Version: 7/24/2001 6:09 PM

FARGOS/SolidState

HTTP Server Adapter
User's Guide

NOTE: The information contained within this document refers to a beta release of the product. Some Application Programming Interfaces may be altered in response to experiences and suggestions obtained in the field.

Powered By FARGOS/VISTA

FARGOS/SolidState HTTP Server Adapter User's Guide

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

Notice of Rights

All rights reserved. This document may be rendered into whatever form is useful for the user, including electronic transmission or printing, so long as the content is not altered.

Trademarks

FARGOS/VISTA, FARGOS/SolidState and FARGOS/SolidConnection are trademarks of FARGOS Development, LLC.

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

Information in this document is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this document, FARGOS Development, LLC shall **not** have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the instructions contained within this document or by the computer software or hardware products described in it.

Contents

1. An Overview of FARGOS/SolidState

FARGOS/SolidState is a technology that implements support for Byzantine-fault-tolerant transactions. This short sentence carries a lot of information content within the phrase "*Byzantine-fault-tolerant*" and readers who are not practicing computer scientists may not be completely familiar with the nuances.

Fault-Tolerant Systems

A fault-tolerant system is able to handle the occurrence of a fault without causing the system to stop or produce an incorrect result. Most systems are not fault-tolerant and the occurrence of a single fault does cause them to stop functioning. A simple example of such a scenario is a PC running a word processing application. If the power fails while a user is typing, his work is lost.

Creating a fault-tolerant system is a complex task and requires some level of redundancy. The basic premise is that a redundant system can take over when the original system has failed. Most efforts to utilize redundancy result in only highly available, but not fault-tolerant systems. Highly available systems provide a means to return the system to an operational state within a period of time shorter than that required to repair the fault in question; however, they do not guarantee that work in progress at the time the fault occurred will be preserved. Continuing with our illustration of the PC user who just lost power: if that user had a gasoline-powered generator left over from his Y2K preparations, he could plug his PC into the generator and continue working after rebooting the PC. Unfortunately, all of his work since the last save (manual or automatic) would have been lost.

For those systems that do provide fault-tolerance, the most common approach is to tolerate a certain class of faults. Faults that are of a different nature than those expected will cause the system to fail. Our hypothetical PC user could have addressed his power problem by having installed an Uninterruptible Power Supply (UPS): this special device would automatically switch over to battery power when the main power failed and the PC would not have seen the loss in power. However, failures caused by the PC's reset button being pressed or the power cord being unplugged from the UPS are both examples of failure modes not handled by the UPS solution and would thus result in the system losing work in progress and halting.

The ability to handle arbitrary (including malicious) faults is termed Byzantine-fault-tolerance. The name arises from a famous paper, "The Byzantine Generals Problem" (see ACM Transactions on Programming Languages and Systems, 4(3), 1982. ACM Digital Library subscribers can obtain the full text online). The problem faced by the hypothetical Byzantine generals was how to successfully coordinate an attack on a city by sending messages to the various army encampments encircling the city under siege. The problem is straightforward if every general was loyal to the crown, but a disloyal general would have the ability to forge new orders, lie in his responses to his superiors, and alter any messages that passed through his hands on their way to a colleague. Cast back into the field of computer science, Byzantine-fault-tolerant systems tolerate faults that may have been caused due to malicious outside interference. Instead of assuming only safe failure modes (e.g., the power fails and the machine quits functioning cleanly), Byzantine-fault-tolerant systems also deal with failures where memory is corrupted or a CPU is computing incorrect results. They also address faults caused by a hostile intruder that has taken over a machine and is attempting to force the system to take certain actions.

Until recently, solutions to this problem have remained of only theoretical interest due to the impracticality of their implementation. FARGOS Development, however, inspired by recent theoretical work by Miguel Castro and Barbara Liskov, has implemented a practical system.

The core implementation of this technology is known as **FARGOS/SolidState**, a reference to the fact that it provides for the reliable maintenance of state.

FARGOS/VISTA

The FARGOS/SolidState technology is implemented on top of **FARGOS/VISTA**, which is a high-performance, transparently distributed, multithreaded, architecture-neutral, object-oriented environment that runs on a variety of hardware and operating system platforms. FARGOS/VISTA-based applications are normally written in Object Implementation Language 2 (OIL2), but they can also be implemented other languages, such as C++. Historical results with the predecessor language OIL showed a 6 to 10-fold improvement in programmer productivity vs. C++ and the bulk of the examples in this document are authored in OIL2 due to its conciseness.

While the FARGOS/SolidState technology can be used by application programmers to build their own Byzantine-fault-tolerant FARGOS/VISTA-based applications, a set of classes already implemented provides such support for the FARGOS/VISTA HTTP server. This document first describes Byzantine fault-tolerance and then how web-based applications can be implemented and deployed using the integration of FARGOS/SolidState and the FARGOS/VISTA HTTP server.

2. Redundant Servers vs. Faults Tolerated

As noted previously, to tolerate a failure, a certain level of redundancy is required. A significant question is "how many machines are required to tolerate a given number of simultaneous faults"? This turns out to be very easy to express as a simple equation:

$$\text{NumberOfMachinesNeeded} = 3 * \text{NumberOfFaults} + 1$$

Thus to tolerate 1 fault, 4 machines are required; to tolerate 2 faults, 7 machines are needed. While it obviously depends on the particular situation and the reliability needed, it is expected that most deployments of a FARGOS/SolidState-based application will use clusters of 4 machines.

3. Designing Byzantine-Fault-Tolerant Applications

Designing an application that is intended to survive a variety of possible fault scenarios, such as power and network failures, damaged memory, failing disk drives or CPUs or even hostile hackers, requires a design slightly different from a conventional application that makes no provision for tolerating a fault. While the FARGOS/SolidState technology hides almost all of the complexity involved in providing Byzantine fault-tolerance, the application designer still needs to consider a few items.

One significant difference is that the application will be running on more than one machine. A design that presupposes that all data items are entered into a master database will stop dead when the machine that hosts the database dies. The goal of deploying technology such as FARGOS/SolidState is to remove single points of failure—having the application services it controls depend on a single machine or service would defeat the purpose.

Obviously, certain actions can only be done once and most organizations have a logically- or physically-centralized database. For example, the picking list for an order should be printed once on the warehouse floor, not four times. Likewise, a customer wants their credit card charged only once, not four times. What this means is that to complete certain types of actions, it may be required that the machines and services that represent single points of failure be operational. If they are not, then the action does not happen. The goal for the web-based application designer is to keep failures in the infrastructure from impacting the end-user. If, for example, the printer server that prints picking lists on the warehouse floor is down, then customer should still be able to enter his order and go about his business, even though the picking list corresponding to his order will not be printed until the printer server is repaired. Note that this is not an example of tolerating a fault—as long as the printer server remains down, the picking lists will not be printed. The point is that the occurrence of this single-point-of-failure fault was not in an application upon which the end user was dependent in a time-critical fashion. A subsequent section in this document provides some suggestions as to how to structure applications and databases to eliminate many time-sensitive single points-of-failure.

The other issue specific to Byzantine-fault-tolerant applications is that each transaction is executed on all of the servers and the results of the transaction are verified for correctness. The operation performed must always produce the same result for a given set of inputs, which are comprised of the current stored state and any arguments provided to the operation. This means that results based upon random numbers, the current time, local server name, etc. are all unsuitable for a Byzantine-fault-tolerant transaction because the result will not be identical across all servers.

4. The FARGOS/SolidState HTTP Server Adapter

FARGOS/SolidState can be used to implement Byzantine fault-tolerant transactions in a wide variety of scenarios, many of which would not doubt be surprising to the developers of the technology. Almost any of these applications will deploy client-side components that are aware that they are interfacing with a FARGOS/SolidState-based infrastructure. While challenging, it is possible to create FARGOS/SolidState-based services that are accessed by clients that are completely unaware of the FARGOS/SolidState infrastructure.

Arguably, the World Wide Web is most widely deployed client/server application that does not permit the server-based components the luxury of dictating the characteristics of the clients that interact with it. Although the vast majority of web-based transactions are read-only (e.g., send an HTML file or GIF image), the small subset dealing with the collection of information provided by the end-user (e.g., e-commerce, forums, comment forms, etc.) is very important. As a rule of thumb, a web site operator never wants to lose information that a customer has taken the time to enter. As a user invests more time at a site, the probability of a failure occurring during her visit continues to increase. If that user was adding items to her shopping cart, the loss of an hour's worth of selections is going to be more significant than that of 2 minutes worth. Thus, the ironic correlation that, as the customer's investment in time and items selected increases, the probability of a failure losing all of customer's work also increases.

A set of classes was developed to integrate the FARGOS/VISTA HTTP server with the FARGOS/SolidState infrastructure so as to enable conventional web browsers to obtain the reliability of Byzantine fault-tolerant transactions. The deployment model assumed is graphically illustrated in Figure 1. The end user is permitted to use a very rudimentary browser: no ability to control the user's browser is assumed other than support for HTML forms and that the browser responds to an HTTP 302 (Moved Temporarily) indication, which requests the browser to load a different page. Technically, the desired HTTP return code should be 303 (See Other); however, in practice this is not handled properly by many web browsers. As a workaround, the HTTP 302 (Moved Temporarily/Found) return code is used instead. Careful readers will discover that this problem is also noted in section 10.3.4 of RFC 2616. These limited functional requirements means support for JavaScript or Java applets is not needed by the FARGOS/SolidState interfaces. Of course, the web site designer might exploit such facilities: the very limited set of required functionality does not prohibit the utilization of advanced browser capabilities..

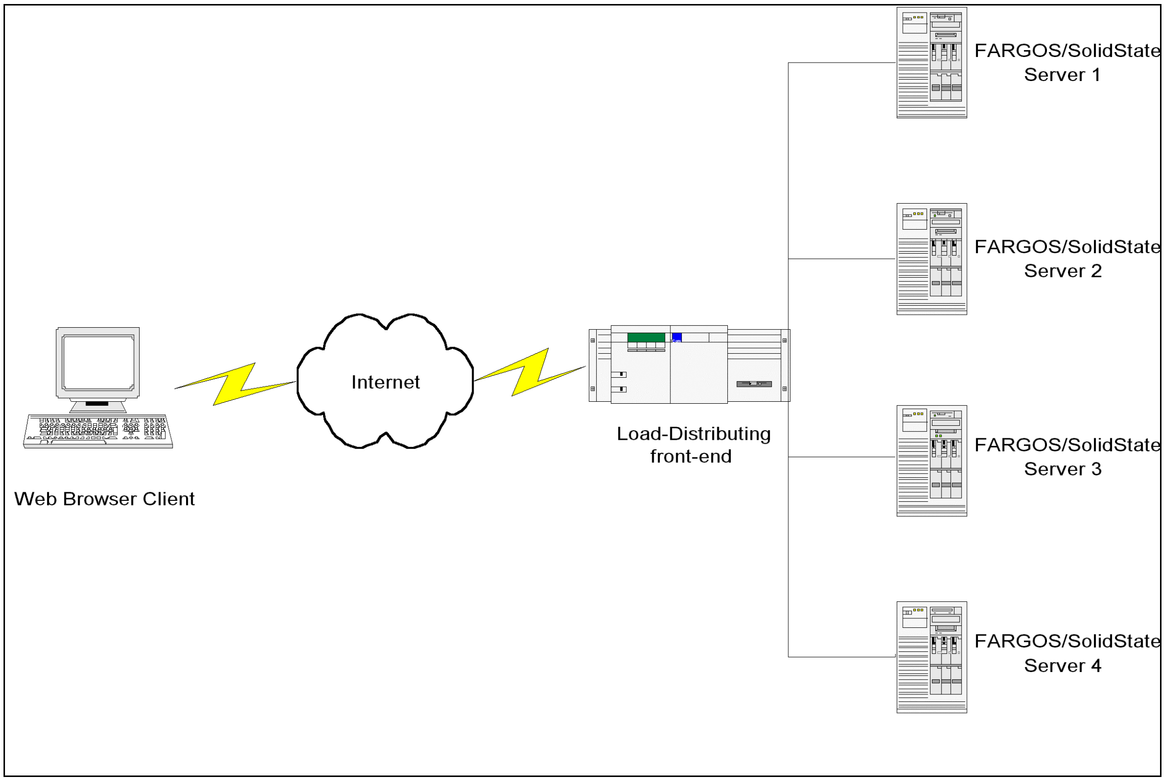
FARGOS/SolidState Server 1
FARGOS/SolidState Server 2
Internet
Load-Distributing front-end
FARGOS/SolidState Server 3
Web Browser Client
FARGOS/SolidState Server 4

Figure 1

Because programmatic logic is not assumed to exist on the web browser client, the detection and initiation of recovery from a failed server will normally be the responsibility of the user: they will have to press for a second time the “Submit” button within the web page's form. If the use of client-side programmatic logic is possible, the user’s browser can execute an auto-recovery procedure locally; however, these capabilities will be found in a much smaller browser population. In addition, the typical human response will be to press the button again, which would interfere with a running applet.

A set of servers, typically four in number, act as the server pool for transactions that are to be fault-tolerant. Sites provisioned with a large number of servers for scaling purposes should use the majority of them in a read-only fashion (e.g., serving up images, HTML documents). The pool of fault-tolerant servers should be used for transactions that maintain state on behalf of the user (e.g., noting the insertion of a particular item into a shopping cart). Alternatively, the server farm can be broken up into orthogonal server pools of 4 servers each.

The logical web site must be front-ended by a load-distribution product to prevent a remote client's web browser from being pinned against a failed server. For testing purposes, the **ForwardConnection** class implements a suitable front-end, but it is not a high-performance solution because it must forward responses from the HTTP servers back to the web browser clients. There are several varieties of high-performance load-balancing front ends on the commercial market and any given offering should be suitable. Most commercial products modify the incoming packet stream and normally avoid being involved in the outgoing data stream, which for a web or FTP site is by far the dominant direction of traffic flow. Note, however, that it would be wise to choose a product that provides support for highavailability; otherwise the load-distribution product itself becomes a single point of failure for the entire site[1].

FARGOS/SolidState Performance Characteristics

By default, FARGOS/SolidState returns results to client applications as soon as the correctness of a given result can be asserted. One benefit is that the speed of transaction processing is not limited by that of the slowest server in the pool. This issue is discussed in more detail later in this document (see Performance vs. Lock-step Synchronization). An even more significant benefit of this default behavior is that when a failure does occur, there is a high probability that no delay at all will be introduced while the system reorganizes itself to compensate for the detected fault. For the most common configuration of 4 servers, the probability of encountering a delay due to the detection of a fault is 25%. When 7 servers are in use, the probability of encountering a delay is less than 15% during the detection of the first failure and less than 17% during the detection of the second simultaneous failure.

It is very important to realize that, if FARGOS/SolidState does detect a failure and reconfigures itself, rather than operate in a degraded, albeit still correct, mode, applications continue to operate at full speed. Thus, reconfiguration is a one-time event that occurs when a fault is first detected and the probability of the reconfiguration introducing a delay that is visible during this one time event is at most 25%![2] Performance issues in the presence of failures are commented upon again elsewhere in this document (see Performance Implications of a Failure).

5. Byzantine-Fault-Tolerant Sessions

Before Byzantine-fault-tolerant operations can be performed on behalf of a user, a session needs to be created. The creation of a session causes a collection of objects to be created within the FARGOS/VISTA environment: these objects will be dedicated to handling Byzantine-fault-tolerant transactions for the user until the session is complete.

The facilities provided by the FARGOS/SolidState HTTP server adapter are intended to be useful for a variety of web-based applications. Consequently, some items of information are passed as arguments when a session is created. These arguments indicate the type of transaction session and the first page to display. The type of the transaction session determines what sort of actions can be performed: an application that implements a shopping cart for an online store would be expected to have different functionality than one that implemented an online brokerage.

The creation of a session and specification of these parameters occurs as a result of an HTTP GET operation. This means that the appropriate request would normally be embedded in an HTML document[3]. An example HTML anchor appears below:

```
<A href="/transactions/createTransaction?page=intialPage&type=transactionType">
```

[1] Sites interested in fault-tolerant transmission of bulk data from the web site to clients (e.g., video streams, downloads of program executables) can consider the **FARGOS/SolidConnection** offering, which also provides such load-distribution facilities.

[2] These performance characteristics are so incredible that they deserve an explanation point.

[3] Clearly, specialized clients that make HTTP requests can be written; however, the focus of this discussion is on the development of applications that only assume the use of a conventional web browser and do not require the installation of additional software on the client machine.

As can be seen, the specification is not very complex. Much of the work of implementing Byzantine-fault-tolerant transactions takes place in HTML pages and does not require any programming.

There are two parameters passed as part of the anchor prototype illustrated above. The first is the *page* parameter. Its value indicates the source of the initial page to be displayed after the transaction is created and is typically the file name of an HTML source file. The other parameter, *type*, indicates the type of transaction to be created. For security purposes, the name of the class that implements a particular transaction type is not used directly. Otherwise, objects of arbitrary classes could be created by hostile third parties using a suitably constructed HTTP GET command[4]. Instead, a class that implements support for web-based transactions is assigned a logical name, which could be exactly the same as the name of the implementation class, and registered with the transaction service. The logical name is used as the value of the *type* parameter. Below is a sample used in a web-based calculator:

```
<A href="/transactions/createTransaction?page=calc.html&type=calculator">
```

6. Implementation of Byzantine-Fault-Tolerant Transactions

As noted above, one of the parameters specified when a new Byzantine-fault-tolerate session is created is the type of the transaction. A given transaction type corresponds to a FARGOS/VISTA-based class that implements the logic for a particular Byzantine-fault-tolerant state variable. Typically, this is where operations like adding an item to a shopping cart are implemented. In general terms, the state variable is passed a variety of parameters that were embedded in an HTML page. These would indicate, for example, what operation is to be performed and any additional information (like a product ID, desired quantity, etc.). The state variable has to perform its work and then it returns a value. To ensure correctness, the Byzantine-fault-tolerant replica controller checks this value for validity against all of the values returned by the other servers.

Although running under the control of FARGOS/SolidState, HTTP-based applications still must integrate with the HTTP server. Programmer's should be aware of facilities provided by the standard FARGOS/VISTA Object Management Environment and they will find that the *FARGOS/VISTA HTTP Server Programmer's Guide* devotes itself to the topic.

Registering Session Types

The value provided for the type parameter is mapped to the actual name of an implementation class that has previously been registered. This registration can be performed creating **RegisterReplicaHTTPclass** objects. For example, in an *rc* file for the FARGOS/VISTA Object Management Environment daemon:

```
RegisterReplicaHTTPclass calculator WebCalculator
```

Multiple **RegisterReplicaHTTPclass** objects can be created to declare a suite of classes that implement state variables that provide support for various transaction types.

Application-Specific State Variables

The FARGOS/SolidState HTTP adapter package provides a convenience class, **HTTPreplicaStateVariable**, from which new classes can inherit. This class provides several useful utility methods that make it convenient to dynamically prepare and register

[4] Sadly, this kind of mistake is made all too often by developer who assume that their specially constructed URLs are too complicated for anyone to decipher.

new web pages that are the result of a particular transaction. While it is not mandatory[5], it is strongly recommended that developers inherit from this base class.

The key method that all FARGOS/SolidState-based state variable objects must implement is **processRequest**. Its prototype is illustrated below:

```
ClassName:processRequest(int transactionID, int clientTransID, array requestData,
assoc options, array uriInfo, array formParams, assoc substitutionList)
{
      // compute result
      return (result);
}
```

The *transactionID* parameter is a value assigned by the underlying replica controller. It uniquely identifies the transaction. The *transactionId* can be used by the transaction logic whenever a unique value is needed that will be shared between the replicas. Note that many normal mechanisms for generating unique identifiers (such as thread object ID) will generate a unique value at each instance of the state variable that is distributed amongst the cooperating members of the server pool and thus be useless for the purposes of a Byzantine fault-tolerant transaction.

The *clientTransID* is a (hopefully) unique identifier assigned by the client that identifies the request it made. Because the FARGOS/SolidState logic caches results, the proper use of client transaction Ids allows a repeated (duplicate) transaction request to be satisfied by returning a previously cached result. In practice, this is a mandatory requirement for applications that use the FARGOS/SolidState facilities directly. Note that since the results returned by applications that use the HTTP adapter are actually web pages, the FARGOS/VISTA HTTP server caches the dynamically generated web page (see descriptions of the classes **HTTPdaemon** and **URLdirectory**). Thus, a repeated request (via an HTTP GET or POST) will be handled by the HTTP server sending back the cached page (see **HTTPcachedObject**) instead of attempting to reissue the transaction via the FARGOS/SolidState HTTP adapter.

The array *requestData* is a parse of the HTTP request line (e.g., a GET or POST) that initiated the transaction. The first subscript (0) is the command; the second is the Uniform Resource Identifier (URI, see RFC 2396 for the format) requested and the third is the HTTP version. Most transaction logic will only be concerned with the URI. It can be conveniently parsed using the **parseHTTPuriData()** function provided by the FARGOS/VISTA Object Management Environment core.

The *options* parameter is an associative array whose subscripts are the names of options that were provided in the HTTP request header. Each name has been converted to lowercase and includes the trailing colon.

The URI specified as *requestData[1]* is already parsed into its component elements and made available as the array *uriInfo*. The first element, subscript 0, is the scheme in use (typically http). The second element is the host and the third is the file that is being requested. The fourth element (subscript 3) contains the options specified (items after a “?”). Any positioning indicators (following a “#”) are placed as the fifth and final element of the array.

Data that was provided in the body of a POST request is provided as the parameter *formParams*. This is an array of associative arrays. Each element of the array corresponds to one item of POST data and maintains the order in which the data was provided. The associative array that is found in each subscript of the array contains a single element, which

[5] It is not mandatory because FARGOS/VISTA supports allomorphism in addition to polymorphism. It is sufficient for a class to implement the interesting methods found in a base class—it does not have to inherit it.

is subscripted by the name of the field. The corresponding value is a string holding the text that was provided as the field's value.

The last argument is an associative array of substitution variables. Each key of the associative array *substitutionList* corresponds to a string pattern that will be used in a global search-and-replace operation against an HTML source file. The value of each element indicates the text that will be substitute wherever the key's pattern was found. Several variables are predefined and will always be present:

- TRANSACTION
- PROTOCOL
- FULL_TRANS_PATH

Other variables may be present as well.

Returning Results

While the underlying FARGOS/SolidState replica controller can handle a value of any kind, the adapter classes that integrate this facility with the FARGOS/VISTA HTTP server impose some structure. The adapter code mandates that the result be returned as an array that has at least two elements. The first element of the array is the transaction ID associated with the transaction. The second is the name of the prepared page to which the user's web browser should now be directed. It is not required, but the computed value should be provided as the third element of the result array. This enables the Byzantine-fault-tolerant replica controllers to verify the correctness of the result. A complete sample appears below:

```
result[0] = transactionID;
result[1] = resultPageName;
result[2] = computedValue;
```

Typically, the contents of the result page will need to be dynamically generated based upon the results of the computation. In such cases, the new result page should be prepared and registered with the web server before the state variable returns its result. The **HTTPreplicaStateVariable** class provides a **loadAndRegister** method that will load an HTML file that acts as a template, perform a set of global search and replace operations against the text based on the contents of a substitution list, and register the resulting dynamically created text with the HTTP server. This method takes four arguments: the page to be loaded as a template, the virtual site name, the name to be assigned to the generated page and the substitution list. This is illustrated below:

```
obj = call “loadAndRegister”(srcPage, options[”host:”], generatedName,
      substitutionList);
```

The Server Side Include processor is automatically utilized as needed (see class **HTTP_SSIprocessor**).

7. Performance Issues

By default, FARGOS/SolidState-based applications are configured to return results to a client as quickly as possible. Some applications may find it desirable to alter the default parameters based on considerations discussed below.

Performance vs. Lock-step Synchronization

Each client that makes use of FARGOS/SolidState-based transaction facilities actually issues requests via an object that is dedicated to the client. One of the responsibilities of this object is to verify the correctness of the results obtained from the members of the server pool. Recall that, unlike previous systems, FARGOS/SolidState provides Byzantine-fault-tolerance, so just returning a result to the client is not sufficient—the result must be correct.

It is important, however, to realize that the actual implementation of this logic was done in such a fashion as to return a result to the client as soon as the correctness of the result can be asserted.

It may not be obvious to the reader, but this assertion can be made after receiving

ToleratedNumberOfFaults + 1

identical results from participating servers. Thus, if one failure is to be tolerated, 2 identical results from the pool of 4 servers are all that is needed to assert the correctness of the result. If two failures are to be tolerated, then 3 identical results are needed from the pool of 7 servers.

This bias towards quickly returning results means that at least half of the participating servers may not have finished their independent computation of a result at the point in time when a response is provided back to the requesting client. For conventional applications, this is not an issue; however, the use of load-distributing front-ends to web sites creates a potential problem: the next request for a web page that is made by a browser client may be sent to one of the servers that has not finished computing its result.

In practice, this will probably not be an issue due to the tendency for server pools to be comprised of identically configured machines, thus providing similar performance characteristics. Also, the latency involved in sending a response across the Internet to the browser client and subsequent reception of the client browser's next request provides an additional window of time in which to allow the slower servers to finish computing their results.

The probability of such an event occurring can be reduced (but not made impossible) by increasing the verification limit from

ToleratedNumberOfFaults + 1

to

3 * ToleratedNumberOfFaults;

however, the overall response time of the system will be somewhat degraded[6].

Performance Implications of a Failure

Many systems that are capable of tolerating a fault continue to produce correct results but run in a degraded performance mode. FARGOS/SolidState-based applications do not suffer from such performance degradations:

- The failure of any server other than the current primary replica controller has no effect on performance.
- The failure of the current primary replica controller introduces a short delay. This delay is caused by the need to detect the actual failure of the primary. Once the failure is detected, the surviving servers recover and a new primary is selected. From this point on, the system continues to run at full speed.

These performance characteristics mean that a FARGOS/SolidState-based application only suffers a performance degradation if-and-only-if the failure affects the current primary replica controller. The probability of this being the case is 1 out of the total number of servers that are cooperating, thus for the common case of 4 servers the probability is ¼ or 25%.

[6] The **ReplicaClientProxy** class provides a **raiseWaitLimit** method that can be used to adjust how many responses are required from participating servers before an answer can be returned to the client.

If the failure does affect the primary replica controller, a short delay is introduced while the failure is detected and the servers reconfigure themselves. This delay is incurred only once. From that point onwards, the system returns to operating with absolutely no degradation in speed.

8. Calculator Example

This example illustrates a simple form-based calculator. The HTML pages that provide the basis of the application are described first.

Creating a Session

The HTML page below creates a new Byzantine-fault-tolerant transaction session: The initial page to be displayed will be “*example2.html*” and the type of the transaction is specified as “*calculator*”.

```
<!DOCTYPE HTML PUBLIC “-//W3C//DTD HTML 3.2 Final//EN”>
<HTML>
<HEAD>
<TITLE>Create New Session</TITLE>
</HEAD>
<BODY>
<p>Start transaction:  click
<A href=”/transactions/createTransaction?page=example2.html&type=calculator”>
      here
</A>
</p>
</BODY>
</HTML>
```

Sending a Request

Input from a user is often obtained using an HTML FORM. The HTML file below corresponds to the “*example2.html*” file referenced above. It provides a form into which a user can enter values and a desired operation. Submission of the form will cause the data provided by the user to be processed by the Byzantine-fault-tolerant state variable associated with this user’s current session.

```
<!DOCTYPE HTML PUBLIC “-//W3C//DTD HTML 3.2 Final//EN”>
<HTML>
<HEAD>
<TITLE>Create New Session</TITLE>
</HEAD>
<BODY>
<h1>Calculation Result</h1>
<!$@RESULT>
<p>
<form name=”ComputeForm”
   action=”$@FULL_TRANS_PATH?page=example2.html&method=compute”
   method=”post”>
      <input type=”hidden” value=”$@TRANSACTION” name=”transactionID”>
      <p>Operand 1:<input type=”text” name=”operand1” size=”24”></p>
      <p><select name=”selectName” size=”1”>
            <option value=”add” selected>+
            <option value=”subtract”>-
            <option value=”multiply”>*
                  <option value=”divide”>/
      </select></p>
      <p>Operand 2:<input type=”text” name=”operand2” size=”24”></p>
      <p><input type=”submit” name=”submitButtonName” value=”Compute”>
      <input type=”reset” value=”Reset Default”>
</form>
</p>
</BODY>
</HTML>
```

Note the use of the <!$@RESULT> comment above. Untouched, this remains a comment in an HTML source file and is not visible to the user when the web browser displays the page. If, however, the substitution list includes an item whose key is <!$@RESULT>, the comment will be replaced with alternative text. The state variable's **processRequest** method can thus replace the comment with arbitrary HTML text that will display the current result. A "trick" such as this allows the same template page to be used for both the initial page (before any transaction has been executed) and subsequent display of results. Some web site design products use a convention of "<!--comment text-->", so the name of the substitution variable may be forced to include the leading and trailing "--" characters.

Processing a Request

Every transaction requires an application-specific class that implements the appropriate functions associated with the Byzantine-fault-tolerant state variable. The source code below implements the calculator logic. The **create** and **delete** methods of the class are responsible for registering and un-registering the name of the state variable with the local system. While the code displayed below can be copied directly into another class, it may help to understand its purpose.

When a Byzantine-fault-tolerant session is created, the appropriate state variables are created on each of the servers participating in the fault-tolerant cluster by the FARGOS/SolidState system. For each session, one state variable is created on each server. Each of the state variables has a unique object identifier and the FARGOS/SolidState replica controller code uses these object Ids to identify and address specific replicas of a state variable. The additional logic that integrates the HTTP server to the facilities provided by FARGOS/SolidState also creates on each server copies of an object that acts as a proxy for the user's browser. It does this so that any server that receives a request to process a transaction will have a locally resident proxy for the client. Again, these objects will have their own unique object Ids, but each one registers with its local web server using the name of the transaction session. Because the creation of these objects takes place as two distinct steps, the client proxy does not know about the object Id of the local copy of its associated Byzantine-fault-tolerant state variable. The solution is to take advantage of a FARGOS/VISTA capability and assign a name to the object and use that name instead of an object Id when addressing the state variable from the client's proxy object.

```
class Local . WebCalculator {
      string      myOIDname;
      oid         urlDirectory;
} inherits from HTTPreplicaStateVariable;


WebCalculator:create(string stateObjName)
{
      myOIDname = stateObjName;

      registerService(myOIDname, thisObject, 0);
}

WebCalculator:delete()
{
      unregisterService(myOIDname, thisObject);
}
```

The remaining method of the class does the actual work each time a request is received by the web server.

```
// HAS TO BE IDEMPOTENT BASED ON ARGV

WebCalculator:processRequest(int transactionID, int clientTransID,
      array requestData,
      assoc options, array uriInfo,
      array formParams, assoc substitutionList)
{
      array       result;
      string      newPage;
      string      page, host, method;
      assoc       queryData, formData, subData;
      string      query, selectName, fullName;
      any         op1, op2, computeResult;
      any         obj;

      query = uriInfo[3];
      queryData = parseHTTPformData(query, assoc);

      // do computations...
      formData = call "collapseForm"(formParams);
      if (indexExists(formData, "method") != 0) {
            method = formData["method"];
      }
      op1 = stringToNumber(formData["operand1"], any);
      op2 = stringToNumber(formData["operand2"], any);
      selectName = formData["selectName"];
      if (selectName == "add") {
            computeResult = op1 + op2;
      } else if (selectName == "subtract") {
            computeResult = op1 - op2;
      } else if (selectName == "multiply") {
            computeResult = op1 * op2;
      } else if (selectName == "divide") {
            if (op2 != 0) {
                  computeResult = op1 / op2;
            } else {
                  computeResult = 0;
            }
      }

      subData = substitutionList;
      subData["<!$@RESULT>"] = makeAsString("<P>Last result=",
                  computeResult, ".</P>");

      // Return page
      newPage = queryData["page"];
      fullName = makeAsString(subData["$@TRANSACTION"], "?page=",
            newPage, "&id=", transactionID);
      obj = call "loadAndRegister"(newPage, options["host:"], fullName,
            subData);

      result[0] = clientTransID;
      result[1] = fullName;
      result]2] = computeResult;

      return (result);
}
```

9. Shopping Cart Example

Many web sites engaged in e-commerce with end-users use the abstraction of a shopping cart to keep track of what items the user has selected for purchase. Browsing an web-based store front's wares and putting together an order can involve the expenditure of significant amounts of time on the part of a potential customer. The last thing a vendor wants is to loose a customer due to a failure of the web site. The example provided here illustrates how such a shopping cart can be implemented using FARGOS/SolidState and thus yield the fault-tolerance currently missing from the web today.

Configuration of the VISTA daemon

Each of the servers in the processor pool will use an *rc* file similar to the following:

```
PeerRegistry
RegisterPoolMembership LocalDemoCluster
HTTPdaemon profile tcp:0.0.0.0:1971
RegisterReplicaHTTPclass shop WebShoppingCart
CreateReplicaHTTPsession HTTPreplicaClientProxy LocalDemoCluster
AcceptPeerConnections tcp:127.0.0.1:1981
ConnectToPeer tcp:127.0.0.1:1982
ConnectToPeer tcp:127.0.0.1:1983
ConnectToPeer tcp:127.0.0.1:1984
```

The creation of a **PeerRegistry** object enables the distributed registry between VISTA daemons. This is followed by the creation of a **RegisterPoolMembership** object. This will register the server as being a member of the indicated pool. A server can participate in more than one pool by having more than one **RegisterPoolMembership** object created with the appropriate arguments. The name of the pool is administratively assigned and it was chosen to be *LocalDemoCluster* for this illustration.

The creation of an **HTTPdaemon** allows HTTP requests to be processed. Note that the traditional port 80 is not used in this example. It could have been, but using a non-standard port drives home the point that a load balancing front-end will be used to distribute incoming requests across the various servers.

Each type of Byzantine-fault-tolerant transaction is implemented using a support class. The class for this example is registered by creating a **RegisterReplicaHTTPclass** object and passing the two arguments that associated the public name for the transaction type that will appear in HTML directives and the actual name of the implementation class. In this case, the type name of "shop" was chosen and the logic is implemented within the class **WebShoppingCart**. The final line specific to this example registers the FARGOS/SolidState HTTP adapter with the web server and specifies that the *LocalDemoCluster* pool of processors is to be used.

The remaining lines accept unsolicited connections from other FARGOS/VISTA peers and actively establish connections to other members of the pool.

Creating a Session

At some point, the user's browser needs to perform an HTTP GET or POST on the URL that causes a new session. That URL is:

```
/transactions/createTransaction?page=nextPage.html&type=shop&method=init
```

The value of *nextPage.html* for the *page* parameter would be set to whatever is appropriate for the site.

The *type* parameter's value is set to "shop", which corresponds to the name used as the first argument to the **RegisterReplicaHTTPclass**. The *method* parameter is set to *init* and acts as an explicit indication that the intent is that the session is being created as the result of this request. This parameter can be examined by the state variable to provide it information with respect to what is expected.

Most of the pages associated with the user's session will contain dynamically generated content. As an illustration, the code below provides a table of available for purchase and lists the contents of the user's shopping cart on the main page. The corresponding HTML used to do this is shown below.

```
<p>Click on an item below to learn more:</p>
<p><!--$@ITEM_LIST--></p>
<p>Your demonstration shopping cart currently contains:</p>
<p><!--$@CART_CONTENTS_LIST--></p>
```

As can be seen, there are two variables used, bracketed by appropriate HTML content that will always be constant.

Adding an Item to the Shopping Cart

When the user decides to purchase an item, they fill out a form with the desired quantity and submit the request. An appropriately constructed HTML form will cause the appropriate POST request to be sent to the HTTP daemon. The HTML form that corresponds to the **WebShoppingCart** class appears below:

```
<p>Picture:</p>
<p><!--$@ITEM_DETAIL_PICT--></p>
<p>Description:</p>
<p><!--$@ITEM_DETAIL_DESC--></p>
<p>Price:</p>
<p><!--$@ITEM_DETAIL_PRICE--></p>
<p>To purchase, enter the desired number of items below:</p>
<p>
<form name="FormName" action="$@FULL_TRANS_PATH?page=page1.html&method=addToCart"
    method="POST">
      <input type="hidden" value="$@ITEM_DETAIL_SKU" name="SKU">
      <label>Desired Quantity:</label>
      <input type="text" name="qty" size="3" maxlength="6"></p>
      <p><input type="submit" value="Modify Cart" name="submitButtonName">
</form>
```

Processing Transactions

Every action taken by a user that causes a change in the state of their shopping cart must be processed by the set of Byzantine-fault-tolerant state variables associated with the user's session. This illustration makes use of the **ReadCSVfile** class, which reads comma- (or tab-) separated values from a text file and provides a level of relational-database like query functionality. The **ReadCSVfile** class assigns column names based on the contents of the first line of the file. A sample file appears below:

```
SKU,Description,Price,Image
dress,A dress,79,dress.jpg
shoes,Shoes to wear,59,shoes.jpg
tackle,Box to tackle,159,tackleBox.jpg
figs,Tacky figurines,19,figurines.jpg
abacus,Fault-tolerant abacus,9,abacus.jpg
booth,Presentation Booth,259,booth.jpg
radios,Walkie-Talkies,139,radios.jpg
watches,Go-Blue watches,45,watches.jpg
house,Two bedroom house,99,house.jpg
```

The **create** method of the **WebShoppingCart** class registers the object as a service whose name was passed as an argument. Any support class utilized by the

HTTPreplicaStateVariable class must perform the same operation. The **delete** method does the inverse by un-registering the service when the object is deleted.

```
global {
      const string DB_FILE = "../catData.csv";
};

class Local . WebShoppingCart {
      string      myOIDname;
      oid   urlDirectory;
      assoc purchasedItems;
      oid   dbObj;
} inherits from HTTPreplicaStateVariable;


WebShoppingCart:create(string stateObjName)
{
      assoc acl;

      myOIDname = stateObjName;
      acl = makeDefaultACL();
      dbObj = send "createObject"("ReadCSVfile", acl, DB_FILE)
            to ObjectCreator;

      registerService(myOIDname, thisObject, 0);
      display("Created WebShoppingCart, ", stateObjName, "\n");
}

WebShoppingCart:delete()
{
      unregisterService(myOIDname, thisObject);
}
```

The **processRequest** method is invoked whenever any modification of the Byzantine-fault-tolerant state variable is performed. This includes the first-time initialization when a new session is created and thus provides an opportunity to redirect the browser to an initial page affiliated with the user's new session.

The first step is to obtain information about the requested transaction. There are two sources for this: the URI passed by the browser and any additional, albeit optional, form-related data that might be present in a POST request. The URI is conveniently pre-parsed and the various options can be extracted as subscript 3 of the *uriInfo* argument. The **parseHTTPformData()** function conveniently parses this and can return an associative array that holds the arguments that were provided. The elements of the *uriInfo* array are:

Subscript	Value
0	Scheme (usually, "http")
1	The host name/address
2	The referenced page
3	Arguments after a "?"
4	Parameter arguments after a "#"

Form-related data is passed as an array of associative elements, which preserves the sequential order in which items were sent. Most form fields are uniquely identified, so often order is not necessary and it is more convenient to access individual fields by their names. The **collapseForm** method of the **HTTPreplicaStateVariable** class is used to convert the form parameters into a single associative array.

```
WebShoppingCart:processRequest(int transactionID, int clientTransID,
      array requestData,
      assoc options, array uriInfo,
      array formParams, assoc substitutionList)
{
      array result, headings, prodList;
      string      newPage;
      string      page, host, method;
      assoc queryData, formData, subData, selectList;
      string      query, selectName, fullName, key;
      string      detailPage, sku;
      any   op1, op2, computeResult;
      any   obj;
      int   nextID, qty;

      display("WebShoppingCart:processRequest transID=", transactionID,
            " clientTransID=", clientTransID, " argc=", argc, "\n");
      display(argv);

      nextID = clientTransID + 1;

      query = uriInfo[3];
      queryData = parseHTTPformData(query, assoc);
      if (indexExists(queryData, "id") != 0) {
            nextID = stringToNumber(queryData["id"], int);
      }

      // do computations...
      formData = call "collapseForm"(formParams);
      if (indexExists(queryData, "method") != 0) {
            method = queryData["method"];
      }
      subData = substitutionList;
```

The operation to be performed is determined based on the value of a *method* parameter specified as part of the URI. A sequence of if-statements is used to take the needed action. Two values are of interest: "*addToCart*", which is used to add an item to the user's shopping cart and "*detail*", which is used to lookup information regarding a particular item and make this information available in substitution variables.

```
      if (method == "addToCart") {
            sku = formData["SKU"];  // hidden field...
            qty = stringToNumber(formData["qty"], int);
            if (qty == 0) {   // remove...
                  deleteIndex(purchasedItems, sku);
            } else {
                  purchasedItems[sku] = formData["qty"];
            }
      }

      if (method == "detail") {
            key = "$@ITEM_DETAIL_SKU";
            sku = queryData["SKU"];
            subData[key] = sku;

            selectList["SKU"] = sku;
            prodList = send "selectRecords"(selectList) to dbObj;

            key = "<!--$@ITEM_DETAIL_PICT-->";
            subData[key] = makeAsString("<IMG src=\"/images/",
                  prodList[0]["Image"], "\">");
            key = "<!--$@ITEM_DETAIL_DESC-->";
            subData[key] = prodList[0]["Description"];
            key = "<!--$@ITEM_DETAIL_PRICE-->";
            subData[key] = prodList[0]["Price"];
      } else { // init or addToCart
            // <!--$@ITEM_LIST-->
            selectList["SKU"] = "*";
            prodList = send "selectRecords"(selectList) to dbObj;
            key = "<!--$@ITEM_LIST-->";
            headings[0] = "SKU";
            headings[1] = "Description";

            detailPage = makeAsString(substitutionList["$@FULL_TRANS_PATH"],
                  "?page=detail.html&method=detail");
            subData[key] = call "formatTable"(headings, prodList, "SKU",
                  detailPage);

            // <!--$@CART_CONTENTS_LIST-->
            key  = "<!--$@CART_CONTENTS_LIST-->";
            subData[key] = call "formatPurchasedTable"();
      }
```

Once the transaction has been processed, a new result page needs to be generated. The page is extracted from the *page* parameter specified as part of the request's URI. Because the same template page is often used as the basis for displaying different content, the new page is qualified with the ID of the transaction, thus generating a unique name. The **loadAndRegister** method of the base class **HTTPreplicaStateVariable** is used to read the template page, perform the global search and replace for the substitution variables and register the resulting page with the HTTP daemon. The name of the new page and the current contents of the shopping cart are returned as the result of the transaction. Since nothing outside of this support class (**WebShoppingCart**) actually understands the meaning of this data, it might seem unproductive to return this information. The actual reason for returning it lies in the nature of the fault-tolerance being provided by FARGOS/SolidState. Because FARGOS/SolidState enables Byzantine-fault-tolerant transactions, correctness of obtained results is also checked. Returning the contents of a shopping cart allows it to be verified and a faulty server that either added or dropped an item from the shopping cart would be detected and tolerated.

```
      // Return page
      newPage = queryData["page"];
      fullName = makeAsString(subData["$@TRANSACTION"], "?page=",
            newPage, "&id=", nextID);
      obj = call "loadAndRegister"(newPage, options["host:"], fullName,
            subData);

      result[0] = clientTransID;
      result[1] = fullName;
      result[2] = purchasedItems;
display("WebShop:processRequest returns ", result);

      return (result);
}
```

The example provided above made use of two additional methods. For completeness, these two methods, which were used to format table data, are presented below. They do not directly deal with the interface to the FARGOS/SolidState HTTP adapter, so they can be skipped by the reader who is uninterested in such low-level programming details.

```
WebShoppingCart:formatTable(array headings, array list, string anchorKeyField,
      string detailPage)
{
      string      result, heading, row, body, elem;
      string      val;
      any   colName;
      int   i, j;

      if (elementCount(list) == 0) return ("");

      heading = "<TABLE><TR>\r\n";
      for(i=0;indexExists(headings, i) != 0; i+=1) {
            elem = makeAsString("\t<TD><B>", headings[i], "</B></TD>\r\n");
            heading += elem;
      }
      heading += "</TR>\r\n"; // end of heading row...
      body = "";
      if (typeOf(list) == array) j = 0;
      else j = nextIndex(list, 0);
      do {
            row = "<TR>\r\n";
            for(i=0;indexExists(headings, i) != 0; i+=1) {
                  colName = headings[i];
                  val = list[j][colName];
                  if (findSubstring(val, ".jpg") != -1) {
                        val = makeAsString("<IMG src=\"/images/",
                              val, "\">");
                  }

                  elem = makeAsString("<TD><A href=\"", detailPage, "&",
                        anchorKeyField, "=", list[j][anchorKeyField],
                        "\">\r\n", val, "</A></TD>\r\n");
                  row += elem;
            }
            body += makeAsString(row, "</A></TR>\r\n");
            j = nextIndex(list, j);
      } while (j != 0);

      result = makeAsString(heading, body, "</TABLE>\r\n");
      return(result);
}

WebShoppingCart:formatPurchasedTable()
{
      int   i;
      string      sku, result, row;

      i = nextIndex(purchasedItems, 0);
      if (i == 0) {     // empty table
            return ("<P><I>Your shopping cart is currently empty.</I></P>");
      }
      result = "<TABLE>";
      do {
            sku = getKeyForIndex(purchasedItems, i);
            row = makeAsString("<TR><TD>", sku, "</TD><TD>",
                  purchasedItems[sku], "</TD></TR>\r\n");
            result += row;
            i = nextIndex(purchasedItems, i);

      } while (i != 0);
      result += "</TABLE>\r\n";
      return (result);
}
```