Barista Callpoint Performance Recommendations

When dealing with performance we're talking about latency and reducing round-trips, and trying to limit the number of times when the application has to come to a complete standstill waiting for data to arrive from the client. No matter how optimized we make Barista, and how consistent we are in setting EM to production settings, one miss-written callpoint can destroy all our optimizations. We must remind ourselves that we're working on 3 tier software, and the days of the monolithic infrastructure is gone.

In general terms, think about client/server network traffic like this:

> (1) Server->Client: This is stuff like editbox!.setText(x$) -- we fire data at the client, but we don't sit around waiting for a response.
>
> (2) Client->Server: This is event traffic, like when the user clicks a button. For the most part, we just fire data at the server, but nobody ever has to sit around waiting. The event object typically contains additional parameters relating to the event, for example the width/height of a resized window, or the x,y location of a moved window. Because this information is fired at the server as part of the act of delivering the event, it's immediately available to the program.
>
> (3) Server->Client->Server (round-trip). This is where your application queries the client for some dynamic information that cannot be cached on the server, so must be retrieved from the client. bbjwindow!.getWidth() and bbjwindow!.getHeight() fall into this category. This needs to be avoided as much as possible, because it means that the application program comes to a complete standstill and has to wait for a round-trip to the client and back. Best-case, assume that each method like this averages 100ms (1/10th of a second) within North America; double that in Europe. Several unnecessary round trips in a row can add up to a second very quickly.

Another thing to think about is why a single round trip can be so very expensive. You probably are already aware of this but it may be worth thinking about these items (especially the second one).

> 1) We batch the commands that are going from the server to the client. Which means that many commands can go in a single payload.
>
> 2) We continue sending commands to the client until we run out of commands or until we get to a command that requires a response. We will often send hundreds of commands to the client and then go back to doing whatever the program needs done. Meanwhile the client has a long queue of commands that it can work on while the server does some other task. But as soon we need a response from the client we have to wait for all the commands that are in the pipeline to be completed by the client before the client can send the response. This means that the server can not go off to do other work until the client has caught up with executing all the commands that have been pipelined.

You can find the rationale for that automagic batching behaviour in this 11-year-old article:

http://www.basis.com/sites/basis.com/advantage/mag-v4n3/lessons.html

That article mentions special batching syntax that originally had to be coded by the programmer. That was our first cut at this problem. Later, we implemented the batching behaviour directly within our client/server protocols, so those comments about a special batching syntax are obsolete. But the concept remains -- we optimize client/server communication as much as possible, automatically building up batches of server->client commands when we can. But as David mentioned, we have to come to a complete standstill and wait for the client to process all pending messages when we hit a command that requires a response from the client. It's that forced round-trip that causes the most noticieable delays for the user.

As a general rule, attention must be given to anything that can change dynamically on the client, not initiated by the server:

- bbjbutton!.getWidth() doesn't have to go to the client -- the user can't change the width of a button control, so we know we can return the width that we set from the server.
- bbjwindow!.getWidth() does have to go to the client because (depending on window settings) the user can resize the window at any instant. Event methods (ResizeEvent!.getWidth(), WindowMoveEvent!.getX(), etc) happen on the server because they're returning data from the event object, which is already on the server.

The following article covers the general concepts to keep in mind when designing a distributed application:

http://www.basis.com/sites/basis.com/advantage/mag-v4n3/lessons.html

Examples

Window Resizing

```
rdForm!.setCallback(rdForm!.ON_RESIZE,"resize_win")
...
resize_win:rem --- Resize Window
rd_new_wwid=rdForm!.getWidth()
rd_new_whgt=rdForm!.getHeight()
if rd_nav_bar$="YES" then
    rdNavWin!.setSize(rdForm!.getWidth(),rdNavWin!.getHeight())
endif
```

The `rdForm!.getWidth()` and `rdForm!.getHeight()` methods force a round-trip to the client. There's no way around that if you get here with `gosub resize_win`, but if you get here from an event, then the BBjResizeEvent object already has those values; the additional round-trip to the client is unnecessary. This area can be tweaked by changing this single subroutine to two subroutines, depending on whether it's invoked as an event handler or through procedural code (gosub). The event handler version would have `rdEvent!=rdSysGUI!.getLastEvent()`, and would use `rdEvent!.getWidth()` and `rdEvent.getHeight()` in place of `rdForm!.getWidth()` and `rdForm!.getHeight()`.

Server Side Collections

When using complex GUI controls such as BBjGrids and BBjTrees, use server side collections like vectors and hashes to store data instead accessing the controls. As with `rdForm!.getWidth()`, `grid!.getCellText(`row, col) forces a round-trip to the client.

Stored Procedures (sprocs)

One of the least often considered performance gains is the use of a sproc to churn through the data at the database level, and only return the results you need to the interpreter and display. SPROCS are invokable from an interpreter, just like they are from a Third Party product.
*** Need more information on this point. Expand to include information on readrecord w/ alt keys instead of full record set.

Callpoint setStatus("REFRESH")

"REFRESH" has two specific issues. From a client performance point of view, it falls into category (1) of the three categories. The data is fired at the client, without waiting for a response. The real performance hit occurs becasue "REFRESH" redisplays all controls on the form, whether they've changed or not. New options have been added to address the issue:

- setStatus("REFRESH:"<column>) - Redisplays only the specified column. However, does not allow developers to specify multiple columns in a single callpoint.
- setColumnData(<alias>.<column>,<value>,<refresh_flag>)) - Redisplays all specified columns. Allows developers to specify multiple columns in a single callpoint. This is the preferred method.

We realize the main issue is retrofitting all of the REFRESHes in code will be a big job.

Miscellaneous Questions

Performance of specific commands/methods:

Q: get/setValue() on group and global namespaces
A: Namespace is entirely server-side.

Q: ctl!.getUserData()
A: getUserData() / setUserData() is entirely server-side.

Q: sysgui!.getWindow(win_ctx).getControl(ctl_id)
A: bbjapi().getSysGui().getWindow(context).getControl(id) is entirely server-side.

Q: ctl!.getX(), ctl!.getY(), ctl!.getWidth(), ctl!.getHeight() for form layout and display purposes.
A: Entirely on the server because control sizing and positioning is initiated by the application.

Q: ctl!.setToolTipText()
A: setToolTipText() falls into category (1) of the three general categories. The data is fired at the client, but we don't sit around waiting for a response.

Q: Control validation callbacks.
A: Definitely a win in thin client if you only register validation callbacks when you know that you will need to validate the control (same for the equivalent focus callbacks for BUI).