Version: 2/3/2002 3:36 PM

# FARGOS/VISTA
HTTP Server Programmer's Guide

NOTE: The information contained within this document refers to a continuously enhanced product. Since the last release of this document, some Application Programming Interfaces may be have been extended and entirely new functionality may have been added. Programmers are strongly encouraged to review the latest documentation that is online.

FARGOS Development, LLC
757 Delano Road
Yorktown Heights, NY 10598
http://www.fargos.net
mailto:support@fargos.net

## Notice of Rights

## Trademarks

## Abbreviations

FARGOS Development, LLC is a Limited Liability Company registered with the State of New York. It is required to identify itself as such in its name, hence the ", LLC" suffix. For purposes of readability in this document, the ", LLC" suffix is sometimes dropped. The phrase "FARGOS Development" always denotes "FARGOS Development, LLC" and is not intended to suggest any alternate form of organization.

## Notice of Liability

## Contents

# 1. Introduction

**FARGOS/VISTA** implements a transparently distributed, multi-threaded, object-oriented environment that supports many kinds of applications. One such application is a Hypertext Transfer Protocol (HTTP) server, which forms a key component of the current suite of Internet-enabled applications. Most distributions of the FARGOS/VISTA Object Management Environment are configured to contain an HTTP 1.1 server implementation (see RFC 2616)[1]. Many FARGOS/VISTA-based applications integrate with the HTTP server to either implement a graphic user interface using World Wide Web browsers or provide support for HTTP-based services.

The FARGOS/VISTA HTTP server has several intrinsic abilities. The most frequently used ability provides read-only access to a document tree that is comprised of a collection of directories located on the local file system. As part of its fundamental feature set, the HTTP server also implements an automatic cache of referenced documents and automatically processes server-side-include directives. The server-side-include processor permits HTML to be generated at runtime based on templates and environment variables.

This Programmer's Guide is intended to be used by FARGOS/VISTA application developers who desire to create HTTP-based applications or services.

## *Another HTTP Server?*

Given the existence of well-respected web servers from a variety of vendors, of which Apache is the most popular, one could be forgiven for asking "why does the world need another web server"? The FARGOS/VISTA HTTP server's primary reason for existence is to provide a tightly coupled mechanism for exporting FARGOS/VISTA-based services via HTTP. For security reasons and as a 24x7 test bed for FARGOS Development, LLC, the FARGOS/VISTA HTTP server has run www.fargos.net from its first day of operation. It has been used as the only server technology for www.alecbaldwin.com, www.vodusa.com and others, with all of the backend operations having been implemented as FARGOS/VISTA-based applications. A FARGOS/VISTA HTTP server replaced an Apache-based installation at www.archiecomics.com in a successful attempt to stabilize a site that was failing every 2-4 days. Once a FARGOS/VISTA server was installed, outages ceased.

Experience has shown that the FARGOS/VISTA HTTP server is quite robust and, since it is part of the FARGOS/VISTA infrastructure, it is quite simple to develop web-based applications that can take advantage of a distributed environment.

[1] The only exceptions are custom OEM configurations.

# 2. HTTP Server Administration

A great deal of functionality is provided by the default HTTP server implementation. The information presented in this section assists a system administrator with the process of configuring and deploying a FARGOS/VISTA HTTP server that will provide access to selected files available on the local system.

## HTTP Server Configuration

A FARGOS/VISTA HTTP server is created by instantiating an object of class **HTTPdaemon**. A single **HTTPdaemon** handles all requests for a particular site. It is possible to have more than one **HTTPdaemon** within the same FARGOS/VISTA address space, so a single FARGOS/VISTA daemon can support multiple web sites. Organizations that provide web-hosting services may find this useful, although running distinct FARGOS/VISTA daemons for each site eliminates the possibility of a faulty application associated with one site affecting other customers. Hence, the ability to run multiple sites within the same address space is most interesting when deploying a population of FARGOS/VISTA daemons on a server farm and dynamically configuring pools of processors to service the current needs of a particular site. See the section on Advanced Usage for details.

There are two other services that are usually used with the **HTTPdaemon**. The **HTTPpurgeCache** class implements a service that periodically removes obsolete documents that were cached by the **HTTPdaemon**. While its use is not mandatory, the storage taken up by cached-but-no-longer-referenced documents will not otherwise be recovered. The **HTTPcommonLogFormat** class implements an optional logging facility. The **HTTPdaemon** sends a message at the completion of every HTTP transaction that indicates the resource that was requested, operation performed, bytes sent, etc. The **HTTPcommonLogFormat** service organizes this data according to the Common Log Format and writes an appropriate record.

These services are normally created in the *rc* file processed at the time the FARGOS/VISTA daemon boots. The services are created in the following order:

1. **HTTPcommonLogFormat**
2. **HTTPdaemon**
3. **HTTPpurgeCache**

A sample *rc* file appears below:

```
HTTPcommonLogFormat /home/httpd/logs/logFile.txt www.domain.com
HTTPdaemon http.profile tcp:0.0.0.0:4321
HTTPpurgeCache 90 www.domain.com
```

The arguments to each of these classes are summarized below. Administrators are encouraged to review the respective class documentation for authoritative and current descriptions that may detail additional functionality.

### HTTPcommonLogFormat

A record of requests made processed by an **HTTPdaemon** can be logged by an **HTTPcommonLogFormat** object. The file name into which records are to be written is always specified as the first argument. An object of class **HTTPcommonLogFormat** registers itself as a named service, which is by default called *HTTP_LOGGER* but can be further qualified if a second argument is provided.

The use of a qualified service name permits multiple independent web sites to reside within the same address space.

## HTTPpurgeCache

An object of **HTTPpurgeCache** checks to see if cached content is still valid and requests the deletion of obsolete objects. The interval at which such checks are performed is specified as the first argument and the unit of measurement is in seconds. The second argument specifies the name of the web site that should be checked. The name of the web site is derived from the value of **ServerName** in the corresponding profile passed to the respective **HTTPdaemon**.

## HTTPprotectedDirectory

System administrators may desire to restrict access to certain sections of the document tree that is exported by the **HTTPdaemon**. The **HTTPprotectedDirectory** class provides a convenient mechanism for setting up such protected regions. The first two arguments to the **create** method of this class specify the web server and section of the document tree to be protected. Anything below the indicated point in the tree is subject to the access restrictions setup by the **HTTPprotectedDirectory** object. The third argument indicates the realm information that will be provided to web browser clients when they access the indicated section of the document tree. Many browsers only support "*Basic*" authentication and it is the default (see RFC 2617 for details on HTTP authentication). The realm name should be some informative text that will help the user understand to where he is providing his authentication information. An enhanced mechanism for transmitting password data can be requested using "*Digest*" mode, which is selected by specifying "**digest**" as the realm name. The next argument can either be a file name specification of a password database file (beginning with "**file:**") or the user name of a user/password pair (the next argument will be treated as a password). The use of a password database is encouraged because it is the only mechanism to allow write methods (such as PUT, DELETE, etc.), conveniently handles an arbitrary number of users, and permits updates that were made by the **HTTPuserAdmin** service to be saved persistently. If user/password pair information is provided as arguments to the **create** method, then the entries can be permanently changed only by editing the *rc* file used to boot the FARGOS/VISTA Object Management Environment process.

```
HTTPprotectedDirectory www.domain.com /admin "Server Administration" adminUser1
password1 adminUser2 password2

HTTPprotectedDirectory www.domain.com /admin "Server Administration"
file:/home/httpd/conf/adminUsers.db

HTTPprotectedDirectory www.domain.com /admin digest
file:/home/httpd/conf/adminUsers.db
```

All of the examples above protect files and services registered under */admin* of the site *www.domain.com*. In the first example, two permanent user Ids (adminUser1, adminUser2) are specified along with their respective passwords. While the **HTTPuserAdmin** service can be used to change the passwords associated with these two users or remove their authorization entirely from a running system, the changes will not be applied to the *rc* file, so a reboot of the system will restore the original settings. It should also be obvious that specifying a large number of users will be unwieldy and the password information is readily visible to anyone who can read the *rc* file since it appears in the clear. This form has its uses, but is not the preferred mechanism for situations in which many distinct users will be granted access to the document tree.

The remaining examples specify a password database instead of explicit user/password pairs. When a password database is used, the **HTTPuserAdmin** application has the ability to keep the database updated to reflect the addition and removal of users or password changes. It also has the benefit that the password information is not so obvious. The password database is processed by a **ParseParameterFile** object. Passwords are placed in a "password" section and each entry has the following format:

```
    username password [ permittedMethod …]
```

An example password file appears below:

```
SECTION password
anonymous "-" GET HEAD POST OPTIONS
user1 pw1 GET HEAD OPTIONS POST PUT DELETE
admin pw2 GET HEAD OPTIONS POST PUT DELETE MKCOL PROPFIND PROPPATCH LOCK UNLOCK
MOVE COPY
```

### HTTPuserAdmin

The **HTTPuserAdmin** class implements a convenient support class that enables system administrators, using a conventional web browser, to add or delete user authorizations for protected sections of a site. The password for an existing user can also be changed. If a password database is in use by the affected section of the document tree, its contents will be automatically updated.

## *HTTP Server Profile*

Most of the static characteristics of a given web site are specified in a profile that is passed to the **HTTPdaemon**. A sample appears below:

```
directoryRoot = dirName1 [dirName2 …]
ServerName = www.domain.com
ServerPort = 80
MIMEtypeFile = /home/httpd/conf/mime.types
TolerateBadHTTP11 = 1
SECTION CustomErrorPages
404 /errors/notFound.html
403 /errors/notAuthorized.html
```

### Directory Roots

Traditionally, there is a straightforward mapping between a file stored on a local file system and a corresponding URL reference. The FARGOS/VISTA HTTP daemon permits multiple directories to be searched for a file of interest. One way this can be exploited is to set up a shadow site to which modifications are applied. The modified content under the shadow site takes precedence over the content found under the production site tree. This permits developers to preview their changes without requiring a complete copy of the original site.

The roots of the document tree are specified by the **directoryRoot** parameter and are listed in order of decreasing precedence.

### Server Name and Port

The official name of the server and the port at which it listens are specified as **ServerName** and **ServerPort** respectively. If **ServerPort** is not specified, it defaults to 80, the normal HTTP port. Following the Common Gateway Interface convention, the value of **ServerName** is made available as the environment variable **SERVER_NAME** and **ServerPort** is made available as the environment variable **SERVER_PORT**.

## MIME Types

The HTTP server always defines a small subset of key MIME types (such as HTML and plain text); the list can be arbitrarily extended by providing an Apache-style MIME type configuration file. The HTTP server will process an Apache-style *mime.types* configuration file as-is. The MIME type configuration file is a plain text file. Comment lines are prefixed with a "#" character. Other lines are treated as MIME type specifications. The first field in such a line is the MIME type name. The second and all other fields on the line are file suffixes. As an example:

```
image/gif      gif
image/jpeg     jpeg jpg
text/html      html htm
text/plain     txt asc
application/x-shockwave-flash    swf
```

It is possible to specify only a MIME type name and no file suffixes. This is a common occurrence in the *mime.types* file provided with the Apache web server distribution; however, such records provide no useful information to the FARGOS/VISTA HTTP server.

## Tolerate Bad HTTP 1.1 Requests

The HTTP 1.1 protocol specification asserts that conformant HTTP servers must reject HTTP 1.1 requests that do not include a Host directive as part of the request header. Unfortunately, some World Wide Web browsers (such as versions of Microsoft's Internet Explorer 4) issue malformed HTTP 1.1 requests. By default, the FARGOS/VISTA HTTP server complies with the HTTP 1.1 specification; however, as a practical matter, this strict conformance can be disabled to not disenfranchise those users who have non-compliant browsers.

**NOTE:** FARGOS Development, LLC recommends that this option be enabled. No known operational problem will arise from tolerating such improperly formed HTTP 1.1 requests. Although tolerating such problems provides no incentive against the continued use of broken browser implementations, most web site owners are not interested in participating in such a crusade at the expense of restricting access to their user community.

## Custom Error Pages

When the FARGOS/VISTA HTTP server encounters an error, such as a request for an unknown file, it returns the corresponding HTTP error code in its response as well as a small HTML document that purports to explain the error. Some sites may wish to provide their own site-specific error pages. This is done by adding a *CustomErrorPages* section to the site's profile that is passed to the **HTTPdaemon** object. Each line of the *CustomErrorPages* section specifies an HTTP error code and the corresponding document that should be sent. The documents will be automatically processed by the Server Side Include processor and the following variables (in Table 1) are set in additional to the normal environment:

**Table 1**

| **Variable** | **Description** |
|---|---|
| HTTP_ERROR_CODE | The numeric HTTP protocol error code, such as 404 for a Not Found error. |
| HTTP_ERROR_CODE_TEXT | The text that corresponds to the HTTP error code, such as “Not Found” |
| HTTP_ERROR_MESSAGE | A text message generated by the server that explains the error in more detail, for example, what file could not be found. |

### *WebDAV Extensions*

The **HTTPdaemon** can be extended with additional capabilities. A useful example is the Web-based Distributed Authoring and Versioning (see RFC 2518) support enabled by the **WebDAVfacility** class. It is normally created using a single argument the specified the name of a profile that defines various configuration parameters. A sample profile appears below:

```
directoryRoot = /tmp
collectionRoot = /tempFiles
serverName = www.domain.com
passwordFile = webdavPW.db
```

**Note:** in contrast to the profile passed to a **HTTPdaemon** object, the *directoryRoot* attribute in a **WebDAVfacility** profile can specify only one directory, not multiple. The reason for this restriction is the ambiguity that would be present when creating new resources with PUT and MKCOL requests.

The *collectionRoot* attribute identifies the logical section of the HTTP servers naming tree under which WebDAV facilities will be provided. An existing section of the tree can be replaced. The most common and significant example is when *collectionRoot* is set to “/”, which adds WebDAV capabilities to the default portions of the entire naming tree. The password file is identified by the **passwordFile** entry and it is mandatory since almost all of the WebDAV-related methods are not permitted to anonymous users.

Only one **WebDAVfacility** object is normally created. It registers the WebDAV-related HTTP extension methods, such as MKCOL, COPY, MOVE, PROPFIND, LOCK, etc. It also creates the root **WebDAVcollection** object corresponding to the root of the WebDAV-enabled document tree. There are three primary WebDAV-related classes: **WebDAVcollection**, **WebDAVfile** and **WebDAVresource**.

### *External Interfaces*

There are some external interfaces that can be used by a system administrator to customize or effect the operation of a running site.

### Reopen Log File

Heavily trafficked web sites will generate log files that become unwieldy due to their large size. The **HTTPcommonLogFormat** service provides a **reopen** method to request that the current log file be closed and a new one be opened. To avoid placing the system into an inconsistent state and handle recovery issues caused by crashed servers, it is more common to first rename the current log file, then issue a

**reopen** request with no argument so that the original log file name is used for the open request.  Since the log file with data was renamed, a new file will be created. Alternatively, a new log file name can be specified.  This approach works on operating systems that prevent the renaming of open files.

The **OMEinvoke** utility program can be used to invoke such a request from external process (e.g., perhaps under control of **cron** or some other scheduler).  For example:

```
$ OMEinvoke tcp:localhost:8765 HTTPlogger reopen
```

or

```
$ OMEinvoke tcp:localhost:8765 HTTPlogger:www.domain.com reopen newFileName
```

# 3. HTTP Server Application Programming Interfaces

While the interfaces exposed to a system administrator are limited in number and complexity, application programmers are provided with a plethora of options. This section discusses the FARGOS/VISTA HTTP server from a programming perspective.

## *Model of Operation*

Each web site hosted by a FARGOS/VISTA daemon is created by the instantiation of an appropriate **HTTPdaemon** object. The **HTTPdaemon** object processes a profile file that describes configuration parameters associated with the site. It then creates two objects. The first is an **URLdirectory** object, which handles the caching of documents, registration of site-specific services, etc. The **URLdirectory** object, not the **HTTPdaemon** object, is really the central object that represents the site.

The other object initially created by the **HTTPdaemon** is an **IOobject** that listens for incoming HTTP requests. The **HTTPdaemon** will be notified by the **IOobject** whenever a new client connection is established. At that point, the **HTTPdaemon** will accept the new connection. It then creates an **HTTPfastReceive** object and passes it the new connection for subsequent processing.

An **HTTPfastReceive** object is responsible for all requests that are sent over a particular HTTP connection. If the client browser is using HTTP 1.0, then only one request per established connection will be expected by default. In contrast, HTTP 1.1 connections are expected to handle multiple requests before a connection is closed.

Once an **HTTPfastReceive** object has received all of the data for a given request, it issues a **loadURL** request to the site's **URLdirectory** object. This locates either the previously cached contents or the handler responsible for the indicated section of the document tree. The **HTTPfastReceive** object then sends the object a message corresponding to the HTTP protocol request:

- getRequest
- headRequest
- postRequest
- putRequest
- deleteRequest
- traceRequest
- optionRequest
- extensionRequest

The *extensionRequest* method is used to handle arbitrary methods that are not defined in the HTTP standard. The **URLdirectory** class provides a **registerNewFeature** method that allows the definition of new headers and supported methods. Perhaps the best illustration of such extensions is found in RFC 2518 (HTTP extensions for distributed authoring — WebDAV).

All of the above methods take the same argument list, as illustrated below:

```
HTTPcachedObject:getRequest(array requestData, assoc options,
   string replyMethodName, oid replyToObject)
```

The *requestData* array is the tokenized parse of the HTTP command issued by the client, thus:

requestData[0] = the command, such as GET, HEAD, POST, etc.
requestData[1] = the URL of interest. URI escapes will have been translated.

requestData[2] = the protocol (e.g., “HTTP/1.0” or “HTTP/1.1”)
The *options* associative array contains the parse of the MIME header associated with the command, CGI-related environment variables and extra data provided with the request (such as form data). Note that, normally, HTTP 1.0 requests will not provide additional MIME header data; however, some browsers request HTTP 1.1 functionality using HTTP 1.0.

Regardless of the case used by the requesting client, all of the MIME header keys are converted to lowercase and the trailing colon is dropped. Thus “*Host:*” will be converted to “*host*” and “*Content-Length:*” will be converted to “*content-length*”. If an entity body is present, which will normally be the case for a POST or PUT command, the data is made available as the “ENTITY_BODY” element of the *options* associative array.

Any application that handles HTTP requests must return a result to the object indicated by *replyToObject* using the method specified by *replyMethodName*. This is illustrated below:

```
send (replyMethodName)(httpReturnCode, httpErrorText, header, entityBody)
   to replyToObject;
```

**Note:** the variable holding the method name is enclosed by parenthesis to prevent the OIL2 compiler from parsing it as an attempted function call reference.

No value will be returned by the target method. The *httpReturnCode* is an integer value corresponding to the appropriate HTTP return code. For example, 200 is OK, 404 is Not Found, etc. The *httpErrorText* argument is a string that provides the text corresponding to the error code, such as “OK”, “NOT FOUND”, etc. Programmers should review the HTTP specification (RFC 2616) to determine the appropriate return code for their situation.

The *header* and *entityBody* arguments are strings that provide the MIME header and body for the response. Do not include a “Connection:” directive in the MIME header to be returned—this is handled as appropriate by the **HTTPfastReceive** object based on the protocol in use, options negotiated and current loading on the server.

### Large Entity Bodies

When working with very large entities, it is prohibitively expensive to cache the contents in memory. For any particular host, a sufficiently large file can exist that is too large to fit into memory, thus making it infeasible to copy the entire file into memory. In such situations, the object Id of an **IOobject** can be passed instead of a string or set of strings. The **sendResult** method of **HTTPfastReceive** will create a **SendFile** object to perform the data transfer. The **setDeleteOnClose** method can be used to have the open **IOobject** automatically deleted on end-of-file.

## *Document Caching*

The FARGOS/VISTA HTTP server makes extensive use of document caching. Whenever reference is made to a file under the site’s document tree, the **URLdirectory** object creates an **HTTPcachedFile** object to read, process and cache the file’s contents. Cached documents are registered with the respective site’s **URLdirectory** object using the **registerObject** method. Before a request will be forwarded to a cached object, a **checkIfStillValid** message will be sent to verify that the cached contents are still valid. No arguments are passed. The cached document’s **checkIfStillValid** method must return a value indicating how many seconds before the object is no longer valid. If the object's contents have expired, -1 is typically returned, but any negative value will suffice.

Cached document objects are periodically probed by the **HTTPpurgeCache** service, which sends a **deleteIfObsolete** message. Two arguments are passed: the current time, as obtained from **getLocalRelativeTime()** and number of seconds until the next verification pass will be made.

If a cached document has expired when it receives a **deleteIfObsolete** message, it un-registers itself by sending the appropriate **URLdirectory** object a **removeFromCache** message with its object Id and aliases as arguments. The **deleteIfObsolete** method returns 1 if the object is obsolete; otherwise, it returns zero. Programmers are cautioned to remember that if the value of *fromObject* is **nil**, no **reply** is expected or possible.

## Access Control

A cached document may be registered under a protected section of the document tree. All objects that register with an **URLdirectory** must implement a **setAccessValidator** method. This method is passed one argument, which is the object Id of a validation object to which access control queries should be sent. If the value of *fromObject* is not **nil**, a zero value should be returned to indicate that the information has been recorded.

When a cached document object is accessed, it should send a **validateAccess** request to the validation object that was set by **setAccessValidator**. The data to be passed is the authentication information provided in the HTTP header. This is readily available as the key "*authorization*" in the MIME header options list. For example:

```
rc = send "validateAccess" (options["authorization"]) to accessValidatorObj;
```

If the value returned by **validateAccess** is zero, access is denied.

## Convenience Classes

While applications that provide HTTP-accessible services may implement all of the necessary methods from scratch, many programmers will find it convenient to inherit from the base class **HTTPcachedObject** and automatically obtain much of the necessary behaviors. The class **HTTPcachedFile** inherits from **HTTPcachedObject**, as does **HTTPreplacedText**. **HTTPcachedFile** is used by the **URLdirectory** to read, perform and necessary processing and cached files found on the local server. **HTTPreplacedText** performs a nearly identical function, with the exception that the document's template contents are provided as an argument instead of a file name. It is thus very useful for caching results that were generated programmatically.

## Support Functions

The FARGOS/VISTA Object Management Environment core contains many functions that may be of use to programmers implementing HTTP-based applications. Some of them are mentioned briefly below. Programmers are referred to the *FARGOS/VISTA Object Management Environment Programmer's Guide* for details.

### parseHTTPuriData

Many web-related applications identify documents using Uniform Resource Identifiers (see RFC 2396). The **parseHTTPuriData()** function parses an HTTP-related URI into its component elements.

### parseHTTPformData

HTML forms are often encoded using a URL-encoding method (see RFC 1738). The **parseHTTPformData()** function decodes such strings into the attribute/value pairs.

### parseMIMEblock

Some advanced applications may need to parse multi-part MIME documents (see RFC 2045). The **parseMIMEblock()** function breaks an entity body into a parsed MIME header and the document data.

### decodeMIMEdata

The **decodeMIMEdata()** function decodes MIME-encoded data into its component elements. It recognizes the following encoding methods:

- application/x-www-form-urlencoded
- multipart/form-data
- multipart/mixed

### base64ToASCII

Some MIME documents are encoded using the base64 encoding method (see section 5.2 of RFC 1521). This function converts a base64-encoded string into its true contents.

## *Example of an HTTP-based Service*

The source below illustrates how a service that processes HTML forms can be implemented and integrated with the HTTP daemon.

```
global {
   const string URL_DIR_PREFIX = "/services/URLdirectory:";
   const int REDIRECT_CODE = 302; // should be 303
   const int REDIRECT_TYPE = "Moved Temporarily"; // should be See Other
   int requestCount;
}

class Local . FARGOSprocessInquiry {
   oid         urlDirectory;
   string      replyMethod;
   oid         replyDest;
   string      pageName;
   array       destEmailAddrs;
   string      acknowledgementPage;
   assoc       formEscapes;
} inherits from Object;
```

```
FARGOSprocessInquiry:create(string serverName, string page, string ackPage, string
addr1)
{
   int   i;

   urlDirectory = lookupLocalService(URL_DIR_PREFIX + serverName);
   if (urlDirectory == nil) {
         send "deleteYourself" to thisObject;
         exit;
   }
   pageName = page;
   acknowledgementPage = ackPage;
   for(i=3;i<=argc;i+=1) {
         destEmailAddrs[i - 3] = argv[i];
   }
   send "registerObject"(pageName, thisObject) to urlDirectory
         from nil;
   formEscapes[" "] = "%20";
}

FARGOSprocessInquiry:delete()
{
   if (urlDirectory != nil) {
         send "removeFromCache"(pageName) to urlDirectory;
   }
}

FARGOSprocessInquiry:deleteIfObsolete(int t)
{
   if (fromObject != nil) return (0);  // keep always...
}

FARGOSprocessInquiry:checkIfStillValid(int t)
{
   return (1);
}

FARGOSprocessInquiry:getRequest(array requestData, assoc options,
   string replyMethod, oid replyDest)
{
   string      body;
   string      hdr;

   body = makeAsString("<HTML><HEAD><TITLE>Method Not Allowed</TITLE></HEAD>\r\n",
         "<BODY>\r\n<P><B>Method Not Allowed: ",
         requestData[1],
         "</B>\r\n<P><HR><P><I>FARGOS/VISTA HTTP server</I></BODY></HTML>\r\n");

   hdr = makeAsString("Content-type:  text/html\r\nContent-length: ",
         length(body), "\r\nConnection-close\r\n\r\n");
   send (replyMethod)(405, "Method Not Allowed", hdr, body) to replyDest;
}

FARGOSprocessInquiry:headRequest(array requestData, assoc options,
   string replyMethod, oid replyDest)
{
   string      body;
   string      hdr;

   body = makeAsString("<HTML><HEAD><TITLE>Method Not Allowed</TITLE></HEAD>\r\n",
         "<BODY>\r\n<P><B>Method Not Allowed: ",
         requestData[1],
         "</B>\r\n<P><HR><P><I>FARGOS/VISTA HTTP server</I></BODY></HTML>\r\n");

   hdr = makeAsString("Content-type:  text/html\r\nContent-length: ",
         length(body), "\r\nConnection-close\r\n\r\n");
   send (replyMethod)(405, "Method Not Allowed", hdr, body) to replyDest;
}
```

```
FARGOSprocessInquiry:postRequest(array requestData, assoc options,
   string replyMethodName, oid replyDestination)
{
   array formInfo, dest;
   assoc acl;
   int   i, j;
   assoc condensedData;
   string      key, subject, message;
   oid   mailObj;

   replyMethod = replyMethodName;
   replyDest = replyDestination;

   formInfo = parseHTTPformData(options["ENTITY_CONTENT"], array);

   message = "";
   for(i=0;indexExists(formInfo, i) != 0;i+=1) {
         j =  nextIndex(formInfo[i], 0);
         key = getKeyForIndex(formInfo[i], j);
         condensedData[key] = formInfo[i][key];
         message = makeAsString(message, key, " = ",
               formInfo[i][key], "\r\n");
   }

   subject = "Customer Inquiry";
   if (indexExists(condensedData, "ProductOrService") != 0) {
         subject += " re: " + condensedData["ProductOrService"];
   }
   acl = makeDefaultACL();
   mailObj = send "createObject"("SendMailViaSMTP", acl,
         "webmaster@fargos.net", destEmailAddrs, subject,
         message, thisObject) to ObjectCreator;
}

FARGOSprocessInquiry:smtpResult(int resultCode)
{
   string      body, hdr;
   string      text, ackPage;
   string      message, title;

   if (resultCode == 1) {
         title = "Email Sent Successfully";
         text = "successfully recorded";
   } else {
         title = "Email Send FAILED";
         text = "failed to record";
   }
   message = makeAsString("We have ", text, " your inquiry.");
   body = makeAsString("<HTML><HEAD><TITLE>", title,
         "</TITLE></HEAD>\r\n<BODY><P><B>",
         message, "</B></P></BODY></HTML>\r\n");
   ackPage = makeAsString(acknowledgementPage, "?title=", title,
         "&message=", message);
   ackPage = substituteText(ackPage, formEscapes);

   hdr = makeAsString("Location: ", ackPage,
         "\r\nContent-type:  text/html\r\nContent-length: ",
         length(body), "\r\nConnection-close\r\n\r\n");
   if (resultCode == 1) {
         send (replyMethod)(REDIRECT_CODE, REDIRECT_TYPE, hdr, body)
               to replyDest;
   } else {
         send (replyMethod)(500, "Internal Server Error",
               hdr, body) to replyDest;
   }
}
```

# 4. Server-Side-Include Processor

The native Server-Side-Include processor is implemented by the class **HTTP_SSIprocessor**. The server side include processor is invoked automatically by many HTTP-related classes when a document that contains server-side-include directives is cached.

The server-side-include processor understands a command language whose elements and structure were taken from that recognized by Apache's *mod_include* server-side-include module. There are some differences: not every *mod_include* directive is supported and there are some additional functions available.

Web page designers can create template pages and add appropriate server-side-include directives that cause content to be dynamically generated and inserted into the body of documents that are sent back in response to various HTTP requests from a client.

All server-side-include directives appear as HTML comments and begin with the pattern:

```
    <!--#
```

The end of the server-side-include directive is terminated by the pattern:

```
    -->
```

Because the server-side-include directives are encapsulated as HTML comments, all web site design tools will be able to read the template files. Web page designers should ensure, however, that their tools do not drop or relocate HTML comment directives.

## Include a Local File

One form of the **include** directive allows retrieval of a file from the local system and its inclusion within the body of a template. While very efficient, such a directive can be used to retrieve arbitrary files under the document tree of the server. Unprivileged users should never be provided the opportunity to place arbitrary HTML files under the roots of the server's document trees.

```
<!--#include file="filename" -->
```

## Include Results of an HTTP Query

A much more powerful, and arguably more secure, form of the **include** directive causes an HTTP query to be issued. The response from the query is then inserted within the body of the template.

```
<!--#include virtual="url" -->
```

## Output an Environment Variable

Many web-based applications make the results of their computations available as environment variables, which can then be inserted at appropriate points in a template file. The **echo** directive inserts the value of an environment variable.

```
<!--#echo var="varName" -->
```

## Set an Environment Variable

Template pages can set environment variables to control processing of dynamically generated queries or the time to live of a page using the **set** directive.

```
<!--#set var=”varName” value=”val” -->
```

## Set an Undefined Environment Variable

Sometimes a programmer only wants to set an environment variable if it has not already been defined.  Rather than use the overhead of an **if**-statement, the **setifnotset** directive can be used to efficiently perform this function.

```
<!--#setifnotset var=”varName” value=”val” -->
```

**Note:** this function is not supported by Apache.