barrier name
sleighState: Object of class "environment" representation of
sleigh state.
state: Object of class "environment" representation of
sleighPending state.
```

Methods

```
checkSleigh signature(.Object = "sleighPending"): ...
initialize signature(.Object = "sleighPending"): ...
waitSleigh signature(.Object = "sleighPending"): ...
```

sleigh

Class representing `sleigh`.

Description

Class representing `sleigh`. Objects can be created by calls in the form `sleigh(nodeList, ...)`. `nodeList` is a list of machine names to start up sleigh workers.

Details

Takes a list of node names (which can be repeated) and a list of options (for `nws host`, `port`, and `space name template`, `launch command`, `user`, and `script directory`). All options have default values.

`nws host` defaults to `localhost`.

`port` defaults to `8765`.

space name	defaults to unique generated name with prefix sleigh_ride.
launch command	defaults to SSH.
user	defaults to the user who starts up R session.
script directory	defaults to the directory where R session starts.

Slots

nodeList: Object of class "character" representation of hosts, where workers will be created
nws: Object of class "netWorkspace" representation of netWorkspace object.
nwsName: Object of class "character" netWorkspace name.
nwss: Object of class "nwsServer" representation of nwsServer object.
options: Object of class "environment" options environment variable.
state: Object of class "environment" representation of sleigh state.
workerCount: Object of class "numeric" number of sleigh workers.

Methods

```
eachElem signature(.Object = "sleigh"): ...  
eachWorker signature(.Object = "sleigh"): ...  
initialize signature(.Object = "sleigh"): ...  
stopSleigh signature(.Object = "sleigh"): ...
```

Examples

```
## Not run:  
# Default option: create three sleigh workers on local host  
s = sleigh()  
# Or,  
s = new('sleigh')  
  
# Create sleigh workers on multiple machines  
s = sleigh(c('n1', 'n2', 'n3'))  
  
# Use LSF instead of SSH for remote login.  
# to login to remote machines.  
s = sleigh(launch=lsfcmd)
```

```
# Use RSH instead
s = sleigh(launch=rshcmd)
## End(Not run)
```

stopSleigh

Shutdown workers and remove netWorkspace.

Description

Shutdown workers and remove netWorkspace.

Usage

```
stopSleigh(.Object)
```

Arguments

`.Object` a sleigh object

Examples

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

waitSleigh

Wait for outstanding tasks to complete and retrieve results.

Description

Wait for all outstanding tasks created by `eachElem` or `eachWorker` to complete and retrieve those results.

Usage

```
waitSleigh(.Object)
```

Arguments

`.Object` a `sleighPending` object returned from non-blocking `eachElem` or non-blocking `eachWorker`.

Details

Retrieve results from invocation of non-blocking `eachElem` or non-blocking `eachWorker`. If sleigh workers are still working tasks, wait for the results to return.

See Also — [eachWorker](#), [eachElem](#).



A

Appendix 13

E

Example Program 5
 Required Software 5
 Setup 5

G

Getting Started 5

I

Introduction 1

N

NetWorkSpaces for R Vignette 13
 checkSleigh 13
 eachElem 13
 eachWorker 16
 netWorkspace 17
 nwsClose 18
 nwsDeclare 18
 nwsDeleteVar 19
 nwsDeleteWS 19
 nwsFetch 20
 nwsFetchTry 21
 nwsFind 22
 nwsFindTry 22
 nwsListVars 23
 nwsListWss 23
 nwsMktempWs 24
 nwsOpenWs 24
 nwsServer 25
 nwsStore 25
 nwsUseWs 26
 nwsWsName 26
 sleigh 27
 sleighPending 27
 stopSleigh 29
 waitSleigh 29

R

R NetWorkSpaces Tutorial 7
R Sleigh Tutorial 10

T

Tutorials 7

SCIENTIFIC
COMPUTING ASSOCIATES
INC.

One Century Tower
265 Church Street, New Haven, CT 06510-7010
Tel: (203) 777-7442 • Fax (203) 776-4074

www.LindaSpaces.com