Byzantine Fault-Tolerant HTTP Services using FARGOS/VISTA

Geoffrey C. Carpenter
FARGOS Development, LLC, 757 Delano Road, Yorktown Heights, NY 10598 USA
(email: gcc@fargos.net)

Abstract

The FARGOS/VISTA™ suite of technologies implements an infrastructure for the development, deployment and non-stop operation of transparently distributed, multi-threaded, architecture-neutral, object-oriented peer-to-peer applications. These capabilities can be applied in a variety of paradigms, ranging from simple client/server applications to more sophisticated applications that are dynamically loaded into to a pool of cooperating host systems, as well as fault-tolerant systems that eliminate single points of failure. With little effort, these capabilities can be exploited to provide conventional applications with unprecedented reliability. This is illustrated by the implementation of Byzantine fault-tolerant transaction monitor for HTTP-based applications.

1. Introduction

FARGOS/VISTA™[1] is a suite of technologies that provide a transparently distributed, high-performance, multi-threaded, architecture-neutral, object-oriented, peer-to-peer environment that runs on a variety of operating systems and hardware platforms. FARGOS/VISTA benefits from years of experience with DRAGONS[2], originally developed by IBM Research in 1990 for the management of proposed the National Research and Education Network and ultimately deployed in the National Science Foundation Network [1] and as the system monitor for the IBM 9076 SP1 scalable parallel processor.

The FARGOS/VISTA Object Management Environment shares many characteristics with its predecessor, the DRAGONS Data Engine. A partial list includes the following:

- Every object (and thread) is accessible from any peer, although the invocation of a method may be denied to a lack of needed authorization. This should be contrasted against environments that only make available a subset of specially prepared objects accessible to remote applications. Such environments require programmers to differentiate between objects that will be of interest only in the local address space and those that are of interest externally.
- All interactions between objects are performed by sending messages, which cause an appropriate method to be invoked on the target object. Because all interactions are performed via the sending of a message, FARGOS/VISTA-based applications make no distinction between local and remote objects.
- Every method invocation is handled by a separate thread of execution. The FARGOS/VISTA Object Management Environment permits the use of native kernel threads, but most applications rely on the high-performance threading facility provided by the environment as it achieves thread creation rates of over 25,000 per second on a 700 MHz Intel Pentium III processor. The development of an event-driven application becomes trivial and symmetric multiprocessing hardware can automatically be exploited with no special investment on the part of the application programmer.
- The environment is self-describing and any primitive type and object can be queried for its type at runtime.
- Class implementations in the form of architecture-neutral object code can be dynamically loaded.
- Code reuse is enhanced by support for polymorphism, allomorphism and reflection [2]. Reflection also enables object-code-only applications to be modified. Allomorphism, in which a class only has to provide methods similar to that of another class, rather than inherit from a common base class, helps overcome a significant impediment to code reuse [3].

FARGOS/VISTA also provides capabilities not found in its predecessor, including access control as fine as per-object/per user/per-method and copy-on-write semantics for complex intrinsic types like strings, sets, sparse and associative arrays.

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

[2] DRAGONS is an acronym for Distributed Reliable Architecture Governing Over Networks and Systems.

2. The FARGOS/VISTA Object Model

The FARGOS/VISTA object model is intended to be simple to understand, consistent and yet powerful. Consider the following rules that pertain to the definition of classes:

- A class defines both the variables that represent the state of a particular object (these are called instance variables) and the operations that can be performed against objects of a particular class (these operations are called methods).
- A class is uniquely named by three elements: a name space, the class name and a version Id. The FARGOS/VISTA Object Management Environment supports the simultaneous use of multiple versions of a class. This helps eliminate the requirement to simultaneously upgrade all peer systems or existing persistent objects and it is an important element required for non-stop operation.
- A class can inherit from one or more classes (i.e., multiple inheritance is supported).
- A class must inherit from the base class **Object** [4]. This can either be explicitly stated or implied as a property of inheriting from another base class that in turn eventually inherits from the class **Object**.
- Every class must have both a **create** and a **delete** method. While most classes have more methods, it is entirely possible to have a useful class that only implements these two methods.

The rules above pertain to the static nature of class definitions. The FARGOS/VISTA object model also includes operational aspects that are unconventional.

- Every FARGOS/VISTA object and thread is identified by a globally unique identifier, which is automatically generated at the time the object or thread is created. This unique identifier is referred to as an *object Id*.
- Two objects or threads can only interact through the sending of a message. This restriction is made visible in Object Implementation Language 2 (OIL2): it does not permit the use of pointers and thus the direct manipulation of another object's instance variables [5]. In OIL2, a message is sent using the **send** statement.
- In general, when a message is received for an object, the indicated method is executed. This process is called a *method invocation*. The indicated method may have a null body, which means that no code is to be executed. The **delete** method of many classes has this characteristic.
- If an object's method body is not null, then its execution is performed by a separate thread. This means that the runtime environment of FARGOS/VISTA objects is one in which parallelism is supported at a fine level of granularity, namely that of a method invocation. In contrast to conventional programming models, this means that FARGOS/VISTA-based applications are composed of collections of active objects.
- By default, only one method can be active on an object at a time. This restriction enforces safe behavior by default and prevents race conditions, a common issue in multi-threaded environments. Except in very complex cases, programmers need take no action to disable the default behavior, but this capability is available.
- If more than one method is active against an object, the other method cannot proceed until the currently active method is suspended. Again, the default is to enforce safe behavior and prevent race conditions, but this can be overridden by setting a thread as preemptable.

FARGOS/VISTA has been used to build a variety of applications, including several that use World Wide Web browsers as a Graphic User Interface. Examples include www.alecbaldwin.com and an integrated document management system for www.dmllc.com.

3. HTTP-accessible Services

As part of its standard distribution, the FARGOS/VISTA Object Management Environment includes an HTTP 1.1 [6] server that 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 locally accessible file systems. As part of its fundamental feature set, the HTTP server also implements an automatic cache of previously referenced documents. The HTTP server also supports server-side-include directives, which enable HTML to be generated at the time of a query based on templates and environment variables.

The HTTP server is implemented as a collection of objects that interact within the transparently distributed environment. A low-level discussion of the object interactions can be found in [7], but a high level description is useful as an aid to understanding the forthcoming discussion of adding Byzantine fault-tolerance. Incoming connections for a HTTP server are accepted by an **HTTPdaemon** [8] object, which in turn creates a distinct **HTTPfastReceive** object to handle all input and output for a given connection. The **HTTPdaemon** object also creates a single **URLdirectory** [9] for the web site whose primary responsibility is to map the name of a requested Uniform

Resource Identifier (URI) [10], such as "*/index.html*", to an object Id. If the object indicated by the URI is not already registered with the **URLdirectory**, a request to retrieve the URI is made of the **URLfileLoader** (or allomorphic equivalent) [11] object that is responsible for the corresponding section of the HTTP server's naming tree. Ultimately, either an object is located (or created as a result of the query) and further processing of the HTTP client's request is handed off or else a "Not Found" error is returned.

Multiple, independent web sites can reside within the same address space merely by creating an appropriate **HTTPdaemon** object for each web site. Since the FARGOS/VISTA Object Management Environment provides a transparently distributed environment, services can be provided by remote hosts without either the HTTP server or the service provider applications being aware of the fact that they are not residing within the same address space or same physical host. This ability can be exploited for a variety of purposes, including load-balancing or secure access to services made available by hosts that are not reachable from the public Internet. For example, a server farm could handle a sudden increase in load for a given customer site by creating appropriate **HTTPdaemon** objects on some elements of the farm that were not heavily utilized. Source documents, templates and images would be automatically cached at their first reference and server-side-including processing would be performed by the new members of the web site's server pool. The discussion below, however, focuses on the implementation of Byzantine-fault-tolerant services.

4. Byzantine Fault Tolerant Transactions

A fault-tolerant system is able to handle the occurrence of a fault without causing the system to stop or produce an incorrect result. 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 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. 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.

The ability to handle arbitrary (including malicious) faults is termed Byzantine-fault-tolerance [12]. 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.

In 1999, theoretical work by Miguel Castro and Barbara Liskov [13] was applied to create a FARGOS/VISTA-based Byzantine fault-tolerant transaction monitor known as FARGOS/SolidState. The FARGOS/SolidState application classes were subsequently interfaced to the FARGOS/VISTA-based HTTP server [14].

5. FARGOS/SolidState HTTP Adapter

To implement a Byzantine fault-tolerant service, at least 4 machines are required. The actual formula is:

machines = 3 *simultaneousFaults + 1

The very simplified flow for a transaction is described below:

- The client application submits a transaction to a coordinator who will handle the processing of the request and ultimately return a response. The interface exposed to the client application is that of the coordinator—the client application is unaware of the Byzantine fault-tolerant algorithms being executed by the coordinator.
- The coordinator forwards the request to the pool of servers participating in the transaction.
- Each server performs the transaction on its own and sends back the computed result to the coordinator.
- The coordinator collects all of the responses and verifies them for correctness. If a correct result can be obtained, this is provided to the requesting client application; otherwise, the client application is informed that an error has occurred.

Given the description above, in the simplest case there are six individual components: the client application, the coordinator and the four members of the server pool. The algorithm for Byzantine fault-tolerance handles failures associated with the servers; however, the client application and the coordinator are single points of failure that are not tolerated. Consequently, the optimal deployment would be to collocate the logic for the coordinator with the client application, thus eliminating the coordinator as a distinct single point of failure and tying the fate of the client and the coordinator together. Collocation is often feasible if the coordinator logic is provided as a library against which a client application must link.

Unfortunately, when dealing with HTTP-based applications, the client application issuing a request will usually be a web browser or similar application. Except

under special circumstances, such as a mandate from a corporate Information Services department, a service provider programmer cannot assume the ability to deploy and collocate the necessary coordinator logic with an arbitrary end user's client application. Note, however, that simply separating the coordinator logic from the client application introduces a single point of failure and, therefore, it can be argued that this would defeat the purpose of attempting to provide Byzantine fault-tolerant services.

The FARGOS/SolidState HTTP adapter uses the capabilities of the FARGOS/VISTA Object Management Environment to solve this problem. When a new Byzantine fault-tolerant session is created, three objects are created on each machine within the server pool:

- An object that will perform the actual application-specific services on behalf of the client. This is often referred to as the state variable.
- A Byzantine fault-tolerant **ReplicaServer** object, which acts as the transaction monitor for the application-specific services provided by the state variable object above.
- A proxy object for the client application that implements the coordinator logic. It receives a request from the client application and interacts with the pool of **ReplicaServer** objects to initiate a transaction, verify the results and return a response back to the client application. Note that a coordinator object is created on each server, so the potential flaw of the coordinator representing a single point of failure has been sidestepped.

Each client proxy object registers itself with the local **URLdirectory** object associated with the web site. When the **HTTPdaemon** receives any HTTP queries that refer to services provided by the Byzantine fault-tolerant service provider, the HTTP server hands the request off to the client proxy object for processing. The client proxy object issues the request to all of the **ReplicaServer** objects and waits to collect and validate the responses. Each individual **ReplicaServer** object invokes the request against its local application-specific service provider object, obtains the result of the work and sends it back to the requesting client proxy object. The requesting client proxy object formats the result as an appropriate response to be shipped back via HTTP (typically, it is a server-side-include processed HTML file).

6. Identifying Replicated Services

Previously, it was noted that every FARGOS/VISTA object is automatically assigned a unique object Id. While it is often necessary to be able to address each distinct object, this same ability poses a significant problem when dealing with replicated objects. Normally, a client application wants to use the local provider of a replicated service. Since each object has a unique Id, applications utilizing replicated services would either have to be aware of the Id of each of the peer objects and determine the local object (which violates transparency) or risk having a method invocation transparently forwarded between hosts. DRAGONS solved this problem by either the duplication and subsequent importation of an object into a remote system or by permitting an object to rename itself to a specified object Id. Either mechanism enabled objects in distinct address spaces to be assigned identical Ids. It also made it impossible to subsequently address a particular replicated object since the unique name was lost. For various reasons, which are not elaborated here, FARGOS/VISTA does not permit an object to change its object Id; however, it does permit the duplication and importation of an object and more than one object Id can refer to a given object (object Ids are actually a many-to-one mapping). The preferred FARGOS/VISTA solution is to permit objects to register themselves as named services. Anywhere an object Id can be used as the target of a method invocation, the name of a service can be used as well. This solution allows replicated objects to be uniquely identified whenever required (such as when they issue an RPC-style method invocation to a potentially remote object) and as a locally resolved service name when a local implementation is desired.

In the context of the Byzantine fault-tolerant services discussed above, both the client proxy object and the state variable are identified using service names rather than object Ids. While a state variable is never directly exposed to the client application, assigning it a service name eliminates the need for a client proxy object to be informed about the object Ids of the state variables that were created by the **ReplicaServer** objects on each of the participating servers. The **ReplicaServer** transaction monitors are able to interact with multiple state variables, so the name of the relevant state variable is passed as an argument, as illustrated below:

```
result = send "issueRequest"("processRequest",
argList, stateVariable) to replicaProxy;
```

Since the request issued to each **ReplicaServer** is identical, it becomes feasible to use multicast vs. individual method invocations to each object.

While not synchronized, the proxy client objects are created on each server to eliminate the single point of failure that would be introduced by a single coordinator and thus implement a replicated service. Consequently, the client proxy objects are identified to client applications using a service name rather than their individual object Ids.

7. Reconnecting a Stuck Client

As noted earlier, it would be ideal to locate the coordination logic with the client application. Unfortunately, this is not possible if one is attempting to provide an HTTP-based service accessible to any potential client, including a individual using **telnet** and entering the HTTP command and headers by hand. Normally, an HTTP client will resolve an address and connect to a specific HTTP server. Since the proxy client objects are replicated across all of the participating servers, it does not matter to which server the client application successfully connects. Unfortunately, the client application may attempt to connect to a failed server or establish a connection to a host that subsequently fails. An additional mechanism must be deployed to enable a client that is unaware of the population of the server pool to get unstuck from the failed server. Several techniques are possible, ranging from crude software tricks like round-robin DNS to load-balancing front-ends, either in software like Narwhal or any one of a variety of hardware-oriented product offerings [15].

Since the coordinator logic is not collocated with the client application, recovery must be manually initiated by the user. In practice, this means the user becomes impatient waiting for a response and clicks on the *submit* button a second time. The second attempt should cause a new HTTP connection to be initiated and redirected to one of the operational members of the server pool. If the transaction had been successfully requested prior to the failure of the original server acting as coordinator, the cached result can be returned immediately; otherwise, the transaction request will be finally issued.

8. Conclusions

The FARGOS/VISTA suite of technologies provides a powerful, flexible and productive environment for the construction and deployment of robust distributed applications that work across a variety of machine architectures and operating systems. By creating a transparently distributed environment in which every object is accessible from any interconnected host, FARGOS/VISTA enables the construction of sophisticated services, such as Byzantine fault-tolerant transactions, even for clients that are strictly client/server-oriented and not under the control of the service provider. Because every method invocation is performed by a separate thread of execution, applications are naturally event-driven and opportunities for parallel execution can be exploited on SMP-capable hardware or physically distinct hosts without additional effort on the part of the application developer, increasing the scalability of the resulting application.

The FARGOS/VISTA Software Development Kit and a demonstration of a web site using a Byzantine fault-tolerant shopping cart are available for downloading from http://www.fargos.net/downloads.html.

9. References

[1] R. Lehman, G. Carpenter and N. Hien, "Concurrent Network Management with a Distributed Management Tool", *USENIX LISA VI conference proceedings*, pp. 235-244, 1992.

[2] J. Ferber, "Computational reflection in class based object-oriented languages", *OOPSLA 1989 conference proceedings*, pp. 317-326, 1989.

[3] S. Yu, "Class-is-type is inadequate for object reuse", *ACM SIGPLAN Notices*, vol. 36 no. 6, pp. 50-59, 2001.

[4] "OIL2 Class Standard.Object", http://ww.fargos.net/classDoc/Standard/oil2Object-0.html.

[5] *FARGOS/VISTA Object Implementation Language 2 Reference Manual*, http://www.fargos.net/documents/manuals/OIL2reference.pdf.

[6] R. Fielding, et. al, "Hypertext Transfer Protocol HTTP/1.1", *RFC 2616*, 1999

[7] *FARGOS/VISTA HTTP Server Programmer's Guide*, http://www.fargos.net/documents/manuals/HTTPguide.pdf

[8] "OIL2 Class Standard.HTTPdaemon", http://www.fargos.net/classDoc/Standard/oil2HTTPdaemon-0.html.

[9] "OIL2 Class Standard.URLdirectory", http://www.fargos.net/classDoc/Standard/oil2URLdirectory-0.html.

[10] "Uniform Resource Identifiers (URI): Generic Syntax", *RFC 2396*, 1998.

[11] "OIL2 Class Standard.URLfileLoader", http://www.fargos.net/classDoc/Standard/oil2URLfileLoader-0.html

[12] L. Lamport, R. Shostak and M. Pease, "The Byzantine Generals Problem", *ACM Transactions on Programming Languages and Systems*, vol 4 issue 3, pp. 382-401, 1982.

[13] M. Castro and B. Liskov, "A Correctness Proof for a Practical Byzantine-Fault-Tolerant Replication Algorithm", Technical Memo MIT/LCS/TM-590, 1999.

[14] *FARGOS/SolidState HTTP Server Adapter User's Guide*, http://www.fargos.net/documents/manuals/SolidStateWeb.pdf

[15] G. Carpenter and G. Goldzsmidt, "Improving the Availability and Performance of Network Mediated Services", *INET'99 conference proceedings*, http://www.isoc.org/inet99/proceedings/4g/4g_3.htm.