MAY 1994

NSL
Technical Note TN-14

Using Tcl to Process HTML Forms

Glenn Trewitt

digital **Network Systems Laboratory** 250 University Avenue Palo Alto, California 94301 USA

Using Tcl
to Process HTML Forms

Glenn Trewitt

March, 1994
External release: January, 2005

Abstract

One of the most interesting uses of the World Wide Web is the creation of interactive applications that let users ask questions, query databases, or place orders, using HTML forms. Most of the work of developing such applications is the creation of programs to process these forms. John Ousterhout's Tool Command Language (Tcl) is an ideal tool for this task. With additional extensions, Tcl can access databases and be distributed across machines.

This paper gives a brief introduction to HTML forms, how to create them, and how to process them using Tcl. We assume that the reader is familiar with the World Wide Web, the operation of Mosaic, and the Hypertext Markup Language, HTML.

Network Systems Laboratory 250 University Avenue Palo Alto, California 94301 USA

Table of Contents

1. Introduction

The World Wide Web provides easy access to documents stored anywhere on the Internet. Web browsers, such as Mosaic, give a "point-and-click" interface to these documents. A recent addition to this interface is the ability to have fill-out forms, which allow users to make complex queries of a Web server. These forms make it possible to build sophisticated interactive applications, using Mosaic as a publicly-available user-interface engine.

The Web interface to the Future Fantasy Bookstore is one such application. Using forms, it allows users to browse the catalog, order books, and send in comments. It is accessible at the Uniform Resource Locator (URL) http://www.commerce.digital.com/palo-alto/FutureFantasy/home.html. Most of the examples in this report are drawn from our implementation of the bookstore's Web "storefront".

If you have access to the Web, you should take some time, now, to look at the bookstore and experiment with the browsing facilities. Please be aware that this is a real bookstore, and that any order you place will be treated as a real order. If you just avoid giving your name and address, you can't accidentally place an order.

This report discusses the following topics:

Chapter 2 -- Overview and Tutorial
How Hypertext Markup Language (HTML) forms are constructed and used to pass data to applications. A brief tutorial example is included.

Chapter 4 -- Using Tcl to Process Forms
How Tcl can be used to process forms, highlights of the language, and a more extensive example.

Chapter 5 -- Tcl Extensions
Additional Tcl extensions that are useful for more sophisticated applications, such as database access and distributed applications.

Chapter 6 -- Building Large Applications
Issues relevant to building large applications using forms.

Chapter 7 -- Additional Details
Miscellaneous details -- useful tricks and nasty pitfalls.

Appendix I -- Building HTML Forms
An in-depth description of the input elements available for forms.

Standards and implementations change rapidly. This report is current as of the following versions of software:

- X Mosaic version 2.4
- Windows Mosaic version 2.0a4
- Macintosh Mosaic version 1.3
- NCSA httpd version 1.2
- Tcl version 7.3

Most of the Tcl-related files mentioned in this report are available by anonymous FTP from **nsl.pa.dec.com** (Digital internal only), in the /pub/tcl directory. Outside of Digital, cgi_tcl is available from **gatekeeper.dec.com** in the /pub/DEC/NSL/tcl directory.

The primary repository for Tcl sources and documentation is **sprite.berkeley.edu**, in the /tcl directory. The master archive of Tcl extensions, as well as a mirror copy of the Tcl sources, is at **harbor.ecn.purdue.edu**, in the /pub/tcl directory. Tcl itself is described in *Tcl and the Tk Toolkit* by John Ousterhout, published by Addison-Wesley, ISBN 0-201-63337-X.

2. Overview and Tutorial

The Hypertext Markup Language (HTML) is the language that "native" web documents use. It provides the formatting directives for Web documents, such as headers, horizontal lines, lists, etc. If you want to see what an HTML file looks like, you can use the **View Source** option in Mosaic’s file window. This displays the HTML source for the current document.

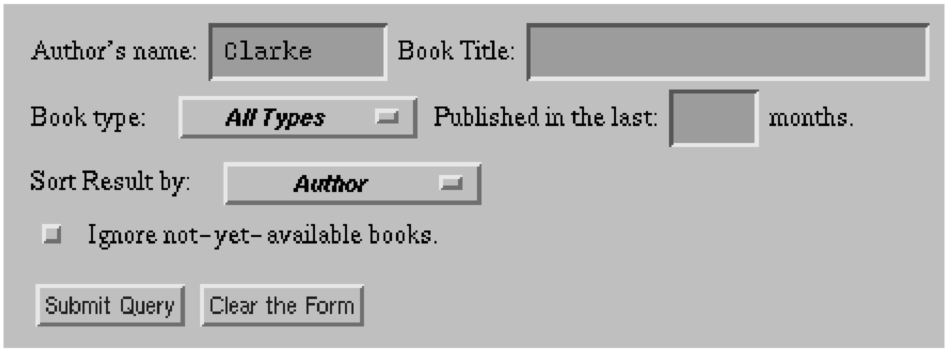
An HTML form is a special portion of an HTML document that defines document components with which the user can interact. Using forms, users can provide input to a remote application. Forms are useful for many applications, such as database lookups, order entry, or general searching.

To write and process HTML forms, you need to understand the following concepts:

- The architecture of the World Wide Web -- the relationship between Web browsers (such as Mosaic) and HTTP servers (such as NCSA httpd). (Section 2.2).
- HTML, for presenting information to the user. http://www.ncsa.uiuc.edu/demoweb/html-primer.html is a good introduction.
- HTML forms, for obtaining information from the user (Appendix I).
- The Common Gateway Interface (CGI), which specifies the environment in which forms-processing scripts operate (Section 3).
- Tcl, which we recommend as the language of choice for forms-processing scripts (Chapter 4 and the Tcl book).

2.1. Introduction to HTML Forms

A form is a collection of simple input elements that the user manipulates to provide input. Here is an example of a simple form, which is used to search the Future Fantasy bookstore’s inventory:



Author’s name: Clarke Book Title:

Book type: All Types Published in the last: months.

Sort Result by: Author

□ Ignore not-yet-available books.

The following input elements are available in HTML forms:

- Single-line text input (echo or non-echo)
- Multiple-line text input, with scroll bars
- Pop-up menus
- Scrolling lists (single or multiple selections)
- Checkboxes (select zero or many options)

- Radio buttons (select exactly one option)
- Invisible text, for holding state information. (More on this, later.)
- Bitmap images, for sending X,Y coordinates.

Two special buttons are available: the **Submit** button, which sends the query to be processed, and the **Reset** button, which resets the form to its default state. These are evident in the example as the **Submit Query** and **Clear the Form** buttons. When the form is submitted for processing, the data from all of the elements in the form is packaged up into a query and sent to a Web server.

The official reference for forms is http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html. Appendix I gives a tutorial on the different form elements.

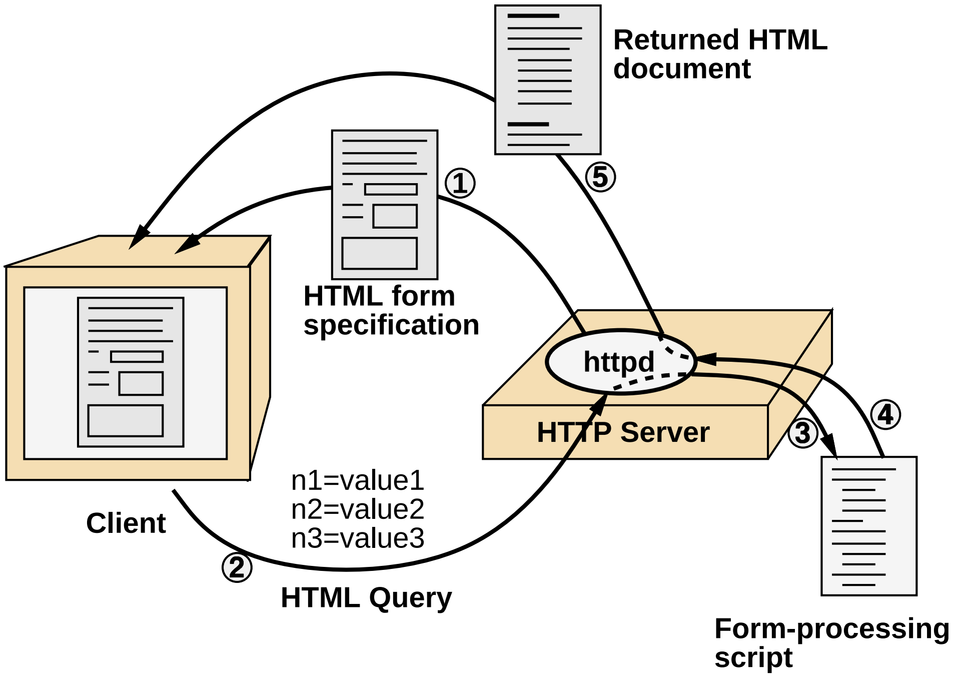
2.2. Query Processing Overview

HTML documents are transferred across the network using the Hypertext Transfer Protocol (HTTP). The two participants in the transfer are the *client*, which is usually a program like Mosaic, and the *server*, which is usually named **httpd**. There are a few different HTTP servers. The one we use comes from the National Center for Supercomputing Applications (NCSA) at the University of Illinois.

An HTTP server supplies HTML documents to a Web client, which formats them for presentation to the user.

An HTTP server initially sends an HTML document it to a Web client in response to a request by the user (**1** in the figure). Usually, the request is the result of the user clicking on a hyperlink in some other document.

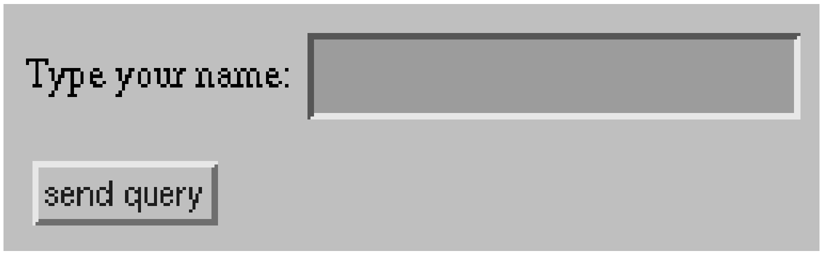
When the user clicks on the **submit** button of an form within the HTML document, the data that the user has entered in the form are packaged into a query and sent to the HTTP server (**2**). Usually, this will be the same HTTP server as the one that sent the form to the user, but the form may contain instructions to use a different server.



When the HTTP server receives the query, it runs a program (**3**) which processes the query. The script output (**4**) is a new document, which is sent back (**5**) and displayed to the user.

2.3. A Simple Example

Here is an extremely simple form and its accompanying Tcl script. This example illustrates most of the features of simple form-processing scripts. The form has a single text input field for entering a name, and a button to submit the form for processing.



Here is the source for the form:

```
<FORM METHOD="POST" ACTION="cgi-bin/simple">
Type your name: <INPUT TYPE="text" NAME="username">
<P>
<INPUT TYPE="submit" VALUE="send query">
</FORM>
```

The HTML form has three components:

1. The `<FORM>` tag, which surrounds the entire form. It has two attributes:
   - `METHOD="POST"`, which makes the query be sent as a `POST` query, with the query contents sent as standard input, rather than as part of the URL. `POST` is the preferred query mechanism.

- ACTION="cgi-bin/simple", which tells the name of the script to execute. In this case, we've given a relative URL, which means that it will look for the script **cgi-bin/simple** in the same directory as the current document. Section 7.1 describes some extra configuration that needs to be done to NCSA httpd to make this script get executed.

2. <INPUT TYPE="text" NAME="username">, specifies an input element for entering text. When the query is sent, the user's input will be associated with the name **username**.
3. <INPUT TYPE="submit" VALUE="send query">, specifies the button that will submit the query for processing.

The **cgi-bin/simple** script that executes the query extracts the user's name from the query and generates a minimal HTML document as a response, confirming what the user entered. Here is the script:

```
#! /usr/local/bin/tcl_cgi

cgi_parse_query [cgi_get_query] Q

puts "Content-type: text/html

<TITLE>Query Response</TITLE>
<H1>Query Response</H1>

Your name is <B>$Q(username)</B>.
<P>
You sent your query from <B>$env(REMOTE_HOST)</B>."
```

It consists of:

1. The first line tells the Unix operating system to execute the program **/usr/local/bin/tcl_cgi** (which is our Tcl interpreter) to run the script. This is what actually causes the script to be executed.
2. The first command uses the CGI extensions to get the query and then parse it, putting the name/value pairs into the array **Q**.
3. The "puts" command (which spans the rest of the script) writes the multi-line string to standard output:
    a. The first line of output should always be "Content-type: text/html" and should be followed by a blank line. This is the header of the HTTP response. Additional header lines are possible, but they are rarely used.

    b. The next two lines are the normal beginning of an HTML document: <TITLE> and <H1> header.

    c. **$Q(username)** is whatever the user typed in the input field.

    d. **$env(REMOTE_HOST)** is the value of the REMOTE_HOST environment variable, which was set by httpd to the name of the host that submitted the query. Section 3.2 lists all of the possible environment variables.

The output of the script is sent back to the Web client that made the request, which formats it and presents it to the user.

2.4. Protection From Abuse

It is easy to write programs to process form queries. Unfortunately, it is not so easy to make sure that your programs can’t be tricked into doing something they were never intended to do. The most important rule to follow is: **Never process user input in a way that it might be evaluated during the process of executing a command.**

This is easier said than done, because there are many, many ways to be tricked. The worst thing you can do is to use **sh**, **csh**, or other Unix shells for writing scripts. Beyond that, there are some simple rules that must be followed carefully:

1. Be careful with the **system()** and **popen()** subroutines. It is common to write something like:

   ```
   sprintf(command, "grep \"^%s\" catalog", userdata);
   system(command);
   ```

   This can easily be abused if the user passes a string that contains a command: "**foo‘rm -rf *‘**". **system** will pass the command to a shell, which will dutifully execute the command "**rm -rf ***", before executing the intended "grep" command.

2. The equivalent construct in the shell is:

   ```
   grep "^$userdata" catalog
   ```

   It has exactly the same problem.

3. In Tcl, be careful with the **eval** command. Don’t use it on any user data, which might contain hidden commands.

Furtunately, Tcl scripts are fairly immune to these types of attacks, because:

- Programs must be invoked explicitly, using the **exec** command.
- Arguments in the **exec** command are not processed through a shell -- they are passed directly to the program. So, the "grep" command, above, would look like:

  ```
  exec grep "^$userdata" catalog
  ```

  It is not vulnerable to hidden commands.
- The **eval** command, which could be abused, is rarely used. Script writers should be very careful when using it.

2.5. Processing Forms With Shell Scripts

Don’t Do It.

We strongly recommend that the conventional shells sh, csh, *etc.*, not be used to process form queries.

If you are willing to believe the claim that the use of **sh**, **csh**, or other Unix shells for processing potentially malicious user input is a bad idea, you can skip this section. The basic problem is that these shells provide a very wide-open environment for interpreting (and executing) user input. Without extreme care, it is easy to accidently program a security hole into your system.

There have already been reports of system break-ins, due to poorly written form-processing shell scripts.

Recall that an HTTP form query looks something like:

```
    n1=value1&n2=value2&n3=value3
```

When the script is started, the query must first be obtained from either standard input or an environment variable. Then, the query must be parsed, first splitting it at the ampersand characters, then breaking apart the name/value pairs, and then processing any escaped characters in the names and values. NCSA httpd comes with software to do this.

Typically, a shell command must be executed to set a shell variable (corresponding to one of the *names*) to the corresponding value. Later on, these variables will be used to execute commands, form command arguments, etc. This is where the biggest vulnerability comes from: if shell meta characters (such as ’, ", \, ‘) are included in the input, a shell syntax error can result. At worst, a malicious (and patient) user can execute arbitrary commands from your script.

A relatively minor disadvantage of shell scripts is that they tend to execute a lot of commands, which makes them significantly slower than most other solutions.

3. The Common Gateway Interface (CGI)

The Common Gateway Interface defines how the HTTP server communicates with programs (which are generally scripts) that are invoked to process a query. It defines mechanisms for passing information to these programs and returning the result.

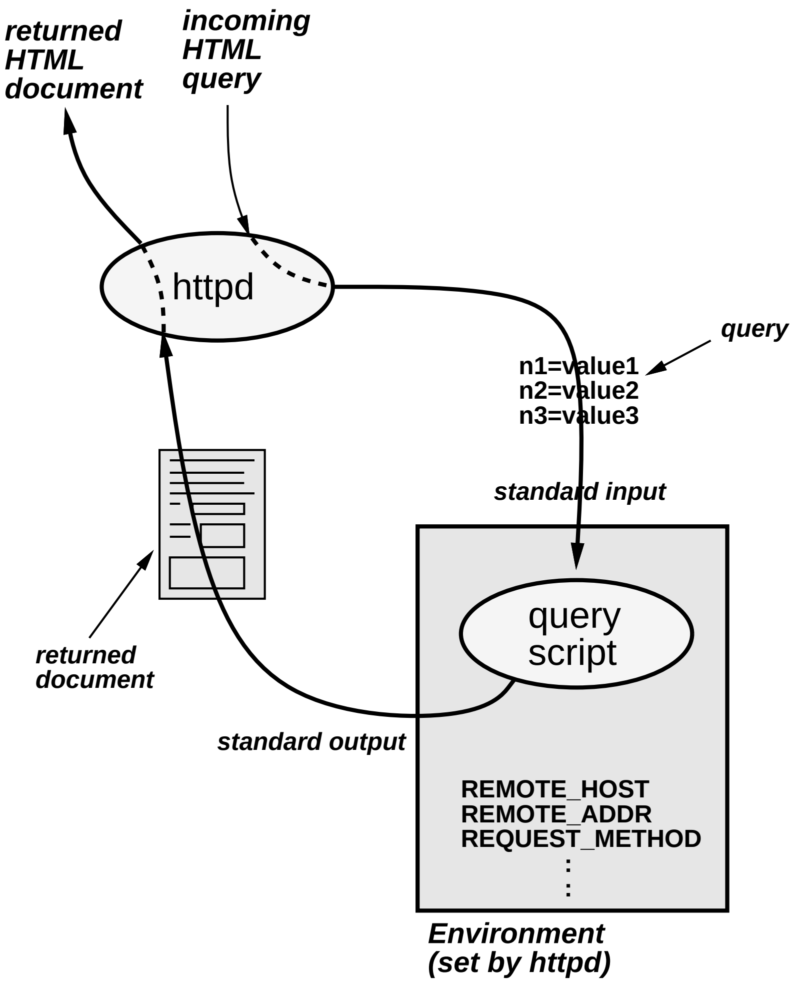
It is the job of the script to obtain the query and other related information (from standard input and/or environment variables), process the request, and generate the response for the user (to standard output). Because tcl_cgi takes care of most of the details, you will usually only need to know how to use the **cgi_get_query** and **cgi_parse_query** commands, and be familiar with the CGI environment variables (Section 3.2).

CGI has additional features that you may occasionally need to achieve some results. Much of this chapter describes these features, which you should be familiar with, just in case you need them.

CGI is implemented by NCSA httpd, versions 1.0 and greater. CGI is intended to be a standard for communication between servers and scripts, so other HTTP servers implement it as well.

Communication between the HTTP server and the script uses:

- Standard input
- Standard output
- Environment variables



The query will either be contained on standard input or in an environment variable, depending upon how the form was written. The Tcl CGI package described in Section 4.2 takes care of getting the query from the right place.

In most situations, the script's standard output is sent back to the user. In some cases, the script output instructs the HTTP server to perform other actions. Section 3.3 describes all of the possibilities.

Except for the query, most of the information about the user's request is passed to the script via environment variables, set by the HTTP server. There are over a dozen standard variables (see Section 3.2). Here are the commonly used ones:

REMOTE_HOST The name of the host making the request. If the name is not available, this is not set.

REMOTE_ADDR The IP address of the host making the request.

HTTP_USER_AGENT
(CGI version 1.1 or later)
The name and version of the client program that submitted the query. For example, the current (as of this writing) version of NCSA Mosaic returns "NCSA Mosaic for the X Window System/2.4 libwww/2.12 modified".

HTTP_REFERER (CGI version 1.1 or later)
The URL of the document from which the current URL was obtained. This is especially useful for debugging obsolete references.

3.1. The Query

When the user clicks on the **submit** button, the current contents of the form are packaged into a query. A query is a set of name/value pairs:

```
    n1=value1&n2=value2&n3=value3
```

The "&" separates the name/value pairs, and and "=" separates the name from the associated value. If the value is empty, the "=" is still present. There is usually one name/value pair for each input element specified in the form. Depending upon how the form is defined, it is possible (and reasonable) for some *names* in the query to be repeated. Appendix I describes how each form input element is encoded into the query.

In the query, special characters are escaped. The encoding and decoding of the query is handled transparently by the Tcl CGI extension, so a script writer doesn't need to worry about the details.

If the FORM tag is specified with the attribute METHOD="POST", the query is passed as the body of the document in the HTTP request. METHOD="POST" is recommended for all forms.

If the FORM tag is specified with the attribute METHOD="GET", the encoded query is appended to the URL, and will be available through an environment variable. METHOD="GET" is provided for backward compatibility, and for "hardwired" queries (see Section 7.2).

3.2. CGI Environment Variables

As of CGI version 1.1, the following environment variables are defined. Most of this list was lifted from http://hoohoo.ncsa.uiuc.edu/cgi/env.html. That document contains the definitive list.

SERVER_SOFTWARE
(name/version) The name and version of the server. *e.g.*, "NCSA/1.0"

SERVER_NAME The hostname or IP address of the server.

GATEWAY_INTERFACE
(name/version) The version of the CGI specification in use. *e.g.*, "CGI/1.0"

SERVER_PROTOCOL
(name/version) The information protocol that was used to send the request. *e.g.*, "HTTP/1.0"

SERVER_PORT The port number on which the request arrived.

REQUEST_METHOD
The method used for the request. For HTTP 1.0, this is either GET, HEAD, or POST. For form processing, only GET and POST are used.

HTTP_ACCEPT The MIME types that the client will accept. This is a list of comma-separated type/subtype pairs. Wildcards (*) are allowed.

SCRIPT_NAME The virtual name of the script being executed. This is the part of the URL after the host name, beginning with the first "/".

PATH_INFO If the script specified extra path components after the name of the script, it is put here. For example, if the script named "/cgi-bin/script" was invoked with a URL of "/cgi-bin/script/extra/stuff", this variable will contain "/extra/stuff".

PATH_TRANSLATED
The server provides a translated version of PATH_INFO, which takes the path and does any virtual-to-physical mapping to it. (I don't understand what the purpose of this is.)

QUERY_STRING The information after the ? in the requested URL, if any. This is how the query is passed to the script, for the GET method.

REMOTE_HOST The name of the host making the request. If the name is not available, this is not set.

REMOTE_ADDR The IP address of the host making the request.

AUTH_TYPE If the server supports user authentication, and the script is protected, this is the protocol-specific authentication method used to validate the user.

REMOTE_USER If the server supports user authentication, and the script is protected, this is the username they have authenticated as.

REMOTE_IDENT If the HTTP server supports RFC 931 identification, then this variable will be set to the remote user name retrieved from the server. Usage of this variable should be limited to logging only.

CONTENT_TYPE For queries which have attached information, such as POST and PUT, this is the content type of the data.

CONTENT_LENGTH
The length of the content, as given by the client.

If the client sends HTTP header lines that are not interpreted by the server, they are placed into the environment. For each HTTP header line, the environment variable name is formed by prefixing the header name with "HTTP_", capitalizing the name, and replacing any "-" characters with "_". For example, the content of the **User-Agent:** header line would be put into an environment variable named **HTTP_USER_AGENT**. This feature is new as of CGI version 1.1 (implemented by NCSA httpd 1.2).

Here are some environment variables, derived from HTTP header lines, that may be of use. Note that these are optional, so they will not always be available.

HTTP_USER_AGENT
The name and version of the client program that submitted the query. For example, the current (as of this writing) version of NCSA Mosaic returns "NCSA Mosaic for the X Window System/2.4 libwww/2.12 modified".

HTTP_REFERER The URL of the document from which the current URL was obtained. This is especially useful for debugging obsolete references.

3.3. Script Output

The output of the script is sent back to the user, to be displayed as the result of the query. It consists of a document header, a blank line, and an optional document body.

There are currently three choices for the document header. They are mutually exclusive:

`Content-Type:` *type of the returned document*
`Location:` *URL to be displayed*
`Location:` *Virtual path of file to return*

If **Content-Type** is specified, the body will be returned to the user as that type of document. The usual content-type is **text/html**. The body that the script generates is then displayed as ordinary HTML. This is the most common way for a script to return results. Other content types are possible, but are beyond the scope of this report.

If **Location** is specified with a URL, the user’s client program will be instructed to retrieve that URL instead. It will be as if the user had originally selected that URL.

If **Location** is specified with a virtual path, that file will be returned. A virtual path begins with a slash (/), and is interpreted starting at the HTTP server’s document root directory. In this case, the server behaves as if it had received that path in a URL, and returns that file, executes the script, or whatever. Note that this path is absolute, so that references relative to the current URL are not possible.

The blank line after the document header is required, even if no body is present.

4. Using Tcl to Process Forms

Forms must be processed using some sort of programming language. The following characteristics are desirable in such a language:

- Clean interface to the form query.
- Ability to program simple tasks quickly.
- Ability to communicate with existing applications, acting as a go-between.
- Good string-processing facilities, for generating HTML documents on the fly.
- Extensibility, so that custom functions can be added.

John Ousterhout's Tool Command Language (Tcl) meets all of these needs, and provides additional features that make it possible to quickly deliver high-quality form-based applications. With additional extensions, Tcl scripts can access databases and be distributed across machines.

The primary repository for information about Tcl, and program sources, is at **sprite.berkeley.edu**, in the /tcl directory. The README file lists the full contents, current version and available documentation. A USENIX paper on Tcl (tclUnenix90.ps.Z) contains a good introduction to the Tcl language.

The rest of this report will be much easier to understand if you understand the basics of Tcl, as described in either the USENIX paper or the first part of the Tcl/Tk book. You can probably get by with the information in Section 4.1, if you have a good imagination.

4.1. The Tcl Language

For a full description of the Tcl language, see the first part of the just-released book *Tcl and the Tk Toolkit* by John Ousterhout, published by Addison-Wesley, ISBN 0-201-63337-X.

Tcl's syntax is very similar to that of the shell or c-shell: A command is one or more words separated by spaces or tabs. The first word is the name of the command to be executed and additional words are arguments to that command. Commands return strings.

Tcl variables are set using the **set** command, and may be either plain variables or array variables. Arrays can be indexed by any string value.

```
    set foo "some text"
    set xref(alpha) "First Reference"
    set xref(beta) "Second Reference"
```

Variables are referenced by prefixing the name with **$**, just as in the shell:

```
    puts "Here is $foo"
```

would print "Here is some text" to standard output.

There are three types of quoting. All of them cause the contents to be interpreted as a single word.

"double quotes" just make the contents be treated as one word. Variable and command substitution are all performed on the contents. Usually used for text, and convenient when the text contains substituted values, as in the example.

{curly braces**}** around words prevent any interpretation of the contents. They are usually used around procedure bodies, where the contents will be evaluated later. Braces are the equivalent of shell single quotes, except that they balance.

[square brackets**]** cause the contents to be executed as a command, substituting the result of the command as a single word. Square brackets are the equivalent of shell back quotes, except that they balance.

Tcl is a fully functional high-level language, with features for:

- looping (**while**, **for**, **foreach**)
- conditionals (**if**, **case**)
- full error handling (**catch**, error backtrace)
- procedure definition, with fixed or variable argument lists
- files (open/close/read/write, plus full access interrogation)
- subprocesses (full pipelines, plus reading/writing to/from variables)
- Full math expression support. Functions written in C can be added in.

Besides these "ordinary" features, Tcl also has a number of other advanced capabilities:

- Full set of operators for operating on lists, lists-of-lists, etc.
- arrays are fully associative
- strings may be of arbitrary length
- regular expression matching and substitution
- ability to "trace" variable accesses -- execute Tcl code when a variable is read, written, or deleted.

Finally, the most important feature is that Tcl can be extended with procedures written in C. All of the Tcl language features are accessible from C-language routines:

- command creation, deletion, and execution
- dynamic strings
- variables and arrays
- parsing and expression evaluation
- generalized hash tables
- process creation and control

4.2. Tcl Extensions for CGI

One of the extensions that we've added to Tcl is a convenient, safe mechanism to obtain and parse the query, using the CGI interface. Two functions are provided:

cgi_get_query

- Obtain the query, either from standard input or from the QUERY_STRING environment variable, depending upon what request method is in use. Because this may read from standard input, it must not be called more than once.

cgi_parse_query [**-strip**] *query array* [*list-names...*]

- For each NAME=VALUE pair in the *query*, set *array*(NAME) to VALUE.

- If a NAME occurs more than once in a form, the last VALUE will overwrite earlier ones. However, in some form designs (*e.g.*, lists of checkboxes), some NAMEs are expected to appear more than once. If a NAME is given as one of the optional *list-names*, those VALUEs will be appended to *array*(NAME) as Tcl list elements. Later on, Tcl list-processing functions can be used to take apart the list.

- If the **-strip** option is given, non-printing characters will be removed from the query VALUEs.

For example, the command `set query [cgi_get_query]` might set **query** to

```
    author=Clarke&isbn=123456&isbn=987654.
```

Then, the result of `cgi_parse_query $query Q isbn` would be:

```
    $Q(author)   =  Clarke
    $Q(isbn)     =  123456 987654
```

Note that `$Q(isbn)` is a list containing all of the values seen in the query for the NAME **isbn**.

Note that both of these commands can produce errors if given bad input from the Web browser. You should definitely check for errors when executing these commands (see Section 7.5). The manual page for tcl_cgi lists all of the possible errors.

Tcl also provides access to environment variables through the array **env**. This array is read/write, meaning that changes to the array are reflected in the environment exported by Tcl.

The result of putting together Tcl and this CGI library is a Tcl "shell" that can be used to interpret Tcl scripts. Those scripts have direct access to the query and the environment variables. By convention, we call the interpreter *tcl_cgi*.

The usual way that such interpreters are used is to make the first line of the script start with "#!", followed by the name of the interpreter used to execute the script. If we put the Tcl interpreter in `/usr/local/bin/tcl_cgi`, then the first line of the script should be:

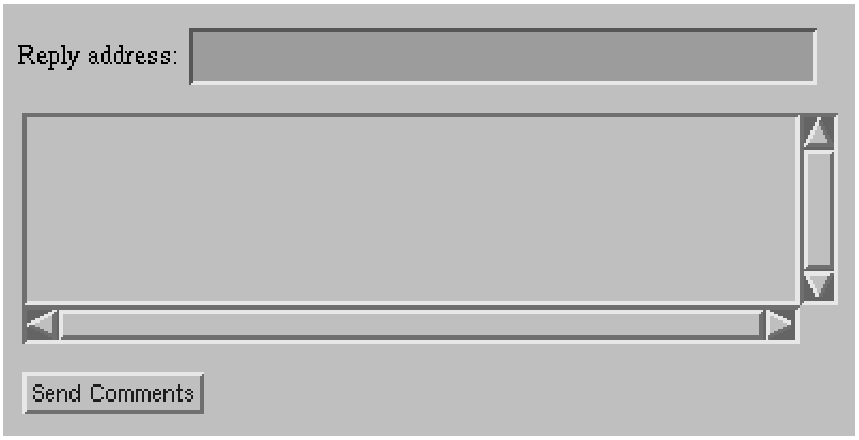
```
    #! /usr/local/bin/tcl_cgi
```

Some Unix systems limit the length of the interpreter name that they will accept to about 24 characters. If the name is longer than that, the shell **/bin/sh** will be used, which will cause confusing (and incorrect) results.

Finally, the script file must be made executable, using the **chmod** command.

4.3. A More Complex Example

Here is a more complex example that takes comments from a user and mails them to an e-mail address. The mail message is set up so that a copy of the message, and any replies, will go to the e-mail address that the user gives. Here's the form that the user fills out:



Here’s the HTML specification of the form:

```
<FORM METHOD="POST" ACTION="cgi-bin/comments">
Reply address: <INPUT TYPE="text" NAME="email" SIZE=40><P>
<TEXTAREA NAME="comments" ROWS=6 COLS=50></TEXTAREA><P>
<INPUT TYPE="submit" VALUE="Send Comments"><P>
</FORM>
```

The script that is executed when the button is pushed is named "cgi-bin/comments", relative to the location of the HTML file containing the form. The first part of the script defines a Tcl procedure "domail" that sends a mail message.

```
#! /usr/local/bin/tcl_cgi

#  This defines the "domail" procedure, which sends
#  a mail message, setting the "to", "cc", "reply-to" and
#  "subject" fields to the arguments provided, with the
#  body as given by the "input" argument.
proc domail {to cc reply subject input} {
    set fid [open "|mail $to" w]
    puts $fid "To: $to"
    if {$cc != ""} {
        puts $fid "CC: $cc"
    }
    if {$reply != ""} {
        puts $fid "Reply-To: $reply"
    }
    puts $fid "Subject: $subject"
    puts $fid ""
    puts $fid $input
    close $fid
}
```

The rest of the script obtains and parses the query, invokes the "domail" procedure, and generates an HTML response for the user.

```
set comments "comments@bigmax"

# get and parse the query
cgi_parse_query [cgi_get_query] Q

# send the e-mail message, using the above procedure.
domail $comments $Q(email) $Q(email) \
"Comments from $env(REMOTE_HOST)" $Q(comments)

puts {Content-type: text/html

<TITLE>Thank You</TITLE>
<H1>Thank You</H1>
Your comments have been submitted.}
if {$Q(email) != ""} {
    puts "<P>A copy has been sent to $Q(email)."
}
```

5. Tcl Extensions

The basic *tcl_cgi* shell provides the basic Tcl language and the ability to use the HTTP CGI to get the query. This is often sufficient, but falls short when the query needs to interface with other systems, such as databases, on-line transaction processing systems, or other existing systems.

One way to access such systems is to run whatever programs are needed using Tcl's process control facilities, sending commands and data via standard input and output. This has the advantage that the existing system doesn't need to be modified, and is just "driven" by the Tcl script. This option is quite powerful and flexible, and shouldn't be overlooked.

The other way to access such systems is to extend Tcl and build in capabilities to access the system. This is done by writing routines in C that communicate with the existing system, and can be invoked from Tcl as ordinary commands. This has the advantage that the system can be accessed directly from Tcl, without the overhead (and possible delay) of invoking the existing program. In addition, the Tcl commands to access the system can be tailored to the needs of the script. The disadvantage is that programming is required, although the Tcl aspects are not difficult.

5.1. Existing Tcl Extensions

Many extensions already exist for Tcl. Here are a few of them:

- **CGI** is available from **nsl.pa.dec.com**, in /pub/tcl/tcl_cgi_1.0.tar.Z.
- **Tk** is the most popular Tcl extension. It provides a Tcl interface to X-windows, so that interactive applications can be constructed quickly and easily. It is common to use Tk/Tcl to build an interactive front-end to an existing program. Tk is not useful for building servers, because they have no use for a windowing interface. However...
- **Tk without X** provides a key **Tk** service, the *event loop*, which allows a Tcl application to be event driven. This is essential for X, but is also useful for non-windowing applications. **Tk-X** is not yet a standard part of Tk, but is available as a patch to it, from nsl.pa.dec.com, in /pub/tcl/tk-noX.patch.Z. This patch should only be applied to Tk version 3.3. Tk version 4.0 will directly support this.
- **Tcl-DP** provides access to TCP/IP networking facilities, allowing a Tcl script to use TCP connections, etc. On top of these facilities, Tcl-DP builds a remote procedure call (RPC) mechanism that is totally integrated with Tcl. Because of the asynchronous nature of networking, Tcl-DP requires either Tk or Tk-X, for the event loop. Use of RPC is discussed in Section 5.3.
- **SQL access** from Tcl is provided by several extensions, which are available for Ingres, Informix, Oracle, and Sybase systems, although they have different command sets.

Most of the Tcl-related files mentioned in this report are available from **nsl.pa.dec.com**, in the /pub/tcl directory. The extensions are available from **harbor.ecn.purdue.edu** in the extensions directory.

5.2. Building Tcl Shells

This section describes how to build a custom Tcl shell, starting with the basic interpreter and adding a collection of extensions. If you are going to use only the CGI extension, then the default *tcl_cgi* shell will suffice.

A Tcl program consists of several parts:

- The Tcl library (libtcl.a).
- A **main()** procedure, that is executed when the program starts. One is provided in the Tcl library.
- A file named **tclAppInit.c**, containing the procedure **Tcl_AppInit()**, which is responsible for initializing the Tcl interpreter and any extensions which are to be included.

Tcl extensions obey a rigorous naming scheme that makes adding them to a basic Tcl shell fairly easy. For example, the "CGI" extension consists of:

- The **libtcl_cgi.a** library, which contains all of the compiled code for the extension.
- The **Cgi_Init()** initialization routine, contained in the library, which initializes the data structures for the extension and adds commands to the Tcl interpreter. All such initialization routines take a single argument, "*interp*".
- A **Makefile**, which controls the compilation and installation process. Most extensions' Makefiles will build the library and a Tcl shell containing just Tcl and that extension.

The procedure for adding the CGI extension to the basic interpreter is simple:

1. In the generic Tcl_AppInit procedure, add a call to **Cgi_Init(interp)**. If you are adding multiple extensions, order may be important.
2. In the **ld** (or final **cc**) command in the Makefile, include the library, by either:
    - Specifying **-lcgi**, if **libtcl_cgi.a** is installed in a standard library area.
    - Giving the name of the library, **libtcl_cgi.a**, on the command line.

   The library should be included on the command line immediately after the **-ltcl** option, which includes the Tcl library.
3. If the extension requires additional libraries, include them on the **ld** (or **cc**) command line, after **-lcgi**.

So, to create a Tcl shell that has both the CGI and SQL extensions, tclAppInit.c would look like:

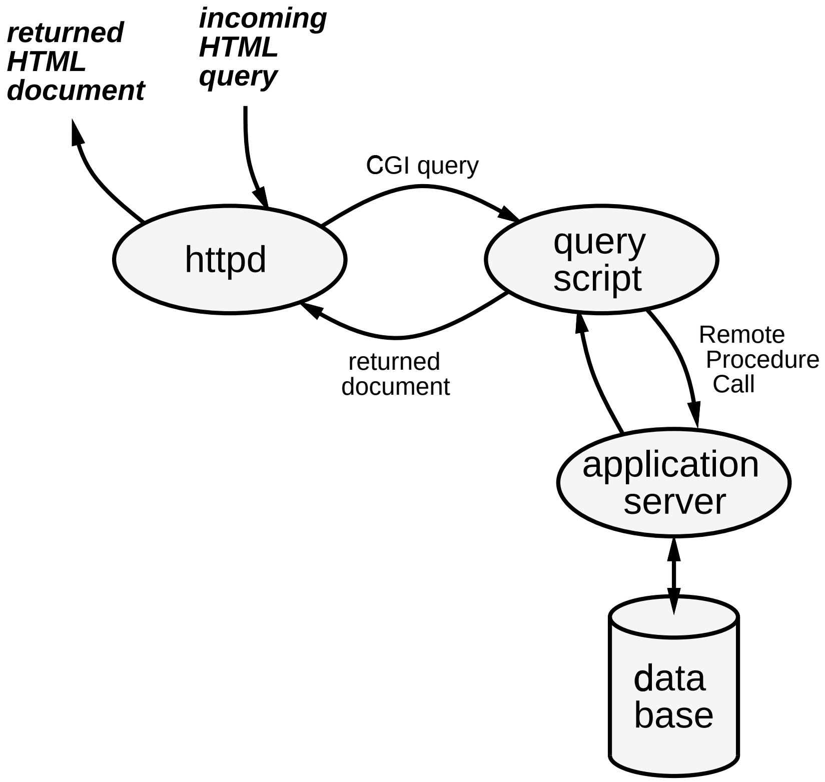
```
int
Tcl_AppInit(interp)
    Tcl_Interp *interp;
{
    /*
     *  Execute a start-up script.
     *  This may need to be loaded before the extensions!
     */
    if (TCL_OK != Tcl_Eval(interp, initCmd))
        return TCL_ERROR;

    /*
     * Calls to init procedures for various extensions.
     */
    if (TCL_OK != Sql_Init(interp))
        return TCL_ERROR;
    if (TCL_OK != Cgi_Init(interp))
        return TCL_ERROR;

    return TCL_OK;
}
```

In the Makefile, the final **cc** line would look something like:

```
cc -o tcl_sql_cgi tclAppInit.c -ltcl -ltcl_sql -ltcl_cgi -lm
```

5.3. Remote Procedure Call -- Distributed Tcl Scripts

If the Tcl scripts must access programs that are large or slow to start up (such as a database system), it may be impractical to start the entire system just to process a single Tcl query. In this case, the Tcl application can be distributed across multiple processes, with one process acting as an "application server" that runs all the time. An incoming query runs a Tcl script which makes an RPC call to the application server, incurring very little startup overhead. The control flow is illustrated in the following figure:



Note that the application server, does not have to be on the same host as the HTTP daemon and Tcl script. If needed, the application server can be replicated.

For distributed operation, two different Tcl shells are needed, one for the client (invoked by httpd) and the other for the application server (which runs all the time). The client needs the CGI, DP, and Tk-X extensions. The server needs DP, Tk-X, and any extensions needed for the application (*e.g.*, SQL).

During operation, the server executes the **dp_MakeRPCServer** command, which listens on a specified TCP/IP port. Options can be specified to restrict which hosts may issue RPC calls, and to restrict what Tcl commands may be executed. Typically, the client and server will agree on a common interface, implemented by a set of purpose-built Tcl commands. After that point, the server just waits for RPC requests to arrive.

When a client starts up, it first establishes a connection to the server, using the **dp_MakeRPCClient** command. The result of **dp_MakeRPCClient** is a handle that identifies the chosen RPC server. (Multiple connections are possible.) The **doRPC** command takes a Tcl command and sends it to a server to be executed. When the command completes, its result is returned as the result of the **doRPC** command.

When the client is finished, **dp_CloseRPC** is used to close the connection cleanly.

Tcl-DP is available from **harbor.ecn.purdue.edu** in the /pub/tcl/extensions directory.

6. Building Large Applications

The mechanisms described up to now are sufficient for building applications that process a single query. A more complex application, such as the Future Fantasy bookstore, requires several rounds of interaction with the user, such as browsing a catalog, accumulating a list of items, and then placing an order. This requires that state be kept, somewhere, about the progress of the interaction.

However, the HTTP model of operation is stateless, which means that the only thing that determines the outcome of the query is the contents of the query and the HTTP request that delivered it. The HTTP server doesn't keep any information about earlier requests. So, for a complex interaction, such as ordering books, some means of storing the state must be devised.

Sections 6.1, 6.2, 6.3, and 6.4 describe a number of issues that require obscure contortions to cope with. Based upon claims made by people who should know, it is our belief that some of these issues will be resolved by changes to the HTTP protocol, as described in Section 6.5, so that passing state will be more robust, flexible, and transparent.

For now, we recommend the maintaining state in the query, as described in Section 6.1.

Section 6.6 describes how servers can be made more efficient. The script-based forms-processing methods described so far have many advantages, but blazing speed isn't one of them. We describe how to construct applications that maintain the flexibility of script-based processing, but allow optimization where necessary.

6.1. Maintaining State -- In the Query

The simplest way to maintain state across queries is to store it in the HTML form and pass it back and forth as part of the HTML document and query.

This technique is simple to implement and always gives the user the expected results. The disadvantage is that, if the state is large, it must be sent back and forth between the client and server on every query.

State can be passed back and forth by embedding it in the form as a VALUE that, if everything goes right, will be returned on the next press of a submit button. There are two choices for how to embed the state: hidden or visible.

Hidden state is reasonable for a handle that refers to some global information, such as the result of validating the user. State that affects immediate outcomes should be visible, or at least evident, to the user.

The state can be completely **hidden** from the user, by using the INPUT type `hidden`:

```
<INPUT TYPE="hidden" NAME="foo" VALUE="here is the data">
```

- `hidden` data is completely invisible. However, if the user is unaware of the state, they may be surprised by actions taken as a result of the state.
- The `hidden` input type is new with Mosaic version 2.2. The behavior of earlier versions is to display it as a `text` field, which will be visible, and can be edited. (But at least it has a chance of working.)

The state can be **visible**, or at least evident, to the user. For example, a checklist for ordering books can embed the state in the VALUE associated with a checkbox:

```
    <INPUT TYPE="checkbox" NAME="isbn" VALUE="034847X329">
```

Only a checkbox is displayed here. The title, author, and price of the book are displayed separately, next to the checkbox. (The checkbox is particularly convenient, because it always gives the user the option to deselect the item.) In this case, the state (the book's ISBN number) isn't visible, but it is evident by the combination of the checkbox and the book information.

It is important to remember that the embedded state will only be passed along if it is in a form that the user clicks the **submit** button for:

- If there are multiple forms in a document, the state must be embedded in each one.
- If the user clicks on a normal hyperlink, no form query will be processed, so the state won't be passed on.

6.2. Maintaining State -- In the Server

A more complex method of maintaining state is to store information about an active transaction in the server, and associate a small handle with it. This handle is passed back and forth between the client and server.

This method offers several advantages:

- Only the handle needs to be passed back and forth.
- If users must authenticate before starting a transaction, and the server associates the stored state with users, then an old transaction can be resumed where it was left off. When a user authenticates, she would be asked whether she wanted to resume the in-progress transaction, or start a new one.

The obvious disadvantage to this scheme is that the server is more complicated: it must store the state and decide when to discard old, useless state that was abandoned by users.

A very subtle problem is that, if only the most recent state is stored, users may get unexpected results if they backtrack through Mosaic's history list and re-process a query from an earlier time. Based upon the contents of the window that she is looking at, the user will expect a particular outcome. However, if the query is processed based upon the most recent state, she will get a different outcome.

In contrast, when state is kept solely in the HTML form, all of the different forms stored in Mosaic's history contain state that is consistent with the appearance of the document. This has the advantage that the user has many different places from which to re-start a query.

A solution to this problem is to alter the handle at each step of the transaction and have the server store all of the previous states in the transaction. Depending upon the application, this can be very complex and expensive.

6.3. Maintaining State -- In the URL

The mechanism described in this section is somewhat speculative -- we haven't used it, and there is not currently enough information to know if it is truly effective. It is presented here as a possible option.

The URL can also be used to store a small amount of state, by embedding a small datum (such as a handle for state maintained by the server) in the URL. For example, one might construct URLs of the form:

`http://nsl.pa.dec.com/BigApp/`*handle*`/cgi-bin/order`

In this case, the server's directory **BigApp/***handle* would be a symbolic link to the single real copy of the HTML sources and scripts. When the script executes, it would examine the SCRIPT_NAME environment variable to determine the value of *handle*.

This scheme requires that either symbolic links be maintained for each possible *handle* or that the httpd server be modified to map all of these possible directories onto a single real directory.

One advantage that this has over the other state-passing mechanisms is that, as long as the URLs are local to the **BigApp/***handle* directory, **all** types of hyperlink actions retain the state -- there's no loss of state if the user clicks on a regular hyperlink.

6.4. Maintaining State -- At the End of URL

The mechanism described in this section is described in the NCSA httpd documentation, and is included here for completeness. However, we don't see any benefits over keeping the state in the query, and we haven't figured out how to use it without giving up the benefits of relative naming.

A normal query looks like

```
http://host.domain/path/name/here/cgi-bin/script
```

where *script* is the name of the script. You can append extra path components after the script name:

```
http://host.domain/path/name/here/cgi-bin/script/extra/path
```

The string **/extra/path** will be passed to the script in the environment variable PATH_INFO.

6.5. Maintaining State -- In the HTTP Header

An addition to the HTTP header has been proposed that would allow a session identifier to be carried in HTTP requests and responses. Presumably, this **Session-ID:** header line would be generated by a server, and the client (*e.g.*, Mosaic) would echo it back in later requests.

This mechanism, if it is adopted and when it is deployed, will solve many of the problems discussed in the earlier sections. Especially, it will allow ordinary hyperlinks to carry session information, so that ongoing state won't be lost.

The status of this proposal is unknown.

Note that the CGI interface will probably have to be updated to pass the session identifier back and forth.

6.6. Efficiency Concerns

The script-based forms-processing methods described in earlier chapters are nicely flexible and easy to implement, but they are not fast. The slowdown is due to two things:

- The inherent slowness of Tcl, which is interpreted. This is only a problem if the script does a lot of computation. It is not a concern if the script runs lots of other programs.
- The overhead of creating a copy of the **httpd** process, to handle each query.
- The overhead of creating a Tcl process for each query.

If Tcl interpretation is truly the bottleneck for an application, critical parts of the application can be rewritten in C, as described in the Tcl book. Or, Tcl can be abandoned altogether, in favor of C or some other compiled language. Obviously, this loses the advantages of Tcl.

The major bottleneck for most forms-processing architectures is process creation. The normal **httpd** forks a copy of itself for each incoming request, then forks again and execs the Tcl script. This overhead can be avoided by building specialized HTTP servers that serve only a single application.

An application-specific HTTP server that just processes critical parts of an application’s queries can listen on a different port than the regular HTTP server. This server can process the query entirely by itself, without invoking other programs, eliminating all start-up overhead. And, it doesn’t ever have to process a "normal" HTTP query, only the ones dedicated to the application. This allows the specialized server to be completely independent of the normal HTTP server. In fact, it is completely reasonable for the two servers to coexist on the same computer.

One of the seldom-used parts of a Uniform Resource Locators can specify the port number to use when connecting to an HTTP server:

```
    http://nsl.pa.dec.com:1234/path/name/here
```

The "**:1234**" part specifies the alternate TCP/IP port.

7. Additional Details

This chapter describes a number of small but important details that script authors should be aware of when designing and implementing their applications. You should be familiar with these issues, so that you will recognize them when you encounter them.

7.1. Making Relative URLs Work

The normal practice when writing large HTML documents is to make the references among the files in the group relative. So, you would have references like:

```
    <A HREF="../home.html">
    <A HREF="reload.html">
    <IMG SRC="../pics/banner.gif">
```

This is a very good thing to do. However, the way that scripts are normally used with *httpd*, (as documented by NCSA), makes the script names absolute:

```
    <FORM METHOD="POST" ACTION="http://x.y.z.com/cgi-bin/script">
```

This means that the script's execution environment is unrelated to the location of the HTML document that invoked the script. This makes it impossible to use relative references in the returned document.

The way to work around this is to put the scripts in a subdirectory of the HTML "application". In the HTML file, the form tag with a relative path name, looks like:

```
    <FORM METHOD="POST" ACTION="cgi-bin/script">
```

For example, the Future Fantasy bookstore's directory tree looks like:

```
    FutureFantasy
            home.html               home page
            index.html              symlink to home.html
            cgi-bin                 scripts
                    setup.tcl       common setup code
                    search          search the database
                    comments        submit comments
                    order           submit an order
            pics                    logos, etc.
                    banner.gif
                    logo.gif
            newsletters
            covers
            inventory               inventory, for searching
            log                     log files we keep
```

In the cgi-bin/comments script, URL references are relative to the **cgi-bin** directory, so a reference like <**IMG SRC="../pics/banner.gif"**> would get the file "banner.gif" from Future Fantasy's "pics" directory.

If the script is going to reference local files (such as setup.tcl), then it must first **cd** to the local **cgi-bin** directory, as follows:

```
# Get the directory name of this script and cd to it.
cd [file dirname $argv0]
source setup.tcl
```

The final need is to configure *httpd* to allow the files in the application's **cgi-bin** directory to be executed. The **ScriptAlias** command in the **/usr/local/httpd/conf/srm.conf** file does this:

`ScriptAlias` *virtual-path file-path*

For the Future Fantasy bookstore, this looks like (all on one line):

```
ScriptAlias /palo-alto/FutureFantasy/cgi-bin/
        /digital/hypertext/palo-alto/FutureFantasy/cgi-bin/
```

7.2. Hardwired Queries

It is often useful to have a regular hyperlink execute a query. For example, in the bookstore's newsletters, all references to authors are hyperlinked to queries that will find the books by that author. So, if **Arthur Clarke** appears as a link, the user can just click on it to execute the appropriate query.

This is done by making the target of the link a URL. The query is specified giving the URL of the script, a **?**, and the query: `cgi-bin/query?quick-author=Arthur+Clarke`. The key components of the query are:

- **?** separates the query from the base part of the URL. Up to the **?**, the URL should name the script that should process the query.
- **&** separates NAME=VALUE pairs in the query.
- **=** separates NAMEs from VALUEs, with the NAME on the left, and the VALUE on the right. If the VALUE is empty, the **=** must still be present.
- **+** replaces spaces. There should be no spaces anywhere in the query.
- **%***xx* is used to represent any special character that might otherwise be interpreted specially. *xx* is two hexadecimal characters. The following characters **must** be escaped this way:
    - newline - **%0a**
    - **" - %22**
    - **% - %25**
    - **& - %26**
    - **+** - **%2b**
    - **<** - **%3c**
    - **=** - **%3d**
    - **>** - **%3e**
    - **? - %3f**

  Any character may be represented by **%***xx* -- there is no harm in being conservative.

7.3. The Evil Reload Button

In the current version of Mosaic, if the **Reload** button is clicked when looking at the result of a form execution, the script will be re-executed, but POST queries will be executed without the query on standard input, making it appear that there is no query. Section 7.4 shows Tcl code to cope with this situation.

7.4. Define a Tcl Setup Script

It is extremely useful to define a Tcl script that is executed by all of your forms-processing scripts. It should do the following things:

- Define common resources, such as e-mail addresses, procedures, *etc.*
- Set the PATH environment variable, so that programs that are **exec**ed have a known environment. It is a bad idea to depend upon httpd to supply a reasonable PATH.
- Output the **Content-type: text/html** line and the required blank line.
- Do common processing, such as parsing the query.

Here is the **setup.tcl** file for the bookstore. Some of the components are described in Sections 4.3, 7.3, and 7.5.

```
#
#  This Tcl script does the setup that is common to all scripts.
#

set orders      "futfan@netcom.com"
set problems    "trewitt@pa.dec.com"
set comments    "futfan@netcom.com, trewitt@pa.dec.com"
set logfile     "../log/queries"

set env(PATH) "/usr/ucb:/bin:/usr/bin"

puts {Content-type: text/html
}
flush stdout

#------------------------------------------------------------
#
#  Proc definitions.
#

#  Get and return the value of an environment variable.
proc getenv {name} {
    global env
    if {[info exists env($name)]} {
        return [format "%-16s %s" ${name}: $env($name)]
    } else {
        return [format "%-16s %s" ${name}: "<missing>"]
    }
}

#  Attempt to execute <body>.
#  If an error occurs, log the error message, send
#  mail about it, and send an apologetic note to the user.
proc handle_error {body} {
    global env problems logfile
    if {[catch {uplevel $body} msg]} {
        puts "<TITLE>Problem Processing Your Query</TITLE>"
        puts "<H1>Problem Processing Your Query</H1>"
        puts "While attempting to process your query,
an unexpected problem was encountered:
<UL>
<LI><B>$msg</B>
</UL>
This error may have been caused by a problem in our software,
but it is more likely that the query sent by the Web browser
you are using was malformed.
We have logged this error and will look into it.
<P>
Thank you for your patience."
        set mail "
Error message:  $msg
[getenv SERVER_PROTOCOL]
[getenv REQUEST_METHOD]
[getenv CONTENT_TYPE]
[getenv CONTENT_LENGTH]
[getenv QUERY_STRING]
Internet host:  $env(REMOTE_HOST) ($env(REMOTE_ADDR))
In script:      [info script]
Active Command: [info level 0]
"
        domail $problems "Unexpected error" "" "<<$mail"
        fappend $logfile [list error
                        [string range [exec date] 4 end]
                        $env(REMOTE_ADDR) $msg]
        exit
        }
}


#  Check for the existance of the named query elements.
#  For any that are missing, send e-mail to the maintainers
#  and set the value to the empty string.
proc check_query {args} {
    global Q problems env query logfile
    set missing ""
    foreach arg $args {
        if {! [info exists Q($arg)]} {
            lappend missing $arg
            set Q($arg) ""
        }
    }
    if {$missing != ""} {
        set msg "
Missing names:  $missing
Query:          $query
[getenv SERVER_PROTOCOL]
[getenv REQUEST_METHOD]
[getenv CONTENT_TYPE]
[getenv CONTENT_LENGTH]
[getenv QUERY_STRING]
In script:      [info script]
Internet host:  $env(REMOTE_HOST) ($env(REMOTE_ADDR))"
        domail $problems "Missing query components" "" "<<$msg"
        fappend $logfile [list error
                        [string range [exec date] 4 end]
                        $env(REMOTE_ADDR)
                        [list missing $missing]
                        $query]
    }
}

#  Send the given file as a fax, to the appropriate number.
#  Normally: return 0, set <result> to the fax job number.
#  On error: return true, set <result> to the error message.
proc dofax {file result} {
    upvar $result rv
    if {[catch {
        exec /usr/local/bin/sendfax -h phax -n -d 855-9963 $file
    } rv] != 0} {
        domail $problems "Future Fantasy fax problem" "" <<$rv
        return 1
    } {
        set rv [lindex $rv 3]
        return 0
    }
}

#  Send mail to the correct people, taking input as specified
#  by <input>, which will usually be one of:
#       <<string        or <<$variable
#       <filename       or <$file
#       <@fileID       or <@$fid
#  <to> says who to send the mail to.
#  <reply> is a reply-to address, which may be null.
#  <subject> is the subject line.
proc domail {to subject reply input} {
    set fid [open "|mail $to" w]
    puts $fid "To: $to"
    if {$reply != ""} { puts $fid "Reply-To: $reply" }
    puts $fid "Subject: $subject"
    flush $fid
    exec cat $input >@$fid
    close $fid
}

#  Append the given text to a file.
proc fappend {file text} {
    set fid [open $file a]
    puts $fid $text
    close $fid
}


#------------------------------------------------------------
#
#  Initial common processing of the query.
#
```

```
    handle_error {set query [cgi_get_query]}

    #  If the query is null, something is really wrong.
    #  Probably Mosaic didn't give anything to us,
    #  which happens in v2.2 when "Reload" is used.
    if {$query == ""} {
        puts "<TITLE>Bug in Mosaic</TITLE>"
        puts "<H1>Bug in Mosaic</H1>"
        puts {Congratulations!
    You have tickled a small bug in Mosaic.
    You probably pushed the "reload" button on a page that
    you got to as the result of filling out a form.
    <P>
    It's nothing to worry about.  To recover...
    <OL>
    <LI>Click on <B>Back</B>, to get to the page with the form.
    <LI>Click on the button you originally used to submit the query.
    </OL>}
        exit
    }

    cgi_parse_query -strip $query Q isbn
```

7.5. Check for and Log Errors

Unlike Unix shells, which continue processing after an error, Tcl is very strict about error handling: If an error occurs, the script will be terminated and the user will almost certainly get a mysterious error message, formatted badly, that they can't do anything about. The Tcl **catch** command executes statements that might produce an error to be executed safely, allowing an alternate action to be taken if the statement fails.

Errors should also be logged, so that corrective action can be taken after errors do occur.

Even a well-tested script can have new errors in a new situation. This is especially true in an environment, like the web, where one of the programs is completely outside of your control. For example, the Future Fantasy scripts had been running fine for several months, when fill-out form support was added to an existing browser. This browser did one thing differently (not wrong) that triggered a latent bug in the **cgi_get_query** code. It also constructed some queries with syntax errors, making **cgi_parse_query** fail sometimes, and omitted query NAMEs at other times.

So: Just because you've defined a form correctly, you won't necessarily get the query you expect.

Here is a routine that executes a Tcl command, checking for errors. If an error occurs, it is logged and an explanation is sent back to the user. Section 7.4 gives a more complete example.

```
#  Attempt to execute <body>.
#  If an error occurs, log the error and
#  send an apologetic note to the user.
proc handle_error {body} {
    global env problems logfile
    if {[catch {uplevel $body} msg]} {
        puts "<TITLE>Problem Processing Your Query</TITLE>"
        puts "<H1>Problem Processing Your Query</H1>"
        puts "While attempting to process your query,
an unexpected problem was encountered:
<UL>
<LI><B>$msg</B>
</UL>
This error may have been caused by a problem in our software,
but it is more likely that the query sent by the Web browser
you are using was malformed.
<P>
Thank you for your patience."
        fappend $logfile [list error
                        [string range [exec date] 4 end]
                        $env(REMOTE_ADDR) $msg]
        exit
        }
}
```

This routine would be used as follows:

```
handle_error {set query [cgi_get_query]}
```

7.6. Server Security

httpd has many features that provide fine-grained control over which files and directories are available.

One of NCSA httpd’s features that is often desirable is automatic generation of indices of directory contents, so that if someone asks for a URL that names a directory, they will get a list of the directory contents. This is often convenient, but may expose information about the server that should be kept private. This feature can be disabled, either globally or locally.

Rather than disabling index generation (which will return an error message to the user), it is nicer to override the automatic generation of the index, by providing an **index.html** file in the directory. When the directory is named in a URL, the contents of that file will be returned. This prevents the automatic index from being generated, and is more friendly to browsers. Our standard practice is to make **index.html** a symbolic link to the top-level page for that directory.

Some directories may contain totally private information, such as Future Fantasy’s log and inventory directories. The file **.htaccess** controls access to the contents of that directory. To make a directory totally private, put:

```
Options none
```

in the **.htaccess** file.

7.7. Transaction Security

Early versions of Mosaic and httpd support password-based authentication. This is of limited use, because it requires users to be registered ahead of time with all of the servers they want to use. This is usually not practical, in the context of the entire Internet. Also, since the password is sent in the clear, the scheme is vulnerable to snooping. Password-based "security" doesn't provide the functionality, generality, or security needed for commercial systems.

Two things are needed:

- Authentication. Each party must know, for sure, that the other party is who they say they are.
- Privacy. Sensitive data, such as credit card numbers, must not be visible to someone who might intercept the request or response.

Security for the Web is evolving very rapidly. Mosaic 2.2 and NCSA httpd 1.2 provide support for cryptographic authentication, but it is not yet standardized. The current specification is in http://hoohoo.ncsa.uiuc.edu/docs/PEMPGP.html.

However, improved security should make very little difference to script writers. It will probably be reflected in the environment variables that are set by httpd. The current scheme sets the AUTH_TYPE and CONTENT_TYPE environment variables from the **Authorization:** and **Content-type:** HTTP header lines.

7.8. What Client is Asking?

It is sometimes useful to identify which client program (*e.g.*, Mosaic 2.1 *vs.* Mosaic 2.4) was used to submit a query. This can be used (indirectly) to determine what features are supported by the client program. For example, named submit buttons don't work in Mosaic 2.2, so a script could choose between using a workaround (such as radio buttons) and using named submit buttons, based upon which client/version is making the request.

Most web clients send the **User-Agent:** in the HTTP request that httpd receives. CGI version 1.1 (implemented by NCSA httpd 1.2 and later) makes this available in the HTTP_USER_AGENT environment variable.

The User-Agent information is also useful for tracking down bugs. It is useful to know if an application bug occurs only when a particular agent is making the query. (Although this doesn't necessarily mean that that program is at fault.)

I. Building HTML Forms

An HTML form is a special portion of an HTML document that contains components with which the user can interact to provide input to a Web-based application. The official documentation for forms is in http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html. This appendix is a more elaborate tutorial.

I.1. The FORM Tag

The `FORM` tag delimits a form:

```
<FORM METHOD=POST ACTION=url>
    contents of the form
        :       :
        :       :
</FORM>
```

Within the `FORM` tag, various *input elements* are used to allow the user to input information, select options, *etc*. When a user submits a form, all of the data in that form are packaged into an HTML query and sent to be processed. A single HTML document can have multiple forms, which will be handled independently -- only the data in a single form will be processed when that form is submitted.

Each input element in a form results in either zero or one NAME=VALUE pair in the query, depending upon the type of the input element. Most input elements require that the `NAME` attribute be specified, because it identifies the data for that element in the query.

The two attributes that are commonly used with the `FORM` tag are:

- `ACTION` specifies the URL to be used to process the form. This should name the script (or program) to be used to process the data in the form. We recommend that a relative URL be used, if possible.
- `METHOD` determines how the query will be sent. `POST` should always be used. The other option, `GET`, is not recommended for use in forms.

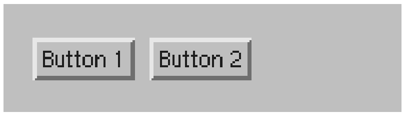
The remainder of this appendix describes the various input elements that are available.

I.2. The Submit Button

The one input element that is common to almost all forms is the `submit` button. When the user clicks on it, the contents of the form is submitted for processing and the user will get a new document containing the results.

Here is an example of the `submit` input element. In this example, we include the `FORM` tag. In later examples, it will be omitted. The `ACTION` URL given here will cause the name/value pairs in the query to be listed, assuming that you are viewing the on-line version of this document.

```
<FORM METHOD="POST" ACTION="cgi-bin/query-details">
<INPUT TYPE="submit" NAME="b1" VALUE="Button 1">
<INPUT TYPE="submit" NAME="b2" VALUE="Button 2">
</FORM>
```



The possible attributes are:

- `TYPE="submit"` specifies that this element is a submit button.
- `NAME` specifies the name for the button's form data. If it is given, then a NAME=VALUE pair is added, corresponding to which submit button was clicked. This attribute is optional, and is not supported in Mosaic versions 2.4 and earlier.
- `VALUE` specifies the label to appear in the button (and be sent in the query). Optional: the default is "Submit".

It is sometimes useful to have multiple possible actions within the same form. For example, when browsing a price list, you might want one button for "do more searching", and another for "place an order". In Mosaic 2.4 and earlier, it is possible to specify multiple **submit** buttons in a form, but scripts can't tell the difference between the buttons, because the buttons don't have names.

The next version of Mosaic (after 2.4) should allow the `NAME` attribute, so this will eventually be a moot point.

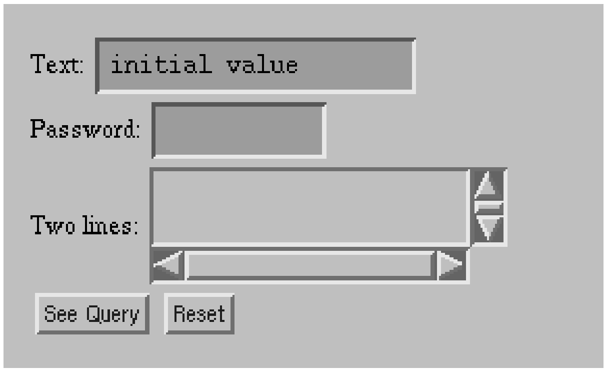
I.3. The Reset Button

The `reset` button resets the data in the form to the default values, which are not necessarily blank. It is specified in exactly the same way as the `submit` button, except that the `NAME` attribute is not used.

I.4. One-Line Text Input

The `TYPE="text"` and `TYPE="password"` elements are used to get single-line text input from the user. `text` input is visible to the user. `password` data is echoed as asterisks, for privacy.

```
Text: <INPUT TYPE="text" NAME="text" VALUE="initial value">
<BR>
Password: <INPUT TYPE="password" NAME="password" SIZE=10>
<BR>
Two lines: <INPUT TYPE="text" NAME="multi" SIZE="20,2">
<BR>
<INPUT TYPE="submit" VALUE="See Query">
<INPUT TYPE="reset">
```



The possible attributes are:

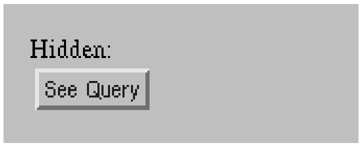
- `TYPE="text"` specifies that this element is a text entry field.
- `TYPE="password"` specifies that this element is a password field, which is just like `text`, except that the input is echoed with asterisks.
- `NAME` specifies the name for the form data. Required.
- `VALUE` specifies initial contents of the field. Optional.
- `SIZE=`*width* specifies the width of the field, in characters. This does **not** limit the amount of text that may be entered -- the field will scroll as necessary to accomodate more data. Optional: the default size is 20.
- `SIZE=`*width*`,`*height* specifies the width and height of the field, in characters and lines. This causes the field to have multiple lines. Optional.
- `MAXLENGTH` specifies the maximum number of characters that will be accepted as input. Optional: the default is unlimited.

I.5. Hidden Data

The `<INPUT TYPE="hidden">` element behaves the same as `TYPE="text"`, except that nothing is displayed and the contents can't be edited. The purpose of this element is to carry hidden data in a form, and is discussed in Chapter 6.

Hidden data is not supported in Mosaic before version 2.2. In earlier versions, it is displayed as an ordinary `TYPE="text"` field, so the data would appear in the form and be editable.

```
Hidden: <INPUT TYPE="hidden" NAME="hide" VALUE="Big Secrets">
<BR>
<INPUT TYPE="submit" VALUE="See Query">
```



The possible attributes are:

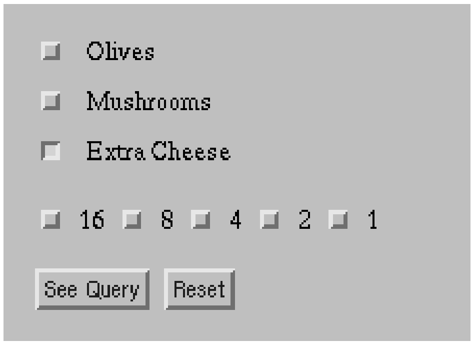
- `TYPE="hidden"` specifies that this element is a hidden data field.

- `NAME` specifies the name for the form data. Required.
- `VALUE` specifies initial contents of the field. Optional.

I.6. Checkboxes

The `TYPE="checkbox"` element is used to get a single on/off value from the user. If the checkbox is checked, the NAME=VALUE pair will be present in the query. Otherwise, it will be absent.

```
    <INPUT TYPE="checkbox" NAME="olives" VALUE="yes"> Olives
    <BR>
    <INPUT TYPE="checkbox" NAME="mushrooms"> Mushrooms
    <BR>
    <INPUT TYPE="checkbox" NAME="cheese" CHECKED> Extra Cheese
    <P>
    <INPUT TYPE="checkbox" NAME="list" VALUE="16">16
    <INPUT TYPE="checkbox" NAME="list" VALUE="8">8
    <INPUT TYPE="checkbox" NAME="list" VALUE="4">4
    <INPUT TYPE="checkbox" NAME="list" VALUE="2">2
    <INPUT TYPE="checkbox" NAME="list" VALUE="1">1
    <P>
    <INPUT TYPE="submit" VALUE="See Query">
    <INPUT TYPE="reset">
```



The possible attributes are:

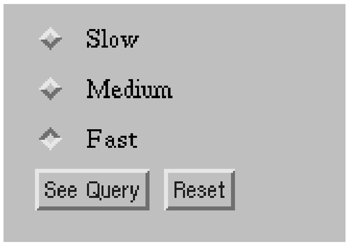
- `TYPE="checkbox"` specifies that this element is a checkbox.
- `NAME` specifies the name for the form data. Required.
- `VALUE` specifies value that will be returned in the query if the checkbox is selected. Optional: the default is "on".
- `CHECKED` specifies that the checkbox should be checked by default. Optional.

If multiple checkboxes have the same `NAME`, one NAME=VALUE pair will be included in the query for each selected checkbox. The second set of checkboxes in the example demonstrates this.

I.7. Radio Buttons

The TYPE="radio" element is used to get a single selection from a set of related items. Each element specifies a single button. Because they act in concert, all radio buttons in a group must have the same NAME.

```
<INPUT TYPE="radio" NAME="speed" VALUE="slow"> Slow
<BR>
<INPUT TYPE="radio" NAME="speed" VALUE="medium"> Medium
<BR>
<INPUT TYPE="radio" NAME="speed" VALUE="fast" CHECKED> Fast
<BR>
<INPUT TYPE="submit" VALUE="See Query">
<INPUT TYPE="reset">
```



The possible attributes are:

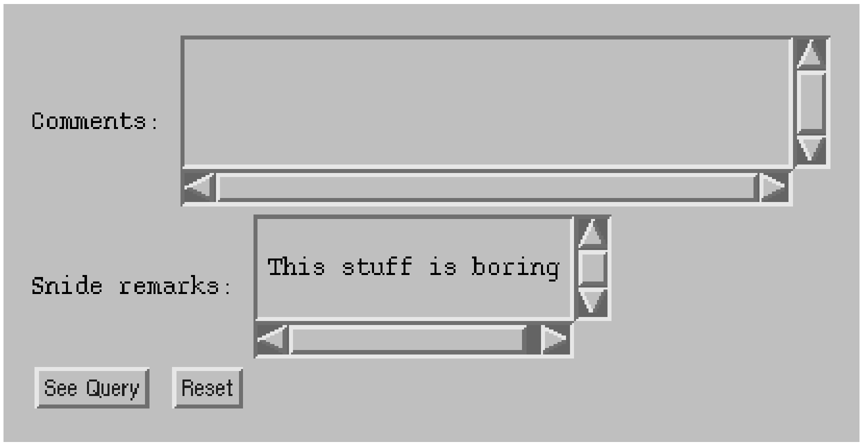
- TYPE="radio" specifies that this element is a radio button.
- NAME specifies the name for the form data. Radio buttons are grouped together by giving them the same NAME. Required.
- VALUE specifies the value that will be returned in the query if the checkbox is selected. Although the default is "on", radio buttons are not of much use unless different VALUEs are specified.
- CHECKED specifies that the radio button should be selected by default. Exactly one button in the group should be CHECKED.

I.8. Multi-line Text Input

The <TEXTAREA> tag is used to get multi-line text input from the user. This is different from <INPUT TYPE="text"> because the text may be scrolled with scrollbars.

The specification is also different, because the tag requires both an opening an closing tag. Initial data can be given by putting it between the tags, as in the second TEXTAREA in the example.

```
Comments:
<TEXTAREA NAME="comments" ROWS=4 COLS=40></TEXTAREA>
<BR>
Snide remarks:
<TEXTAREA NAME="snide" ROWS=3 COLS=25>
This stuff is boring!
</TEXTAREA>
<BR>
<INPUT TYPE="submit" VALUE="See Query">
<INPUT TYPE="reset">
```



Comments:

Snide remarks:

This stuff is boring

Note that the newlines inside the TEXTAREA tag are significant. That is why the "Snide remarks" text starts on the second line.

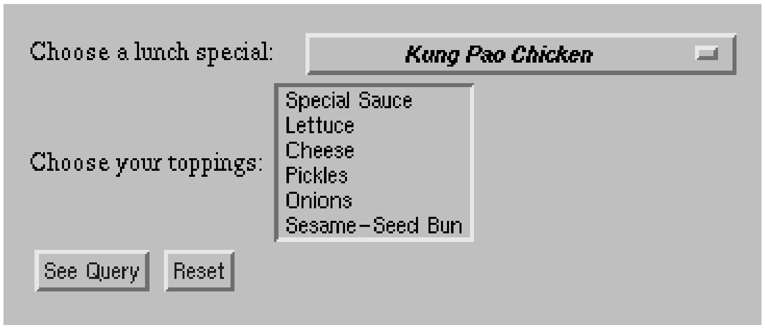
The possible attributes are:

- NAME specifies the name for the form data. Required.
- ROWS specifies the height of the field, in lines. Optional.
- COLS specifies the width of the field, in characters. Optional.

I.9. Menus and Scrolling Lists

The <SELECT> tag is used to get a selection(s) from a list of options. The options are displayed as either a pop-up menu, or a scrolling list. The list of options are specified between the opening and closing <SELECT> tags. Each option is prefixed with the <OPTION> tag.

```
Choose a lunch special:
<SELECT NAME="pop-up">
<OPTION> Kung Pao Chicken
<OPTION> Twice Cooked Pork
<OPTION> Dry Braised Shrimp in Chili Sauce
<OPTION> Szechwan Pepper Chicken
<OPTION> Broccoli in Spicy Garlic Sauce
<OPTION> Sweet and Sour Pork
<OPTION> Shrimp with Assorted Vegetables
<OPTION> Vegetable Delight
</SELECT>
<BR>
Choose your toppings:
<SELECT NAME="list" SIZE=6 MULTIPLE>
<OPTION> Special Sauce
<OPTION SELECTED> Lettuce
<OPTION> Cheese
<OPTION> Pickles
<OPTION SELECTED> Onions
<OPTION VALUE="ssb"> Sesame-Seed Bun
</SELECT>
<BR>
<INPUT TYPE="submit" VALUE="See Query">
<INPUT TYPE="reset">
```


Choose a lunch special: Kung Pao Chicken

Choose your toppings: Special Sauce, Lettuce, Cheese, Pickles, Onions, Sesame-Seed Bun

Notes:

- Use *Shift*-click to select a range of items. Use *Control*-click to select multiple discontiguous items.
- If none of the list items are selected, that NAME will not be present in the query.
- The NAME **list** is given as a *list-name* in the **cgi_parse_query** command, so the "toppings" are accumulated into a Tcl list.
- Note that "Sesame-Seed Bun" is represented as "ssb" in the query, because the VALUE attribute is given for it.

The possible attributes for the SELECT tag are:

- NAME specifies the name for the form data. Required.
- SIZE specifies the number of rows in the list. If SIZE is 1 or omitted, the SELECT will be represented with a pop-up menu, otherwise with a scrolling list with the given number of rows.

- `MULTIPLE` specifies that the `SELECT` should allow multiple selections. `MULTIPLE` forces the `SELECT` to be represented with a scrolling list, with a default `SIZE` of 5. Optional.

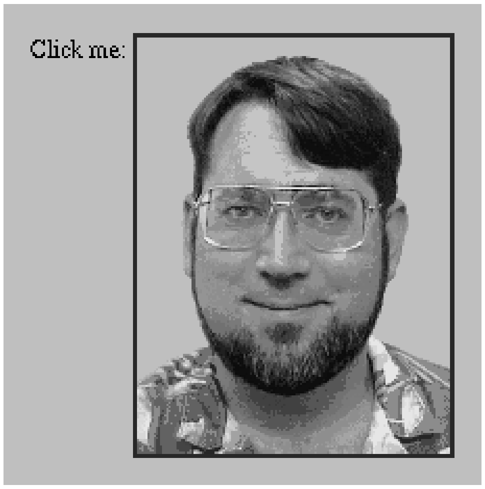
The possible attributes for the `OPTION` tag are:

- `SELECTED` specifies that this option is selected by default. Multiple options can be selected, if the `MULTIPLE` attribute is present in the `SELECT` tag.
- `VALUE` specifies the value to be returned in the query when this option is selected. By default, the value is the same as the text in the option. `VALUE` is new as of X Mosaic version 2.2.

I.10. Images

NCSA Mosaic version 2.2 allows bitmap images to be used as input elements in forms. However, it is not a part of the official HTML specification, so its use may not be supported in all HTML viewers.

```
Click me:
<INPUT TYPE="image" NAME="image" ALIGN=TOP SRC="myself.gif">
```



When the user clicks on the displayed image, the (x,y) coordinates are put into the query as *name***.x** and *name***.y and the query is immediately sent**. For example:

```
test.x=35
test.y=82
```

Coordinates are relative to the upper-left corner of the image, measured in pixels.

Unlike all of the other FORM input elements, the **submit** *button is not required, because clicking on the image submits the query.*

The possible attributes are:

- `TYPE="image"` specifies that this element is an image.
- `NAME` specifies the name for the form data. Required.

- `SRC` specifies the URL of the image to display. This is subject to the same restrictions as any other inline image. *i.e.*, the format may only be GIF (.gif) or X-bitmap (.xbm). Required.
- `ALIGN` can be either `TOP` or `BOTTOM`, specifiying how to align the image relative to the surrounding text. Optional: the default is `BOTTOM`.