RedeR: bridging the gap between hierarchical network representation and functional analysis.

Mauro A. A. Castro, Xin Wang, Florian Markowetz *
http://www.markowetzlab.org/software/networks.html
florian.markowetz@cancer.org.uk

March 21, 2012

Contents

*Cancer Research UK - Cambridge Research Institute, Robinson Way Cambridge, CB2 0RE, UK.

1 Overview

RedeR is an R-based package combined with a Java application for dynamic network visualization and manipulation. It implements a callback engine by using a low-level R-to-Java interface to build and run common plugins. In this sense, *RedeR* takes advantage of ***R*** to run robust statistics, while the R-to-Java interface bridge the gap between network analysis and visualization: for **R Developers**, it allows the development of Java plug-ins exclusively using R codes; for **Java Users**, it runs R methods implemented in a stand-alone application, and for **R Users** *RedeR* interactively displays R graphs using a robust Java graphic engine embedded in *R*.

RedeR use different strategies to link R to Java:

- Data interface: implements the callback engine to make calls from R via xml-rpc protocol. It sets *R* as client and *RedeR* as server.
- Graphic interface: implements the callback engine to make calls from Java via dynamic libraries. It wraps R graphics into RedeR classes.

The design of the software is depicted from Figure 1. One unique feature of this concept is how R methods can be wrapped and exported. For example, in a few lines of code *RedeR* sends R methods to the Java app using the *submitPlugin* function, which gives rise to a new Java plugin. Also, complex graphs with many attributes can be transferred from-and-to *R* using *addGraph* and *getGraph* functions.

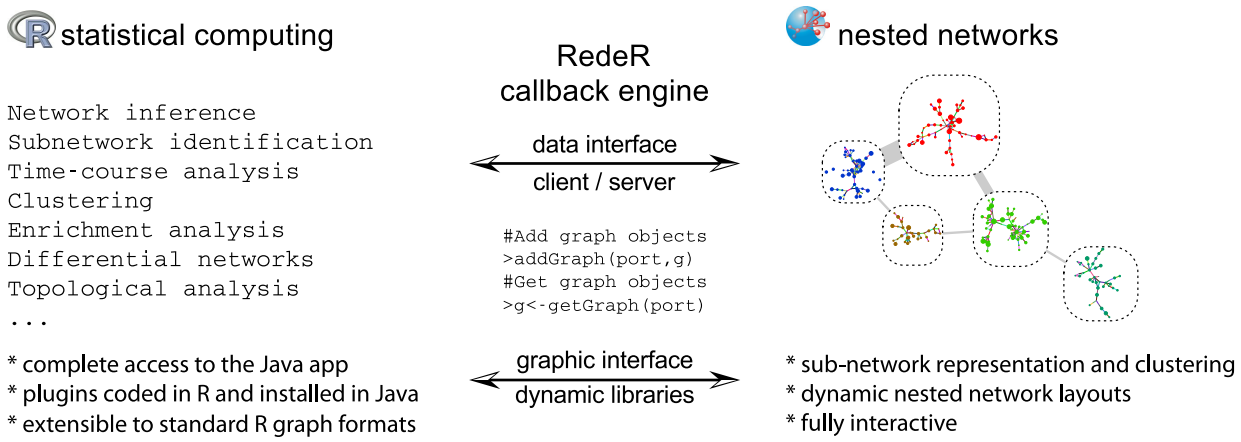


Figure 1: Schematic representation of RedeR calls. In the low-level interface, packages like XMLRPC[1] and rJava[2] are used to link R to Java.

2 Quick start

2.1 Main callback methods

The first step is to build the server port, which will be required in all remote procedure calls. By default the constructor *RedPort* should set all details:

```
>    library (RedeR)
>    rdp <- RedPort ()
```

Next, invoke RedeR using the method *calld*:

```
>    calld(rdp)
```

Within an active interface, then the method 'addGraph' can easily send R graphs to the application. For example, the following chunk adds an *igraph*[3] object:

```
>    g1 <- graph.lattice(c(5,5,5))
>    addGraph( rdp, g1, layout.kamada.kawai(g1) )
```

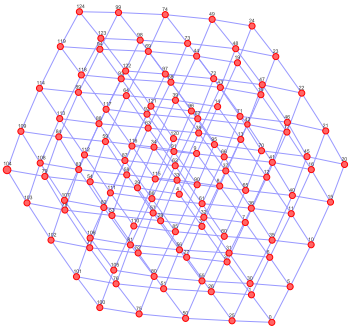


Figure 2: A toy example added to *RedeR* by the *addGraph* function.

Conversely, RedeR graphs can be transferred to R and wrapped in *igraph* objects:

```
>    g2 <- getGraph(rdp)
>    resetd(rdp)
```

The interface accepts additional graph attributes, as for example edge direction, edge width, edge weight, node shape, node size, node color etc. In *igraph* objects, vertex and edge attributes can be assigned as arbitrary R objects. In order to pass these extensible features to *RedeR* the attributes must be provided in a valid syntax. [1]

Another strategy is to wrap graphs into containers and then send it to the Java app. Next, the subgraphs g3 and g4 are assigned to different nested structures (Fig.3).

```
>    g3 <- barabasi.game(10)
>    g4 <- barabasi.game(10)
>    V(g3)$name<-paste("sn",1:10,sep="")
```

[1] See *getGraph* and *addGraph* specification for additional details.

```
>    V(g4)$name<-paste("sm",1:10,sep="")
>    addGraph(rdp, g3, isNest =TRUE, gcoord=c(25,25), gscale=50)
>    addGraph(rdp, g4, isNest =TRUE, gcoord=c(75,75), gscale=50 )
```

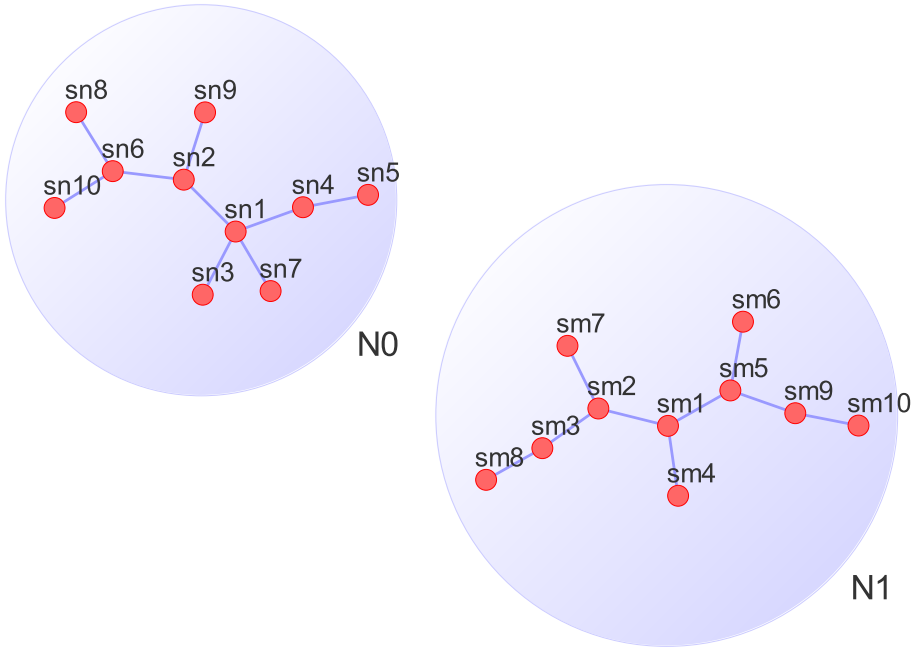


Figure 3: Graphs nested in *RedeR* by the command *addGraph*.

In this case, the subgraphs can be handled apart from each other. For example, the following chunk selects all nodes assigned to the container "N0" and then gets back the subgraph (the selection step can also be done interactively!).

```
>    selectNodes(rdp,"N0")
>    g5 <- getGraph(rdp, status= "selected")
>    resetd(rdp)
```

As a suggestion, try some RedeR features in the Java side (e.g. open samples s2 or s3 in the main panel and enjoy the dynamic layout options!).

2.2 Interactive work

The next chunk generates a scale-free graph according to the Barabasi-Albert model[3] and sends the graph to RedeR without any layout information.

```
>    g6 <- barabasi.game(500)
>    addGraph(rdp, g6, zoom=20)
```

Then using the "relax" options available in the app you can tune the graph layout as presented in Figure 4.

```
>    relax(rdp,p2=400,p5=30,ps=T)
```

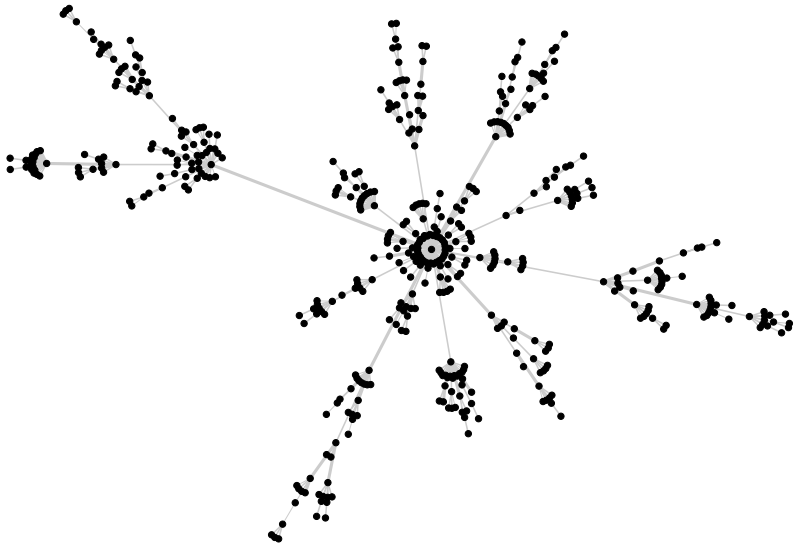
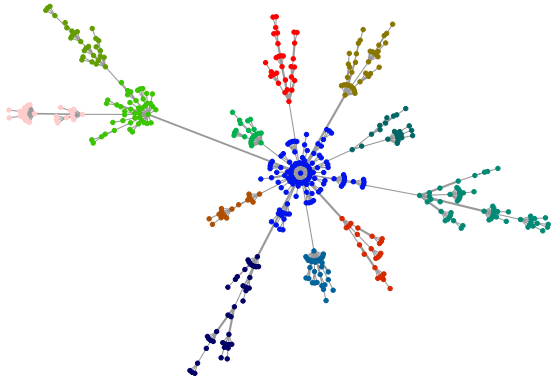


Figure 4: Scale-free graph according to the Barabasi-Albert model[3].

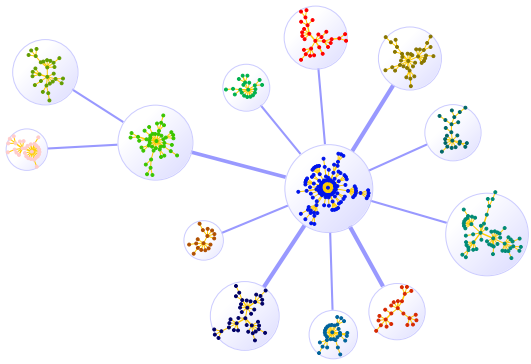
In Figure 5a the same graph is used to exemplify the community structure mapped by the edge-betweenness function available in RedeR. In Figure 5b these communities are nested to containers, which are objects of the same class of the nodes but with additional behaviors: it can be hidden and anchored to the main panel (Fig.5c). You can build these containers either using *R* or *Java* functions (see options available in the *clustering* main menu and in the shortcuts of nested objects, i.e., right-click a container!).

For the next example you will need to reproduce in *RedeR* app the graph from Figure 5b (or any graph with containers), then select one of the communities and run the chunk below: a simple degree distribution should be plotted in the R side (Fig.6). This is the first step to illustrate how to build an interactive plugin.

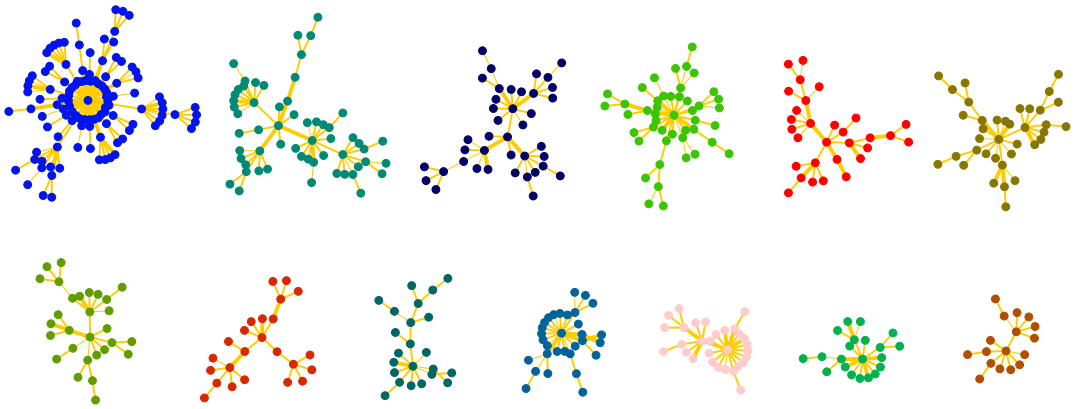
```
>    g <- getGraph(rdp, status= "selected")
>    if(vcount(g)>0)plot(degree.distribution(g), xlab = "k", ylab = "P(k)", pch=19)
>    resetd(rdp)
```



(a) Communities



(b) Graphs into containers



(c) Subnetworks

Figure 5: Community structure: (a) subgraphs detected based on edge betweenness; (b) nested communities into containers; (c) subnetworks in hidden containers.

3 Workflow samples

This section provides a sequence of steps that illustrate how users might integrate its own pre-processed data with a given network in order to visualize subgraphs and nested networks. The full description soon will be available in a new communication!

3.1 Subgraphs

- 1 - start the app (i.e. 'calld' method).
- 2 - get a dataframe and an interactome:

```
> data(ER.limma)
> data(hs.inter)
> dt <- ER.limma
> gi <- hs.inter
```

- 3 - extract a subgraph and set attributes to RedeR (i.e. logFC from t3-t0 contrast):

```
> gt3  <- subg(g=gi, dat=dt[dt$degenes.t3!=0,], refcol=1)
> gt3  <- att.setv(g=gt3, from="Symbol", to="nodeAlias")
> gt3  <- att.setv(g=gt3, from="logFC.t3...t0", to="nodeColor", breaks=seq(-2,2,0.4), pal=2)
```

ps. some genes will not be found in the interactome!

- 4 - extract another subgraph and set attributes to RedeR (i.e. logFC from t6-t0 contrast):

```
> gt6  <- subg(g=gi, dat=dt[dt$degenes.t6!=0,], refcol=1)
> gt6  <- att.setv(g=gt6, from="Symbol", to="nodeAlias")
> gt6  <- att.setv(g=gt6, from="logFC.t6...t0", to="nodeColor", breaks=seq(-2,2,0.4), pal=2)
```

- 5 - extract another subgraph and set attributes to RedeR (i.e. logFC from t12-t0 contrast):

```
> gt12 <- subg(g=gi, dat=dt[dt$degenes.t12!=0,], refcol=1)
> gt12 <- att.setv(g=gt12, from="Symbol", to="nodeAlias")
> gt12 <- att.setv(g=gt12, from="logFC.t12...t0", to="nodeColor", breaks=seq(-2,2,0.4), pal=2)
```

- 6 - add subgraphs to the app:

```
> n0 <- addGraph(rdp, gt3, gcoord=c(10,25), gscale=20, isNest=TRUE, theme='tm1', zoom=30)
> n1 <- addGraph(rdp, gt6, gcoord=c(20,70), gscale=50, isNest=TRUE, theme='tm1', zoom=30)
> n2 <- addGraph(rdp, gt12, gcoord=c(70,55), gscale=80, isNest=TRUE, theme='tm1', zoom=30)
```

- 7 - nest subgraphs (i.e. overlap time-series):

```
> nestNodes(rdp, nodes=V(gt3)$name, parent=n1, theme='tm2')
> nestNodes(rdp, nodes=V(gt6)$name, parent=n2, theme='tm2')
```

- 8 - assign edges to containers

```
> mergeOutEdges(rdp)
```

- 9 - relax the network

```
> relax(rdp,50,400)
```

- 10 - add a color legend (other types are available): [2]

```
> scl <- gt3$legNodeColor$scale
> leg <- gt3$legNodeColor$legend
> addLegend.color(rdp, colvec=scl, labvec=leg, title="node color (logFC)")
```

[2] Legends are set only via command line in the current version.

- 11 - select a gene:

```
> selectNodes(rdp,"RET")
```

- 12 - reset graph:

```
> resetd(rdp)
```

3.2 Nested networks and clustering

- 1 - get a dataframe and an igraph object:

```
> data(ER.deg)
> dt <- ER.deg$dat
> sg <- ER.deg$ceg
```

- 2 - map the dataframe to the graph:

```
> sg <- att.mapv(sg, dat=dt, refcol=1)
```

- 3 - set attributes to RedeR (i.e. gene symbols and two available numeric data):

```
> sg <- att.setv(sg, from="Symbol", to="nodeAlias")
> sg <- att.setv(sg, from="logFC.t3...t0", to="nodeColor", breaks=seq(-1,1,0.2), pal=2)
> sg <- att.setv(sg, from="ERbdist", to="nodeSize", nquant=10, isrev=TRUE, xlim=c(5,40,1))
```

- 4 - add graph to the app:

```
> addGraph(rdp,sg)
```

- 5 - compute a hierarchical clustering using standard R functions:

```
> hc <- hclust(dist(get.adjacency(sg, attr="weight")))
```

- 6 - map the hclust object onto the network (pvclust objects are also compatible!):

```
> nesthc(rdp,hc, cutlevel=3, nmemb=5, cex=0.3, labels=V(sg)$nodeAlias)
```

...at this point nested objects from the network should also be mapped to a dendrogram! different levels of the nested structure can be set by the nesthc method. Additionally, significant clusters can be computed by the pvclust package, which is already compatible with RedeR interface.

- 7 - assign edges to containers:

```
> mergeOutEdges(rdp)
```

- 8 - relax the network:

```
> relax(rdp)
```

- 9 - add color and size legends: [3]

```
> scl <- sg$legNodeColor$scale
> leg <- sg$legNodeColor$legend
> addLegend.color(rdp, colvec=scl, labvec=leg, title="diff. gene expression (logFC)")

> scl <- sg$legNodeSize$scale
> leg <- sg$legNodeSize$legend
> addLegend.size(rdp, sizevec=scl, labvec=leg, title="bd site distance (kb)")
```

- 10 - reset graph:

```
> resetd(rdp)
```

[3] Legends are set only via command line in the current version.

4 Plugin builder

RedeR plug-ins have two main sections: methods and add-ons. The 'methods' section can be regarded as the plug-in trigger. When installed in the Java app, this trigger is used to start a given analysis by unfolding the R expressions wrapped in the methods. Add-ons use the same strategy, but remains hidden in the app – and it is optional. Formal functions can be passed to add-ons as additional arguments. Prior to the main call, all functions are automatically loaded in R, making the source code available to the subsequent analysis.

A simple example

- Wrap methods in functions

  Here, the degree distribution is used in the same way as illustrated in the previous section but – prior to execute the analysis – the plugin requires two commands: *RedPort* to set the interface, and *dynwin* to wrap R graphics in RedeR Java classes. Additionally, the "JavaGD" package should be available in R. [4]

```
>       mt1 <- function()
+       {
+           rdp <- RedPort('MyPort')
+           dynwin(rdp)
+            g <- getGraph(rdp, status= "selected")
+         if(vcount(g)>0)plot(degree.distribution(g), xlab = "k", ylab = "P(k)", pch=19)
+        }
```

- Initiate plugin skeleton:

  Now the method *mt1* is added to the *PluginBuilder*, and is ready to be sent to the application.

```
>     plugin <- PluginBuilder(title="MyPlugin", allMethods=list(mt1=mt1))
```

- Submit new plugins to RedeR:

  The submission is straightforward: in the Java side, the plugin will be displayed in the main menu provided that the interface is properly set to find *R*.

```
>   submitPlugin(rdp, plugin)
>   updatePlugins(rdp)
>   exitd(rdp)
```

[4] Due to build/check errors, this package was not available at the occasion of Bioconductor release; therefore, it had to be removed from RedeR dependencies. It can be installed by command line, as indicate in *Installation* section.

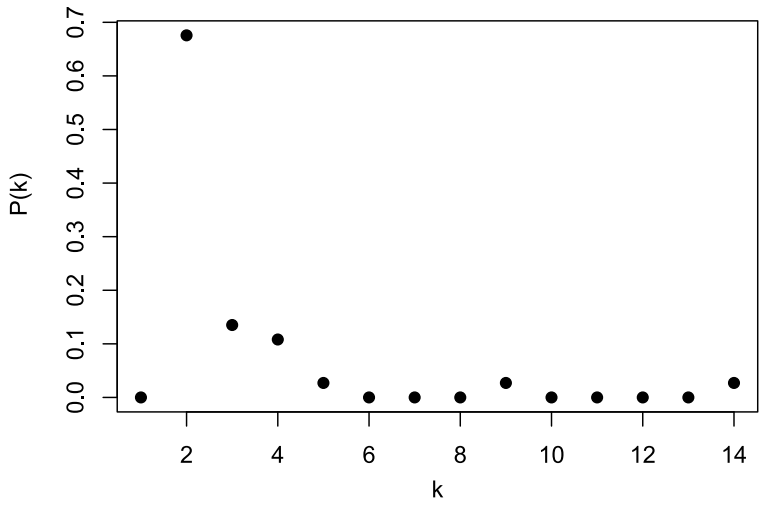


Figure 6: RedeR plugins: a simple example to illustrate the concept (degree distribution of selected subgraphs). One subgraph from Figure 5b was interactively selected in *RedeR* and then plotted either in the R side or in the Java side of the interface using the plugin described above.

5 Installation

Release version

The RedeR package is freely available from Bioconductor at `http://bioconductor.org/packages/release/bioc/html/RedeR.html`.

Developing version

The developing version is available at `http://bioconductor.org/packages/devel/bioc/html/RedeR.html`. In order to install RedeR devel please download the *Package Source* (the .tar.gz file!!) and install all dependences in R:

```
> install.packages(c("RCurl", "XML", "rJava", "igraph", "JavaGD"), repos = "http://cran.r-project.org")
> install.packages("XMLRPC", repos = "http://www.omegahat.org/R", type = "source")
> install.packages('/devel.tar.gz', type='source')  ## i.e. the path to the downloaded .tar.gz file!
```

The JAVA application

The RedeR jar file is already included in the R package and, as usual, to run Java applications your system should have a copy of the JRE (Java Runtime Environment, version>= 5). The RedeR software can be used as a stand-alone application, or even embedded in other softwares, but in order to use plugins the interface must be set properly in the Java side to find R (e.g. path to R home). This should be a simple task, but if your R environment deviates a lot from the default installation then it might be necessary to add some path manually. If you find any difficulty, please contact us.

6 Session information

```
R version 2.14.2 (2012-02-29)
Platform: i386-pc-mingw32/i386 (32-bit)

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

other attached packages:
[1] RedeR_1.0.14   XMLRPC_0.2-4   XML_3.9-4.1    RCurl_1.91-1.1 bitops_1.0-4.1
[6] igraph_0.5.5-4

loaded via a namespace (and not attached):
[1] tools_2.14.2
```

References

[1] Duncan Temple Lang. *XMLRPC: Remote Procedure Call (RPC) via XML in R*, 2010. R package.

[2] Simon Urbanek. *rJava: Low-level R to Java interface*, 2010. R package.

[3] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.