The Lessons of BASIS b-Commerce™

By Jim Douglas and Greg Smith

Our electronic commerce application, BASIS b-Commerce™, is the first application implemented with the BBj Thin Client™ technology. It helped us to test and optimize BBj™ and the BBj Thin Client component, and it showed us what application programmers need to know in order to design and implement a system that works efficiently in a thin client environment.

The first thing we noticed when we started testing b-Commerce over the Internet was "It works! Cool!" The second thing we noticed was that it was too slow to be usable.

The reason was clear enough in retrospect. We developed b-Commerce using GUIBuilder®, which was written with an implicit assumption that the resulting program would run under Visual PRO/5®. Client/server issues weren't a factor in the design of GUIBuilder because Visual PRO/5 isn't a client/server platform. It turns out that when the client side (screen, keyboard and mouse) is separated from the server side (the business rules and database) by thousands of miles of copper and fiber, the application needs to work differently.

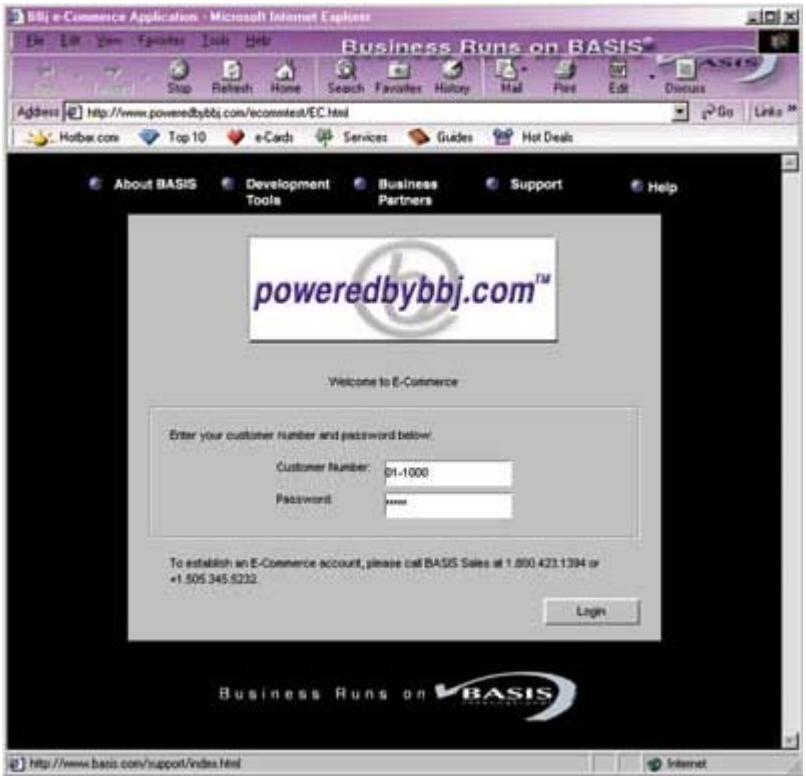
When you're writing a distributed application, you need to take bandwidth and latency issues into account. Bandwidth is a measure of how much data you can send in a given time. It's typically 28.8 kilobits per second (kbps) up to 53 kbps for an analog modem. Other technologies such as integrated services digital networks (ISDN), digital subscriber lines (DSL) and cable modems offer substantially higher bandwidth. The smaller the bandwidth, the more important it is to reduce the volume of data being transferred.

Bandwidth

One way to reduce transmission volume is to make image files as small as possible. BBj supports BMP format, but it also supports GIF, PNG and JPG, and they typically produce smaller files. The b-Commerce login screen uses an image that was originally stored as a 729,132-byte bitmap file, logo.bmp. When we saved the image using the exact dimensions required for the login screen, the size went down to 41,878 bytes. Then we tried various formats. Saving as a 16-color bitmap cut the size in half, but it ruined the clarity of the image.

We tried logo.gif (13,193 bytes) and finally settled on logo.jpg (4,715 bytes). Assuming a 53-kbps connection, we can transmit the final version of logo.jpg in less than one second, as opposed to more than two minutes for the original version of logo.bmp. We also added image caching to the thin client implementation, so any given image file is transmitted once and cached locally.

The login screen from BASIS b-Commerce™. The application was completely built in Visual PRO/5® and runs in BBj™



Latency

Latency is the time it takes for a single packet of information to make a round trip from the client to the server and back again. You can measure your current network latency to BASIS with the command: `ping www.basis.com`. For example:

```
[c:\basis]ping www.basis.com
Pinging www.basis.com [207.66.35.68] with 32 bytes of data:
Reply from 207.66.35.68: bytes=32 time=120ms TTL=240
Reply from 207.66.35.68: bytes=32 time=100ms TTL=240
Reply from 207.66.35.68: bytes=32 time=91ms TTL=240
Reply from 207.66.35.68: bytes=32 time=110ms TTL=240
[c:\basis]
```

In other words, each time you exchange any information with the BASIS Web server, even a single byte, the round trip will take, on average, a minimum of 100 milliseconds (ms), or 1/10th of a second. This contrasts with typical ping times of 10 ms for an ISDN connection and 0.3 ms for a local Ethernet connection. A widely distributed application should be designed to minimize the number of times the client and server need to talk to each other. With this principle in mind, we analyzed what the b-Commerce application was doing.

When we considered code like the following, we realized that each PRINT statement requires a network access, so this block of code was taking 500 ms to execute:

```
print (gb__sysgui)'context'(gb__win.login)
  print (gb__sysgui)'enable'(0),'show'(0),'focus'(0),'raise'
  print (gb__sysgui)'focus'(ctl_id)
  print (gb__sysgui)'context'(gb__win.main),'hide'(0)
 print (gb__sysgui)'context'(gb__win.login)'
```

We rewrote this code by combining multiple PRINT statements into a single command:

```
print (gb__sysgui)
  : 'context'(gb__win.login)'
  : 'enable'(0),'show'(0),'focus'(0),'raise','focus'(ctl_id),
  : 'context'(gb__win.main),'hide'(0),
: 'context'(gb__win.login)
```

This is functionally identical to the first version, but assuming a network latency of 100 ms, it will execute in 0.1 seconds as opposed to 0.5 seconds. We made this kind of change about 70 times in the b-Commerce program.

Then we considered the number of network accesses we were doing each time we update a screen. Applications developed with GUIBuilder typically use code like this to update the screen:

```
rem ' Initialize data structures for the login screen
 win_id_login$ = fngb__win_id$(gb__win.login)
 dim win_login$:fngb__template$(win_id_login$)

 rem ' Clear a couple of fields on the screen
 win_login$ = fngb__get_screen$(win_id_login$,win_login$)
 win_login.user_id$ = ""
 win_login.password$ = ""
 win_login$ = fngb__put_screen$(win_id_login$,win_login$)
```

This code does the following:

1. Reads the entire screen into a string variable
2. Clears a couple of fields in the string variable
3. Updates the entire screen based on the contents of the string variable

This strategy works well in Visual PRO/5, and it works well in BBj when the application is running on a single machine. But in a distributed application, we need better control over the number of network accesses and the volume of data being transmitted. So we added two new functions that allow us to limit the update to selected fields. With these new functions, we can update two fields on the screen with code like this:

```
rem ' Clear a couple of fields on the screen
 win_login.user_id$ = ""
 win_login.password$ = ""
 win_login$ = fngb__put_fields$(win_id_login$,win_login$,"user_id,password")
```

This code is more efficient because we've eliminated the initial screen read and we've limited the screen update to just the two fields being changed.

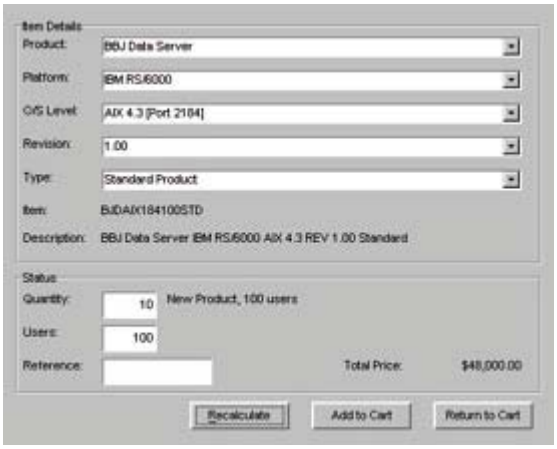
This optimization improved performance significantly, but we kept looking for ways to make it even faster. When we considered what these functions were doing internally, another optimization occurred to us. From a high-level point of view, these functions work with the screen as a complete unit: you read a screen in a single operation and you update it in a single operation. But the functions are implemented to iterate through the fields one at a time and do whatever is necessary to read or update that individual field. For example, it would take at least 3 seconds to read or write a screen containing 30 fields, assuming each field incurred a single network access of 100 ms.

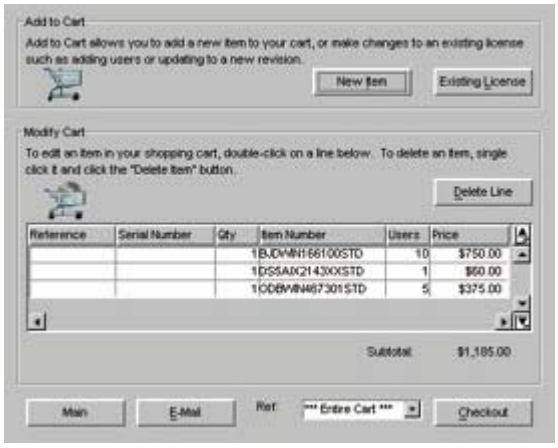
The solution was to enhance the SYSGUI device in BBj with a concept called batches. Batches allow you to assemble a group of SENDMSG() and/or CTRL() functions and execute them all at once, with a single network access. We rewrote all four functions to perform batch updates. This change makes a huge difference. No matter how many fields are on a given screen, a call to any of these functions translates to a single network request.

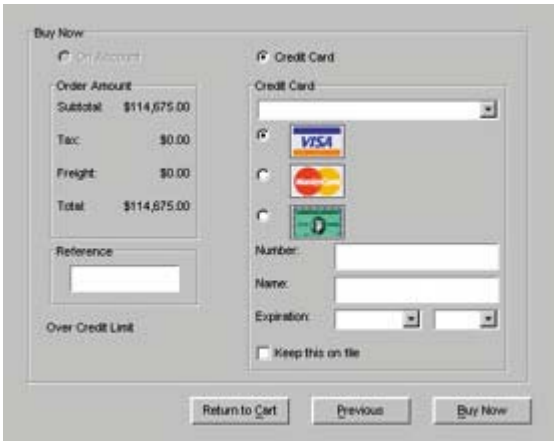
After we added the batch syntax, we looked for other places in the application where we could combine multiple commands into a single batch. Any block of SENDMSG() or CTRL() functions is a candidate; we found several data-aware grids where the code could be optimized in this way. In each case, we were able to combine five SENDMSG() functions into a single batch.

Other Optimizations

In testing the application, we realized that the core of the event loop, READRECORD(GB__SYSGUI,SIZ=10)GB__SYSGUI$, was also subject to network latency delays. We changed the BBj implementation so that event information is constantly being sent from the client to the server in a background thread. This can make a difference with, for example, list button events because making a selection from a list causes several events to be fired in rapid succession. Of course, when we were designing the b-Commerce resource file, we were also careful to set the event masks so as to minimize the number of events that the event loop would have to read and discard.







We also found a few application strategies to eliminate some network accesses entirely. For example, some status messages can appear in either black or red. Instead of explicitly setting the color every time we update the field, the application keeps track of the current color and only resets it if the color needs to be changed. With all of these changes and testing in place, we finally got the performance that we wanted. When we started testing, screen transitions within the application were taking as long as 10 to 15 seconds. After working through these optimizations, we reduced this to a more acceptable 1 to 2 seconds. All of the changes we made in GUIBuilder and the application are performance-neutral when the application is run locally, so there's no need to maintain two versions of the application. If you keep these principles in mind when you're writing your application, it'll work well both locally and in a distributed environment.

Implementing the System

It took a lot of planning and testing to bring the BASIS b-Commerce application to the point we felt comfortable releasing it. The act of selling our products involves different internal organizations and different types of Customers. We wanted to make sure that we had an application that would satisfy as many requirements as we could with an initial release.

The approach we took began with our internal groups, including Accounting, Customer Service, Systems Administration and Sales.

Our Accounting group had to deal with a major change in credit card processing, which was done through a third-party processor. We needed to make sure that we could track invoices and credit card payments all the way through the system. For orders that required shipment of CDs, we had to be sure that the items had shipped before the credit card processing was completed. This part of the b-Commerce application dealt mostly with learning more of the credit card processor's system and how to do all of these necessary activities.

Our Systems Administration group made sure that the business logic, product listings, and serial number information and modification processes worked correctly. Through this, we discovered and added functions that our Customer Service staff did intuitively that weren't previously implemented in the system.

We also needed to ensure the consistency and accuracy of orders entered through the automated system. To do this, we used a type of parallel systems test. Using a duplicate set of Customer data, Sales entered the orders into the b-Commerce system as well as continuing to write the orders manually. Our Customer Service staff entered the manual orders and then compared them to the b-Commerce orders entered by Sales.

Once we were satisfied that the b-Commerce system was correctly generating orders, we went "live" using our main accounting system database. We tested the process for a couple of weeks ourselves and then put the system into the hands of our international distributors.

The last and final stage was to open BASIS b-Commerce up to all BASIS Customers. We have initially implemented a level of functionality necessary to handle about 90% of the transactions we perform. But we plan many additional features to provide even more value to Customers as we evolve the application.

Using BBj for our own Web-based business operations has turned out to be an exceptional testing situation. BASIS b-Commerce gives us the ability to accurately gauge real-world requirements for BBj in distributed computing. And as we continue optimizing BBj and its thin client interface, BASIS b-Commerce improves in its efficiency and capabilities.