HowTo access a file repository

Jeff Gentry

January 23, 2004

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 for a set of interactive tools to communicate with R repositories as well as tools to handle automatic library management.

A 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:

```
> 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)

Synching your local package management information ...
```

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"
```

```
                                                    BIOCRel1.3
"http://www.bioconductor.org/repository/release1.3/package"
                                                     BIOCDevel
     "http://www.bioconductor.org/repository/devel/package"
                                                      BIOCData
                "http://www.bioconductor.org/data/metaData"
                                                   BIOCCourses
           "http://www.bioconductor.org/repository/Courses"
                                                       BIOCcdf
           "http://www.bioconductor.org/data/cdfenvs/repos"
                                                    BIOCprobes
         "http://www.bioconductor.org/data/probes/Packages"
```

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

```
                                                         CRAN
               "http://www.bioconductor.org/CRANrepository"
                                                    BIOCRel1.3
"http://www.bioconductor.org/repository/release1.3/package"
                                                     BIOCDevel
     "http://www.bioconductor.org/repository/devel/package"
                                                      BIOCData
                "http://www.bioconductor.org/data/metaData"
                                                   BIOCCourses
           "http://www.bioconductor.org/repository/Courses"
                                                       BIOCcdf
           "http://www.bioconductor.org/data/cdfenvs/repos"
                                                    BIOCprobes
         "http://www.bioconductor.org/data/probes/Packages"
```

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`.

```
> z <- getReposEntry("http://www.bioconductor.org/repository/sample/package")
> z

Repository Information:
Repository Listing:
        Repository:  Sample Package Repository
                Type:  package

     pkgs        vers types
[1,] annotate    1.0  Source, Win32
[2,] Biobase     1.0  Source, Win32
[3,] geneplotter 1.0  Source, Win32
```

6 Installing a package from a repository

As you can see, that repository contained 3 packages, one of which was *Biobase*. Suppose you would like to install this package, you would use the command `install.packages2`, and pass in a set of package names (in this case simply *Biobase*), the local library to install to and your `ReposEntry` object.

Here we use the `ReposEntry` `z`, give the name of the package, and a 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. Also note the usage of the parameter `force=TRUE`. Normally `force` is set to `FALSE`, and generally it is a good idea to leave it set this way. Setting it to `TRUE` will override automatic dependency checking for a package, which is useful in this example but not in general.

```
> install.packages2(pkgs = "Biobase", repEntry = z, lib = tmpLib,
+     force = TRUE, searchOptions = FALSE)

Note: You did not specify a download type.  Using a default value of: Source
This will be fine for almost all users

Note: Couldn't find a local library database in library /tmp/Rtmp29298/file446b .  Synching

Synching your local package management information ...
Note: No locliblisting in R library /tmp/Rtmp29298/file446b
Note: No locliblisting in R library /tmp/Rtmp29298/file446b
Note: destDir parameter missing, using current directory
[1] "Attempting to download Biobase from http://www.bioconductor.org/repository//sample/pack
```

```
[1] "Download complete."
[1] "Installing Biobase"
[1] "Installation complete"

Synching your local package management information ...
Packages which have been added/updated:
        Biobase


From URL:  http://www.bioconductor.org/repository//sample/package
        Biobase version 1.0
```

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.

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:

```
> BiobasePkg <- buildPkgInfo("Biobase", "1.0")
> BiobasePkg

[1] "Biobase: 1.0"
```

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 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 then 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 `ReposEntry` or the `repositories` option. This is to help lessen the load on some of the larger repositories such as *CRAN*.

```
> update.packages2(repEntry = z, lib = tmpLib, searchOptions = FALSE)
```

```
Note: You did not specify a download type.  Using a default value of: Source
This will be fine for almost all users
```

8 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:

```
> remove.packages2("Biobase", lib = tmpLib, force = TRUE)

[1] "Removing package Biobase from system ...."
[1] "Removal complete"
```

Finally, we will remove the temporary directory.

```
> unlink(tmpLib, recursive = TRUE)
```

9 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 `ReposEntry`, 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 lookt 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 then the user is running. (e.g. if the package was built for R 1.6, and the user is running R 1.5).