Mayday RLink – The best of both worlds

Florian Battke, Stephan Symons, Kay Nieselt

battke@informatik.uni-tuebingen.de

July 8, 2009
Outline

1. Motivation
2. Design
3. Implementation
4. Evaluation
5. Outlook
Motivation

Mayday – An extensible visualization platform

- Basic data structure is a numeric matrix
  - columns are observations, rows are "features" of interest
  - Aim is to find (full-width) submatrices with common features
Motivation

Mayday – An extensible visualization platform

**Strengths**

- Cross-platform: Written in **Java**
- Structured display of submatrices
- **Plugin-based** → fast integration of new methods
- **Interactive** visualizations, different views are linked
- Visualizations can be *enhanced* by **meta-data**
- Focus: **visual** data exploration and hypothesis generation

**One big deficit**

No live programmers’ access to the data.

→ “Power-users” often need to move data to R and back
Mayday – An extensible visualization platform

Strengths
- Cross-platform: Written in Java
- Structured display of submatrices
- **Plugin-based** → fast integration of new methods
- **Interactive** visualizations, different views are linked
- Visualizations can be *enhanced* by *meta-data*
- Focus: **visual** data exploration and hypothesis generation

One big deficit
No live programmers’ access to the data.
→ “Power-users” often need to move data to R and back
Integration of an interactive \texttt{R} shell into Mayday

- Live access to Mayday’s data
- Efficient data management
- Memory-safe data manipulation
- Objects behave as much like real \texttt{R} objects as possible
Possible solutions

**Self-made interface**
- e.g. using pipes
  - no process limit
  - could be interactive
- slow
- a LOT of work

**RServe / RSJava**
- using sockets
  - no process limit
  - no direct dependency
  - Java accessing R
  - no interactive session
  - still lots of work

**JRI + RJava**
- R embedded in JVM
  - only one R instance
  - shared memory
  - R accessing Java
  - interactivity built in
  - very fast

---

**Short overview: JRI+RJava**
- Using Java objects in R: rJava
- Embedding R in Java: JRI
- One process (JVM), memory shared between VM and R
- R event loop waiting for input from Java callbacks
Possible solutions

<table>
<thead>
<tr>
<th>Self-made interface</th>
<th>RServe / RSJava</th>
<th>JRI + RJava</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. using pipes</td>
<td>using sockets</td>
<td>R embedded in JVM</td>
</tr>
<tr>
<td>+ no process limit</td>
<td>+ no process limit</td>
<td>– only one R instance</td>
</tr>
<tr>
<td>+ could be interactive</td>
<td>+ no direct dependency</td>
<td>+ shared memory</td>
</tr>
<tr>
<td>– slow</td>
<td>– Java accessing R</td>
<td>+ R accessing Java</td>
</tr>
<tr>
<td>– a LOT of work</td>
<td>– no interactive session</td>
<td>+ interactivity built in</td>
</tr>
<tr>
<td></td>
<td>– still lots of work</td>
<td>+ very fast</td>
</tr>
</tbody>
</table>

Short overview: JRI+RJava

- Using Java objects in R: rJava
- Embedding R in Java: JRI
- One process (JVM), memory shared between VM and R
- R event loop waiting for input from Java callbacks
# Possible solutions

<table>
<thead>
<tr>
<th>Self-made interface</th>
<th>RServe / RSJava</th>
<th>JRI + RJava</th>
</tr>
</thead>
<tbody>
<tr>
<td>e.g. using pipes</td>
<td>using sockets</td>
<td>R embedded in JVM</td>
</tr>
<tr>
<td>+ no process limit</td>
<td>+ no process limit</td>
<td>– only one R instance</td>
</tr>
<tr>
<td>+ could be interactive</td>
<td>+ no direct dependency</td>
<td>+ shared memory</td>
</tr>
<tr>
<td>– slow</td>
<td>– Java accessing R</td>
<td>+ R accessing Java</td>
</tr>
<tr>
<td>– a LOT of work</td>
<td>– no interactive session</td>
<td>+ interactivity built in</td>
</tr>
<tr>
<td></td>
<td>– still lots of work</td>
<td>+ very fast</td>
</tr>
</tbody>
</table>

Short overview: JRI+RJava

- Using Java objects in R: rJava
- Embedding R in Java: JRI
- One process (JVM), memory shared between VM and R
- R event loop waiting for input from Java callbacks
## Possible solutions

### Self-made interface
- e.g. using pipes
  - no process limit
  - could be interactive
- slow
- a LOT of work

### R Serve / RS Java
- using sockets
  - no process limit
  - no direct dependency
- Java accessing R
  - no interactive session
  - still lots of work

### JRI + R Java
- R embedded in JVM
  - only one R instance
- shared memory
- R accessing Java
- interactivity built in
- very fast

---

**Short overview: JRI+RJava**
- Using Java objects in R: rJava
- Embedding R in Java: JRI
- One process (JVM), memory shared between VM and R
- R event loop waiting for input from Java callbacks
Some thoughts on memory management

Pointers
- no copying needed
- very fast
- uncontrolled access
- GC issues

Copied objects
- slow
- memory-intensive
- controlled access
- hard too keep in sync

“Controlled references”
- Lightweight S3 objects, containing
  - Identifier (integer), used by Java as object reference
  - Type/Class (string), used by R to resolve function calls
- copy data as needed, still very fast
- Java program decides what to expose to R
Some thoughts on memory management

Pointers
- no copying needed
- very fast
- uncontrolled access
- GC issues

Copied objects
- slow
- memory-intensive
- controlled access
- hard too keep in sync

“Controlled references”
- Lightweight S3 objects, containing
  - Identifier (integer), used by Java as object reference
  - Type/Class (string), used by R to resolve function calls
- copy data as needed, still very fast
- Java program decides what to expose to R
## Some thoughts on memory management

### Pointers
- no copying needed
- very fast
- uncontrolled access
- GC issues

### Copied objects
- slow
- memory-intensive
- controlled access
- hard to keep in sync

### “Controlled references”
- Leightweight S3 objects, containing
  - Identifier (integer), used by Java as object reference
  - Type/Class (string), used by R to resolve function calls
- copy data as needed, still very fast
- Java program decides what to expose to R
Thoughts on user-friendliness

Fetching a value from a `HashMap<String, Integer>`

- **JAVA**
  ```java
  int ret = hashMap.get("Key")
  ```

- **native rJava**
  ```java
  key <- .jnew( "Ljava/lang/String;", "Key" );
  ret <- .jcall( hashMap,
                  "Ljava/lang/Object;",
                  "get",
                  .jcast(key, "Ljava/lang/Object")
  )
  ret <- .jcast( ret, "Ljava/lang/Integer" )
  ret <- .jcall( ret, "I", "intValue" );
  ```

- **Our aim for RLink**
  ```r
  ret <- hashMap[["Key"]]
  ```
Command translation and data flow

Mayday (Java)  interactive R session
VM code (Java)  R functions (R)
VM memory mgr, GC (Java)  R library (native), MM, GC
Java VM core (native), JNI Communication

One object “ref” is shared between Mayday and R

Example: (int) ret ← hashMap[["Key"]], with class “rlink.hm” and id “5”

1. R resolves operator [[ for class “rlink.hm”
2. [[.rlink.hm(hashMap, "Key") uses rJava
3. .jcall(ref, "hmget", 5, .jnew("Ljava/lang/String", "Key"))
4. rJava/JRI transfer
5. ref.hmget(5,"Key") resolves "5" to an actual object o,
calls o.get("Key") and packages the return value
6. rJava/JRI transfer
7. rJava/jcall.map.hashMap."Key" unpacks the return value
   and uses rJava functions to convert to a native type (or another "wrapped" object)
Command translation and data flow

One object “ref” is shared between Mayday and R

Example: (int) ret ← hashMap[[ "Key"]]

with class “rlink.hm” and id “5”

1. R resolves operator [[] for class “rlink.hm”
2. [[.rlink.hm(hashMap, "Key") uses rJava:
   .jcall(ref, "hmget", 5, .jnew("Ljava/lang/String","Key"))
3. rJava/JRI transfer
4. ref.hmget(5,"Key") resolves “5” to an actual object o,
   calls o.get("Key") and packages the return value
5. rJava/JRI transfer
6. [[.rlink.hm(hashMap, "Key") unpacks the return value
   and uses rJava functions to convert to a native type (or another “wrapped” object)
One object “ref” is shared between Mayday and R

Example: (int) ret ← hashMap[["Key"]]

with class “rlink.hm” and id “5”

1. R resolves operator [[ for class “rlink.hm”
2. [[.rlink.hm(hashMap, "Key") uses rJava:
   .jcall(ref, "hmget", 5, .jnew("Ljava/lang/String","Key"))
3. rJava/JRI transfer
4. ref.hmget(5,"Key") resolves “5” to an actual object o,
   calls o.get("Key") and packages the return value
5. rJava/JRI transfer
6. [[.rlink.hm(hashMap, "Key") unpacks the return value
   and uses rJava functions to convert to a native type (or another “wrapped” object)
One object “ref” is shared between Mayday and R

Example: (int) ret ← hashMap[['Key']] with class “rlink.hm” and id “5”

1. R resolves operator `[[` for class “rlink.hm”
2. `[[.rlink.hm(hashMap, "Key")` uses `rJava`:
   ```java
   .jcall(ref, "hmget", 5, .jnew("Ljava/lang/String","Key"))
   ```
3. rJava/JRI transfer
4. `ref.hmget(5,"Key")` resolves “5” to an actual object `o`,
calls `o.get("Key")` and packages the return value
5. rJava/JRI transfer
6. `[[.rlink.hm(hashMap, "Key")` unpacks the return value and uses `rJava` functions to convert to a native type (or another “wrapped” object)
Command translation and data flow

Mayday (Java) → interactive R session
VM code (Java) → R functions (R)
VM memory mgr, GC (Java) → R library (native), MM, GC
Java VM core (native), JNI Communication

One object “ref” is shared between Mayday and R

Example: (int) ret ← hashMap[["Key"]]
with class “rlink.hm” and id “5”

1. R resolves operator [[ for class “rlink.hm”
2. [[.rlink.hm(hashMap, "Key") uses rJava:
   .jcall(ref, "hmget", 5, .jnew("Ljava/lang/String","Key"))
3. rJava/JRI transfer
4. ref.hmget(5,"Key") resolves “5” to an actual object o,
calls o.get("Key") and packages the return value
5. rJava/JRI transfer
6. [[.rlink.hm(hashMap, "Key") unpacks the return value
   and uses rJava functions to convert to a native type (or another “wrapped” object)
Mayday (Java) | interactive R session
VM code (Java) | R functions (R)
VM memory mgr, GC (Java) | R library (native), MM, GC
Java VM core (native), JNI Communication

One object “ref” is shared between Mayday and R

Example: (int) ret ← hashMap[["Key"]]

with class “rlink.hm” and id “5”

1. R resolves operator [[] for class “rlink.hm”
2. [[.rlink.hm(hashMap, "Key") uses rJava:
   .jcall(ref, "hmget", 5, .jnew("Ljava/lang/String","Key"))
3. rJava/JRI transfer
4. ref.hmget(5,"Key") resolves “5” to an actual object o,
calls o.get("Key") and packages the return value
5. rJava/JRI transfer
6. [[.rlink.hm(hashMap, "Key") unpacks the return value
   and uses rJava functions to convert to a native type (or another “wrapped” object)
Command translation and data flow

One object “ref” is shared between Mayday and R

Example: (int) ret ← hashMap[["Key"]]

with class “rlink.hm” and id “5”

1. R resolves operator [] for class “rlink.hm”
2. [[].rlink.hm(hashMap, "Key") uses rJava:
   .jcall(ref, "hmget", 5, .jnew("Ljava/lang/String","Key"))
3. rJava/JRI transfer
4. ref.hmget(5,"Key") resolves “5” to an actual object o,
   calls o.get("Key") and packages the return value
5. rJava/JRI transfer
6. [[].rlink.hm(hashMap, "Key") unpacks the return value
   and uses rJava functions to convert to a native type (or another “wrapped” object)
Command translation and data flow

One object “ref” is shared between Mayday and R

Example: \((\text{int}) \text{ret} \leftarrow \text{hashMap}[\text{"Key"]} \) with class “rlink.hm” and id “5”

1. R resolves operator \([[]\) for class “rlink.hm”
2. \([.\text{rlink.hm(hashMap, "Key")}) \text{ uses rJava:} \) .jcall(ref, "hmget", 5, .jnew("Ljava/lang/String", "Key")
3. rJava/JRI transfer
4. ref.hmget(5, "Key") resolves “5” to an actual object o, calls o.get("Key") and packages the return value
5. rJava/JRI transfer
6. \([.\text{rlink.hm(hashMap, "Key")}) \text{ unpacks the return value} \) and uses rJava functions to convert to a native type (or another “wrapped” object)
One object “ref” is shared between Mayday and R

Example: \(\text{(int) ret} \leftarrow \text{hashMap}[\text{"Key"]} \) with class “rlink.hm” and id “5”

1. R resolves operator \(\text{[]}\) for class “rlink.hm”
2. \(\text{[[.rlink.hm(hashMap, "Key")]}\) uses rJava:
   \(\text{.jcall(ref, "hmget", 5, .jnew("Ljava/lang/String","Key")})\)
3. rJava/JRI transfer
4. \(\text{ref.hmget(5,"Key")}\) resolves “5” to an actual object \(o\),
   calls \(o.get("Key")\) and packages the return value
5. rJava/JRI transfer
6. \(\text{[[.rlink.hm(hashMap, "Key")]}\) unpacks the return value
   and uses rJava functions to convert to a native type (or another “wrapped” object)
Operations of interest

- **All objects**
  - summary, print

- **List-like objects**
  - length
  - names, names←
  - `[c]` (select) and `[c]←` (replace)
  - `[c]` (sublist)
  - `lapply`, `sapply`

- **Matrix-like objects**
  - `nrow`, `ncol`, `dim`
  - `rownames`, `colnames`, `rownames←`, `colnames←`
  - `[c]` (submatrix) and `[c]←` (replace)
  - `apply`

- ... and object-specific methods

Overloading depends on context
⇒ We do it dynamically
Operations of interest

- **All objects**
  - summary, print

- **List-like objects**
  - length
  - names, names←
  - `[[` (select) and `[]←` (replace)
  - `[` (sublist)
  - `lapply, sapply`

- **Matrix-like objects**
  - `nrow, ncol, dim`
  - `rownames, colnames`, `rownames←`, `colnames←`
  - `[` (submatrix) and `[]←` (replace)
  - `apply`

- ... and object-specific methods
Operations of interest

- **All objects**
  - summary, print

- **List-like objects**
  - length
  - names, names←
  - `[[]` (select) and `[[<-` (replace)
  - `[` (sublist)
  - lapply, sapply

- **Matrix-like objects**
  - nrow, ncol, dim
  - rownames, colnames, rownames←, colnames←
  - `[` (submatrix) and `[<-` (replace)
  - apply

⇒ We do it dynamically

... and object-specific methods
Operations of interest

- **All objects**
  - summary, print

- **List-like objects**
  - length
  - names, names ←
  - `[[` (select) and `[[<-` (replace)
  - `[` (sublist)
  - `lapply, sapply`

- **Matrix-like objects**
  - nrow, ncol, dim
  - rownames, colnames, rownames ←, colnames ←
  - `[` (submatrix) and `[<-` (replace)
  - `apply`

- ... and object-specific methods

Overloading depends on context
⇒ We do it dynamically
Operations of interest

- **All objects**
  - summary, print

- **List-like objects**
  - length
  - names, names ←
  - `[[]` (select) and `[[<- (replace)`
  - `[` (sublist)
  - lapply, sapply

- **Matrix-like objects**
  - nrow, ncol, dim
  - rownames, colnames, rownames ←, colnames ←
  - `[` (submatrix) and `[[<- (replace)`
  - apply

- ... and object-specific methods

Overloading depends on context
⇒ We do it dynamically
Mayday’s R terminal

- Multi-line editor
  - syntax highlighting
  - auto-completion
  - brace matching
- History
  - multi-line entries
  - storable
- Live list of user objects
Example

Simulated data:
- 3000 rows (probes), 100 columns
- 1000 probes with random oscillations
- 1000 probes each for two different frequencies
Example (2)

```
TestData <- mayday[['Example']];
submatrix <- TestData[['Complete DataSet']]  # <<-- get reference from Mayday
clusterByFFT( submatrix, 50 );  # <<-- select submatrix reference

clusterByFFT <- function( probelist , minsize=10 ,
     parentName="FFT Clustering", prefix="Strongest:" ) {

  f <- probelist[,T]

  # perform fft on each row-vector, find strongest factor
  f.fft<-Mod(t(apply(f,1,fft)))
  f.fftrank<-t(apply(-f.fft[,1],1,rank, ties="first"))
  f.fftrankbest<-apply(f.fftrank,1,
    function(i) which(i==1)+1)

  ds <- getDataSet( probelist );
  group <- addProbelistGroup(ds, parentName, probelist);  # <<-- create hierarchical structure

  factors <- unique(f.fftrankbest);
  clusters <- sapply(factors, function(factor) {
    cluster_i <- names(which(f.fftrankbest==factor))
    if (length(cluster_i)>minsize) {
      name <- paste(prefix,factor)
      return (addProbelist(ds, name, cluster_i, group));
    }
    return(-1);
  });

  # color the results nicely
  clusters <- clusters[clusters>-1];
  callPlugin( ds, "PAS.core.RecolorProbelists", clusters );  # <<-- call another Mayday plugin
  invisible();
}
```
Example (2)

```r
TestData <- mayday[["Example"]]; # <<< get reference from Mayday
submatrix <- TestData[["Complete DataSet"]]; # <<< select submatrix reference
clusterByFFT(submatrix, 50);

clusterByFFT <- function(probelist, msize=10, parentName="FFT Clustering", prefix="Strongest:") {

  f <- probelist[,T] # <<< extract submatrix

  # perform fft on each row-vector, find strongest factor
  f.fft <- Mod(t(apply(f, 1, fft)))
  f.ffdrank <- t(apply(-f.fft, -1, 1, rank, ties="first"))
  f.ffdrankbest <- apply(f.ffdrank, 1,
                         function(i) which(i==1)+1)

  ds <- getDataSet(probelist); # <<< create hierarchical stucture
  group <- addProbelistGroup(ds, parentName, probelist);

  factors <- unique(f.ffdrankbest);
  clusters <- sapply(factors, function(factor) {
    cluster_i <- names(which(f.ffdrankbest==factor))
    if (length(cluster_i)>msize) {
      name <- paste(prefix, factor)
      return (addProbelist(ds, name, cluster_i, group)); # <<< add a new cluster to Mayday
    }
    return(-1);
  })

  # color the results nicely
  clusters <- clusters[clusters>-1];
callPlugin(ds, "PAS.core.RecolorProbelists", clusters); # <<< call another Mayday plugin
  invisible();
}
```
Example (3)

- Complete Dataset [2] 3000
- FFT Clustering [2] 1926
  - Strongest 4 1020
  - Strongest 6 905
- Global 3000

Global This is the global probe list.
Further wishes

- separation of Java and R at the process level
- parallel R instances
- network transparency
- complex R calculations on dedicated machines

Possible solution

Adding an RMI layer → Very few changes needed.
Further wishes

- separation of Java and R at the process level
- parallel R instances
- network transparency
- complex R calculations on dedicated machines

Possible solution

Adding an RMI layer → Very few changes needed.
Summary

- Integration of R and Mayday
- Wrapped Java objects behave like native R objects
- Controlled interface between Mayday and R
- Mode of communication can be changed easily
- Very user-friendly R shell

Mayday is freely available at http://microarray-analysis.org/
Directions for future work

What we can do

- Generic framework for object wrapping
- Register R functions into Mayday’s plugin manager
- Make more Mayday plugins available in R
- Use R to script Mayday

Nice to have

- Multithreaded R core
- More crash-resistant JRI
Directions for future work

What we can do

- Generic framework for object wrapping
- Register R functions into Mayday’s plugin manager
- Make more Mayday plugins available in R
- use R to script Mayday

Nice to have

- Multithreaded R core
- More crash-resistant JRI
Acknowledgements

The Mayday team

The R developers

The rJava/JRI developers

The Federal Ministry of Education and Research
Mayday RLink – The best of both worlds

Florian Battke, Stephan Symons, Kay Nieselt

battke@informatik.uni-tuebingen.de

http://microarray-analysis.org/
Creating overloaded method “X” for objects of class “C” depends on existing definitions of “X”.

No previous definition for X
- X is primitive
- X is an S3 method

⇓
new S3 method: X.C()

X is an S4 method

⇓
new S4 method for C

We automatically determine which is needed
⇒ functions are built dynamically
Creating overloaded method “X” for objects of class “C” depends on existing definitions of “X”.

No previous definition for X
- X is primitive
- X is an S3 method

⇓

new S3 method: X.C()

X is an S4 method

⇓

new S4 method for C

We automatically determine which is needed
⇒ functions are built dynamically
Operator overloading

Creating overloaded method “X” for objects of class “C” depends on existing definitions of “X”.

No previous definition for X
X is primitive
X is an S3 method

⇓

new S3 method: X.C()

X is an S4 method

⇓

new S4 method for C

We automatically determine which is needed
⇒ functions are built dynamically
Creating overloaded method “X” for objects of class “C” depends on existing definitions of “X”.

No previous definition for X
- X is primitive
- X is an S3 method

⇓

new S3 method: X.C()

X is an S4 method

⇓

new S4 method for C

We automatically determine which is needed
⇒ functions are built dynamically
Limitations

Shared process
- limits memory on 32 bit systems
- Makes JVM vulnerable to crashes in R code
- only one instance of at a time
- blocking, no parallel execution

Installation
- Requires C and Java compilers, R headers
- Superuser privileges needed
- Can’t easily be automated
- So far not working on MacOS with 64 bit Java
Limitations

Shared process
- limits memory on 32 bit systems
- Makes JVM vulnerable to crashes in R code
- only one instance of $\text{R}$ at a time
- blocking, no parallel execution

Installation
- Requires C and Java compilers, R headers
- Superuser privileges needed
- Can’t easily be automated
- So far not working on MacOS with 64 bit Java
RMI Connections

We can easily replace the connection between Mayday and R.

Mayday (Java) running RLink server

| + Multiple parallel instances
| + Unlimited memory
| + More stable
| + Installation is much simpler
| − More work to start a session
| − Somewhat slower

interactive R session
rJava running RLink client

RMI Communication
Java VM core (native)
R library (native), MM, GC
We can easily replace the connection between Mayday and R.

- **Mayday (Java)** running RLink server
- **interactive R session**
- **rJava running RLink client**
- **Java VM core (native)**
- **R library (native), MM, GC**

**RMI Communication**

+ Multiple parallel instances
+ Unlimited memory
+ More stable
+ Installation is much simpler
  - More work to start a session
  - Somewhat slower
We can easily replace the connection between Mayday and R.

**Mayday (Java)**
running RLink server

**interactive R session**

**rJava running RLink client**

**Java VM core (native)**

**R library (native), MM, GC**

- Multiple parallel instances
- Unlimited memory
- More stable
- Installation is much simpler
  - More work to start a session
  - Somewhat slower