Abstract
While the web is putting a new face on many universities, very few have developed more than an ad hoc strategy for deploying WWW access to administrative systems. This is in part due to the present distribution of applications across multiple platforms, in part due to the rapid change in server/browser software, and in part due to the lack of maturity of WWW server application development tools with secure and multi-protocol back doors to host applications. The present migration of administrative systems to open platforms and multi-tier architectures assures that business logic will be distributed across disparate environments for several years. In the meantime, how does an institution develop a WWW-to-business data access strategy that can support moving targets on both ends? This paper will examine how Georgia State University, building on a middleware strategy developed for Interactive Voice Response (IVR) has extended this multi-host, multi-peripheral strategy to address the WWW.
Putting a New Spin on Web/Host Applications

Spider webs come in infinite varieties. And perhaps the same can be said for WWW implementations. In establishing the web as an entry point to administrative systems, one might quickly conclude that the same could be said for host applications — that every application is different. At Georgia State University, building on ideas for a generalized client which emerged from twelve years of Interactive Voice Response (IVR) experience, we determined to spin our web application with the same stuff of which our seasoned client applications were made. We decided to identify the problems common to any WWW / host interface, and to provide a client infrastructure that would minimize the application programmer’s and WWW form designer’s investment in each new WWW front end.

In this paper, we will reason through our decision to build WWW administrative applications on top of an existing robust client interface. We will show how we are reaping the benefits of that decision with quick and consistent deployment of WWW entry points to a variety of host applications.

About Georgia State University
Located in the heart of Atlanta, just four blocks from the new Olympic Stadium, Georgia State University, is a thriving urban institution. With a Fall enrollment of 24,000, GSU is home to over 34,000 students annually, enrolled in 50 majors and more than 200 programs of study. Historically a commuter campus, in the Fall of 1996, GSU for the first time opened its doors to residents. Occupying the former Olympic Village, Georgia State’s 1500 new residents represent a dramatic new enrollment trend for GSU as well as new vitality for Atlanta’s central city.

Over 50% of Georgia State’s undergraduates are transfers. Over 30% of GSU’s students are enrolled in graduate programs. The average student age is 28. Georgia State awards over 4500 degrees per year — over 1900 at the graduate level. With a largely commuter population, telephone registration and registration via the Internet quickly became essential components in the delivery of enrollment services.

How the IVR Interface Led to Applications for WWW
Georgia State’s interest in deploying a WWW front end to the administrative database has its roots in GSU’s 1985 implementation of IVR registration. As the first public school in the nation to implement IVR technology for registration, GSU made a significant investment in hardware to support an application that was idle for much of the term. Consequently, there was a desire to leverage the IVR investment and to diminish idle capacity by implementing additional administrative applications. As additional enrollment applications were identified, the development group wished to minimize the redundant implementation of IVR-specific logic in business application software (i.e., we did not want logic specific to IVR hardware embedded within application software). So, in 1984, the concept of an IVR/host interface was born. It is a concept which has continued to evolve at GSU — in the early years, as a do-it-yourself project, and more recently utilizing IVR vendor-supplied application scripting software. As GSU’s IVR applications have evolved over 12 years, the following have evolved as goals for the IVR/host interface:

1. Optimize IVR/host interface for scalability and flexibility so that new host applications (e.g., Financial Aid Applicant Status Checking) may be added incrementally to the existing IVR implementation.
2. Integrate IVR automated attendant and voice bulletin board applications — spoken text that is independent of the host database, with IVR/host applications like registration, so that all three services are available in the same phone call, through the same software interface.
3. Maximize the reusability of code by modularizing the IVR interface into generic functions that serve a variety of host applications.
4. Maximize the reusability of business application logic (business rules) by designing an application interface that permits the same business logic to serve IVR users, terminal users, and WWW users.

5. Leverage the IVR/host interface, if possible, to serve additional end-user interfaces beyond IVR (e.g., character-based interfaces and WWW).

6. Leverage the IVR resource by distributing development of new IVR applications among several applications programmers. Distribute IVR application development by creating a multi-user IVR application development environment and by minimizing the learning curve for new IVR application developers. This eliminates potentially paralytic single-threaded application support.

7. Broaden the scope of the host interface to enable accesses to multiple hosts and multiple databases by a single IVR transaction.

8. Control the flow of the end-user transaction within the IVR/host interface enabling business logic to be stateless and as a result, transaction driven.

The IVR host/interface concept was expanded to serve a character-based terminal registration system, TEMPO, in 1992. Accessible as a Telnet application through the Internet, TEMPO has provided a direct, visual registration medium. Presently, 62% of registrations are by IVR and 26% are by TEMPO. TEMPO posed a number of interface challenges that had not been issues for IVR, and so the GSU interface model was expanded to address additional transaction flow requirements. Among these was the need to match the practical capacity of the host application, the database, and the interface itself, with a potentially unconstrained volume of incoming Internet traffic. In plain English, there was no implicit limit to the number of students who could concurrently attempt to register by TEMPO. By contrast, in touch tone, concurrent sessions was constrained by the number of telephone lines attached to the IVR unit. This added the requirement to constrain the number of concurrent Internet connections.

It was this experience and investment in an IVR and then a TEMPO host/interface that put GSU in a strong position to implement a WWW front end to administrative applications. From the business application perspective, the logic was already present in half a dozen business applications to address both IVR and TEMPO — it would only be necessary to enhance the interface to serve an additional end-user medium.

**Motivation and Concerns Prompting Interface Design**

Before discussing the mechanics of our WWW client interface, it is helpful to discuss our motivation and the concerns we were addressing. These are grouped into four objectives:

- Utilize the emerging universal access standard
- Provide for security and context in application flow
- Maximize re-use of existing business logic and gateways
- Ensure flexibility and extensibility

A primary motivation for developing a WWW interface was to leverage the features of this new medium. While our existing application platforms (IVR and the Telnet-based TEMPO) are fine for many applications, they are not well suited for tasks which require large amounts of text input — whether that be text data entry, or voluminous informational displays. Examples of such text-based applications include applications for admission and housing (or any applications requiring entry and correction of names and addresses). The web and HTML offer a rich tool set, aimed primarily toward presenting information, and hence are well suited for text intensive applications. Also, by utilizing the forms feature of HTML, which allows for a two way flow of information, we saw an opportunity to more tightly integrate our applications with the presentation of related information (e.g. linking course name to appropriate course description and prerequisite list in the General Catalog). We further wanted to
capitalize on the emerging web browser standard — the fact that these applications would be accessible to any Internet-connected, browser-equipped computer without the need for special client software.

In moving to this new platform we were confronted with significant obstacles in the areas of security and flow control (i.e., providing for context within the flow of the application). Because the web and HTTP evolved more as a vehicle for information presentation than as an application platform, security and application flow were not design priorities. As a result, HTTP came to be a connection-less, stateless protocol with relatively few security features. The underlying standard notwithstanding, it was critical, in designing an application structure for display and update of potentially sensitive data, to address real security concerns. It was also necessary to maintain the user’s context within an application in order to provide for a coherent approach to application navigation. As a connection-less protocol, HTTP requests or transactions arrive at the server with no indication of what has transpired immediately before. To provide for context, it is necessary to ensure that a reference to context exists for every stateless transaction that arrives for processing.

We sought to utilize the tiered architecture that was already present for IVR and Telnet access methods by re-using existing business logic and protocol gateways. It was apparent that maximizing the re-use of existing business logic and protocol gateways would decrease our investment in WWW technology. The LU6.2 gateway to legacy host data and an application bus, components of the TEMPO/IVR host interface would be requirements of a WWW interface as well. On the host, there were business logic components already interfaced to TEMPO and IVR that could link to a web application, too. As we develop new applications with WWW front-ends and add more web entry points to legacy applications, it is equally important to design the business logic of these applications for applicability to the other access media as well.

Finally, we wanted to ensure that our web application platform remained flexible and extensible. As our applications were webbed, we needed the ability to adapt quickly to feedback from testing and early production experience. And with many new technologies on the horizon, such as new directions in the use of IVR, and our intent to employ emerging web tools for ORACLE SQLNet, we needed our host interface to represent a sufficient level of abstraction to permit extension in these new directions. This required a robust data structure with the ability to handle many different types of data flow into and out of the various applications and hosts.

**Building WWW Middleware on GSU’s Existing Application Interface**

In attempting to best address these motivations and concerns, we chose to build upon our previous successes by extending our existing application architecture with the integration of the WWW interface. As our TEMPO, Telnet-based, on-line registration interface already performed many functions required of a WWW interface, it seemed natural as a starting point. But there were differences that had to be addressed. TEMPO is a line-at-a-time medium. HTML is forms. TEMPO establishes a session while HTML’s relationship to its server is connectionless and stateless.

The following overview of the interface, elaborated in a later section, is useful to the present discussion. The existing tiered architecture interface included presentation-specific processes for IVR and TEMPO. This architecture shared common communication modules within the interface and common business logic on the host (see Figure 1). The application bus, providing queued access to communication modules, was already in use by IVR and TEMPO. Because of the application bus’ generic pipe-like functionality, providing service to an additional presentation medium, HTML, was straightforward. It was simply a matter of attaching the additional presentation medium’s process to the bus and related communication modules.

The interface control process addresses two major functions: provide application security and maintain the user’s state in the application flow. Not only is security important from a protection of information
assets perspective, but security also plays a critical role in session continuity. Security for continuity sake is provided through an exchange of keys technique in which a unique key is generated and embedded in the browser-bound HTML. Later, when the user submits that form, the same key is returned to and verified by the control process assuring that the user should be allowed to perform the requested next function. If the transaction is validated, the requested service is performed, a new key is generated and imbedded in the HTML exchange, and the sequence starts over. By generating a new key with each transaction, in conjunction with utilizing the data encryption provided by the Web Server, sufficient security is provided to assure that user authentication achieved early in a session remains reliable with each subsequent form submission. The exchange of keys also assures timeliness in the exchange — each key has a limited useful life. When the interface issues a form to the browser, that form is set to expire in a short time. After say, two minutes or ten minutes, that form is no longer valid for input, and the user must go through the login sequence again.

Not only does exchange of keys assure continuity of user (same user for entire session), it assures from the application perspective — actually from the middleware perspective — that the interface control knows the context of the user within the application. To clarify, when a form arrives, it is imperative that the interface control software knows the user’s context as it may be relevant to requested navigation. Returning to a previous menu or backing up in an application where forms are generating updates may not be a simple matter of using the browser’s backup key as in a typical web application. Here, the interface control software is utilizing data structures in its memory space to maintain context on many concurrent users — all whose sessions have not yet expired. This operates much as it did previously for TEMPO.

Figure 1: Overview of Processes
The functions and text, except for actual variable values which reside with each individual client process, are stored in shared memory for use by the WWW interface clients. These functions and text are stored in "byte-code" form. This partially compiled, compact form of representing functions is similar to the form in which a Java application is transmitted across the Web to a browser. A middleware client process springs into existence for each incoming transaction, actually interpreting this byte-code (which is why you will see these middleware processes referred to as byte-code interpreters or BCIs throughout this discussion). While this shared memory approach is not absolutely necessary in accomplishing our goals, it has greatly increased the efficiency with which our processes perform over earlier strategies.

**Developing a New Host/WWW Application**

Before we delve too deeply into the conceptual structure of our application interface, it may be helpful to take a step back and look at the interface from a very practical perspective: What is involved in developing and deploying a new host application over the WWW?

Utilizing the GSU application interface and the tools designed to make the interface practical, here are the relatively simple steps required to develop an application (path to existing administrative data) for the web platform:

- Identify application flow.
- Design HTML for data entry (user input form) and response screens (application feedback).
- Develop or modify business logic, if necessary, using API’s through the communications gateway.
- Write scripts.
- Compile scripts into Byte Code.

Following is a sample application flow for User Authentication as utilized in GSU’s WWW application for registration, financial aid, and housing applications.

**Example - User Authentication**

- Present data input screen to collect student number and PIN.
- Perform business logic (verify student).
- Respond — either positively or present error and repeat process.

The steps required to implement this task include: designing the input screen (HTML), designing the various responses messages (also HTML) needed to handle input errors, and writing the script (application flow). It was not necessary to develop business logic. It existed for use with other interfaces and required no modifications.

The compile step creates the new version of the byte code for shared memory. The compile program is run against the newly created scripts, along with those already existing, to incorporate the new elements into the application. The byte code for the new function(s) is created, the text elements are placed in the text section, and the functions and text are indexed. The compiler does not create an executable program like most compilers, but rather byte code to be executed by the byte code interpreters (BCI) that are initialized on behalf of each incoming transaction.

In order to better illustrate the components, we will study them in the context of the application flow. When a student clicks on the hypertext link that is the entry-point for our application he is actually clicking on a link such as this:

```
<a href="https://www.gsu.edu/cgi-bin/simple-form.pl?func=access-html">.
```

Which causes our web server (www.gsu.edu) to execute the CGI program "simple-form.pl", sending it
the string "func=access-html". Our CGI program executes the module corresponding to the function
name it received as input, "access-html", listed below.

```
#module access-html;
#call access-web-header;
#call access-web-form;
#call access-web-footer;
#quit;
#return;
```

As you can see, this module does nothing but call three other modules. Those three modules, and the two
that they call, are listed below in the order that they are called.

```
#module access-web-header;
#call web-header-1;
<CENTER><H1>Access Verification</H1><P><P>
#return;

!*****************************************************************

#module web-header-1;
Content-type: text/html

<html>
<head>
<title> Georgia State University Web Generator! </title>
</head>
<body
BACKGROUND="https://iweb.gsu.edu/projects/regweb/images/bg3.jpg">
#return;

!*****************************************************************

#module access-web-form;
#var {OUTPUT; name=key; length=20; }
<form method=POST action="https://iweb.gsu.edu/cgi-bin/simple-form.pl">
<input name="func" type=hidden value="ssn-access">
<input name="key" type=hidden value="^" >
Student Number: <input name="ssn" type="text" rows=1 cols=9
size="9"><br>
Access Code: <input name="access" type="password" rows=1 cols=4
size="4"><p>
<input name="submit" type="submit" value="Submit"> <input
name="reset" type="reset" value="Reset">
#return;

!*****************************************************************

#module access-web-footer;
#call web-footer-2;
#return;
```
The result of the execution of these modules is the HTML listed below:

```html
<HTML>
<HEAD>
<TITLE>Georgia State University Web Generator!</TITLE>
</HEAD>
<BODY BACKGROUND="https://iweb.gsu.edu/projects/regweb/images/bg3.jpg">
<CENTER><H1>Access Verification</H1><P>
<FORM METHOD=POST ACTION="https://iweb.gsu.edu/cgi-bin/simple-form.pl">
<INPUT NAME="func" TYPE=hidden value="ssn-access">
<INPUT NAME="key" TYPE=hidden VALUE="00000000200000012754">
Student Number: <INPUT NAME="ssn" TYPE="TEXT" ROWS=1 COLS=9 SIZE="9"><BR>
Access Code: <INPUT NAME="access" TYPE="PASSWORD" ROWS=1 COLS=4 SIZE="4"><P>
<INPUT NAME="submit" TYPE="SUBMIT" VALUE="Submit"> <INPUT NAME="reset" TYPE="RESET" VALUE="Reset">
</FORM>
</CENTER>
</BODY>
</HTML>
```

You can see how each module’s output is contained in this HTML document. When a browser renders this HTML, the result is the access verification form shown below (Figure 2):
When a student enters the requested information and submits the form, the CGI program is executed and passed the following data: "func=ssn-access,key=9999999999,ssn=999999999,access=9999". Note, this data is encrypted by the browser for transmission and decrypted upon arrival at the web server. The CGI program executes the module "ssn-access" (listed below) which operates on the remaining data (ssn and access).

```
#module ssn-access;
#host {func=access; dialog=oasml308; var=ssn; var=access; }
#if resp="OK";
#    call stu-menu-html;
#else_if resp="STU-NOF";
#    call stu-nof;
#else_if resp="INVALID-ACCESS";
#    call invalid-access;
#else;
#    call fatal-html;
#end_if;
#quit;
#return;
```

This module consists of a call to the host to execute the function, "oasml308" (a dialog within CA-IDMS/R database) using the variables "ssn" and "access". Oasml308 is the same program used to verify access for IVR and TEMPO users. It simply compares the user-provided id number and pin number with those stored in the student database. If the host responds favorably ("OK"), the student menu is presented. Otherwise, one of the following modules is executed depending on the returned value from the host.

![Figure 2: "Access Verification" Screen](image)

To EXIT, please close this window by clicking on the "X" at the top of the screen.
Which results in (Figure 3):

![Invalid Access Code Screen](image)

Figure 3: "Invalid Access Code" Screen

Or:

#module stu-nof;
#call access-web-header;
<CENTER><H1>STUDENT NUMBER NOT FOUND.<BR>Please re-enter your student number and access code.</H1></CENTER><P><P>
#call access-web-form;

The student then re-enters the information and the process is repeated. It should be noted that, by breaking down the script logic required to create these screens into components (headers, footers, etc.), modules can be re-used, minimizing coding effort and maintenance.

Components Driving the Host/Web Interface
In order to best illustrate the structure of the application and the functioning of each component (Figure 5), we will walk through the illustration below, using our prior example: a student attempting to enter the application by validating their student number and PIN. Recall that the student requested the form by submitting the URL "http://www.gsu.edu/cgi-bin/simple-form.pl?func=access-html". We will discuss the details of this request shortly, for now suffice it to say that information reached the control process informing that a client wished to access the application. The control process checks its administrator configured "maximum connections" value against the number of users currently in the application. If the maximum has not been reached, the controller process returns a unique key, creates a data structure identified by that key, stores the "end time" (the maximum duration of the session, also administrator
configured) for this user’s session in the data structure, and returns the key to the client. The user is then presented with a form consisting of two input fields, one for student number and one for PIN. If you examine the HTML source code for the form (which you can if you look back to the example), you will notice that there are also two hidden fields. One is labeled "function" and contains the value "ssn-access". The other is labeled "key" and contains a literal string value. Both hidden fields will be discussed later. These hidden fields are not visible when viewing the form in the browser window, but as you will see, they play an important role.

The student inputs the student number and PIN and presses the submit button, sending the web server the URL "http://www.gsu.edu/cgi-bin/simple-form.pl" (which is the value in the current "action=" portion of the form definition, and is the URL for the one and only CGI program) followed by "name = value" pairs for the two hidden fields and the two user supplied fields. The web server realizes that, since the URL calls for a resource ("cgi-bin/simple-form.pl") located in the cgi-bin directory, it should execute the program "simple-form.pl" and pass the data along to it. As "simple-form.pl", our CGI program, is really nothing more than a pipe, its execution results in its connecting and passing data to the byte code interpreter (BCI) on the application server machine.

The BCI server, upon accepting the connection from "simple-form.pl", initiates (in UNIX terms, initiates an EXEC) an individual BCI, as it does with every incoming transaction. The BCI stores the incoming data in its memory for use during execution. As is also the case with every incoming transaction, the BCI places the key in a portion of shared memory known as the "controller mailbox" and contacts the control process. It is here that the first of the two hidden fields, the key, comes into play. With the exception of the initial request for the access verification form, every incoming transaction contains a unique key. As stated earlier, this key, created by the control process and sent out to the client as a hidden field embedded in the HTML, is used by the control process to accomplish two objectives: provide security and maintain a user’s state in the application.

Security is provided by the fact that each incoming transaction will contain a unique key. As the web server provides data encryption between the web server and the user’s browser, we can with considerable confidence know that only the user’s browser who received this particular key could be submitting it back to the application. The fact that each individual transaction contains a unique key assures us that an unknown user has not used the "back" button of a browser to back up to a screen cached by the browser. This could happen in a computer lab situation where one student walks away from an open browser session, leaving cached web forms accessible to a subsequent student. If the subsequent student attempts to re-submit the form data from a cached form, he will be submitting a "stale" key — one already submitted and hence now invalid.

Maintaining the user’s state in the application is also accomplished through the user’s key. As mentioned before, when the control process created the key in our example, a data structure identified by that key value was also created. As HTTP over the web is a connectionless protocol, this data structure is the only persistent element of the application. When the control process retrieves the key from the section of shared memory designated as the controller mailbox, the key is used to locate the user’s data structure. If there is no matching key, the user is presented with an error message. An error message could arise because of (1) a stale key (a reused key as described above), (2) a user has taken too long to respond and the control process views the form as expired, or (3) a security breach is being attempted. Whatever the reason, the error generates an HTML form containing an error message plus the data entry fields for initial entry to the application (student number and PIN). If these errors are not detected, a new key is generated, and returned to the controller mailbox.

The BCI, executing on behalf of this transaction, retrieves the new key from the mailbox and stores it. If no error condition was returned by the control process, the BCI continues it’s processing, attempting to carry out the user’s request. The BCI uses the "function" argument (recall "function=ssn-access") to
index into the byte code portion of shared memory. Once the module "ssn-access" is located, the byte code at that location is interpreted and executed by the BCI. This module, the source for which is listed in our earlier example, instructs the BCI to retrieve the values "ssn" and "access" from its memory and communicate this data to the database, accessed through a particular business logic module, "oasml308".

This communication takes place through the functioning of two additional components of the interface: the application bus and the communication processes. The BCI places its request data on the application
bus, along with sufficient address data to get this data to the proper host and return address data to get it back to the proper BCI. The host-specific communications processes, which are constantly polling the bus for new transaction requests, see and respond to new data on the bus. The appropriate communication process, as determined by the host address data included in the packet, picks up the data from the bus, stores the BCI process identifier and connects to the host. This connection remains open as the data is passed to and acted upon by the host (e.g., host attempts to verify the student id and PIN against the database), and as a response ("response=OK" followed by information pertaining to the student, such as name, etc.) is returned. The communication process, receives the host response and places it back on the application bus, addressed to the respective BCI. The respective BCI retrieves its host response from the bus.

As the BCI continues execution of its module, "ssn-access", it compares the host’s response to the possible responses in the if-else section of the module. Upon finding a matching response (we will use "OK" in our example), the module’s byte code, "if response=ok; call stu-menu-html", instructs the BCI to retrieve from shared memory the byte code for module, stu-menu-html. This module, consisting mostly of text, and stored in the text section of shared memory is to be submitted as HTML to the browser to render a new form. Within "stu-menu-html", which again is mostly HTML, the BCI may detect embedded variables, for which it substitutes the respective values. The purpose of the form, "stu-menu-html", is to function as a menu of options from which the student may choose to request various services provided by the application. The aforementioned variable values (e.g., student name, the key unique to this submission of this form across the web, etc.) are substituted into the HTML in preparation for transmission to the client. As the execution of the function, "stu-menu-html", results in a form of type “menu”, a special process takes place. The BCI informs the control process, via a "push menu" to place this function on the menu stack in the event that the user subsequently attempts to back up to a higher level menu (or "pop menu"). This communication takes place in much the same was that the key is communicated and updated. Once this process is complete, the HTML generated by the BCI is transmitted back to the CGI-program on the web server and the BCI program terminates. The CGI program, being merely a pipe, passes the HTML back to the web server which in turn passes it back to the client’s browser for use by the student. As there is now no persistent data left behind by the BCI, there would be no way of maintaining a client’s state in the application were it not for the data structure maintained by the control process on behalf of the client.

Benefits of a Standard Web/Host Interface

Through the use of a standard interface between our web server and applications on several hosts, we have reduced the mean time to deploy new web/host applications by nearly an order of magnitude. The initial University Housing application was deployed in eight months. The second application, Registration and Fee Collection, which was far more complex, was deployed in five months. The next suite of applications — all student-related: Personal Information Display/Update, Transcript Request, Reentry Application, Grade Display, Financial Aid Application Status and Award Display, were deployed in four months. As an example, the Grade Display, a retrieval-only application which had already existed in TEMPO and IVR was deployed on the Web in less than two hours. The next suite of applications we intend to address are Human Resource applications, Professor Roll Retrievals, and Grade Recording.

In the mid-1980’s, lacking standards for interfacing host applications to IVR, we developed our own IVR interface software. That software served us well for seven years, at which time it was clear that vendor supplied interfaces had reached a level of robustness and functionality, that it no longer made economic sense to maintain a home-grown interface. Again, Georgia State finds itself on the early portion of the industry learning curve as we are connecting a variety of host applications to the Web. As in our IVR experience, there will come a time when industry standards will make it possible for vendors to supply economically viable solutions to this problem as well. In the meantime, we will continue to expand our service to customers by deploying Web applications to meet their ever changing and ever expanding requirements.