an article for The Smalltalk Report

by Patrick Mueller

Patrick Mueller is a member of the IBM Smalltalk Distributed team in IBM Cary. He co-authored the HP/IBM submission to OMG for Smalltalk mappings to CORBA. Patrick can be reached via e-mail at pmuellr@vnet.ibm.com.

Business Phone 919-254-4307

Notes to editor:

For Table of Contents:

With all the interest in the Internet lately, it's only natural for folks to want to write applications in Smalltalk to access the vast wealth of information out there. Patrick provides some introductory information on how to use tcp/ip sockets to access a Gopher server from IBM Smalltalk. He also describes how the IBM Smalltalk user interface is programmed through widgets.

Pull Quotes:

If you're already familiar with Motif Widgets, you're already familiar with IBM Smalltalk's Widgets. If not, don't worry - it's a simple and elegant model.

Graphics: I've put frames in two locations for graphics. One is for gomenu.tif, the other for gotext.tif. I'm actually going to send along 4 files: gomenu.tif - color, gomenum.tif - mono; gotext.tif - color, gotextm.tif - mono. I'd prefer if the color versions can be used, but use the mono if you must.

Formatting: the only real formatting I've done is to mark Smalltalk code with the styles StCode and StBlock. StCode is used for in-line stuff like Class and method names, StBlock is used for blocks of code.

Building a Gopher from Sockets and Widgets

No, this isn't an article on Cyber-Biology. I'll be describing how to build an Internet Gopher client within IBM Smalltalk, using the Widgets user interface programming model and the sockets communications protocol used on the Internet. Plan on learning a couple of things after reading the article:

- what an Internet Gopher looks like
- how the Gopher protocol works
- an introduction to socket programming in IBM Smalltalk
- an introduction to user interface programming in IBM Smalltalk

What is Gopher?

First let's talk about Gopher. If you don't already know what Gopher is, the best way to learn is to play with a Gopher client. Ask you local internet guru for a test drive. In case you don't have one locally available, here's a description.

Insert goMenu.tif or goMenuM.tif here.

Figure 1. Sample Gopher Menu

You start up a gopher client by running the gopher program and specifying a gopher server to start at. The gopher client will contact the server and ask for a list of items. Those items will be displayed by the gopher client, with some kind of user interface for you to select items. Each item in the list is typed: common types are

- another gopher menu
- a text file
- a graphics file

When you select one of the items, the gopher client will send a command to get the appropriate type of item from the server and return it to the client. In the case of another gopher menu, another menu will be displayed. In the case of a text file, a text editor will be displayed with that text. You get the picture. It's a very simple program to use. And there's lots of information available. Within IBM, for example, we have over 60 well-known gopher servers, servicing over 8000 different menu items (sorry folks, this is primarily IBM-only information).

The Gopher Protocol

The protocol a gopher client and server use to exchange information is one of the simplest used over the internet. To get a gopher item from a server, the client needs to know three pieces of information: the name of the server (tcp/ip hostname), the tcp/ip port for the server, and a selector string. Most gopher servers use port 70. The main menu for a gopher server uses an empty selector string. So, to get the main menu from a gopher server, you really only need one piece of information: the name of the gopher server.

The client creates a new tcp/ip socket and connects it to the server at the port requested. It then writes the selector string, followed by carriage return and linefeed to the socket. At this point, it starts reading from the socket, terminating when the socket is closed by the server. The data returned by the server is interpreted depending on the type of the item. After receiving the data, the client closes the socket.

The most common type of gopher item is a menu; that is the type of item returned for the main gopher server, when passed an empty selector string. The data returned for a menu consists of a set of lines, separated by carriage return and linefeed characters, up to the line which contains nothing but a period ("."). For each line, the first character is a type indicator. The rest of the string is tab delimited. The field after the type indicator is a string to display in the user interface for the item. The next field is the selector. The next is the server name, and the last is the port. The selector, server name and port are all used to get that item. The type indicator (first character in the line) indicates what type of item this is (eg, menu, text, graphics, etc).

For the text type, the data returned from the server is just the text to display back to the user. For the graphics type, the contents of a GIF or TIFF file might be returned.

Classes implemented

First a little class hierarchy creation. We're going to implement a class called **GopherItem**, with a subclass for each of the gopher data types. **GopherItem** is defined with instance variables:

display	description of the item to display to the user
selector	selector to send to the server
host	name of the server
port	port number for the server
data	data returned by the server

Besides defining accessors for these variables **GopherItem** contains

- the logic to get the data for an item from a server
- the logic to parse a line of menu information returned from the server
- the logic to determine what type of data a particular line is

We'll create the following subclasses of **GopherItem**:

- **GopherItemMenu** to display menus
- **GopherItemSearch** to prompt for a search string, and display a resulting menu (used to search phone books, for instance).
- **GopherItemText** to display textual information
- **GopherItemUnknown** to handle Gopher data our client does not understand.

We'll create a class named **Gopher** to manage the user interface.

Using TCP/IP Sockets in IBM Smalltalk

Now we'll actually implement the main processing of the gopher client: connecting to a server to get the some data.

If you aren't already familiar with sockets, here's a brief overview. Sockets are a lot like file handles. You open them, read from them, write to them, and close them. Except, instead of having a disk drive to read from or write to, there's another program over the network who is reading what you are writing, or writing what you are reading. And instead of specifying a file name, you specify a host name and a port number to connect to.

The logic to get the data for an item from the Gopher server is implemented in the instance method **GopherItem>>getData**. **GopherItem** supplies instance methods to return the host, port and selector of the Gopher server we want data from. Example 1 contains the Smalltalk code for this method.

```
getData
  "Set data instance variable to the data returned for the
  gopher menu item."

  | abtHost abtPort abtSock dataChunk allData |

  self data: ''.

  self port isNil ifTrue: [self port: 70].
  self selector isNil ifTrue: [self selector: ''].

  abtHost := AbtTCPInetHost getHostByName: self host.
  abtHost isCommunicationsError ifTrue: [^nil].

  abtPort := AbtTCPPort usingHost: abtHost portNumber: self port.
  abtPort isCommunicationsError ifTrue: [^nil].

  abtSock := AbtSocket newStreamUsingPort: abtPort.
  abtSock isCommunicationsError ifTrue: [^nil].

  abtSock bufferLength: 8192.

  (abtSock connect) isCommunicationsError ifTrue: [^nil].

  (abtSock sendData: (self selector, Cr asString, Lf asString))
    isCommunicationsError ifTrue: [^nil].

  allData := ''.
  [abtSock isConnected] whileTrue: [
    dataChunk := abtSock receive.
    dataChunk isCommunicationsError ifTrue: [^nil].
    allData := allData, dataChunk contents asString
    ].

   abtSock disconnect.
   self data: allData.
```

Example 1. GopherItem>>getData method

getData first initializes its data to an empty string, and defaults it's port and selector if not set. It then obtains an instance of **AbtTCPInetHost**, **AbtTCPPort**, and **AbtSocket** from the host and port information. **AbtTCPInetHost** is used to convert host names into tcp/ip addresses. **AbtPort** is used to associate a tcp/ip port with a tcp/ip address. **AbtSocket** is used to manage the actual socket, based on the **AbtPort** it was created with.

Up to this code, we have defined what we want to connect the socket to, but haven't actually connected it. Sending **connect** to the socket will cause the socket to connect to the server.

Once connected, we send the selector, followed by carriage return and linefeed, then start reading from the server. As long as the socket is connected, we receive the data from the socket and append it to the end of a local variable. When the server finally closes the socket, **isConnected** will return false. At this point, we close our end and set the data to the entire string returned from the server.

That’s the only tcp/ip related code in the entire gopher client. Each gopher item subclass is responsible for interpreting the data received by this code.

Using Widgets in IBM Smalltalk

Widgets are the programming interface used for user interface programming in IBM Smalltalk. The terminology comes from Motif, upon which the user interface classes are based on. If you’re already familiar with Motif Widgets, I have real good news for you - you’re already familiar with IBM Smalltalk’s Widgets. If not, don’t worry - it’s a simple and elegant model.

Widgets are used to model all the visual building blocks needed to create a user interface:

- the shell, to contain the frame, system menu, title bar, and minimize/maximize buttons
- main windows to contain the menu bar
- forms and bulletin boards to contain other widgets
- core widgets like buttons, list boxes, text fields, etc.

Each type of widget is a subclass of **CwWidget**. There are two primary ways to change the behavior of a widget: through resources and through callbacks.

Resources control the basic state of a widget, such as color and font information. Most widgets have a unique set of resources associated with them, and resources are inherited down the **CwWidget** class hierarchy. Resources are set and queried via instance methods named after the resource. For instance, to query the width of a widget, send it the message **width**.

Callbacks are a way to get feedback from the user when they interact with the system. Like resources, each widget class implements its own set of callbacks, which are inherited down the CwWidget class hierarchy. As an example, to be notified when the user presses a button, the following code may be used.

```
buttonWidget
  addCallback: XmNactivateCallback
  receiver: self
  selector: #pressed:clientData:callData:
  clientData: nil.
```

Each callback has a name, in this case **XmNactivateCallback**. This particular callback is invoked when a button is pressed. When the button is pressed, the message **pressed:clientData:callData:** will be sent to the object which executed this code (since the receiver was specified as self). The callback message is passed the widget, the client data specified when the callback was added (in this case, nil), and an object containing information specific to this type of callback.

Ok, so those are the basics, let’s dive right into our gopher client. Our user interface is going to be a new window, with a read-only text field at the top giving a description of the current gopher menu item we’re viewing and a list box containing the items available on this gopher menu. Gopher text items will be displayed in a separate window (a Workspace), which is not described here.

The widgets we’ll need are:

- a shell, to contain the frame, system menu, title bar, etc.
- a form, to contain the text field and list box
- a text field

- a list box

A form is a widget which knows how to resize the widgets contained within it. We're using it to allow the user to resize the window and have the widgets contained in the form automatically resize themselves.

As mentioned before, we'll be implementing a class called **Gopher** to handle the user interface. **Gopher** is defined with the following instance variables:

data	to hold the data associated with the menu items (ie,the selector, server, and port of the menu items)
listWidget	to hold our list box widget
textWidget	to hold our text field widget
shellWidget	to hold our shell widget
menuStack	to keep track of where we came from, so we can backtrack through the gopher.

The instance method createWindow is used to create and setup all the widgets. Example 2 contains the code for this method.

The first thing we do is create a shell window. This is done with the message **CwTopLevelShell class>>createApplicationShell:argBlock:**. The first parameter is the name of the widget. All widgets have a name, which is usually not externally visible to the user. The second parameter is a block used to set resources when the widget is created. In this case, we're going to set the title of the shell window, which will be placed in the frame's window bar, and the width of the frame, making it half the size of the screen.

You might be wondering why we use the **argBlock** parameter (and the **setValuesBlock:** later in the code) to set our resources.

The message to create the shell widget could also have been written as

```
shell := CwTopLevelShell
  createApplicationShell: 'gopherMenu'
  argBlock: nil.
shell
  title: 'Gopher Menu';
  width: (CgScreen default width) // 2
```

In IBM Smalltalk, widget resources are 'hot' - that is, when changed, the user interface is immediately updated. In order to allow the system to optimize changes to a widget, the **argBlock** parameter and **setValuesBlock:** message are the recommended ways to set resource values for a widget.

Next, we create the form. Most widgets are created using widget creation convenience methods named **createXXXX:argBlock:**, where XXXX is the type of widget to create. These messages are sent to the widget which will contain the widget to be created. In this case, we'll create a form with the name 'form', and don't need to set any resources.

After the widget is created, we send it the message **manageChild**. This is a Motif-ism, which you don't need to be too worried about, but will need to call it after creating your widgets. Managing and mapping widgets allows some interesting behaviors, such as causing widgets to instantly appear and disappear as needed.

Contained within the form will be a label widget, created with **createLabel:argBlock:**. We'll set the initial text of the label to a blank string.

Also contained within the form is a list box, created with **createScrolledList:argBlock:**. The **selectionPolicy** resource sets the type of selection allowed - single select, multiple select, etc. The **visibleItemCount** resource sets the initial size of the list box, eg. the list box will be sized to contain 20 items.

As mentioned previously, we're using a form so that the widgets inside the form can be automatically resized. In order to make this happen, we have to attach the widgets to the form.

For each of top, bottom, left and right, there are three basic types of attachment:

- attach the widget to the edge of the form
- attach the widget to a position in the form (position based on 100 - setting to position 50 attaches the widget to the middle of the form)
- attach the widget to another widget.

In our case, we attach the label widget to the top, left, and right sides of the form. We don't need to attach the bottom, since a label field has a default height (the height of the font the text is being displayed in). The list box is attached to the bottom, left and right sides of the form, and it's top is attached to the label widget. Note also an offset is specified for aesthetic reasons (to keep the user interface from looking as if it's all crammed together).

Now when the window is resized, the label and text windows will have their widths changed automatically, since they are attached to the sides. When the height changes, the label won't change size but the list box will, since it's attached to the label widget at the top and the form on the bottom.

As a further example of attachments, if we change the label widget to attach the bottom as in:

```
bottomAttachment: XmATTACHPOSITION;
bottomPosition: 25;
```

the label widget would take the top ¼ of window and the list box would have take the bottom ¾.

Note that for the listbox, we send **setValuesBlock:** to the parent of list, not list itself. This is because a **CwScrolledList** widget is a list box with a set of scrollbars around it. It's the widget (which we don't see) which contains the list box and scrollbars which we need to attach to the form.

In order to be able to execute some code when an item in the list is selected, we need to use a callback. In the code above, the **XmNdefaultActionCallback** is used on the list widget. This callback is invoked when an item is double-clicked in the listbox. We specify sending the message **selectItem:clientData:callData:** to self. The actual callback is implemented as follows:

```
selectItem: widget clientData: clientData callData: callData
  "Callback sent when an item is selected.  Open a viewer
  for the appropriate GopherItem subclass for the item."

  | pos menuItem |
  pos := callData itemPosition.
  menuItem := (self data) at: pos.
  menuItem view: self.
```

callData is an object containing information specific to this callback; in this case, sending it itemPosition answers the 1 based offset of the item within the menu which was selected.

The data instance variable of **Gopher** contains an ordered collection of **GopherItem** instances returned from the server. We just get the appropriate menu item and tell it to view itself.

Finally, we tell the shell to realize itself, which causes it to be displayed, and set our instance variables.

The contents of the listbox are maintained with the **items** resource. The data associated with this resource is an **OrderedCollection** of **Strings**. For instance, to set the contents of a list box to the the items ‘a’, ‘b’, and ‘c’, you would use the following code

```
listWidget items: (OrderedCollection with: ‘a’ with: ‘b’ with: ‘c’)
```

```
"Create the gopher menu window"

| shell main form text list |

shell := CwTopLevelShell
  createApplicationShell: 'gopherMenu'
  argBlock: [:w| w
    title: 'Gopher Menu';
    width: (CgScreen default width) // 2
    ].

form := shell
  createForm: 'form'
  argBlock: nil.
form manageChild.

text := form
  createLabel: 'label'
  argBlock: [ :w | w
    labelString: ' '
    ].
text manageChild.

list := form
  createScrolledList: 'list'
  argBlock: [ :w | w
    selectionPolicy: XmSINGLESELECT;
    visibleItemCount: 20
    ].
list manageChild.

text setValuesBlock: [:w | w
  topAttachment: XmATTACHFORM; topOffset: 2;
  leftAttachment: XmATTACHFORM; leftOffset: 2;
  rightAttachment: XmATTACHFORM; rightOffset: 2
  ].

list parent setValuesBlock: [:w | w
  topAttachment: XmATTACHWIDGET; topWidget: text;
  bottomAttachment: XmATTACHFORM; bottomOffset: 2;
  leftAttachment: XmATTACHFORM;leftOffset: 2;
  rightAttachment: XmATTACHFORM; rightOffset: 2
  ].

list
  addCallback: XmNdefaultActionCallback
  receiver: self
  selector: #selectItem:clientData:callData:
  clientData: nil.

shell realizeWidget.

self listWidget: list.
self textWidget: text.
self shellWidget: shell.

self menuStack: OrderedCollection new.
```

Example 2. Gopher>>createWindow method

Final Notes

The source for the gopher client is available via anonymous ftp to st.cs.uiuc.edu, and will work on OS/2 and Windows, with IBM Smalltalk or VisualAge with the Communications Component.

Insert goText.tif or goTextM.tif here

Figure 2. Example Gopher Text Window