

v1.4

13/04/2011

# CytoscapeRPC Tutorial

Jan Bot

## About

This tutorial gives an overview of how Cytoscape and CytoscapeRPC can be used to visualize networks. It assumes you have Cytoscape and CytoscapeRPC (version 1.4 or higher) installed and working. Details on how to do this can be found on the [NBIC wiki](#). As all the function calls are done through XML-RPC the programming language you choose to do these exercises with doesn't really matter as long as it has an XML-RPC client implementation. Most major programming languages (Java, C, Python, Perl, PHP, Ruby, R etc.) have such an implementation. For a complete overview of programming languages which support XML-RPC see [xmlrpc.com](http://xmlrpc.com).

[Cytoscape](#) is a graph visualization and analysis tool. It is especially well suited for visualization of interactions between biological entities.

[CytoscapeRPC](#) is a plugin for Cytoscape which allows users to create, modify and query Cytoscape networks. It provides a relatively simple API through a XML-RPC interface so that users can access the Cytoscape functionality from all major (and many minor) programming languages.

## Notes

Cytoscape and CytoscapeRPC are implemented in Java and therefore very picky about variable types, *e.g.* when a double is needed you cannot enter an integer. Keep this in mind when using the plugin from a dynamic programming language. Furthermore, parameters are often provided as lists or hashes (a.k.a. maps or dictionaries). Getting these hashes right can be challenging. Take a look at the plugin documentation when you're not sure what type of data is needed.

Every CytoscapeRPC function has a return value, this can either be something meaningful like the identifier of an object you just created, or a boolean indicating that the function completed successfully. The latter does not really contain any information but is needed as most XML-RPC clients require a return value from every function call.

When using the javadoc of CytoscapeRPC, keep in mind that the only class that is available through XML-RPC is the `CytoscapeRPCCallHandler`. All the other classes implement functions that are called by the handler class.

## Interactive use

It's useful to store CytoscapeRPC calls in scripts so that they can be reused. However, to try out the plugin and play around with Cytoscape it is easier to do so interactively. The [Ipython](#) shell is especially well suited for this as it supports tab-completion of the XML-RPC commands. When using Ruby the commands can be interactively handled using the IRB shell. The plug-in is partially self documenting, use the `getFunctions` function to get a list of all the available functions in the handler class.

## Definitions

Cytoscape consists of a number of parts with which you should be familiar. A basic graph with nodes and edges is called a *CyNetwork* or network for short. One network keeps track of all the nodes and edges that belong to it. Cytoscape can manage multiple networks simultaneously, however, the underlying code is unable to make the distinction so all the nodes and edges belong to a common `RootGraph`. One of the problems with this approach is that nodes and edges are never 'really' removed. So when you are running out of memory your only way out is restarting Cytoscape<sup>1</sup>.

---

1 This is no longer true, CytoscapeRPC (from version 1.3) supports clearing the `RootGraph` using the `resetRootGraph` function. It's not something that Cytoscape natively supports so your mileage may vary.

The visualization of a network is done by the *CyNetworkView*, one network can have multiple views. Nodes and edges are the basic elements of a network and don't need any further explanation. Nodes, edges and networks can have attributes, these can be of type Integer, String, Boolean, Double, Array and Map. We'll just be using the first four as they are easier to deal with. The attributes of nodes and edges can influence what these nodes and edges look like, as we will see in exercise 6.

## Documentation

Multiple types of documentation are available, see the following links:

1. Javadoc of the plugin: <http://bioinformatics.tudelft.nl/content/cytoscapercp>
2. CytoscapeRPC wiki page: <https://wiki.nbic.nl/index.php/CytoscapeRPC>
3. CytoscapeRPC wiki install page: [https://wiki.nbic.nl/index.php/CytoscapeRPC\\_install](https://wiki.nbic.nl/index.php/CytoscapeRPC_install)
4. Cytoscape documentation: <http://cytoscape.org/features2.php>

## Activating the plugin

A not really difficult but certainly necessary step is activating the plug-in. If the plugin is installed correctly the 'Plugins' menu should contain an option '**Cytoscape RPC**'. From this menu select the '**Activate CytoscapeRPC**' option and keep the default port. Make sure there are no firewall rules which interfere with local traffic to that port.

The '**Deactivate CytoscapeRPC**' option in the CytoscapeRPC plugin menu actually does what the name implies.

With the '**Settings**' option you can tell CytoscapeRPC to start directly upon Cytoscape startup. You can also specify the port on which CytoscapeRPC needs to listen.

Mac users might experience problems when activating the plugin. This is due to an old Java version which is supplied with MacOS X. If the plugin cannot be started use the 'Write default config' option in the CytoscapeRPC plugin menu and restart Cytoscape.

## Exercise 1: Setting up the connection

How you should set up the connection to CytoscapeRPC depends on the programming language that you use and the module / package which supplies XML-RPC functionality. Programming languages such as Perl, R, PHP and Java have multiple XML-RPC implementations. As we cannot list them all only a few examples are given. See the CytoscapeRPC page on the NBIC wiki for more examples and feel free to add your own.

### Python example

```
1. import xmlrpclib
2. server = xmlrpclib.ServerProxy("http://localhost:9000")
```

The xmlrpclib is part of the standard python distribution so no additional packages have to be installed.

### Python 3 example

```
1. from xmlrpc import client
2. server = client.ServerProxy("http://localhost:9000")
```

All the Python 3 commands are the same as those used in the 2.x version.

### Perl example

```
1. use Frontier::RPC2;
2. use Frontier::Client;
3. my $url = "http://localhost:9000/Cytoscape";
4. my $client = Frontier::Client->new(url => $url, debug => 0);
```

The Frontier packages have to be installed as they are not included in the standard Perl distribution. Use [CPAN](#) or the package manager of your distribution to install the necessary modules.

## Exercise 2: Testing the connection

The first function you should test is the test function. This function takes no arguments and should return 'It works!' after calling. All CytoscapeRPC functions are stored in the Cytoscape namespace so the call should be prefixed with 'Cytoscape'. Once this is working you can begin creating your own network.

### Perl example

```
1. my $response = $client->call('Cytoscape.test');
2. print $response . "\n";
```

### Python example

```
1. print server.Cytoscape.test()
```

### Ruby example

```
1. puts server.call("Cytoscape.test")
```

From now on all exercises will contain only one example. The goal of the rest of the tutorial is for you to rewrite these examples in the programming language of your choice.

## Exercise 3: Creating a network

To create a network use the `createNetwork` function. It takes one string argument indicating the title of the network. The function itself returns a string giving you an ID to identify the network with. You can use the ID to modify that particular network in other functions. Most functions also support not specifying the network ID and assume that you want to modify the current (selected) network. When you call the function a new window is opened within Cytoscape which has the title you specified earlier.

You can now start adding nodes, to do this use the `createNode` or `createNodes` functions. The `createNode` function takes a string as input and returns a string as output. You can optionally specify, as a first argument, the network to which the node should be added. The `createNodes` function works the same but, as the plural form implies, takes a list of identifiers (strings) as input.

Once there are nodes you can begin creating edges. To do this use the `createEdge` or `createEdges` functions. The `createEdge` function takes a number of arguments: (optional) the network id (string), from node id (string), to node id (string), the type of the link (string) and whether the connection is directed (boolean). The `createEdges` function takes the same arguments but then in list form and has an additional parameter 'forgive' (boolean) which indicates whether to stop or continue when a node is not found. There is also a shortcut available: use the `createEdges` function with two lists (from and to node IDs) to create edges. All these functions return the identifiers of the edges as they are known in Cytoscape. You'll need these later on to be able to add attributes to the edges.

### Perl code to create a network:

```
1. # A coder is needed to deal with Perls automatic type casting.
2. my $coder = Frontier::RPC2->new;
3.
4. # Create a network
5. my $networkID = $client->call('Cytoscape.createNetwork', 'Test network');
6.
7. # Add nodes
8. my $done;
9. $done = $client->call('Cytoscape.createNodes',
10.     $coder->string("$networkID"), # supply networkID
11.     ["a", "b", "c", "d", "e", "f", "g"] # list of node names
12. );
13. my $nrNodes = $client->call('Cytoscape.countNodes');
14. print "There are now ", $nrNodes, " nodes present in Cytoscape.\n";
15.
16. # Add edges
17. $done = $client->call('Cytoscape.createEdges',
18.     ["b", "b", "c", "d", "d", "d", "d"], # from nodes
19.     ["b", "c", "d", "a", "e", "f", "g"] # to nodes
20. );
21. my $nrEdges = $client->call('Cytoscape.countEdges');
22. print "There are now ", $nrEdges, " edges present in Cytoscape.\n";
```

## Exercise 4: Applying a layout

The network you just created does not contain any location information so Cytoscape does not yet know where to put the nodes and edges. Layout algorithms compute how the nodes should be laid out in 2d space. You can perform a layout by using the Layout menu but you can also do it via CytoscapeRPC.

The `performLayout` function takes as arguments the network ID of the network that you want to layout and the name of the algorithm that should be used. The problem here is that the name of the algorithm is actually unknown and does not necessarily correspond to the name given in the Layout menu. To get all the names of the layouts that you can use call the `getLayoutNames` function or see appendix A. Due to licensing issues none of the yFiles layouts can be used.

Play around with different layouts to familiarize yourself with them.

### Python code to perform a layout

```
1. # Get the current network index
2. networkid = server.Cytoscape.getCurrentNetworkIndex()
3. # Show the names of the layouts which are available to us.
4. print server.Cytoscape.getLayoutNames()
5. # Perform a layout (in this case the 'kamada-kawai' layout)
6. server.Cytoscape.performLayout(networkid, 'Kamada-Kawai')
```

## Exercise 5: Adding attributes

You can add attributes to nodes and edges. These attributes can be used by plugins to *e.g.* calculate enrichment with. They can also be used to change the way nodes and edges are displayed. This will be shown in this and the following exercise.

To add attributes to nodes you can use the `addNodeAttribute` and `addNodeAttributes` functions. `addNodeAttribute` takes as inputs the name of the attribute to add (string), the type of the attribute (one of [STRING, INTEGER, FLOATING, BOOLEAN]), the identifier of the node and the value of the attribute. `addNodeAttributes` works the same but takes a hash of (node identifier – value) pairs to set the attributes and an additional `forgive` parameter which tells the plugin to report an error when a node cannot be found. For edges it works the same but then with the `addEdgeAttribute` and `addEdgeAttributes` respectively. The same goes for network attributes. Try adding some attributes to the edges.

There is also an alternative implementation to add attributes to nodes, edges and networks, these functions all have the following form:

```
add[VarType][AttrType]Attribute(s)
```

where `VarType` is the type of variable [Boolean, Double, Integer, String] and `AttrType` is one of [Node, Edge, Network]. These functions take the following parameters: the name of the attribute, IDs of the entities to add the attributes to and a list of values of the type specified in the function name. For example, to add integer attributes to nodes you can use:

```
addIntegerNodeAttributes('number', ['a', 'b', 'c'], [10, 100, 2])
```

To see the attributes click on the '**Select Attributes**' button in the Data Panel (bottom panel) and select the attributes to display.

## Ruby code to add attributes

```
1.  server.call(  
2.      "Cytoscape.addNodeAttribute",  
3.      "MyAttribute",  
4.      "STRING",  
5.      "a",  
6.      "aa"  
7.  )  
8.  
9.  server.call(  
10.     "Cytoscape.addNodeAttributes",  
11.     "YourAttribute",  
12.     "STRING",  
13.     {  
14.         "a"=>"one", "b"=>"one", "c"=>"two",  
15.         "d"=>"two", "e"=>"three", "f"=>"three",  
16.     },  
17.     true  
18.  )  
19.  server.call(  
20.     "Cytoscape.addNodeAttributes",  
21.     "pvalue",  
22.     "FLOATING",  
23.     {  
24.         "a"=>-0.5, "b"=>0.98, "c"=>0.12, "d"=>0.002,  
25.         "e"=>0.2, "f"=>0.33, "g"=>1.5  
26.     },  
27.     true  
28.  )
```

## Exercise 6: Changing the visual style

Visual styles influence the way nodes and edges are displayed based on attributes of those nodes and edges. There are multiple types of visual styles, you can try them out by clicking on the VizMapper tab in the left panel.

The *continuous mapper* changes a *visual property* of a node or edge by interpolating between multiple values, this could *e.g.* be the color or size of a node which is adjusted based on the importance of that node. This mapper thus works with ranges in which the attribute value that a certain node has should be within.

The *discrete mapper* takes (attribute value – visual property) pairs to change the appearance of nodes and edges. This mapper can *e.g.* be used to change the shape of a node (triangle, square, etc.) based on the type of the biological entity (gene, protein, etc.).

Now that we have added attributes we can use them to influence the node shape. We'll use the `createDiscreteMapper` function to do this. It takes the following arguments: the name of the style you want to change, the attribute name to use for the mapping, the visual property to change, the default shape to use and a map with (attribute value – visual property) pairs.

You can get a list of the available node shapes by calling the `getNodeShapeNames` function.

## Python code to set a discrete node shape visual style

```
1. server.Cytoscape.createDiscreteMapper(  
2.     'default',          # VizMap to change  
3.     'YourAttribute',   # Attribute name to use with the mapper  
4.     'Node Shape',  
5.     'triangle',        # Default value  
6.     {                   # (attribute value-shape) map  
7.         'one':'octagon', 'two':'round_rect', 'three':'rect'  
8.     },  
9. )
```

Note that node 'g' has no attribute 'YourAttribute' and will therefore get the default shape (triangle).

The same can be done using a continuous mapper. Here you specify multiple points between which Cytoscape can interpolate a visual attribute. In this example we'll change the node color.

## Python code to set a continuous visual style

```
1. server.Cytoscape.createContinuousMapper(  
2.     'pvalue',  
3.     'Node Color',  
4.     [0.0, 1.0],  
5.     ['#000000', '#FF0000', '#FFFFFF', '#000000']  
6. )
```

Here, the first and last entries ('#000000') indicate black and give colors to the nodes which have values outside the specified range while the middle entries ('#FF0000' and '#FFFFFF') indicate the colors between which Cytoscape should interpolate. The black color is used to indicate that the values for that attribute do not make sense: the p-values we use to color the nodes with should be between 0 and 1.

## Exercise 7: Overwriting the visual style manually

Sometimes it's convenient to be able to change the appearance of nodes and edges without using a VizMap (as we did in the previous exercise). Cytoscape provides some functions to bypass the visual mapping and change the visual properties directly. It is important to know that the changes you make using this method *will not be overwritten* by default when using a VizMap. You'll have to manually remove these attributes if you want the VizMap to control that property again. This only applies to nodes and edges for which you have set this property, all other nodes and edges will still display as configured by the VizMap.

To overwrite the visual style of one node you can use the function `setNodeProperty`. This function overwrites one visual property for one node. To get a list of all the visual properties that can be changed use the function `getVisualStyleModifiables`, it returns a list of the names of all properties that can be overwritten. The `setNodeProperty` expects the following parameters: node ID (string), name of the property to change (string) and the property value (string).

To make the change visible the `redraw` function needs to be called. This is not done in the `setNodeProperty` function because that would generate too much overhead when overwriting many visual properties.

The overwritten properties can be viewed in the data panel by adding them to the variables that should be displayed (click on the '**Select Attributes**' button).

## Perl code to overwrite a visual style

```
1. $done = $client->call(
2.     'Cytoscape.setNodeProperty', # function to call
3.     'a',                          # node which needs to be changed
4.     'Node Shape',                  # visual property to change
5.     'triangle'                     # value of the visual property
6. );
7. $done = $client->call('Cytoscape.redraw');
```

## Exercise 8: Queries & Selections

You can use the plugin to query Cytoscape so you can use the information from Cytoscape in your script. You can query the IDs of all nodes (`getNodeIds`) or the selected nodes (`getSelectedNodes`). The same holds for edges. You can also query which nodes are the neighbors of one or more nodes by calling `getNodeNeighbors` and `getNodeNeighbors` respectively.

You can use this information to *e.g.* change the selection using `setNodeSelection`.

### Ruby example of changing the selection

```
1. # Setup code
2. server.call("Cytoscape.clearSelection")
3. server.call("Cytoscape.setSelectedNodes", ['a', 'b'])
4. server.call("Cytoscape.redraw")
5. selectedNodes = server.call("Cytoscape.getSelectedNodes")
6. puts "Selected nodes: #{selectedNodes * ' '}"
7.
8. # Select the neighbors of the currently selected nodes
9. neighbors = server.call("Cytoscape.getNodeNeighbors",
10.    server.call("Cytoscape.getSelectedNodes"))
11. server.call("Cytoscape.setSelectedNodes", neighbors)
12. server.call("Cytoscape.redraw")
13. selectedNodes = server.call("Cytoscape.getSelectedNodes")
14. puts "Selected nodes: #{selectedNodes}"
15.
16. # Get some network properties
17. nrNodes = server.call("Cytoscape.countNodes")
18. nrEdges = server.call("Cytoscape.countEdges")
19. puts "Number of nodes: #{nrNodes}"
20. puts "Number of edges: #{nrEdges}"
```

## Exercise 9: Exporting a view

Cytoscape allows you to save a network as a number of different image formats. This can be done with CytoscapeRPC through the `exportView` function. This function takes the filename (string), filetype (string) and scaling factor (double) as inputs. The pictures end up in the Cytoscape directory.

### Python example of exporting a view

```
1. server.Cytoscape.exportView(  
2.     "test.jpg", # filename  
3.     "jpg", # file type  
4.     2.0 # scaling factor  
5. )
```

The output of this command should look similar to Illustration 1.

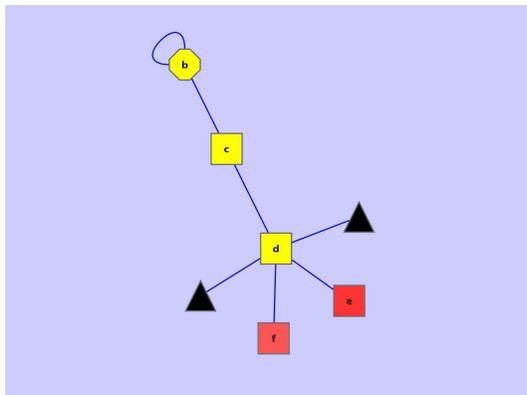


Illustration 1: Output network

You can use the Linux commandline tool 'convert' to create a movie from a number of image files. Doing this for different selections, VizMappers and layouts is left as an exercise for the reader.

## Exercise 10: Collaboration

CytoscapeRPC is not picky about where the inputs come from. Use your social engineering skills to acquire the ip-address of one of your neighbors and send him a message (use the `showMessage` function).

The default behavior of CytoscapeRPC is to accept incoming calls from any computer. The plugin can be instructed to only accept connections from the machine that Cytoscape is running on by selecting the 'Local only' option in the configuration menu.

This was the last exercise, the following pages list some useful functions which help you get more out of Cytoscape.

## Additional functions

### *Modifying network views & controlling windows*

When you are creating multiple networks with associated views you want to be able to control which window is currently being shown. To raise a window to the top you can use the `'raiseNetworkView(String networkID)'` function. To modify the size of a view use `'resizeNetworkView(String networkID, int width, int height)'`. You can close the view without destroying the network: `'closeNetworkView(String networkID)'` or close all network views: `'closeAllNetworkViews()'`.

When you are changing the size of the view you will probably also want to change what part of the network is being shown. To show the entire network use `'fitContent(String networkID)'` to show only the selected nodes use: `'fitSelectedContent()'`.

### *Setting layout parameters*

Some layouts require that the user sets a number of parameters. This can also be done using CytoscapeRPC. You've seen in exercise 4 that you can query all the layout names using `'getLayoutNames()'`, the same can be done for the parameters of an individual layout: `'getLayoutParameters(String layoutName)'`, this returns a list of property names. To change the parameters you will need to know what the current values and the types of the properties are. Use `'getLayoutPropertyValues(String layoutName)'` and `'getLayoutPropertyTypes(String layoutName)'` respectively.

## Appendix A: CytoscapeRPC Constants

### *Layout names*

The output of the `getLayoutNames` function. This is the list of layouts which are available in Cytoscape.

<code>jgraph-circle</code>	<code>Fruchterman-Rheingold</code>	<code>isom</code>	<code>jgraph-tree</code>
<code>attribute-circle</code>	<code>Kamada-Kawai</code>	<code>jgraph-moen</code>	<code>force-directed</code>
<code>jgraph-annealing</code>	<code>jgraph-gem</code>	<code>jgraph-sugiyama</code>	<code>degree-circle</code>
<code>jgraph-radial-tree</code>	<code>hierarchical</code>	<code>attributes-layout</code>	<code>jgraph-spring</code>
<code>Kamada-Kawai-Noweight</code>	<code>circular</code>	<code>grid</code>	

### *Node shape names*

The output of the `getNodeShapeNames` function. This is list of node shapes available in Cytoscape.

<code>trapezoid</code>	<code>rect_3d</code>	<code>trapezoid_2</code>
<code>round_rect</code>	<code>diamond</code>	<code>rect</code>
<code>ellipse</code>	<code>parallelogram</code>	<code>vee</code>
<code>triangle</code>	<code>octagon</code>	<code>hexagon</code>

### *Arrow shape names*

The output of the `getArrowShapeNames` function.

<code>No arrow</code>	<code>T</code>
<code>Diamond</code>	<code>Circle</code>
<code>Delta</code>	<code>Half Arrow Top</code>
<code>Arrow</code>	<code>Half Arrow Bottom</code>