GraphRight

Open Application Program Interface

Introduction

Imagine adding the power of a graphics package to any custom application. People who are familiar with Improv will remember how Improv and Presentation Builder worked together as though they were a single application, well GraphRight's API allows for any custom application to take full advantage of a rich charting and graphing package in the same manner. GraphRight already works with popular packages such as Mathematica, Mesa, and will work as a replacement for Presentation Builder for Improv.

One big advantage of our API is that when Watershed Technologies improves GraphRight, you will not have to re-link in any libraries or palettes into your custom applications, everything will "just work" when you upgrade.

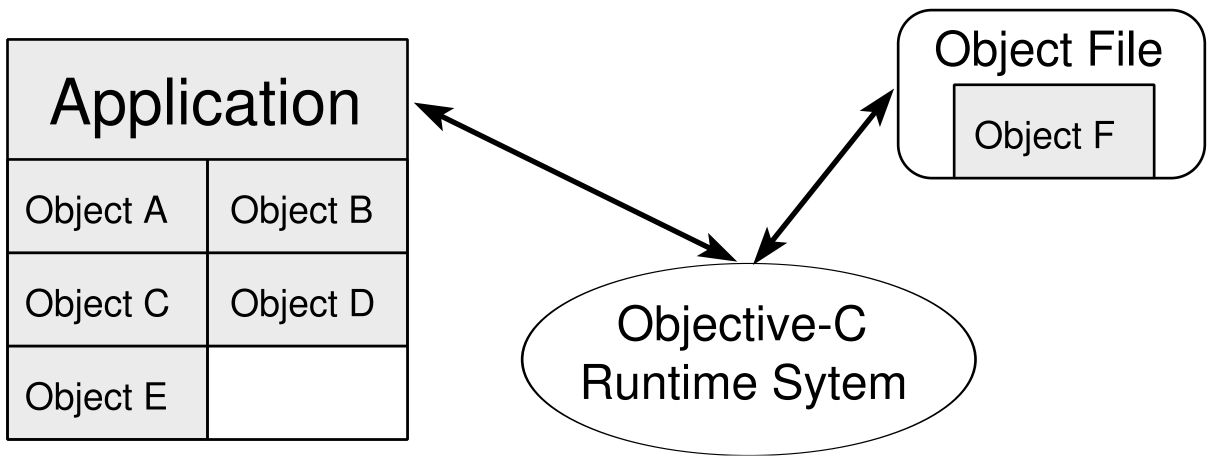
The GraphRight API is based on NeXT's Distributed Objects. Distributed Objects (DO) give a great amount of flexibility while requiring only a minimal learning curve. If you have worked with Objective-C, learning DO takes almost no time. Methods defined in DO's are called in exactly the same manner as object methods within a stand alone applications. With DO's objects no longer have to know about one another at compile time, an object can "live" anyplace, even on another machine on the network.

Watershed Technologies is very committed to customer satisfaction, if your custom application requires methods not provided please contact Watershed Technologies with your request. Our email address is: graphright@watershed.com

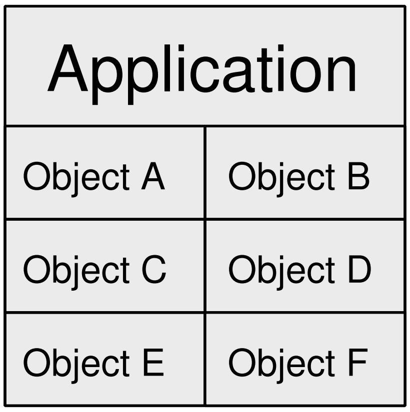
The rest of this document assumes familiarity with NeXTSTEP, Objective-C, Project Builder, and Interface Builder. Some exposure to Distributed Objects would be very beneficial. Recommended reading includes: Chapter 6 in NeXT's General Reference manual, this manual is also online on all developer systems in the : /NextLibrary/Documentation/NextDev/GeneralRef/06_DistributedObjects folder.

Introduction to Distributed Objects

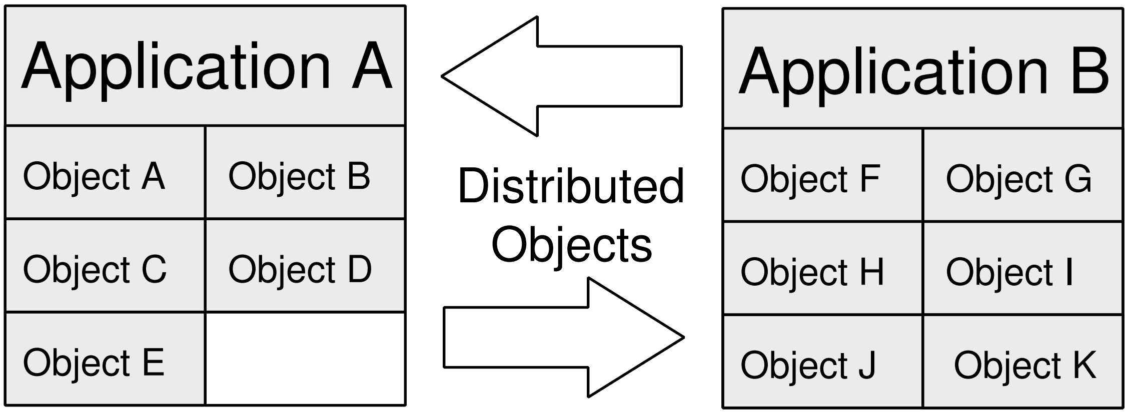
Distributed Objects implement a method for a way to share objective-c objects across networks. Where an object "lives" is no longer important. This gives many benefits, it allows the correct object to "live" on the hardware that best matches it's task. Traditionally even though Objective-C supported run-time binding, this binding had to take place within the same application. So at run-time an application could load code and resource "bundles" and have them act as though they were part of the original program.



Resulting in the program now functioning as though Object F were part of the original program.



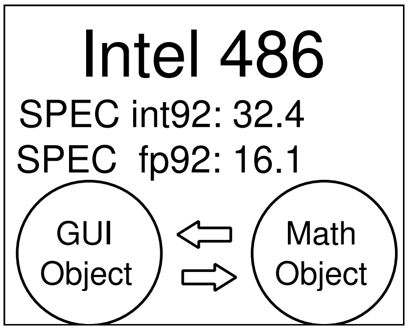
This provides great power, but does not provide for a way for objects in separate applications to work with one another. Distributed Objects add this key piece of functionality, with them two separate applications can share objects and make use of each others functionality.



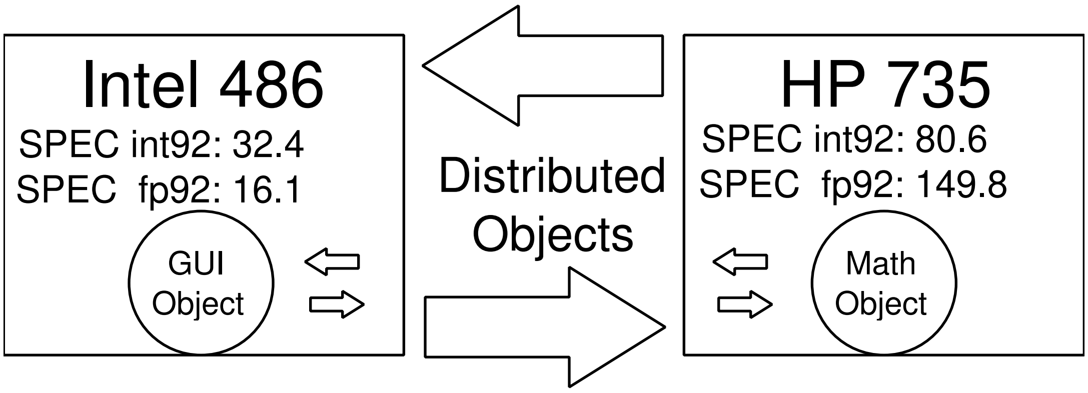
Now Application A can access the object it needs from Application B through Distributed Objects, all that must be defined is a common protocol.

For example, if you were doing a large calculation that takes enormous amounts of time on a 486 machine, this object could be run on an HP 735 machine resulting in much improved performance. The front end to the application that say displayed a graph of the resulting calculations could then run on the less powerful 486.

So a model that would look like this:



Would turn into something like this:



To some degree this is an extreme example, not only are the objects not running on the same machine, but they are also running on different architectures. This shows the incredible flexibility that Distributed Objects bring to custom applications.

Making the Connection

When working with Distributed Objects, the first thing one must do is create a connection from the client to the server. The first thing to do would be to launch GraphRight, this can also be done from the client program.

```
BOOL dirLaunch = [[Application workspace] launchApplication:GraphRight.app];
```

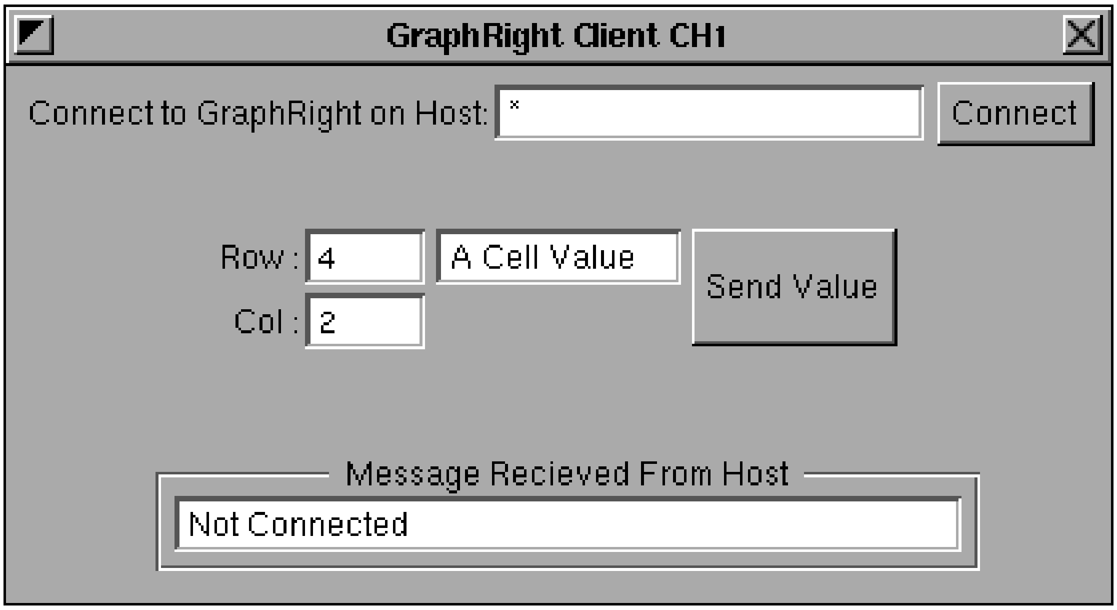
If the workspace was able to launch the application then YES is returned, if not NO is returned. The server application is assumed to be in the normal application path. Note that if the server is already running then NO will be returned.

The next thing we need to do is create the connection to the server.

```
server = [NXConnection connectToName:"GraphRightServer" onHost:"*"];
```

If the host is known, it can be substituted for "*", but if "*" is specified NXConnection will begin to traverse your netinfo domain looking for a connection with the name "GraphRightServer". This single line of code is all that is required to establish a Distributed Object connection between two objects. Distributed computing has never been easier.

So lets create our first program to talk to the GraphRight server (A finished version is included under the Example 1 folder in the API folder). With Project Builder create a new project and create it's interface to look like:



Create a new subclass of Object and call it GraphRightClient, to it add six instance

variables:
hostName, messageFromServerField, cellValue, row, col, and grServer.

Now Instantiate the object with Interface Builder and connect hostName to the upper Form Field, connect messageFromServerField to the lower Text Field, connect row and col to the corresponding Form Fields, and connect cellValue to the center Text Field. Now create a method named connectToServer:, and create a message called sendValue:. Connect the "Connect" button to the connectToServer method, and connect the "Send Value" button to the sendValue method. Save everything and un-parse the file in Interface Builder. GraphRightClient.h and GraphRightClient.m will be created, to them we will now add some code.

To the - connectToServer method we will add code to launch GraphRight if it is not already launched, and we will add the one line that will allow us to begin to send messages to GraphRight

```
- connectToServer:sender
{
    char *serverHostname = [hostName stringValue];
    BOOL dirLaunch =[[Application workspace]
                      launchApplication:GraphRight.app];
    grServer = [NXConnection connectToName:"GraphRightServer"
                                   onHost:serverHostname];
    if(!grServer)
    {
      [messageFromServerField setStringValue:"No GraphRight Server Found"];
      return self;
    }
    return self;
}
```

We are now ready to finish this example by adding the last message, sendValue:.

```
- sendValue:sender
{
    id notebook = NULL;
    id listOfNotebooks = NULL;
    int count;
    const char *cellContents = [cellValue stringValue];
    int r = [row intValue];
    int c = [col intValue];
```

```
    listOfNotebooks = [grServer GR_currentNotebooks];
    count = [listOfNotebooks count];

    if(count < 1) /* if there are no notebooks create a new one */
      notebook = [grServer GR_createNewNotebook];
    else /* just grab the first one */
      notebook = [listOfNotebooks objectAt:0];

    if(!notebook)
    {
      [messageFromServerField setStringValue:"Unable to connect to a 
notebook"];
      return self;
    }

    [notebook GR_setCell:r :c string:cellContents];

    return self;
}
```

One last thing we need to do is to make sure GraphRight is running before we begin try to connect to it. So we will add that to the appDidInit: method. Make sure that the GraphRightClient object is the delegate of the File's Owner Object (Application Object), so that we receive the appDidInit: message.

```
- appDidInit:sender
{
    BOOL didLaunch =[[Application workspace]
                        launchApplication:"GraphRight.app"];
    return self;
}
```

One thing to be aware of (and this is something that everyone stumbles on) is that this is a one way connection from the client to the server. The client can at any time send the server a message, but the server can not send messages to the client object at any time. The server object can only call client methods during the context of a message to the server from the client. As an example:

Object A is the client, and Object B is the server.
At any time Object A can send a message to Object B.
Object A sends message

```
@implementation ObjectB
- bar
{...
      [objectB foo:self];
...}

@end
```

Now while Object B is executing foo

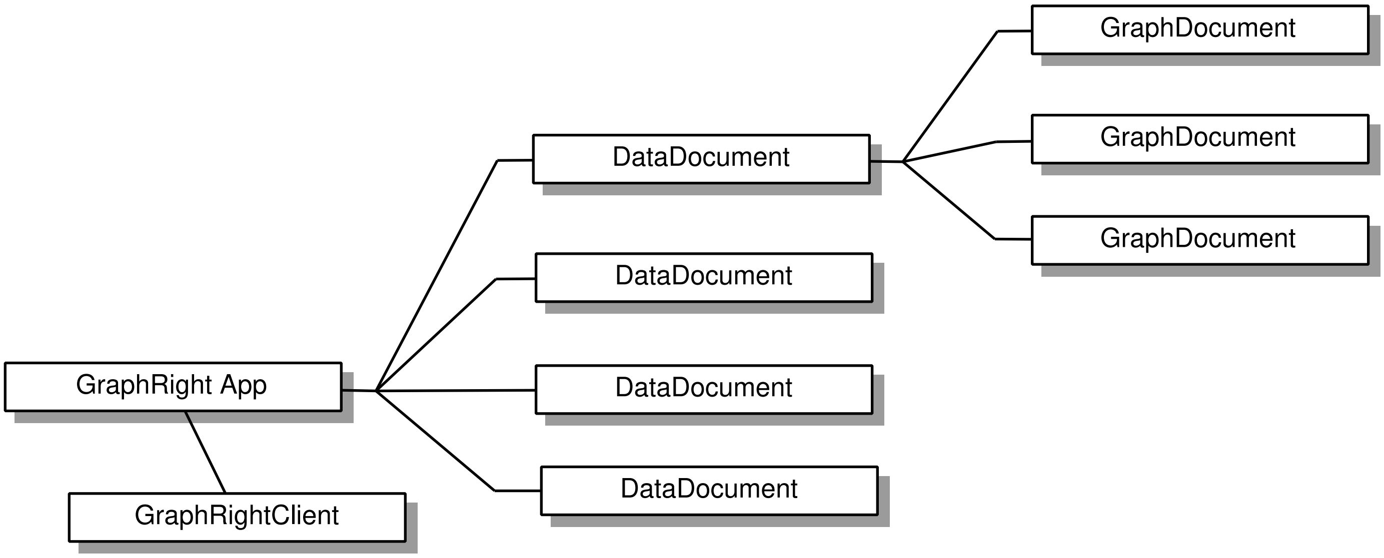
```
@implementation ObjectB

- foo:sender
{
...
      [sender valueFor:n];
...
return self;
}

@end
```

As you can see while it is executing the method foo, Object B needs to know some value from Object A , since it was called by Object A it can send a message to Object A.

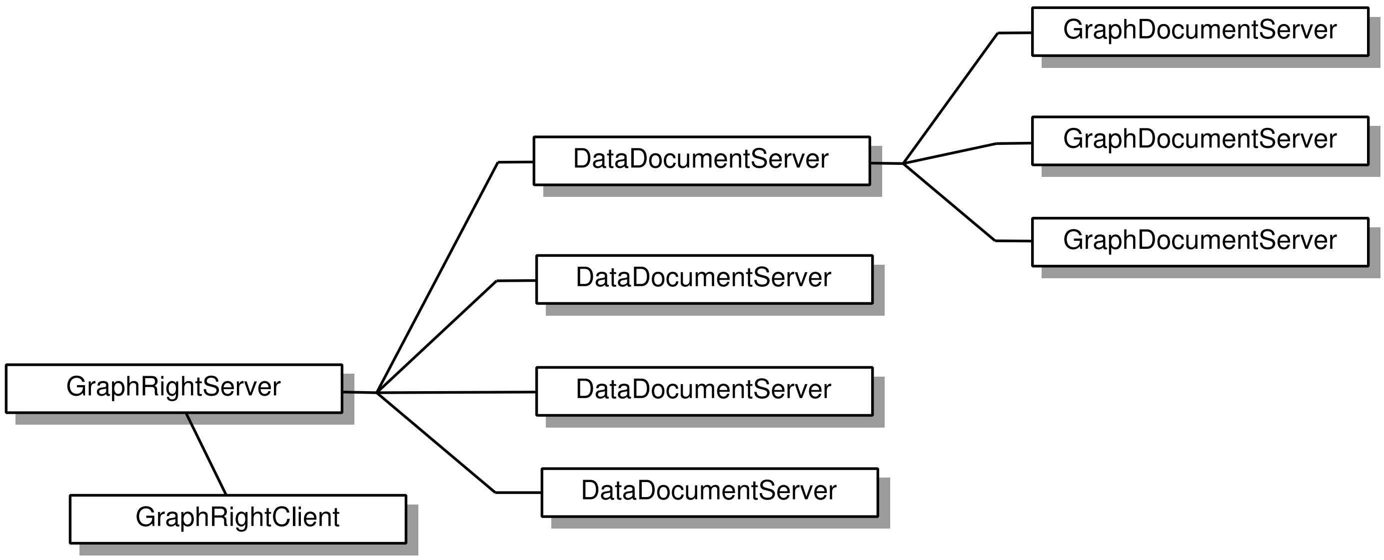
While this limitation may be fine for most applications, we wanted GraphRight to be able to send messages back to it's client at any time. Thus we added a simple initialization method that sets everything up. But we will cover that later in the documentation. For now all we want to do is connect to the server send in some data and create a graph.

Working with GraphRight Server

At this time it would be good to cover the class structure of the GraphRight server. To understand how the API interacts with GraphRight, it would be beneficial to see how GraphRight organizes things internally. As you can see from the diagram, the GraphRight Application owns a set of Data Documents that hold the data for the user and present it in a simple tabular fashion. Likewise each DataDocument owns a set of GraphDocuments. Each GraphDocument is a single "page" that can in turn hold multiple Graphs and any number of Graphics (Lines, Boxes, Circles, etc). In this way a Graph can never loose the data that it came from as with many traditional graphing packages where the Graph refers back to a set of data stored in a separate location.



The GraphRight API structure is really quite simple, and four main classes make up the entire API structure. Thus from the point of view of a client application the structure of GraphRight looks like this. Each DataDocument and each GraphDocument has an API server that acts on it’s behalf.



One thing to note here is that the GraphRightClient is optional if the client application does not wish to receive events that occur while the user is interacting with the program (changing the value in a cell, closing a Notebook (what the API calls a DataDocument), etc.).

Creating a Graph

Now that we have some understanding of how GraphRight handles it's API connections, we can go onward with our goal of creating a graph with the API.

The first thing to do is create a copy of our Example 1 folder and call it Example2.

Our goals for this project are to:
1) Connect to a GraphRight Server
2) Create a new DataDocument
3) Insert some data into the document
4) Create some graphs form that data

The number of lines we need to add to this example will also be minimal. The first thing we need to add is a way of working with the same notebook in GraphRight, we don’t want to over write data in some random notebook. So we will create a notebook called "GR Client Notebook" and always send new data to it. So when the user clicks on send new data it will first try to find the named notebook, if it is not found then it will be created. Here is the code for that method.

```
- sendData:sender
{
    id notebook;
    if(!grServer)
    {
      [messageFromServerField setStringValue:"Not Connected toGraphRight
Server"];
      return self;
    }
    /* we are connected, so now check for the notebook we want */
    notebook = [grServer GR_getOpenNotebook:"GR Client Notebook"];

    if(!notebook)
    {
      /* we could not find a notebook called "GR Client Notebook" on
         the server, we need to create a new notebook and save it
         in our home directory
       */
      notebook = [grServer GR_createNewNotebook];
      if([notebook GR_setPath:"~/GR Client Notebook"] != NO_ERROR)
      {
          /* for some reason we can’t create the notebook, or the
             path given was bad... were dead...
           */
          [messageFromServerField setStringValue:"Could not create Notebook :
GR Client Notebook"];
          return self;
      }
    }
    /* now we know we have a valid notebook to work with */
    return self;
}
```

Add the method to the header file, and re-parse it in Interface Builder. Now make create a connection in Interface Builder from the "Send" button to this new method (you can also remove the row and column fields since we will not be using them this time). Save everything and compile, this program will then launch GraphRight, form a connection to it and create a new Notebook with the name "GR Client Notebook". We are now ready to send data into the created notebook.

Add a new method called -sendNewData to the GraphRightClient.m file. What we need to do is encode the data in a GRCell write it to a typed stream (so that any machine can read the data on the receiving end, even if it is of a different architecture), and then send the encapsulated data to the GraphRight server. Here is the code:

```
- sendChunckOfData:sender
{
    int i,j;
    int n = 20;
    int m = 6;
    GRCell cell;
    NXTypedStream *ts;
    NXStream *stream = NXOpenMemory(NULL, 0, NX_WRITEONLY);
    char *buffer;
    int len, maxLen;
    id data;

    ts = NXOpenTypedStream(stream,  NX_WRITEONLY);
    i = m * n;
    NXWriteType(ts, "i", &i); /* Write the number of cells we are sending */
    for(i=0;i<n;i++)
      for(j=0;j<m;j++)
      {
          cell.r = i;
          cell.c = j;
          cell.isValue = YES;
          cell.val = random()/10000000;
          cell.string = NXCopyStringBuffer("");
/* This MUST be done, it seems that there is a bug in Distributed Objects when
 * sending a NULL string. OD’s don’t check for NULL and just try to do a
 * strlen(string) on the NULL, this crashes the client
 */
          NXWriteType(ts, "{iicf*}", &cell);
      }

    NXCloseTypedStream(ts);
    /* encapsule the data */
    NXGetMemoryBuffer(stream, &buffer, &len, &maxLen);
    data = [[NXData alloc] initWithData:buffer size:len dealloc:YES];
    /* send to client, NXData objects are automatically sent ’bycopy’ */
    printf("Sending Data\n");
    [currentDoc GR_setCells:data]; /* currentDoc is a Proxy to a DataDocument
*/
    [data free];

    return NO_ERROR;
}
```

Now we need to add a line that will call the method from the - sendData: method, before the line:

```
return self;
```

add:

```
[self sendChunckOfData:self];
```

Let's finish by finally creating a graph of the data. The first thing we need to do it inform GraphRight what data we wish to create a graph of. In GraphRight any portion of a data set can be graphed, one does not have to the limitation of graphing all the data in a data table. Let us now create a new method that will select the data to be graphed, and then create a new Graph Page from that data.

Add a new menu to the NIB file and call it "Create New Graph", now add a new method to both GraphRightClient.h and GraphRightClient.m and call it -createGraph:sender. We will now fill in the example source for it.

```
- createGraph:sender
{
    id notebook;
    GRAddress upperLeft, lowerRight;

    if(!grServer)
    {
      [messageFromServerField setStringValue:"Not Connected to GraphRight
Server"];
      return self;
    }
    /* we are connected, so now check for the notebook we want */
    notebook = [grServer GR_getOpenNotebook:"GR Client Notebook"];

    if(!notebook)
    {
      [messageFromServerField setStringValue:"Could not find GR Client
Notebook"];
      return self;
    }

    /* Ok we have found it, now select the data we know is there and graph it
*/

    upperLeft.r = 0;
    upperLeft.c = 0;
    lowerRight.r = N;
```

```
    lowerRight.c = M;

    [notebook GR_selectRange:upperLeft bottom:lowerRight];

    /* Now lets create the Graph */

    if([notebook GR_createGraphType:GR_STACKEDCOLUMN_CHART] == NULL)
      [messageFromServerField setStringValue:
       "Error: can't create graph: invalid data selected"];
    else
      [messageFromServerField setStringValue:"Successfully created graph"];

    return self;
}
```

With this method finished, we have accomplished our goal of witting a client application that creates a new notebook on the GraphRight server, sends data into the notebook, and creates a graph from that data. In this example we just created a stacked column chart, it is left as an exercise to the reader to modify the example so that it can create any number of graphs from the data we send it.

Sending Images and Text

We should not overlook some other important methods for importing images and rich text into a graph page. This makes creating custom reports with company logs and such very easy to do. The two relevant methods are:

- **GR_importImage**:(const char *)imagePath
:(float)x
:(float)y

and

- **GR_setRichText**:(char *)data
andPlaceAt:(float)x :(float)y
allowWidth:(float)width

The first imports a TIFF (.tiff) or EPS (.eps) file into the Graph Page at location x y. Both x and y are in inches and are measured from the bottom left corner of the page. Borders are not counted, measurements are in absolute units from the farthest left bottom edge of the paper. Returns a Proxy to an NXImage object work-a-like on success, nil on failure. Do not use any methods that are not part of the GraphRight API on the returned object.

The second method imports Rich Text at location x y. Both x and y are in inches and are measured from the absolute top left corner. Borders are not counted, measurements are in absolute units from the edge of the paper. If width = 0, then the text will be sized such that the first row fits on a single line, all subsequent rows will be wrapped if they are wider. If a width is given then all the text will be wrapped to that width. The data should be raw RTF text, it should have a NULL at the end of the char array. Returns a Proxy to a Text object work-a-like on success, nil on failure. Do not use any methods that are not part of the GraphRight API on the returned object.

Receiving Client Events

As we mentioned before one limitation of Distributed Objects is that the connection is only a one way connection from the client to the server. Since it is feasible that the client program might want to know of events that occur on the server side, such as the user changing the value of a cell, we have extended the API such that two way communications are very simple. This part of the API need not be implemented for all custom programs, only those that wish to know of user events in GraphRight.

The first thing that you need to do to set up two way communication is register your application as a receiver for Distributed Object messages. These three methods will set up everything for you:

```
grClient = [NXConnection registerRoot:self withName:"GraphRightClient"];
[grClient registerForInvalidationNotification:self];
[grClient runFromAppKit];
```

grClient is an instance variable for our GraphRightClient object. This then sets up our object to receive all GraphRightClient messages. The string "GraphRightClient" should be replaced with a unique name that identifies the client application from other clients that might want to connect to other copies of GraphRight on the network. Now we must just inform GraphRight that we wish to actually receive the messages and inform it of our port name and our host name.

This is the code to do that:

```
gethostname(clientHostname, MAXHOSTNAMELEN);
error = [grServer GR_registerWithServer:self
                                forName:"GraphRightClient"
                                 onHost:clientHostname];
```

This must be after the initial connection is made to the GraphRight server.

Advanced Topics

The GraphRight Open API is constantly changing to accommodate customer requests. We welcome all comments and suggestions on improving our API so that it fully meets your particular applications needs.

For further help in either learning the GraphRight Open API or for custom work contact Watershed Technologies for a list of members of the Watershed Partners Program in your area. Watershed Partners are consultants who have registered with Watershed Technologies and are knowledgeable in our API. Help and custom programming is also available from Watershed Technologies directly. Service contracts are also available on a yearly basis directly from Watershed Technologies.