

HowTo access a file repository

Jeff Gentry

May 17, 2005

1 Overview

This article demonstrates how you can make use of the *reposTools* package available in the Bioconductor project to quickly and easily access, download and install files stored in R repositories. This package provides a set of interactive tools to communicate with R repositories as well as tools to handle automatic library management.

An R file repository is a set of packages, vignettes, or other data that can be accessed interactively. Users can obtain automatic updates of packages, data and other R related resources. The *reposTools* package allows users to easily locate and communicate with R repositories, and download/update/install the available objects on the remote repository to the local system with attention paid to dependencies, platform, version, etc. A key aspect for this package is automatic library management for the user's R installation, which is the focus of this article.

2 Getting Started

To start using *reposTools*, you will need to obtain the package and install it. The package is available from www.bioconductor.org. Now, load it with the `library` command:

```
> origLibPaths <- .libPaths()
> library(reposTools)
```

For this vignette, we will create a temporary directory to simulate your R package library directory.

```
> tmpLib <- tempfile()
> dir.create(tmpLib)
```

We will be using this location to install (and then delete) packages in this vignette, so that it will not interfere with your primary R installation.

3 Initial setup of local system

In order to keep track of what packages are currently installed, *reposTools* maintains a small database in the R library directories. The `syncLocalLibList` command can be used to initialize this database, or to make sure it is up to date.

The `.libPaths` function is used as the default for this and most other functions in *reposTools* which require a library option. It is important that if one regularly uses another library directory that they have it in their `.libPaths`, so as to keep *reposTools* working accurately.

In this example, however, we only want to update the temporary library that we just created. Currently, that library is empty, however this will initialize the database in that library directory.

```
> syncLocalLibList(tmpLib)
```

This function determines which packages (and what versions of those packages) are currently installed and generates the database file, `liblisting.Rda`. If this file already exists, it will be updated to reflect the current information - for instance, if packages were manually installed, updated, or deleted, these changes will now be reflected in the user's database.

The `syncLocalLibList` command can be run at any time by the user, such as after manually installing or removing a package. It will automatically be called every time the user loads the *reposTools* package, as it is called from the `.First.lib` function of the package.

It is generally a good idea to try to avoid manual package management, but it should not cause any real problems with the automatic package management system. Utilizing the *reposTools* functions as much as possible will help insure that all dependencies are met for package installs and removals, and if desired, that package updates are obtained from the same repositories from which they originally were downloaded to be sure that one is following the same branch of code.

4 Locating Repositories

An initial set of default repositories are provided to the user by the *reposTools* package - these are used by default for every function which takes a set of repositories as input.

These can be accessed by the `getReposOption` command or directly by viewing the `repositories2` option:

```
> getReposOption()
```

```
CRAN
"http://www.bioconductor.org/CRANrepository"
BIOCRrel1.5
```

```

"http://www.bioconductor.org/repository/release1.5/package"
      BIOCRel1.6
      "http://www.bioconductor.org/packages/bioc/1.6"
      BIOCDevel
      "http://www.bioconductor.org/packages/bioc/devel"
      BIOCDATA
"http://www.bioconductor.org/packages/data/annotation/stable"
      BIOCDATA
      "http://www.bioconductor.org/packages/data/annotation/devel"
      BIOCDATA
      "http://www.bioconductor.org/packages/data/experiment/stable"
      BIOCCourses
      "http://www.bioconductor.org/repository/Courses"
      BIOCCdf
"http://www.bioconductor.org/packages/data/annotation/stable"
      BIOCCdf
      "http://www.bioconductor.org/packages/data/annotation/stable"
      BIOCCdf
      "http://www.bioconductor.org/packages/omegahat/stable"
      BIOCCdf
      "http://www.bioconductor.org/packages/lindsey/stable"

```

```
> getOption("repositories2")
```

```

      CRAN
      "http://www.bioconductor.org/CRANrepository"
      BIOCRel1.5
      "http://www.bioconductor.org/repository/release1.5/package"
      BIOCRel1.6
      "http://www.bioconductor.org/packages/bioc/1.6"
      BIOCDevel
      "http://www.bioconductor.org/packages/bioc/devel"
      BIOCDATA
"http://www.bioconductor.org/packages/data/annotation/stable"
      BIOCDATA
      "http://www.bioconductor.org/packages/data/annotation/devel"
      BIOCDATA
      "http://www.bioconductor.org/packages/data/experiment/stable"
      BIOCCourses
      "http://www.bioconductor.org/repository/Courses"
      BIOCCdf
"http://www.bioconductor.org/packages/data/annotation/stable"
      BIOCCdf
      "http://www.bioconductor.org/packages/data/annotation/stable"
      BIOCCdf
      "http://www.bioconductor.org/packages/data/annotation/stable"
      BIOCCdf

```

```
"http://www.bioconductor.org/packages/omegahat/stable"
      BIOCLinds
"http://www.bioconductor.org/packages/lindsey/stable"
```

This results in a named vector of repositories, with both the symbolic name and its associated URL. The order of this vector is important, as some functions will view this as the order in which repositories should be searched for a particular package. By default, the first (and thus primary) repository is a mirror of CRAN where a repository was built from the packages. After that are the repositories for the current release version of Bioconductor, the developmental version of Bioconductor, as well as the repositories for the Bioconductor data and course packages.

If a user has a repository that they'd like to access frequently, it is easiest to manually edit the `repositories2` option and place this repository in the position which it is desired. Repositories which are only going to be accessed once in a while are easier accessed via `getReposEntry`, which is discussed below.

5 Initial Access

For the rest of this document, we will be using a sample repository which is hosted on the Bioconductor website. This repository will be used in several functions as the `repEntry` parameter. If a user wishes to just use the default repositories as discussed above, simply do not provide this `repEntry` parameter.

The first step in accessing a repository is to download the repository information for the repository that you wish to utilize. This information is contained in an object of class `ReposEntry`.

The primary function to access a `ReposEntry` is `getReposEntry`, where one specifies a URL and it will get the `ReposEntry` object from that URL. A more convenient mechanism, if one wishes to access one of the builtin repositories is the function `repositories`, which provides the user with a menu of repositories, and will return the `ReposEntry` of the entry that was selected.

In this example, we will first use the `repositories` function (if this document is being processed in an interactive manner), as well as specify a particular URL to the `getReposEntry` function. The result of the latter call will be used throughout the rest of this document for other examples.

```
> if (interactive()) {
+   a <- repositories()
+   exampleRepos <- getReposEntry("http://www.bioconductor.org/repository/example")
+   exampleRepos
+ }
```

6 Installing a package from a repository

As you can see, the example repository contains two packages, *helloWorld* and *goodbyeWorld*. You can use the `install.packages2` function to install a package from a repository by passing it a list of packages names, the local library to install to and the `ReposEntry` object representing the repository that contains the desired packages.

Here we use the `ReposEntry` `exampleRepos`, specify the *helloWorld* package and give the location to install it to. For the purposes of this vignette, we are placing it in the `tmpLib` directory we created above. In normal usage, one can generally remove this parameter.

```
> if (interactive()) {  
+   install.packages2(pkgs = "helloWorld", repEntry = exampleRepos,  
+   lib = tmpLib)  
+ }
```

Leaving the package argument out will install all the packages available in the repository. Likewise, one can provide just the `pkgs` parameter and only the default repositories (see section *Locating Repositories*) will be searched.

An alternative to providing a package name, the user can provide a `pkgInfo` object. This class provides a way to store both package name and version information together. To create a `pkgInfo` object:

```
> if (interactive()) {  
+   helloPkg <- buildPkgInfo("helloWorld", "1.0")  
+   helloPkg  
+ }
```

Note that this allows the user to specify a particular version of a package to install. The `pkgs` parameter will accept either the package name, a `pkgInfo` object, or a list that can be comprised of a mixture of the two.

One can also specify what type of package (e.g. `Source`, `Win32`) using the `type` argument, as well as toggle the `recurse` argument — which if set to `TRUE` will progress through subrepositories (and their subrepositories) if it can't find a desired package in the current repository. For the latter two options, the default is `recurse=TRUE` and `type=Source`. See below for a more verbose explanation of all arguments.

7 Handling potential typos, slightly wrong names, etc

The `reposTools` package will note any packages which were not found and attempt to find any possible approximate matches (using the `agrep` function against the available packages) in case users have made typo mistakes when specifying packages or simply got the name of a package slightly wrong. The

way to take advantage of this is to assign the output of `install.packages2` or `update.packages2` to a variable, which will be of class `pkgStatusList` and the potential matches can be accessed via its `matchesList` slot with the `matchesList` method. The slot contains a list with an element for each package which was not found originally that had potential matches. Each element contains a vector of those matches, which users can then look at to see if they spot their mistake. For instance:

```
> if (interactive()) {
+   out <- install.packages2(c("bar", "world"), repEntry = exampleRepos,
+     lib = tmpLib)
+   matchesList(out)
+ }
```

Here we can see that there are no matches for *bar*, but that there are two matches for *world*.

8 Dealing with release levels

As of the Bioconductor 1.4 release, *reposTools* now provides for the notion of a **release level** for packages. Specifically, two levels are provided: **release** for released software and **devel** for developmental packages. By default, the `install.packages2` and `update.packages2` functions will ignore any packages marked as being **devel** even if that is a package the user requests. For instance, if someone calls `update.packages2("Biobase")`, any version of *Biobase* in any of the repositories searched will not be installed. Repositories themselves are flagged with a default release level, and this is used for any package that does not explicitly label itself. In most cases, all packages in a repository will be using the default value of the repository for the release level.

For this example, we will demonstrate the issues of downloading **devel** packages by attempting to download the *hgu95av2* package from the developmental metadata repository hosted by Bioconductor.

```
> if (interactive()) {
+   devRep <- getReposEntry("http://www.bioconductor.org/data/metaData-devel")
+   "hgu95av2" %in% repPkgs(devRep)
+   repReleaseLevel(devRep)
+   install.packages2("hgu95av2", repEntry = devRep, lib = tmpLib,
+     develOK = TRUE)
+ }
```

9 Installing a theme from a repository

Beyond single packages, there is a notion in *reposTools* of a **repository theme**, which is a collection of packages specified by the repository and given a single

name which can be referenced. Requests to install one of these themes will selectively obtain the packages specified by the theme from the given repository. This allows for larger repositories to effectively provide sub-repositories in a single space as well as a convenient mechanism for overlapping packages in sub-repositories.

To utilize a theme, one specifies the `theme` argument to `install.packages2`. If this is done, then the `pkgs` argument is ignored, and packages listed as being a part of the specified theme will be used as the set of packages instead. In the following example, we will be installing the theme `TestTheme` from our repository. This theme contains the *Biboase* and *geneplotter* packages, but not *annotate*.

```
> if (interactive()) {
+   install.packages2(repEntry = exampleRepos, theme = "Testing",
+     lib = tmpLib)
+ }
```

10 Updating packages from a repository

The process of updating packages to the latest version is similar to the `install`. There are a few key differences: If no packages are specified, all installed packages in the users `lib` argument are used for the update. Speaking of R libraries, one can specify more than one to update from (default is `.libPaths()`, which is all that R knows about); if `prevRepos=TRUE`, the system will attempt to update from the repository it was last acquired from before attempting to look at any supplied `RepoEntry` or the `repositories` option. This is to help lessen the load on some of the larger repositories such as *CRAN*.

```
> if (interactive()) {
+   update.packages2(repEntry = exampleRepos, lib = tmpLib)
+ }
```

11 Removing a package from your system

Removing a package is quite simple. Using the command `remove.packages2` (note: as before, the '2' is temporary), and passing in a set of packages to remove as well as which library to remove them from:

```
> if (interactive()) {
+   remove.packages2("goodbyeWorld", lib = tmpLib)
+ }
```

Finally, we will remove the temporary directory.

```
> unlink(tmpLib, recursive = TRUE)
> .libPaths(origLibPaths)
```

12 Usage Information

`install.packages2(pkgs, repEntry, lib, recurse = TRUE, type="Source", force=FALSE, syncLocal = TRUE, searchOptions = TRUE, versForce = FALSE)`

`update.packages2(pkgs=NULL, repEntry, libs=.libPaths(), recurse=TRUE, prevRepos=TRUE, force=FALSE, upTest=getNewerPkgs, type="Source", syncLocal=TRUE, versForce=FALSE, searchOptions=TRUE)`

`download.packages2(pkgs, repEntry, destDir, recurse=TRUE, type="Source", searchOptions=TRUE, versForce=FALSE)`

`remove.packages2(pkgs, lib, force=FALSE)`

repEntry An object of type `RepoEntry`, if the user wishes to specify a particular repository to use.

pkgs A character vector of package names to act upon.

lib(s) A directory to use for installing/removing the package. `update.packages2` takes `libs`, which allows for multiple lib directories to be specified.

destDir Identical to `lib`, just used for `download.packages2`

prevRepos A logical. If `TRUE` (default), will preferentially update a package from the repository it was last acquired from.

force A logical. If `FALSE` (default), will check to insure that updating/installing/removing a package will not break any dependencies on the current status quo. If `TRUE`, will force the requested action.

upTest A function taking a `VersionNumber` object and a repository data.frame, to determine if a package should be updated or not. The default merely checks to see if there is a higher version, and if so uses the highest version number available.

recurse If `TRUE` (default), will look through any listed subrepositories when searching for a package.

type Notes what type of package the user is looking for (i.e. a "Source" (.tar.gz) package, a "Win32" (.zip), etc).

syncLocal If `TRUE` (default), and the system is given a `lib` that has not previously had `syncLocalLibList` run on it, will run this function before any work is done. The system can not install/update/remove packages without a local library management system in place.

searchOptions If `TRUE` (default), will look at `getOption("repositories")` to provide alternate repositories for searching.

versForce If `FALSE` (default), the system will not allow the user to install/update R binary packages (.zip) that were built for a different version of R than the user is running. (e.g. if the package was built for R 1.6, and the user is running R 1.5).