Getting Started with RLink in *Mayday*

Florian Battke

1 Downloading and Installing the Required Software

1.1 General outline

Installation of RLink is a bit more complicated than that of other *Mayday* plugins due to need for native libraries. This section explains all the requirements for using RLink while the next section provides a step-by-step installation tutorial. Depending on your operating system, some of these general steps may not be needed.

1. `R` must be installed[1].
2. `R` must be configured to run with Java.
3. The `rJava` package for `R` must be installed. This package links Java applications to `R` and vice-versa. It consists of two parts: the rJava `R` package allowing `R` to call functions inside a Java Virtual Machine (JVM) and the JRI library allowing Java programs to start and interact with an embedded `R` process. JRI consists of a native library (`libjri.so` or `jri.dll`) and Java bindings (`JRI.jar`).
4. The JRI library must be placed in a directory where Java can find it. The easiest way is to place it in the system's default library location.
5. The JRI jar file must be in a directory where *Mayday*'s plugin scanner can find it.
6. The `R_HOME` environment variable needs to be set to point to the `R` installation directory so that Java can start `R` processes.
7. Download the RLink package from the *Mayday* website and extract it in *Mayday*'s plugin directory.

1.2 Installation in Microsoft Windows XP, 32bit

1. Installing Java
   You'll need the Java Runtime Environment which can be found at `java.sun.com`
   We'll assume that you installed Java 6 into C:\Program files\Java\jre6\bin\client. For the following steps, replace JPATH with the path to your Java installation.

[1] `R` is available at `http://www.r-project.org/`

2. Installing `R`
   Install R binaries from `www.r-project.org`. We'll assume you installed version 2.7.0. For the following steps, replace **RPATH** with the path to your R installation, i.e. C:\Program files\R\R-2.7.0.

3. Installing the `rJava` binary package
   Start R (via the start menu or via RGui.exe in **RPATH**\bin\).
   ```
   > install.packages("rJava")
   ```
   Close R.

4. Make sure *Mayday* can find the JRI jar file.
   Copy the file `JRI.jar` from **RPATH**\library\rJava\jri\ to your *Mayday* plugin directory[2].

5. Set the `PATH` variable so that the jri libraries are found
   Open System control from the Control Panel. Select the "Advanced" tab, click on "Environment variables". In the list "System variables", select the variable "Path" and click "Edit".
   To the end of the second line, add the following:
   ```
   ;RPATH\bin;JPATH;
   ```

6. Download the RLink package from the *Mayday* website and extract it in *Mayday*'s plugin directory.

1.3 Installation in Ubuntu Linux 9.04

1. Installing Java
   You'll need the Java Runtime Kit to run *Mayday*.
   ```
   sudo apt-get install sun-java6-jre
   ```

2. Installing `R` and rJava/JRI
   ```
   sudo apt-get install r-base-core
   ```

3. Installing the `rJava` package
   ```
   sudo apt-get install r-cran-rjava
   ```
   This will install several additional files, among others it may install the openjdk java framework. We have to make sure that SUN Java is used as default.
   ```
   sudo update-alternatives --config java
   ```
   Please select java-6-sun from the list.

4. Make sure Java can find the JRI library
   We will simply create a symbolic link from `/usr/lib` to point to the correct path:
   ```
   sudo ln -s /usr/lib/R/site-library/rJava/jri/libjri.so /usr/lib/
   ```

[2]The plugin directory can be changed in *Mayday* by clicking "Mayday"–"Preferences"–"Plugins"

5. Make sure *Mayday* can find the JRI jar file.
   We create a symbolic link from *Mayday*'s plugin directory[3] to the original location of `JRI.jar`. For the remainder of this guide, we'll assume that your plugin directory is set to `/mayday`.
   ```
   ln -s /usr/lib/R/site-library/rJava/jri/JRI.jar /mayday
   ```

6. Define the `R_HOME` environment variable
   We add the definition to the system-wide bash configuration
   ```
   sudo nano /etc/bash.bashrc
   ```
   Add this line at the end of the file
   ```
   export R_HOME=¨/usr/lib/R¨
   ```

7. Download the RLink package from the *Mayday* website and extract it in *Mayday*'s plugin directory.

1.4 Installation *from source* in Ubuntu 9.04

This section gives a step-by-step guide to installing RLink on a Ubuntu Linux 9.04 system. Other Linux distributions may need slightly different package names, the package manager (apt in Ubuntu) maybe different (e.g. yum) and some paths may not be exactly the same. In general, installing RLink on a Linux system should be similar. Ubuntu uses the `sudo` command to execute programs with root user privileges. If this isn't working for you, use `su` to get a root shell and execute the commands without the `sudo` prefix.

1. Installing Java
   You'll need the Java Development Kit to compile RLink.
   ```
   sudo apt-get install sun-java6-jdk
   ```

2. Installing `R`
   ```
   sudo apt-get install r-base-core
   ```

3. Configuring `R` to work with Java
   ```
   sudo R CMD javareconf
   ```

4. Installing the `rJava` package
   ```
   sudo R
   > install.packages(¨rJava¨)
   > q()
   Save workspace image? [y/n/c]: n
   ```
   This step will install the rJava package and compile the JRI library `libjri.so` as well as the Java bindings in `JRI.jar`. The compiled files are placed in a subdirectory of the `R` library path.

5. Make sure Java can find the JRI library
   We will simply create a symbolic link from `/usr/lib` to point to the correct

[3] The plugin directory can be changed in *Mayday* by clicking "Mayday"–"Preferences"–"Plugins"

path:

```
sudo ln -s /usr/local/lib/R/site-library/rJava/jri/libjri.so /usr/lib/
```

6. Make sure *Mayday* can find the JRI jar file.
   We create a symbolic link from *Mayday*'s plugin directory[4] to the original location of `JRI.jar`. For the remainder of this guide, we'll assume that your plugin directory is set to `/mayday`.

   ```
   ln -s /usr/local/lib/R/site-library/rJava/jri/JRI.jar /mayday
   ```

7. Define the `R_HOME` environment variable
   We add the definition to the system-wide bash configuration

   ```
   sudo nano /etc/bash.bashrc
   ```

   Add this line at the end of the file

   ```
   export R_HOME=¨/usr/lib/R¨
   ```

8. Download the RLink package from the *Mayday* website and extract it in *Mayday*'s plugin directory.

1.5 Installation in Mac OS 10.5 (Leopard)

1. Installing Java
   Java 6 is only available as 64 bit version for MacOS.

2. Installing `R`
   Install R binaries from `http://r.research.att.com/`. Make sure to install the 64 bit version of R.

3. Installing the `rJava` package
   Start R in 64 bit mode and install the package.

   ```
   sudo R --arch x86_64
   > install.packages(¨rJava¨)
   > q()
   Save workspace image?  [y/n/c]:  n
   ```

4. Make sure Java can find the JRI library
   We will simply create a symbolic link from `/usr/lib` to point to the correct path:

   ```
   sudo ln -s /Library/Frameworks/R.framework/Versions/Current/
         Resources/library/rJava/jri/libjri.jnilib /usr/lib/
   ```

5. Make sure *Mayday* can find the JRI jar file.
   We create a symbolic link from *Mayday*'s plugin directory[5] to the original location of `JRI.jar`. For the remainder of this guide, we'll assume that your plugin directory is set to `/mayday`.

   ```
   ln -s /Library/Frameworks/R.framework/Versions/Current/
         Resources/library/rJava/jri/JRI.jar /mayday
   ```

[4] The plugin directory can be changed in *Mayday* by clicking "Mayday"–"Preferences"–"Plugins"
[5] The plugin directory can be changed in *Mayday* by clicking "Mayday"–"Preferences"–"Plugins"

6. Define the R_HOME environment variable
   This can be done temporarily in a Terminal by entering
   export R_HOME=/Library/Frameworks/R.framework/Resources/
   You can also add the definiton to the system-wide bash configuration
   sudo nano /etc/bashrc
   Add this line at the end of the file
   export R_HOME=¨/Library/Frameworks/R.framework/Resources/"

7. Download the RLink package from the *Mayday* website and extract it in *Mayday*'s plugin directory.

2 Using RLink

2.1 Opening and using the RLink console

Start RLink from *Mayday*'s "Mayday" menu. You can use the internal console, or start RLink in server mode (see more in section 3). The internal console works almost like the original R console, with a few changes:

- You can enter multiline commands (use CTRL-Enter to start a new line).
- To navigate your command history, use PageUp and PageDown, respectively.
- Auto-complete is available for R functions and objects, use the Tab key once to complete your input, twice to see a list of possible completions.
- You can not interrupt R during lengthy computations – CTRL-C does not work here.
- While R is busy (indicated by a red border around the input field) you can enter more commands. These will be executed as soon as R has finished whatever it was doing.

2.2 An example

The next section introduces the RLink objects in detail. Let's look at a very simple example first.

1. We will create a random dataset with 100000 probes in 20 experiments, starting from an R matrix of normally distributed values.

```
data <- matrix(ncol=20, nrow=100000, rnorm(2000000))
probenames <- paste("Probe No.", 1:nrow(data))
experimentnames <- paste("Experiment No.", 1:ncol(data))
rownames(data) <- probenames;
colnames(data) <- experimentnames
ds <- addDataSet("Example", data);
```

2. Now let's add some meta information. We'll compute the standard deviation of each probe.

```
stdDevs <- apply(data, 1, sd, na.rm=T)
addProbeMIOs( getMIManager(ds), stdDevs, "Standard Deviations" )
```

3. The 10% probes with the highest variance would make a very interesting probe list.

```
stdDevOrder <- order(stdDevs, decreasing=T)
best10pc <- stdDevOrder[ 1:(length(stdDevs/10)) ]
ds[[ "Interesting Probes" ]] <- names(stdDev)[best10pc]
```

2.3 Some background information

A few things need to be kept in mind when working with RLink.

1. Pointer-like objects and local copies
Usually, R hides the true nature of objects from the user. For example, if m is a huge vector, the command k<-m copies a pointer while m[5]<-7 actually creates a clone of m, and then replaces the fifth element. Thus, R can get around expensive copy operations if they're not needed (clone on modification). In RLink, all *Mayday* objects can be regarded as pointers to the live data structures in the *Mayday* core. To make RLink efficient, data is only copied from *Mayday* into the R process as needed. It is up to you to decide when to use the pointer and when to create a local copy of the data it points to.
Advantages of using pointers are speed, memory efficiency and the fact that they always point to the current content of the *Mayday* core. Creating local copies has the advantage that repeat access of values is a lot faster on internal R matrices than accessing the *Mayday* MasterTable via the RLink pointer. Furthermore, it shields your R program from concurrent changes in *Mayday*, e.g. due to other plugins modifying core data structures at the same time.
In general, local copies of RLink objects can be obtained by using the [[]] (or sometimes []) operator:

```
ds <- mayday[[1]]    # get the first open dataset in mayday
ds2 <- ds            # ds2 and ds point to the same core
                     #  data structure
ds3 <- ds[]          # ds3 is a R matrix, a local copy of
                     # the DataSet's MasterTable
```

2. Virtual Machine Memory
Mayday displays the current usage of available memory in the Java Virtual Machine. The JVM startup parameters[6] determine how much memory *Mayday* can use for

[6] more specifically, the -Xmx switch

its data structures. An `R` process running as part of RLink does not consume any JVM memory. `R` objects consume memory as part of the JVM's total memory consumption, but are not stored inside the memory managed by the JVM. They are stored in the memory the operating system manages *for* the JVM. Thus you can use more memory in RLink as allowed for *Mayday*. However, it is your responsibility to remove `R` objects that you no longer need, or they will stay in memory until you close *Mayday*. Use `R`'s `rm()` and `gc()` commands.

3. Terminating `R`

Currently, `R` running inside another applications process can not be terminated without terminating the enclosing process. Furthermore, only one `R` process can be run at any time inside an application. If you close the RLink console window, `R` will remain active, consuming memory. Reopening the console will just connect to the previous `R` process, not create a new one. Again this means that you are responsible for any memory consumed by the `R` process.

3 RLink RMI Server

Using RLink's server mode, you can connect any R session to a running *Mayday* instance, both locally and over the network. Several parallel connections are also possible, so that lengthy R computations do not stop you from working with your data.

3.1 Starting the server

From *Mayday*'s "Mayday" menu, start RLink and select the server mode. Make sure the host name is correct (or remote connections won't be possible). The `rmiregistry` program must be running for the server to work. Mayday will try to start it automatically. If this doesn't work for you, you can start `rmiregistry` manually. The `CLASSPATH` environment variable must be set such that it contains the RLink classes.

3.2 Connecting as a client

- Allow Java to access remote RMI services. Create a new file in your user home directory, called `.java.policy` with the following content:

```
grant {
  permission java.security.AllPermission;
};
```

- Start R and prepare the virtual machine. We assume that you extracted the RLink archive in `/path/rlink`.

```
R
> library(rJava);
> jinit();
> .jaddClassPath( ``/path/rlink'' )
```

- Set the host you want to connect to, either by host name or IP address[7]:

```
> mhost=``192.168.1.2'';
```

- Load the RLink code

```
> source( ``/path/rlink/mayday/rlink/RConnector.R '');
```

4 RLink objects

4.1 Object descriptions

4.1.1 mayday

The `mayday` represents the connection from `R` to *Mayday*. It provides access to all open DataSets. New DataSets can be added. Available commands:

- `print`, `summary` provide information about the current state of *Mayday*
- `lapply`, `sapply` can be used to iterate over all open DataSets
- `length` returns the number of open DataSets
- `names` returns a vector of DataSet names
- `mayday[[ i ]]` returns the ith DataSet. If i is omitted, a local copy of the DataSet list is returned.
- `addDataSet( name, matrix )` creates a new DataSet. If another DataSets with the same `name` exists, the user is asked to enter a new name. `matrix` must be a numeric matrix of probe values with `rownames` containing Probe names and `colnames` containing experiment names. Probe names must be unique. If `matrix` is `NULL`, the resulting DataSet is empty. Otherwise, a global ProbeList is added. The result of this function is a new DataSet object.

[7] If you chose any port number n other than 1099 in the RLink server settings, you must also supply `mport=!`n

4.1.2 `dataset`

Dataset objects have a dual nature, being a list of ProbeLists and a Matrix of expression values at the same time. Due to the design of the *Mayday* core, access to Probe values is only possible using Probe names as indices. Numeric indices are not allowed.

1. Functions related to the DataSet

   - `print`, `summary` provide information about the object
   - `getName( dataset )` returns the DataSet's name
   - `setName( dataset, name )` changes the DataSet's name. If another DataSet with the same name exists, the user is asked to supply a unique name.
   - `removeDataSet( dataset )` closes the DataSet and removes it from *Mayday*.

2. Functions related to the DataSet's ProbeLists

   - `lapply`, `sapply` can be used to iterate over all ProbeLists in that DataSet
   - `length` returns the number of ProbeLists
   - `names` returns a vector of ProbeList names
   - `dataset[[ i ]]` returns the ith ProbeList. If `i` is omitted, a local copy of the list of ProbeLists is returned.
   - `dataset[[ i ]] <- v` adds or removes ProbeLists depending on `i` and `v`.
     Removing a ProbeList: `i` must be the index or name of an existing ProbeList, `v==NULL`.
     Adding a ProbeList: `i` must be the name of the new ProbeList, `v` must be a character vector of Probe names.
     ProbeList replacement is not supported.
   - `addProbeList( dataset, name, content, parent )` adds a new ProbeList. `name` is a string containing the new ProbeList name, `content` is a character vector of Probe names. `parent` can be omitted. If it is given, the new ProbeList is added as a child of `parent` (in case parent is a ProbeList Group) or as a sibling of `parent`. This function returns the newly created ProbeList.
   - `addProbeListGroup( dataset, name, parent )` adds a new ProbeList group. `name` is a string containing the new ProbeList name. `parent` can be omitted. If it is given, the new ProbeList is added as a child of `parent` (in case parent is a ProbeList Group) or as a sibling of `parent`. This function returns the newly created ProbeList group.

3. Functions related to the DataSet's expression values (Probes)

   - `apply` can be used to iterate over all Probes in that DataSet
   - `nrow` returns the number of Probes
   - `ncol` returns the number of experiments
   - `rownames` returns the names of the Probes. Probe names can be changed using the assignment operator (`rownames( dataset ) <- x`). Keep in mind that Probe names need to be unique!
   - `colnames` returns the names of the experiment. Experiment names can be changed using the assignment operator (`colnames( dataset ) <- x`).
   - `addProbes( dataset, matrix )` appends new values to the expression matrix. `matrix` must have the same number of columns (experiments) as the DataSet's expression matrix. The `rownames` of `matrix` must be unique and must not contain Probe names already used in the DataSet. If Probes are added to an empty DataSet, `matrix` must have a `colnames` attribute of experiment names.
   - `removeProbes( dataset, probes )` removes values from the expression matrix. `probes` must be a vector of Probe names. Removed Probes may still be referenced from and contained in open ProbeLists, however.
   - `dataset[ i, j ]` returns the expression value for Probe(s) i in experiment(s) j. `i` must be a character vector of Probe names, `j` a numeric vector of experiment indices. If `i` is omitted, all Probes are returned, if `j` is omitted, all experiment values are returned. `dataset[]` returns the whole expression matrix.
   - `update( dataset, matrix )` replaces the expression values in the dataset with those in the matrix. Matrix row names are used to find the probes to modify. New probes are added when no probe of the respective name exists.

4. Functions related to Meta Information Objects (MIOs)

    - `getMIManager( dataset )` returns the Meta Information Manager instance for this DataSet.

5. Functions related to Mayday Plugins

    - `callPlugin( dataset , PlumaID, ProbeLists )` calls the plugin identified by `PlumaID` on `ProbeLists` in this DataSet. The result is a list of (new) ProbeLists that are not automatically added to the DataSet.

4.1.3 `probelist`

ProbeList objects behave just like character vectors of Probe names.

1. Functions related to the ProbeList

    - `print`, `summary` provide information about the object
    - `getName( probelist )` returns the ProbeList's name
    - `setName( probelist, name )` changes the ProbeList's name. ProbeList names need not be unique.
    - `removeProbeList( probelist )` closes the ProbeList and removes it from the DataSet. Probes that are not contained in any other ProbeList are also removed from the expression matrix.
    - `getParent( probelist )` returns the ProbeList group that is this ProbeLists parent.

2. Functions related to the ProbeList content

    - `lapply`, `sapply` can be used to iterate over the Probe names
    - `length` returns the size of the ProbeList
    - `addProbes( probelist, probes )` includes Probes in the ProbeList. `probes` must be a vector of Probe names.
    - `removeProbes( probelist, probes )` removes Probes from the ProbeList. `probes` must be a vector of Probe names.
    - `probelist[ i ]` returns the ith Probe name. If `i` is omitted, a local copy of the ProbeList is returned.
    - `probelist[ i , v ]` returns the columns `v` for the probes i. Set `v=T` to get all columns or use any other indexing method. Omitting `i` returns the values for all probes in the probelist.

4.1.4 `mimanager`

This object refers to a DataSet's meta information manager and can be used to add and remove meta information groups (MIGroups).

1. Functions related to the MIManager
    - `print`, `summary` provide information about the object
2. Functions related to the MIGroups contained in the DataSet
    - `lapply`, `sapply` can be used to iterate over the MIGroups
    - `length` returns the number of MIGroups
    - `names` returns the names of all MIGroups
    - `mimanager[[ i ]]` returns the *i*th MIGroup. If `i` is omitted, a local copy of the list of MIGroups is returned.
    - `addMIGroup ( mimanager, name, plumatype, path )` adds a new MIGroup with the given `name`. Naming conflicts are resolved by the *Mayday* core. The data type of the MIGroup has to be supplied as a valid *Mayday* plugin id (defaults to `PAS.MIO.Double`). Optionally, the `path` of the new MIGroup in the MIGroup hierarchy can be provided as a string (defaults to the root path). If successful, the function returns the new MIGroup object.
    - `addProbeMIOs( mimanager, values, groupname, plumatype, path, stepping )` is a fast convenience method to create a new MIGroup and add a large number of Probe MIOs. The `groupname`, `plumatype` and `path` arguments are passed to `addMIGroup` with the same defaults as described there. `values` must be a vector the new MIO values with a `names` attribute containing unique Probe names of the Probes to attach the values to. `stepping` is an optional parameter defining how many values should be passed to *Mayday* in one function call. Defaulting to 1000, larger values can greatly speed up the function while consuming more memory for the transfer. See the description of `createMIO` in section 4.1.5 for details the `values` object.

4.1.5 `migroup`

A MIGroup attaches meta information to objects (Probes, ProbeLists or DataSets). It is implemented as a list mapping objects to values. However, list iteration and access to the full list is not possible.

1. Functions related to the MIGroup
    - `print`, `summary` provide information about the object
    - `length` returns the number of contained MIOs
    - `getName( migroup )` returns the MIGroup's name

- `setName( migroup, name )` changes the MIGroup's name. Name clashes are resolved by *Mayday*.
- `getType( migroup )` returns the *Mayday* plugin identifier for the MIGroup's content type
- `getPath( migroup )` returns the path of this MIGroup in *Mayday*'s MIGroup hierarchy
- `removeMIGroup( migroup )` completely removes the MIGroup from the DataSet

2. Functions related to the MIOs

- `migroup[ object, extractValue ]` returns the MIO value for the given object(s). `object` can either be a character vector of Probe names or a single ProbeList or DataSet object. If `extractValue` is set to `TRUE`, RLink will try to return `R` objects instead of Java objects subclassing class `MIType`. Extracting values only works for MIOs implementing the `GenericMIO<T>` interface where the Java type `T` corresponds to a native `R` type (Double→numeric, Integer→integer, String→character, Boolean→logical, ...).
- `addMIO( migroup, object, mio )` adds meta information to an object. `object` can be a Probe name, or a ProbeList or DataSet object. `mio` must be a Java object obtained by `createMIO`, the type of the MIO must correspond to the type of the MIGroup.
- `removeMIO( migroup, object )` removes the meta information for `object` from this MIGroup. See `addMIO` for valid `object` values.
- `createMIO( type, value )` creates a new MIO object. `type` must be a valid *Mayday* plugin id (e.g. `PAS.MIO.Double`). `value` will be cast to a string and must contain the desired MIO value in serialized form (as produced by `MIType.serialize(SERIAL_TEXT)`).

4.2 Command matrix

This matrix gives an overview over all RLink commands. Abbreviations are DS=DataSet, PL=ProbeList, MG=MIGroup, Pb=Probe, EM=Expression Matrix

	Mayday	DataSet	ProbeList	MIManager	MIGroup
print, summary	✓	✓	✓	✓	✓
getName	—	✓	✓	—	✓
setName	—	✓	✓	—	✓
getPath	—	—	—	—	✓
getParent	—	—	✓	—	—
getType	—	—	—	—	✓
length	no. DS	no. PL	no. Pb	no. MG	—
names	DS names	PL names	—	MG names	—
lapply, sapply	over DS	over PL	over Pb	over MG	—
`[[`	DS	PL	—	MG	—
`[[<-`	—	add/remove PL	—	—	—
apply	—	over EM	—	—	—
`[`	sublist DS	EM values	Probes	sublist MG	—
nrow,ncol,dim	—	EM size	—	—	—
colnames	—	Experiments	—	—	—
rownames	—	Probes	—	—	—
addProbes	—	✓	✓	—	—
removeProbes	—	✓	✓	—	—
update	—	✓	—	—	—
adding objects	addDataSet	addProbeList addProbeListGroup		addMIGroup addProbeMIOs	addMIO
removing objects		removeDataSet	removeProbeList		removeMIGroup removeMIO
		getMIManager			createMIO
using plugins		callPlugin			

This Mayday How-To was written and edited by Florian Battke. If you have comments or questions please contact the author via email, *battke@informatik.uni-tuebingen.de*. The latest version of this document can be found at *http://www.zbit.uni-tuebingen.de/pas/mayday*.