Version: 7/6/2001 5:30 PM

FARGOS/VISTA

An Overview

FARGOS/VISTA: An Overview

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

An Overview of FARGOS/VISTA

FARGOS/VISTA[1] is a new, next-generation platform intended for the development and support of massively distributed global applications. It builds upon a decade of experience with state-of-the-art distributed object-oriented operating systems, and includes numerous features that exceed the capabilities of other systems.

A Short History

Technologies like Java, CORBA, and message queues are promoted by some organizations as key technologies, but none of the three dominates the arena of rapid development and/or deployment of distributed applications. Consequently, any programming organization responsible for writing and integrating applications that operate within a networked environment would be well served by investigating the functionality provided by the FARGOS/VISTA system and comparing it against other available technologies.

Beyond the mere mechanics of providing a technology infrastructure that enables the creation of transparently distributed applications, FARGOS/VISTA also pays attention to the development lifecycle: for example, how code is created and reused by multiple, potentially independent developers; internationalization; deployment of updated or new code into running systems.

Cooperative Applications

The FARGOS/VISTA infrastructure focuses on the development of distributed applications that can cooperate with one and another. Such applications can be written independently of each other and a previously written application can interact with new applications without being modified or even re-linked. This is a radical departure from conventional technologies and provides a host of capabilities:

- A FARGOS/VISTA-based application from one vendor can be extended later by code written by a different vendor without requiring access to the source code of the original application.
- Applications can be developed by different organizations and interact without requiring the exchange of header files and the corresponding burden of keeping libraries in sync.
- An application can be upgraded without requiring re-linking and re-deployment of applications that make use of functionality exported by the enhanced program.

As illustrated above, one feature of the FARGOS/VISTA technology is to enable multiple, yet independent, developers to implement applications that cooperate without requiring them to closely coordinate their development and synchronize their deployment activities. This is powerful functionality directly applicable to most environments, whether they use off-the-shelf software from multiple vendors or develop their own mission-critical applications in-house. FARGOS/VISTA provides the infrastructure for these capabilities (and many others) by utilizing a different application paradigm than that pursued for the past 20 years.

Distributed Application Paradigms

Networks of interconnected machines provide opportunities for new types of applications, increased reliability, utilization of idle resources, etc. If an application is to take anything other than limited advantage of such opportunities, it must be aware of the distributed nature of the environment.

---

[1] FARGOS/VISTA is an acronym for Fantastic And Really Great Operating System / Various Interconnected Systems with Transparent Access.

There are many middleware packages available in today's market and most implement their functionality using a Remote Procedure Call (RPC) paradigm. Two examples are the Object Management Group's CORBA "standard" and Microsoft's DCOM. Unfortunately, the RPC paradigm is, by design, a poor approach for building distributed applications that can cooperate.

The RPC paradigm makes the execution of a function on a remote machine appear as if it was a local function call. In simple terms, the RPC paradigm makes everything appear local, thus the RPC paradigm intentionally hides the distributed nature of an application. This is seductive, but ultimately very limiting and contributes to the development of fragile systems. Examples of some problems it creates:

- Programmers need to explicitly identify what functions are to be accessible by remote processes and implement them as remote procedure calls. This requires correctly identifying the appropriate interfaces and writing the server-side code. It frequently involves significant work to maintain state on the remote (server) side of the application. Design decisions that prove to be incomplete can require significant reengineering of the application and typically require maintaining old interfaces in a deprecated state. Such changes in turn contribute to application bloat and place increased burdens on software maintenance.
- Failures of a remote procedure call due to server or network problems are difficult to handle correctly, due to the illusion that the function call appears to be local. The consequence is fragile code that can fail spectacularly in the presence of an overloaded server or worse problem.
- Among other problems, it forces application programmers to write monolithic applications in a conventional fashion. The disadvantages of large monolithic programs are well known and some people hold out hope for using middleware packages as an infrastructure for building componentized applications.

Componentization of Applications

One trend that is promoted today is the use of components to build applications. Great claims of increased programmer productivity due to the reuse of previously written components are often touted. This can indeed be the case, but it assumes that the components are designed in such a way as to be suitable for reuse by other application programmers. While feasible, in practice most programmers are not skilled at delivering reusable code. For those with the necessary skills, many work in environments where the pressures of dealing with their immediate problems mean that short-term results are valued more than the optimization of productivity in the long term.

Another claim in favor of componentization of an application is that it creates an opportunity for increased scalability. An application faces a scaling problem when some finite resource required by it is exhausted. Such resources might be easily measured and predicted quantities like the amount of virtual memory or free disk space available on a host. They can also be resources that are influenced by a variety of factors, such as CPU cycles devoted to an application or available network bandwidth. There are two broad, complementary approaches to avoiding scaling problems. The first is to be frugal with the available resources. By using as little as possible, one can stretch the finite resources farther, allowing for larger problems sizes than would otherwise be possible. The second approach is to break the application into distinct pieces and distribute these pieces amongst physically separate systems. Such an approach can be effective because many of the resources are constrained by limits imposed by a given physical system (like a host or LAN). It is thus often possible to nearly double the scarce resource by adding a duplicate of the physical system in question.

The second approach creates its own set of difficulties. When the resource in question being doubled is physical hosts, there is a problem in that while many resources are doubled,

additional CPU cycles and network bandwidth are required to communicate between the hosts. If any significant communication takes place between hosts, the increased overhead may stall the application completely. Consider Microsoft's published numbers for DCOM (rounded to nearest magnitude):

- 3 million calls per second in the same process
- 2000 calls per second between processes on the same machine
- Less than 400 calls per second between process on different machines

A reduction in throughput from over 3 million calls-per-second to less than 400 just by breaking apart a monolithic application and distributing the pieces on distinct machines is a very significant performance reduction. This demonstrates that only components of applications that have very little interaction are suitable for distribution amongst multiple hosts; otherwise, the inter-machine communication overhead will dominate the workload and bring the system to a standstill. This very real issue should be at the forefront of any assertion that breaking a given system into pieces would help make it scale. Doubling the available CPU resources at the cost of introducing overhead that makes the system run 8000 times slower is not an improvement.

Breaking an application into distinct pieces, scattering them across machines and having them cooperate in a distributed fashion introduces new problems. These include determining the location of a given application or isolating the underlying cause of a problem when one of the applications is not working properly due to a hardware failure or incorrectly configured machine. It should be obvious that setting up and maintaining security in a distributed framework is much more difficult than doing the same for a single host.

Thus while componentization of application code is very attractive, components that use an RPC-based infrastructure rest on awkward and unstable ground.

In contrast to the RPC paradigm, FARGOS/VISTA provides an elegant design paradigm in which applications are built from the ground up in such a fashion as to enable their interaction with other applications in a transparently distributed environment. Instead of having to explicitly decide what functions should be accessible from remote systems, as is the case with RPC- or message queue-based systems, every item of data and every associated function is accessible from anywhere in the distributed system[2].

The FARGOS/VISTA Object Model

The FARGOS/VISTA runtime environment provides the infrastructure upon which transparently distributed applications can be built. Like the RPC paradigm, FARGOS/VISTA-based applications can invoke functions without regard to physical location: no distinction is made between local or remote. FARGOS/VISTA provides an object-oriented environment and FARGOS/VISTA-based applications are composed out of objects. It is reasonable to view these objects as miniature components. The object-oriented nature of the environment is not unique: this simply means that logically related pieces of data are represented as objects and the exposed APIs are functions, not the physical layout of the data in question. That said, there are many degrees of sophistication that can be associated with an object-oriented environment: the threshold required to use this buzzword is quite low, thus when comparing models, it is crucial to focus on the features provided and not merely the declaration of object-orientation.

In the FARGOS/VISTA object model, all objects are instances of a class. A class describes the data associated with an object and the operations that can be performed on such objects (these functions are referred to as methods). A class may inherit from other classes. Some

[2] Enforcement of access control may prevent a given user from interacting with a specific object.

systems only permit a class to inherit from one other class, but the FARGOS/VISTA object model supports multiple inheritance. Inheritance can be used to obtain additional functionality (such as inheriting a class that implements object persistence) or to handle special case behavior.

FARGOS/VISTA objects interact with each other by sending messages. This results in a method being run against the destination object as a separate thread of execution. Such fine-grained parallelism provides unique advantages; however, conventional operating systems are not able to provide fast enough performance to make this feasible[3]. The FARGOS/VISTA runtime environment is able to use native kernel threads, but the FARGOS/VISTA runtime environment also provides its own ultra-high-performance threading technology that is over 250 times faster than using native kernel threads. The use of native kernel threads provides increased performance on parallel processors as well as easy integration with legacy code, but for obvious reasons, the FARGOS/VISTA-specific threads are the mechanism of choice.

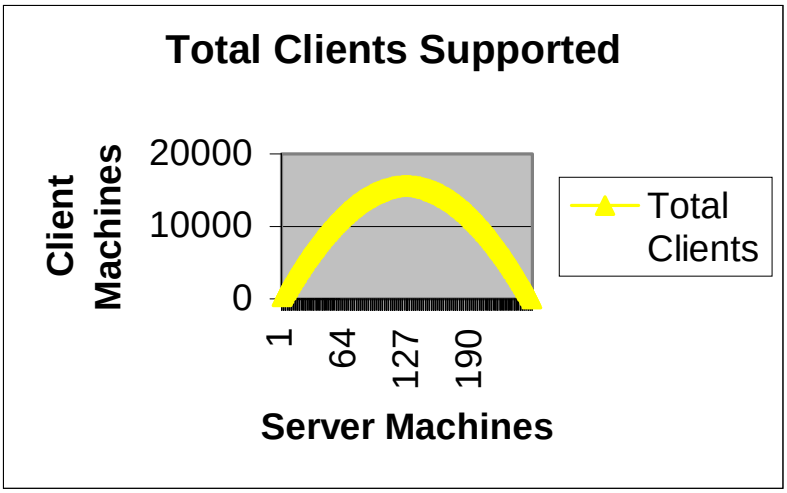
Each object in an FARGOS/VISTA-based system is accessible from any host participating in the distributed system. In contrast to most other middleware systems, the two objects do not have to reside on hosts that are directly connected. One result of this is that an object residing on a host in a TCP/IP-only domain can interact with an object residing in an IPX-only or SNA-only domain. This enables the integration of existing applications that were based on older networking technologies with those based on newer protocols. Another important aspect pertains to scaling. Middleware systems that require direct connection between the servers that support the distributed object environment do not have good scalability. As an illustration of this point, consider the following characteristics of a CORBA-compliant product from a well-known vendor:

> The vendor in question recommends that each management server handle a maximum of approximately 200 clients. This surprisingly low limit is imposed by a file descriptor limit found in many operating systems; however, this does not mean that the server will be lightly stressed. Indeed, it is recommended that the typical server machine be configured with a minimum of 64 megabytes of real memory and 128 megabytes of swap space for virtual memory.
>
> Following conventional recommendations, handling 10,000 clients would require a minimum of 50 servers. Because routing between management servers is not supported, if a truly distributed system is required a file descriptor will be used for each inter-server link, which would bring the number of servers required up to approximately 66.
>
> The graph below demonstrates how the total number of client machines handled can be increased by adding additional servers, but only up to a point. Not only does the benefit of adding new servers decrease, but at a certain point it actually reduces the capacity of the system. The result is that one cannot assert that a fully connected system will continue to scale by increasing the number of dedicated servers[4].

---

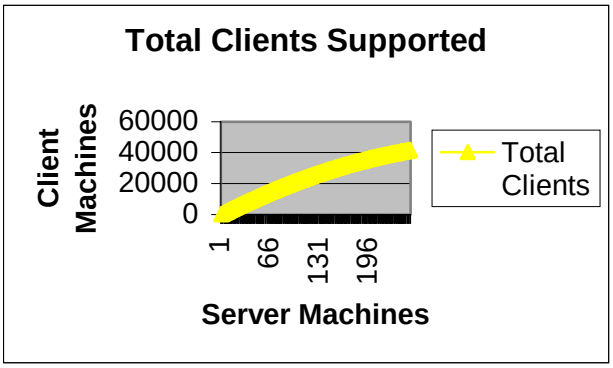
[3] Thread creation rates of around 20 per second are typical for kernel-provided threads.

[4] The product in question also has another critical deficiency, namely that servers exchange object registration tables when they connect. In a modest setup, this requires 30 minutes per server pair. For a network of N servers, there will be ½ N (N - 1) links between server pairs. With only 5 servers (serving around 1200 clients), this is 10 inter-server links and yields an expectation of 5 hours to initialize the system. Restart of a server will require 4 links to be established and would be expected to take about 2 hours. It should be clear from this example that a given middleware technology that works well in a small prototype environment may suffer scaling problems that render it useless for actual deployment in production environments. Unfortunately, this may only be discovered after significant effort has been invested in developing new applications and their deployment was attempted.



The data graphed was for a case that assumed 250 free file descriptors for each server process (approximately 20% over the suggested limits). In this case, the maximum number of client machines is 15625 using 125 servers. Note, however, that while some operating systems support more than 250 free file descriptors, others may support far less. Unless special care is taken, Windows NT-based applications will end up with FD_SETSIZE set to 64, and the **WaitForMultipleObjects()** kernel call is limited to waiting on at most MAXIMUM_WAIT_OBJECTS objects--which not surprisingly happens to be 64. In this case, assuming 60 free file descriptors, the maximum client total is 900 using 30 servers.

In contrast, FARGOS/VISTA can be deployed in such a fashion as to provide near-linear scalability. The graph below, using the same assumptions as the prior example, illustrates this:



Beyond its raw performance and scaling advantages over other technologies, FARGOS/VISTA focuses on the effort expended by programmers to develop new applications and maintain existing code. Programmer productivity is actually the most important aspect of FARGOS/VISTA: the intent of FARGOS/VISTA is to make programmers more productive and permit them to write applications previously beyond their reach. Programmers who used the predecessor to FARGOS/VISTA consistently obtained 6 to 10-fold improvements in productivity.

Independent Development

The development of a large application generally involves the efforts of more than one developer. Conventional technologies require such developers to cooperate closely and keep their code in sync. The FARGOS/VISTA system is designed to enable the utilization of code

written by different developers without requiring the exchange of header files. A developer can correct or enhance a piece of code, and applications that utilize it do not need to be recompiled or re-linked. These powerful capabilities enable the creation of non-stop systems that not only tolerate hardware failures but also permit the replacement of code without requiring a restart. This can be used by an organization that operates 24 hours-a-day, 7 days-a-week to upgrade existing or introduce new production applications without causing a service outage.

Polymorphism and Allomorphism

In typical object-oriented systems, inheritance is used to create specialized classes with generic interfaces. This is called polymorphism: an object can appear to be of more than one class. Applications can usefully interact with objects by treating them as instances of their base class, but specialized methods will override the default implementation provided in the base class. Inheritance is a powerful mechanism; however, some systems only provide support for single inheritance instead of the more general multiple inheritance. Providing support for only single inheritance makes it difficult for a developer to provide generic facilities that can be combined through inheritance. FARGOS/VISTA provides support for multiple inheritance, thus introducing no artificial restrictions on functionality. In addition to support for polymorphism, FARGOS/VISTA also supports allomorphism: a class can provide method interfaces that look like those of another class, but it does not need to inherit from the look-alike class. Code that expects to interact with objects of the original class can thus interact with objects of the new class. This capability is very powerful, but rarely found in other object-oriented technologies.

The elegance of the FARGOS/VISTA paradigm contributes directly to programmer productivity. It is complemented by the extensive and often unique functionality provided by the FARGOS/VISTA runtime environment.

Name Spaces and Versioning

FARGOS/VISTA classes are uniquely identified by three attributes:

1) The name space in which the class is defined
2) The name of the class
3) The version ID of the class

A name space is a text string that is used to identify a collection of classes. Its primary purpose is to provide a mechanism to prevent name collisions between classes created by independent development organizations. When creation of an object is requested, the name space of the desired class can be specified to remove any ambiguity. It can also be left unspecified and in that case, a series of name spaces will be searched to find the indicated class. This is one way to perform replacement of a class implementation with a locally enhanced version without requiring access to the original source code.

Each class also has a version ID associated with it. More than one version of a given class may be simultaneously supported within the FARGOS/VISTA infrastructure. This is a rarely found feature, but it is really a fundamental requirement for any system supporting evolving applications and persistent data[5]. This is an interesting capability for non-

[5] If a system does not support multiple versions of a class, changes to the storage layout of a persistent object are a significant problem. It requires shutting the system down, running an offline procedure to convert the old saved data to the new layout and then restarting the system. It is a complicated enough procedure if the data is all on a single server, much less scattered across many hosts.

persistent objects as well: an application can be upgraded by deploying the new versions of its classes and they will not interfere with versions that are already executing.

Security

FARGOS/VISTA allows the implementation of distributed applications that work in environments that are not completely trusted. Every FARGOS/VISTA-based object has access control lists associated with it that indicate who is allowed to invoke a particular method against the object, so security is an inherent aspect of the environment. While this imposes a non-zero amount of overhead on every method invocation, the implementation has been done in such a way as to make the overhead very small and thus make it practical to provide such fine-grained access control.

Method Overloading

An FARGOS/VISTA class can have more than one implementation of a given method name if the distinct implementations take different arguments. This capability has long been known to C++ programmers as overloaded functions. For a given positional argument, a specific type (like an integer or floating point number) may be required or any type may be acceptable. It is easy to provide a default implementation by providing a method that will accept any type of data. The functionality provided by FARGOS/VISTA is thus a combination of the style provided by C++ and that used in its own implementation.

In addition to supporting multiple implementations of a given method name, the same method body can be used to implement methods with different names. This is referred to as an "alias". Many times programmers implement methods that take different arguments, but contain virtually identical code. The use of an aliased method permits the method body to be written only once. Writing a piece of code only once has in turn several obvious benefits: faster development, less opportunity for bugs, easier future maintenance.

Self-Describing Environment

As previously mentioned, FARGOS/VISTA supports the development of applications that manipulate data whose type is not known at compile time. This is easy to do in FARGOS/VISTA because all data is tagged and its type can be inquired at runtime. This is an incredibly powerful capability and it is doubtful that it can be fully appreciated unless one has had the opportunity to utilize such a system.

FARGOS/VISTA does not just support tagged data, it implements a complete self-describing environment. Applications can determine what classes are loaded in the environment at runtime and their characteristics.

Reflection

Reflection is another powerful capability intrinsic to the FARGOS/VISTA runtime infrastructure. It permits the processing of a method invocation against an object to be delegated to another object, called a "meta-object". FARGOS/VISTA supports reflection on both a per-object and a per-class basis. The uses for reflection are limited only by a programmer's imagination, but some examples include: tracing method invocations against an object for debugging or profiling purposes, patching a running application, implementing a routine that can handle an invocation of any method name, etc.

Per-class reflection is useful to enable a single object to handle method invocations against all objects of a particular class. As an illustration, consider a set of persistent objects whose storage layout needs to be changed due to deployment of a new, enhanced version of its class.

The new and old version of the class are uniquely identified by their version Ids and, as noted above, FARGOS/VISTA permits multiple versions of a class to be supported simultaneously. A per-class meta-object can be associated with the old version of the class. At this point, whenever one of the objects of the old class is accessed, the method invocation is reflected to the meta-object. The code associated with the meta-object can perform a procedure to convert the old object's data to conform to the new class format and then invoke the intercepted method against the newly constructed object. Since the newly created object is of a different class version, the meta-object will not intercept future method invocations against the object, so no future overhead is incurred. The net effect is to permit objects to be updated on demand[6].

Capabilities that sound similar to reflection have started to appear in other middleware technologies (like Microsoft's DCOM). It is worth noting that true reflection requires a self-describing environment. This necessitates the ability to examine all aspects of a method invocation (method name, destination object, method arguments, from object, etc.) as well as to perform arbitrary operations based on the intercepted method invocation.

Dynamically Loaded Code

FARGOS/VISTA-based classes can be dynamically loaded into the environment. They can be loaded from the local file system or easily sent from another host (one of the benefits of a transparently distributed environment). FARGOS/VISTA supports a dynamically loaded architecture-neutral object code format. It also allows the dynamic loading of native object-code in addition to statically linked code for environments that are sensitive to maximizing performance.

Intrinsic Support for Internationalization

In this era of globalization, internationalization of applications is an issue for both multi-national corporations and even small software development organizations. Internationalization of an application is typically performed by externalizing all of the language-specific messages and providing a message catalog for each of the languages that are supported. At run time, the application retrieves relevant message text from an appropriate catalog based on the locale in which the application runs. This works reasonably well for applications that are not distributed. It becomes a little more difficult when dealing with a distributed system. Massive distributed systems of the order supported by FARGOS/VISTA can be deployed in such a fashion as to span multiple countries. It is entirely possible for a user in the United States to make use of results produced by servers in Italy or Germany. The English-speaking user needs his messages in English, even though the servers executing portions of his application were started under Italian and German locales. Trivial distributed systems ignore this problem by assuming that the user and the servers he utilizes share identical locales.

FARGOS/VISTA addresses the problem by treating an internationalized message to be a special data type with the same importance as an integer, floating point or string. This does make it very easy for programmers to provide native language support in their applications. More importantly, the ultimate result is to allow a server, say sitting in Milan, to simultaneously provide results to users in the United States, France, and Germany in their respective languages.

---

[6] The actual procedure would go something like the following: 1) the data is extracted from the old object. 2) a new object of the class version is created using the extracted data. 3) the old object is deleted. 4) the newly created object is renamed to have the same object Id of the deleted object.

Not only do FARGOS/VISTA Native Language Messages assist individuals that speak different languages, it provides the capability for application programs to understand these same messages. This is very useful for systems management-related applications that need to recognize the meaning of selected messages and automatically take appropriate actions.

The Power of a Peer-to-Peer Architecture

As noted earlier, the prevalent distributed paradigm is that of the remote procedure call, and many programmers are familiar with the resulting client/server architecture. Some middleware packages also include support for events, which conceptually can be used in special cases to achieve a level of functionality equivalent to the method invocation style used in FARGOS/VISTA. Given these common capabilities, when presented with the transparently distributed, peer-to-peer architecture created by an FARGOS/VISTA-based system, it is natural to attempt to map its features into familiar concepts. While this can assist in understanding, it can cause the opportunity to create new types of applications to be ignored.

As an illustration of the inherent power of transparently distributed, peer-to-peer architecture, consider the following application example.

Fault-tolerant Web Server

One advantage of a transparently distributed, peer-to-peer architecture is that mobility is naturally supported. An external application can change its connection point to the distributed system, but still interact with objects that previously were local but now are remote. Likewise, an FARGOS/VISTA-based application can move among servers, perhaps to escape a server that is to be brought down for maintenance or move closer to the data it is manipulating.

This can be exploited in numerous ways, but one example is to provide fault-tolerant service for simple applications. As an example, consider an external application interfacing with an FARGOS/VISTA-based infrastructure. If the host running the FARGOS/VISTA process fails, the external application can connect to an alternate FARGOS/VISTA process and continue operation. Of course, objects hosted by the failed server would be inaccessible unless they were replicated, so replication of some form is important for applications intending to be fault-tolerant.

As an illustration, consider a FARGOS/VISTA-based implementation of a fault-tolerant web server (see the *FARGOS/SolidState HTTP Server Adapter User's Guide* for details). Many, but not all, implementations of web-based "shopping carts" can tolerate the crash of a user's web browser. One way this is done is by storing a cookie on the user's machine and subsequently retrieving it to obtain information about the user's pending transaction. The cookie thus maintains on the user's machine the small amount of state that is needed to reconnect the user to the transaction in progress. As anyone who has had a browser crash in the middle of making a purchase from a web site can attest, this can save a lot of frustration on the part of a purchaser. However, it does not help the user when the vendor's web server or link to the Internet fails.

Ideally, a vendor serious about non-stop operation has multiple servers at physically distinct locations. Failure of a given web server would normally cause the loss of all transactions that it had in progress, but with an FARGOS/VISTA-based infrastructure supporting the backend, a failed request from the user could be reissued against an operational server[7].

[7] Accessing an alternate operational server is a non-trivial issue. HTML forms send their data to the URL specified in the declaration of the form. Resubmitting the form will work

Consequently, instead of a major service outage causing a user's entire purchase to be lost, at most the user is inconvenienced by having to retype a few fields on a form.

Further Reading

Further details regarding the FARGOS/VISTA-affiliated suite of technologies can be found in the following resources:

FARGOS/VISTA Installation Guide

FARGOS/VISTA Object Management Environment Programmer's Reference

FARGOS/VISTA Object Management Environment Classes

Object Implementation Language 2 Reference

An Introduction to Programming using OIL2

FARGOS/VISTA HTTP Server Programmer's Guide

FARGOS/VISTA Examples

FARGOS/SolidState HTTP Server Adapter

---

with sites that use a network-level redirector, assuming the redirector remains operational. As a fallback, the HTML page can include links to alternate sites: a successful connect to an alternate site would result in a receipt of a blank form.