Version: 4/12/2002 9:49 PM

FARGOS/VISTA

Examples

FARGOS/VISTA Examples

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

Notice of Rights

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. Programming FARGOS/VISTA-based Applications

FARGOS/VISTA is the name given to a suite of technologies that enable the development of architecture-neutral, transparently distributed applications. While FARGOS/VISTA itself is a young technology, it is benefactor of a decade worth of research, development and implementation experience in the following areas:

- Transparently-distributed, peer-to-peer applications
- Active objects
- High-performance multi-threading
- Architecture-neutral applications
- Persistent objects
- Fault-tolerance
- Non-stop operation with updates applied to running systems
- Reusable components
- Loosely-coordinated development efforts
- Self-describing environments
- Reflection
- Rapid development
- Code maintenance

Despite the plethora of programming technologies available today, there are a few reasons why FARGOS/VISTA should be in the repertoire of any programmer:

- It is easy to learn: much of its power comes from the deceptively simple object model.
- It is productive: while every development effort is unique, 6-to-10-fold improvements in productivity were the norm with the predecessor technology.
- A FARGOS/VISTA-based application runs on any supported platform: if you develop under Sun Solaris, your FARGOS/VISTA application runs under Microsoft Windows or Linux, OpenBSD, etc.
- It enables the development of sophisticated, fault-tolerant distributed applications that would otherwise be prohibitively expensive to develop.

The examples that follow are intended to assist programmers who are experimenting with FARGOS/VISTA as well as peak the interest of those who are learning of its existence for the first time.

FARGOS/VISTA Software Development Kit

Almost all FARGOS/VISTA-affiliated applications run within the FARGOS/VISTA Object Management Environment, which is sometimes abbreviated as OME. The FARGOS/VISTA Object Management Environment can be thought of as a transparently distributed object-oriented operating system. It runs on top of a computer's native operating system. For example, a Sun SPARC computer will normally be running Sun's Solaris Operating Environment and many Intel Pentium-based machines run some variant of Microsoft Windows. Reader's familiar with IBM's VM operating system for mainframes can view the FARGOS/VISTA Object Management Environment as a guest operating system. Details regarding the model of operation and standard facilities are provided in the *FARGOS/VISTA Object Management Environment Programmer's Guide*.

Applications that run within the FARGOS/VISTA Object Management Environment are implemented as objects that are instances of classes, which describe all of the necessary application logic. These classes must be compiled from source code into some form of object code. The source code language is selected by the application

programmer. The preferred choice for FARGOS/VISTA applications is Object Implementation Language 2, but a language such as C++ is a viable alternative.

The choice of source language will impact the form of object code that can be generated. If a language other than OIL2 is used, the application developer will be restricted to using native object code. (e.g., Sun SPARC or Intel x86 instructions), which is generated by a C++ compiler, such as Microsoft's Visual C++ compiler (for Microsoft Windows) or the GNU C++ compiler (available for all supported Unix distributions). If OIL2 is used, then not only can native object code be generated but also an architecture-neutral format (OIL2 ANF) as well as documentation. Native object code can be statically linked into a custom executable or dynamically loaded into a running FARGOS/VISTA Object Management Environment process (e.g., using the **LoadObjectFile** class). Architecture-neutral code is always dynamically loaded and the standard mechanism is to use the **LoadOIL2File** class.

Regardless of the choice of implementation language, developers must install the FARGOS/VISTA Software Development Kit and configure their environment appropriately. Instructions are provided in the *FARGOS/VISTA Installation Guide*.

Getting Started

Consider an example that will serve to verify that the FARGOS/VISTA Software Development Kit has been properly installed. The following performs the classic "hello world" example:

```
%include "OMEcore.o2h"

class HelloWorld {
} inherits from Object;

HelloWorld:create()
{
      display("hello world!\n");
}

HelloWorld:delete() {}
```

Compile:

```
oil2_parse –oil2 hello.oil
```

Using the following *rc* file:

```
LoadOIL2File "hello.o2o"
HelloWorld
```

Test by issuing:

```
vista hello.rc
```

Something very similar to the following should appear:

```
FARGOS/VISTA Object Management Environment
Copyright © 1999 – 2002 FARGOS Development, LLC.  All rights reserved.
hello world!
```

Compared to the equivalent C/C++ program, the amount of effort involved is slightly greater (5 extra text lines and an *rc* file):

- The application was implemented by a class, which is not mandated in C (or C++), so the declaration of the **HelloWorld** class was required. Good programming style required the implementation of the **delete** method, even though the object was not deleted in the example.

- An *rc* file had to be prepared to load the object code of the class and create the object of interest. With a conventional application, the executable would have been invoked directly.

When the application at hand is slightly more involved, the cost of a conventional approach quickly exceeds that of a FARGOS/VISTA-based application.

Creating an Object

Everything within a FARGOS/VISTA Object Management Environment is represented as an object. Consequently, one of the more useful activities that an application can perform is the creation of an object. Objects are created by sending an appropriate message to the **ObjectCreator** object. The method most often used is **createObject**. The **createObject** method takes at least two arguments:

- The name of the object's class
- An Access Control List

An object is deleted by sending it a **deleteYourself** message.

Access Control Lists

Security is an integral part of the FARGOS/VISTA Object Management Environment. The predecessor to FARGOS/VISTA[1] was developed for and deployed into environments that assumed a great deal of trust. Consequently, access was granted on an all-or-nothing basis. In contrast, FARGOS/VISTA was designed to support a wide variety of trust. Ultimately, security involves the denial of access to unauthorized parties. In the FARGOS/VISTA object model, every object is protected by at least one access control list that determines who is permitted to invoke methods on that object. The granularity of access can be as fine as per-object/per-user/per-method. In general, most applications create objects using a default access control list that permits the creator of the object full access and denies access to all others. These default access control lists are conveniently created using the **makeDefaultACL()** function.

Object Ids

All objects are identified by and referred to using an object Id (**oid**). An object Id is an opaque handle and is somewhat analogous to a pointer in conventional programming environments. There are, however, significant differences. A FARGOS/VISTA **oid** is globally unique, thus no two objects have the same **oid**. Unlike many object-oriented systems, a FARGOS/VISTA object can be referenced using more than one **oid**. Stated formally, a FARGOS/VISTA **oid** provides a many-to-one mapping to a physical object.

Because all inter-object interactions are performed by sending messages and all objects are referenced using object Ids, all inter-object activity takes places without regard to physical location. It is this characteristic that forms the basis of the transparently distributed nature of the FARGOS/VISTA infrastructure. It should be noted, however, that anywhere an object Id can be used as the target of a method invocation, it is possible to use the name of a well-known service.

The example below illustrates the creation of several objects and their deletion.

[1] Distributed Reliable Architecture Governing Over Networks & Systems (DRAGONS).

```
%include "OMEcore.o2h"

class HelloWorld2 {
} inherits from Object;

HelloWorld2:create()
{
      int   j;
      assoc acl;
      oid   obj;

      for (j=1;j<argc;j+=1) {
            acl = makeDefaultACL();
            obj = send "createObject"("HelloWorld2", acl, argv[j])
                  to ObjectCreator;
      }
      if (argc >= 1) {
            display("obj=", thisObject, "argv[0]=", argv[0],"\n");
      } else {
            display("no arguments\n");
      }
      send "deleteYourself" to thisObject;
}

HelloWorld2:delete() {}
```

Doing Several Things at Once

One of the novel characteristics of the FARGOS/VISTA Object Management Environment is that every method invocation is handled by a separate thread of execution. This is in stark contrast to a conventional programming environment, which has a single thread of execution that performs the work of an entire application. Applications running in a FARGOS/VISTA Object Management Environment have separate threads of execution performing items of work that roughly correspond to function calls in a conventional environment. Most of the time, programmers can be oblivious to the fact that a separate thread of execution performs every method invocation. It does, however, provide a powerful environment that makes the creation of asynchronous, event-driven applications trivial to implement. Likewise, the automatic exploitation of symmetric multiprocessor (SMP) hardware can be done with no effort on the part of the application writer.

Well-Known Services

Every object that is created inside a FARGOS/VISTA Object Management Environment is assigned a globally unique object Id. Since a message cannot be sent to an object unless its object Id is known, objects that provide well-known services must make their existence known. This is done using the **registerLocalService()** function provided by the FARGOS/VISTA Object Management Environment core. An application can locate a service by using the **lookupLocalService()** function. Alternatively, a message can just be sent to the named service. Thus the following:

```
oid   obj;
obj = lookupLocalService("HTTPlogger");
send "reopen" to obj;
```

is equivalent to:

```
send "reopen" to "HTTPlogger"
```

Well-known service names have additional uses that may not be immediately obvious. Persistent objects can benefit from not having to alter their instance data when restored into a new process image—although the object Id of a service will have changed, its well-known name remains the same. An example of this usage is provided by the Email Mailbox Storage example presented later in this document. Replicated services should always make a service name the definitive means of addressing the service: this permits requests to be handed off to any of the duplicate providers rather than being pinned to a particular instance[2].

A well-known service is un-registered using the **unregisterService()** function.

Debugging using a Web Browser

The standard FARGOS/VISTA Object Management Core includes an HTTP server and an integrated object browser application. The use of the HTTP server is described in some detail in the *FARGOS/VISTA HTTP Server Programmer's Guide*.

An object browser session is implemented by the class **HTTPobjectBrowser**. The actual display of an object's contents is performed by the class **HTTPdisplayObject**, objects of which are created as needed by the session's **HTTPobjectBrowser** object. Neither of these classes is normally used directly. Instead, a new browsing session is created by the helper class **HTTPcreateObjectBrowserSession**. The normal usage is demonstrated in the **vista** *rc* file below:

```
AcceptPeerConnections tcp:127.0.0.1:8765
HTTPcommonLogFormat
HTTPdaemon $VISTA_ROOT/config/localHTTP.profile tcp:0.0.0.0:4321
HTTPpurgeCache 50 localhost
HTTPcreateObjectBrowserSession localhost /debug/newBrowser
```

By browsing the URL http://localhost:4321/debug/newBrowser, developers will be presented with a list of all well-known services. Clicking on a link will cause the object's contents to be displayed.

Timers

Occasionally, an application may want to perform tasks periodically. One terrible way to achieve this would be to constantly monitor the current time until enough time had passed. The wasteful consumption of CPU cycles makes such an approach infeasible. The FARGOS/VISA Object Management Environment provides two mechanisms for waiting until a certain amount of time has passed. The approach most commonly used by application programmers is to create a **TimerEvent** object. An application can create as many **TimerEvent** objects as needed. When a timer goes off, the client object is sent a **timerExpired** message and passed any extra data that was previously associated with the timer.

As an alternative, the **sleepForSecond()** call can be used to put the application's thread to sleep for the requested number of seconds. This approach is convenient when the delay is required in the middle of a looping construct.

Input Buffers and Asynchronous Input/Output

The FARGOS/VISTA Object Management Environment provides access to almost all input/output-related facilities using the class **IOobject**. Rather than provide distinct

[2] A discussion of issues related to replication is provided in the paper "Byzantine Fault-Tolerant HTTP Services using FARGOS/VISTA".

classes for file access, TCP connections, UDP ports, etc., an **IOobject** is able to deal with a wide variety of devices that are distinguished using transport schemes. While the total set of recognized transport schemes varies from platform to platform, the standard schemes always include:

- "file:" for files accessible via the local file system
- "tcp:" for TCP connections using IP version 4 (same as "tcp4:")
- "udp:" for UDP ports using IP version 4 (same as "udp4:")

Other transport standard schemes recognized by many FARGOS/VISTA Object Management Environments include:

- "unix:" for Unix file domain sockets
- "raw:" for raw IP version 4 sockets (same as "raw4:")
- "tcp6:" for TCP connections using IP version 6
- "udp6:" for UDP ports using IP version 6
- "raw6:" for raw IP version 6 sockets
- "ipx:" for IPX ports
- "spx:" for SPX connections

The actual argument passed to the **create** method is structured like a Uniform Resource Identifier, as specified in RFC 2396. The FARGOS/VISTA Object Management Environment can be locally extended to support additional schemes by using the C++ function **OMEregisterIOscheme()**.

Load Balancing Front-End for TCP-based Services

One of the very powerful features of the FARGOS/VISTA programming model is the use of multiple threads of execution for every method invocation. Among other things, this makes it trivial to handle asynchronous events, such as the arrival of data.

The example source code provided below implements a load balancing front-end to an arbitrary TCP-based service, such as an HTTP server. An ultimate high-performance solution would utilize dedicated routers that modify the in-bound packets; the application-level load balancing "solution" illustrated here is intended to demonstrate the simplicity of the resulting application that is good enough for many environments.

The code below is complete "as-is". Note that it is able to deal with an arbitrary number of servers and simultaneous clients and it will tolerate the failure of a server.

```
class Local . ForwardConnection {
/*!
This is an experimental class that forwards TCP connections in a round-robin
fashion to to a pool of servers.
!*/
   array destAddrList;
   int   destinationCount;
   int   nextDestIndex;
   string      listenAddress;
   oid   listenObj;
} inherits from Object;

ForwardConnection:create(string listenAddr, string connAddr1, string connAddr2)
{
/*!
The <b>create</b> method takes a minimum of two arguments.  The first argument
specifies the transport address at which a listen port should be created.
The second and all remaining arguments indicate transport addresses to which
new connections should be forwarded.
!*/
   assoc acl, connACL;
   int   i;

   listenAddress = listenAddr;
   destinationCount = 0;
   for(i=1;i<argc;i+=1) {
         destAddrList[destinationCount] = argv[i];
         destinationCount += 1;
   }
   acl = makeDefaultACL();
   connACL = makeDefaultACL();
   listenObj = send "createObject"("AcceptConnection", acl,
         listenAddress, thisObject, connACL) to ObjectCreator;
}

ForwardConnection:delete()
{
   if (listenObj != nil) send "deleteYourself" to listenObj;
}

ForwardConnection:connectionAccepted(oid source)
{
   oid   fwdObj;
   assoc acl;

   acl = makeDefaultACL();
display("connect via ", destAddrList[nextDestIndex], " index=",
nextDestIndex,"\n");
   fwdObj = send "createObject"("ConnectAndForward", acl, thisObject,
         source, destAddrList[nextDestIndex], 0) to ObjectCreator;
   nextDestIndex += 1;
   if (nextDestIndex >= destinationCount) nextDestIndex = 0;
}
```

```
ForwardConnection:failedConnection(string destAddr, oid src, int tries)
{
   string      dest;
   assoc acl;
   oid   fwdObj;

display("Connection attempt to ", destAddr, " failed.\n");
   if (tries >= destinationCount) {
display("Tried all servers, closing source\n");
         send "deleteYourself" to src;
         exit;
   }
   do {
         dest = destAddrList[nextDestIndex];
display("\tnext choice: ", destAddrList[nextDestIndex], " index=",
nextDestIndex,"\n");
         nextDestIndex += 1;
         if (nextDestIndex >= destinationCount) nextDestIndex = 0;
   } while (dest == destAddr);
display("\tWent with ", dest, "\n");
   acl = makeDefaultACL();
   fwdObj = send "createObject"("ConnectAndForward", acl, thisObject,
         src, dest, tries + 1) to ObjectCreator;
}
```

```
class Local . ConnectAndForward {
   oid   forwardController;
   string      destinationAddr;
   oid   srcObj;
   oid   sinkObj;
   oid   connObj;
   int   sidesOpen;
   int   attemptCount;
} inherits from Object;

ConnectAndForward:create(oid fwdCntrl, oid source, string destAddr, int tries)
{
   assoc acl, connACL;

   forwardController = fwdCntrl;
   destinationAddr = destAddr;
   srcObj = source;
   attemptCount = tries;

   acl = makeDefaultACL();
   connACL = makeDefaultACL();

   connObj = send "createObject"("EstablishConnection", acl, destAddr,
         thisObject, connACL) to ObjectCreator;
}

ConnectAndForward:delete()
{
   if (srcObj != nil) send "deleteYourself" to srcObj;
   if (connObj != nil) send "deleteYourself" to connObj;
   if (sinkObj != nil) send "deleteYourself" to sinkObj;
}

ConnectAndForward:connectionAttemptFailed(int rc)
{
   send "failedConnection"(destinationAddr, srcObj, attemptCount)
         to forwardController;
   srcObj = nil;     // we no longer own it...
   send "deleteYourself" to thisObject;
}
```

```
ConnectAndForward:connectionEstablished(oid ioObj)
{
   send "deleteYourself" to connObj;   // its job is done...
   connObj = nil;
   sinkObj = ioObj;
   sidesOpen = 2;
   send "selectForRead"(thisObject) to srcObj;
   send "selectForRead"(thisObject) to sinkObj;
}

ConnectAndForward:canRead(oid obj)
{
   oid   dest;
   any   data;
   int   rc;

   data = send "readBytes"(4096) to obj;
   if (obj == srcObj) {
//display("reading from browser, forwarding to web server\n");
         dest = sinkObj;
   } else {
//display("reading from web server, forwarding to browser\n");
         dest = srcObj;
   }

   if (typeOf(data) == string) { // normal forwarding...
         // TO DO:  make sure all were written...
         send "writeBytes"(data) to dest from nil;
         send "selectForRead"(thisObject) to obj;
         exit; // all done...
   }
   // EOF was detected...
   send "closeForRead" to obj from nil;
   send "closeForWrite" to dest from nil;
   sidesOpen -= 1;
display("EOF on connection to ", destinationAddr, " sidesOpen=", sidesOpen, "\n");
if (obj == srcObj) display("\tcame from source\n");
else display("\tcame from sink\n");
   if (sidesOpen == 0) {
         send "deleteYourself" to thisObject;      // we're done...
   }
}
```

SMTP Server

While file-based I/O usually seems to be performed without noticeable delay, I/O over network connections to a remote host invariably involves delays. Such delays arise from unavoidable network latencies and the occasional retransmission of data due to lost packets. Consequently, an application that desires a certain number of bytes of data cannot assume that its request will be satisfied in full if it reads directly from a stream represented by an **IOobject**. In a similar vein, many applications need to read variable length records from a stream, such as lines of text in a file. Both of these problems are handled by the very useful **ReadBuffer** class. The example below is a complete Simple Mail Transfer Protocol (SMTP, see RFC 2821) server. It integrates with the **SMTPmailboxService** describes in the section entitled “Email Mailbox Storage”. The application is broken into two classes, an **SMTPserver** which creates an **AcceptConnection** object to listen for incoming SMTP connections and an **SMTPclientConnection** class, which handles an individual SMTP session. The **AcceptConnection** class sends its **SMTPserver** client object a **connectionAccepted** message and provides the object Id of an **IOobject** that represents a new SMTP connection.

```
%include <OMEcore.o2h>

global SMTP {
   const string SMTP_SERVER_ID = "FARGOS/VISTA SMTP Server 1.0";
   const int MAX_IDLE_TIME = 300;
};

class Local . SMTPserver {
   assoc configParams;
   oid   listenObj;
   int   connectionsAccepted;
} inherits from Object;

SMTPserver:create(string listenAtAddr, string hostName)
{
   assoc acl, connACL;
   string      smtpHostName, listenAddress;

   if (length(hostName) > 0) {
         smtpHostName = hostName;
   } else {
         smtpHostName = getSystemInfoAttribute("hostName");
   }
   configParams["hostName"] = smtpHostName;

   if (typeOf(listenAtAddr) == string) {
         listenAddress = listenAtAddr;
   } else {
         listenAddress = "tcp:0.0.0.0:25,l";
   }

   connACL = makePermitEveryoneACL();
   acl = makeDefaultACL();
   listenObj = send "createObject"("AcceptConnection", acl,
               listenAddress, thisObject, connACL) to ObjectCreator;

   send "addNotifyOnShutdown"(thisObject) to "ShutdownService";
display("SMTPserver operating on ", listenAddress, "\n");
}

SMTPserver:delete()
{
   if (listenObj != nil) send "deleteYourself" to listenObj;
}

SMTPserver:systemShutdown()
{
/*!
Because it provides a long-running service, class=SMTPserver objects
register themselves with the class=ShutdownService,
which sends a method=systemShutdown notification when a graceful shutdown
is requested.
!*/
   send "deleteYourself" to thisObject;
}
```

```
SMTPserver:connectionAccepted(oid newSocket)
{
/*!
When new HTTP connections are received, the method=connectionAccepted
method creates a class=HTTPfastReceive object to handle the I/O and
processing of any requests.
!*/
   assoc acl;

// becomePseudoUser();
   connectionsAccepted += 1;
   acl = makeDefaultACL();
   send "createObject"("SMTPclientConnection", acl, newSocket,
         configParams)
         to ObjectCreator from nil;    // don't bother with response
}
```

```
class Local . SMTPclientConnection {
   enum ParseStates { HELLO, MAIL, DATA, DONE };
   oid   connObj;
   oid   readBfr;
   oid   mailboxService;
   string      peerAddress;
   string      claimedPeerName;
   int   lastRequestTime;
   assoc serverConfig;
   oid   timerObj;
   int   currentParseState;
   set   recipients;
   set   recipientAddrList;
   set   senders;
   set   mailBody;
   int   sessionClosed;
   oid   shutdownOID;
} inherits from Object;
```

```
SMTPclientConnection:create(oid newSocket, assoc configParams)
{
   assoc acl, currentTime;
   int   t, rc;
   string      date, initialLine;

   connObj = newSocket;
   serverConfig = configParams;
   peerAddress = send "getPeerAddress" to connObj;
display("New STMP connection from ", peerAddress, "\n");

   mailboxService = lookupLocalService("MailboxDirectory");
   if (mailboxService == nil) {
         send "writeBytes"("421 No MailboxDirectory\r\n") to connObj from nil;
         send "deleteYourself" to thisObject;
         exit;
   }

   lastRequestTime = getLocalRelativeTime();
   currentTime = convertLocalRelativeTimeToAbsolute(lastRequestTime, 0);
   date = rfc1123Date(currentTime);

   initialLine = makeAsString("220 ", serverConfig["hostName"],
         " ", SMTP_SERVER_ID, "; local time is ", date, "\r\n");
   rc = send "writeBytes"(initialLine) to connObj;
display("wrote len=", rc, "\n");
   if (rc <= 0) {
display("Could not announce to client:  ", initialLine, "\n");
         send "deleteYourself" to connObj;
         send "deleteYourself" to thisObject;
         exit;
   }

   currentParseState = HELLO;

   acl = makeDefaultACL();
   readBfr = send "createObject"("ReadBuffer", acl, connObj)
         to ObjectCreator;

   // NOTE:  end of line delimeter must be set to CR/LF for correctness.
   // RFC 2821, section 4.1.1.4 requires that a single LF MUST NOT
   // be acccepted, even if it would seem to tolerate incorrect
   // implementations.  This is one case where being tolerant in
   // what one accepts is explicitly disallowed.
   send "setDelimeter"("\r\n") to readBfr;

   send "selectForRead" to readBfr;

   timerObj = send "createObject"("TimerEvent", acl,
         thisObject, MAX_IDLE_TIME) to ObjectCreator;

   shutdownOID = thisObject;
   send "addNotifyOnShutdown"(shutdownOID) to "ShutdownService";
}

SMTPclientConnection:delete()
{
   if (readBfr != nil) send "deleteYourself" to readBfr;
   if (shutdownOID != nil) {
         send "removeNotifyOnShutdown"(shutdownOID) to "ShutdownService";
   }
}
```

```
SMTPclientConnection:timerExpired(oid timerObj)
{
   assoc acl;
   int   t, delta;

   if (sessionClosed == 1) {
display("Session was closed, time to delete\n");
         send "deleteYourself" to thisObject;
         exit;
   }

   t = getLocalRelativeTime();
   delta = t - lastRequestTime;
   if (delta > MAX_IDLE_TIME) {
display("SMTP client connection with ", peerAddress, " timed out\n");
         send "deleteYourself" to thisObject;
         exit;
   }

   acl = makeDefaultACL();
   timerObj = send "createObject"("TimerEvent", acl,
         thisObject, MAX_IDLE_TIME, t) to ObjectCreator;
}
```

```
SMTPclientConnection:canRead(oid bfr)
{
   any   line, tokens;
   string      cmd;
   int   rc;

   lastRequestTime = getLocalRelativeTime();
   line = send "readLine" to readBfr;
display("SMTP client line=", line, "\n");
   if (line == nil) {      // EOF
display("Unexpected EOF from SMTP client ", peerAddress, "\n");
         send "deleteYourself" to readBfr;
         readBfr = nil;
         sessionClosed = 1;
         exit;
   }
   if (currentParseState == DATA) {
display("accepting message body\n");
         if (line != ".") {
               if (length(line) > 0) {
                     if (midchar(line, 0) != '.') mailBody += line;
                     else { // delete initial period
                           mailBody += midstr(line, 1, length(line) - 1);
                     }
               }
               mailBody += "\r\n";
         } else { // process mail message...
               call "_processMail"();
         }
   } else {
         tokens = tokenizeString(line, " \t\r\n", 0);
display("tokenized line=", tokens);
         if (elementCount(tokens) > 0) {
               cmd = convertCase(tokens[0], 0);
         } else {
               cmd = "";   // bogus command
         }
```

```
         if (cmd == "HELO") {
               call "_helo"(tokens);
         } else if (cmd == "EHLO") {
               call "_ehlo"(tokens);
         } else if (cmd == "MAIL") {
               call "_mail"(tokens);
         } else if (cmd == "RCPT") {
               call "_rcpt"(tokens);
         } else if (cmd == "DATA") {
               if (elementCount(tokens) != 1) {
                     send "writeBytes"("501 No argument expected\r\n") to connObj
from nil;
               } else {
                     send "writeBytes"("354 Send mail body; end with
<CR><LF>.<CR><LF>\r\n") to connObj from nil;
               }
               call "_addTraceLine"();
               currentParseState = DATA;
         } else if (cmd == "QUIT") {
               if (elementCount(tokens) != 1) {
                     line = "501 No argument expected\r\n";
               } else {
               line = makeAsString("221 ", serverConfig["hostName"],
                     " is now closing transmission channel\r\n");
               }
               send "writeBytes"(line) to connObj from nil;
               send "deleteYourself" to readBfr;
               readBfr = nil;
               sessionClosed = 1;
               exit;
         } else if (cmd == "RSET") {
               call "_reset"();
               if (elementCount(tokens) != 1) {
                     send "writeBytes"("501 No argument expected\r\n") to connObj
from nil;
               } else {
               send "writeBytes"("250 OK Reset was done\r\n") to connObj from nil;
               }
         } else if (cmd == "VRFY") {
               call "_vrfy"(tokens);
         } else if (cmd == "EXPN") {
               call "_expn"(tokens);
         } else if (cmd == "HELP") {
               call "_help"(tokens);
         } else if (cmd == "NOOP") {
               send "writeBytes"("250 OK Did nothing\r\n") to connObj from nil;
         } else {
               send "writeBytes"("500 Command not implemented\r\n") to connObj from
nil;
         }
   }
   send "selectForRead" to readBfr;
}
```

```
SMTPclientConnection:_reset()
{
   mailBody = emptySet;
   recipients = emptySet;
   recipientAddrList =  emptySet;
   senders = emptySet;
   currentParseState = HELLO;
   return (0);
}

SMTPclientConnection:_addTraceLine()
{
   assoc currentTime;
   string      traceLine;
   int   requestTime;

   requestTime = getLocalRelativeTime();
   currentTime = convertLocalRelativeTimeToAbsolute(lastRequestTime, 0);
   traceLine = makeAsString("Received: from ", claimedPeerName, " ([",
         peerAddress, "])", "\r\n",
         "        ", "by ", serverConfig["hostName"],
               " (", SMTP_SERVER_ID, ")",
               "\r\n",
         "        ", "via TCP", " ", "with SMTP", "\r\n",
         "        ", "for ", recipientAddrList,
         "; ", rfc1123Date(currentTime), "\r\n");

   mailBody += traceLine;
   return (0);
}
```

```
SMTPclientConnection:_helo(array tokens)
{
   string      response;

   call "_reset"();

   if (typeOf(tokens[1]) == string) claimedPeerName = tokens[1];
   else claimedPeerName = "didNotSay";

   response = makeAsString("250 ", serverConfig["hostName"], " Hello ",
         peerAddress, "\r\n");
   send "writeBytes"(response) to connObj from nil;
   return (0);
}

SMTPclientConnection:_ehlo(array tokens)
{
   string      response;

   call "_reset"();

   if (typeOf(tokens[1]) == string) claimedPeerName = tokens[1];
   else claimedPeerName = "didNotSay";

   // Announce support for RFC 1652 - 8Bit-MIMEtransport
   response = makeAsString("250-", serverConfig["hostName"], " Hello ",
         peerAddress, "\r\n",
         "250 8BITMIME\r\n");
   send "writeBytes"(response) to connObj from nil;
   return (0);
}

SMTPclientConnection:_help(array tokens)
{
   set   lines;
   string      l;

   lines += "214-FARGOS/VISTA SMTP Server\r\n";
   lines += "214-Commands:  HELO ELHO MAIL RCPT DATA QUIT\r\n";
   lines += "214-           VRFY EXPN\r\n";
   lines += "214-           RSET NOOP HELP\r\n";
   lines += "214 End of HELP output\r\n";
   l = makeAsString(lines);
   send "writeBytes"(l) to connObj from nil;
   return (0);
}
```

```
SMTPclientConnection:_mail(array tokens)
{
   oid   mailbox;
   string      response, fromCmd, user;

   if (currentParseState != HELLO) {
         send "writeBytes"("503 Transaction already in progress\r\n")
               to connObj from nil;
         return (0);
   }
   if (elementCount(tokens) < 2) {
         send "writeBytes"("501 FROM: argument required\r\n") to connObj from nil;
         return (0);
   }
   if (tokens[1] == "FROM:") {
         user = tokens[2];
   } else {
         fromCmd = midstr(tokens[1], 0, 5);
         fromCmd = convertCase(fromCmd, 0);
         if (fromCmd != "FROM:") {
               send "writeBytes"("501 Mailbox Syntax error -- no FROM:\r\n")
                     to connObj from nil;
               return (0);
         }
         user = midstr(tokens[1], 5, length(tokens[1]) - 5);
   }

   senders += user;
   send "writeBytes"("250 OK\r\n") to connObj from nil;
   currentParseState = MAIL;
   return (0);
}
```

```
SMTPclientConnection:_rcpt(array tokens)
{
   oid   mailbox;
   string      response, toCmd, user;
   assoc info;

   if (currentParseState != MAIL) {
         send "writeBytes"("503 Need to issue MAIL before RCPT\r\n")
               to connObj from nil;
         return (0);
   }
   if (elementCount(tokens) < 2) {
         send "writeBytes"("501 TO: argument required\r\n") to connObj from nil;
         return (0);
   }
   if (tokens[1] == "TO:") {
         user = tokens[2];
   } else {
         toCmd = midstr(tokens[1], 0, 3);
         toCmd = convertCase(toCmd, 0);
         if (toCmd != "TO:") {
               send "writeBytes"("553 Mailbox Syntax error -- no TO:\r\n")
                     to connObj from nil;
               return (0);
         }
         user = midstr(tokens[1], 3, length(tokens[1]) - 3);
   }

   mailbox = send "lookupMailbox"(user) to mailboxService;
   if (mailbox == nil) {
         response = makeAsString("550 ", user, " user unknown\r\n");
         send "writeBytes"(response) to connObj from nil;
         return (0);
   }
   recipients -= mailbox;  // prune out any duplicates -- deliver once
   recipients += mailbox;
   info = send "getMailboxInfo" to mailbox;
   if (recipientAddrList != emptySet) {
         recipientAddrList += ", ";
   }
   recipientAddrList += makeAsString("<", info["fqName"], ">");

   response = makeAsString("250 OK for ", info["fqName"], "\r\n");
   send "writeBytes"(response) to connObj from nil;
   return (0);
}
```

```
SMTPclientConnection:_vrfy(array tokens)
{
   // 550 userName user unknown
   // 250 FirstName LastName <account@host.domain>
   oid   mailbox;
   string      response;
   assoc info;

   mailbox = send "lookupMailbox"(tokens[1]) to mailboxService;
   if (mailbox == nil) {
         response = makeAsString("550 ", tokens[1], " user unknown\r\n");
         send "writeBytes"(response) to connObj from nil;
         return (0);
   }
   info = send "getMailboxInfo" to mailbox;
   response = makeAsString("250 ", info["fullName"],
         " <", info["fqName"], ">\r\n");
   send "writeBytes"(response) to connObj from nil;
   return (0);
}

SMTPclientConnection:_expn(array tokens) alias for _vrfy;

SMTPclientConnection:_processMail()
{
   oid   mailbox;
   string      mailMessage, response;

display("recipients=", emptySet + recipients);
   mailMessage = makeAsString(mailBody);     // convert to big string
   for mailbox in recipients do {
         send "storeMail"(mailMessage) to mailbox from nil;
   }
   response = "250 OK\r\n";      // all done...
   send "writeBytes"(response) to connObj from nil;

   call "_reset"();
   return (0);
}
```

While improbable, a very slow email transfer may be in progress at the same time an administrator decides to shutdown the system. The SMTP RFC does address this case and specifies that a "421" error notification should be sent and then the connection should be closed (see section 3.9 of RFC 2821). To comply with the RFC, an **SMTPclientConnection** object can request notification of a system shutdown by sending an **addNotifyWhenShutdown** message to the **SystemShutdown** service (see the last line of the **SMTPclientConnection:create** method). The **systemShutdown** method below handles such notifications.

```
SMTPclientConnection:systemShutdown()
{
   string      line;
   int   rc;

   line = makeAsString("421 ", serverConfig["hostName"],
         " is shutting down\r\n");
   rc = send "writeBytes"(line) to connObj;
   send "deleteYourself" to readBfr;
   readBfr = nil;
   sessionClosed = 1;
   shutdownOID = nil;
}
```

Applications with Meta-Objects

The FARGOS/VISTA Object Management Environment supports a concept called *reflection*[3]. A meta object can be associated with an existing object, which permits the meta object to enhance the existing object. In computer science theory, reflection permits higher-level layers to re-implement the lower layers. When a meta object *M* is associated with an object *O*, any message sent to object *O* is redirected to meta object *M* for processing. The meta object must implement a **reflectedMessage** method, which is always passed four arguments:

1. the destination object Id
2. a string that specifies the name of the method that was to be invoked
3. an array of arguments for the method
4. the source of the message (e.g., **fromObject**)

The example below illustrates a variation of the services provided by the **TraceInvocations** debugging class. The **create** method issues a **setMeta** request (implemented by the base class **Object**) against the target object. The **reflectedMessage** method displays the method invocation data and forwards the request as-is to the associated object, thus yielding a per-object tracing service that does not affect the operation of an application.

```
%include <OMEcore.o2h>

global {
const string srcID = "$Id$";
};

class Local . TraceInvocations {
   oid   clientObj;
} inherits from Object;

// client is public address...
TraceInvocations:create(oid client)
{
   array methodList;

   clientObj = client;
   send "setMeta"(thisObject) to clientObj;
}

TraceInvocations:delete()
{
   if (clientObj != nil) {
         send "setMeta"(nil) to clientObj;
   }
}

TraceInvocations:disableTrace()
{
   if (clientObj != nil) {
         send "setMeta"(nil) to clientObj;
         clientObj = nil;
   }
   if (fromObject != nil) return (0);
}

TraceInvocations:reflectedMessage(any destination, string methodName,
   array methodArgv, any fromObj)
{
   display("reflectedMess:method=", methodName, " argv=",argv);
   send (methodName)(arrayToSet(methodArgv)) to clientObj from fromObj;
}
```

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

The use of the debugging facility demonstrated above is illustrated by the **TestTrace** class below. It makes use of a **TestTraceObj** class that serves as a target object for method invocations.

```
class Local . TestTrace {
   oid   traceObj;
} inherits from Object;

TestTrace:create()
{
   oid   obj;
   assoc acl;
   int   i;

   acl = makeDefaultACL();
   obj = send "createObject"("TestTraceObj", acl) to ObjectCreator;
   traceObj = send "createObject"("TraceInvocations", acl, obj)
         to ObjectCreator;
   display("Start test\n");
   for(i=1;i<=10;i+=1) {
         send "method1"(i) to obj;
   }
   i = send "disableTrace" to traceObj; // make sure it's deleted...
   send "method1"(-123) to obj;
   send "deleteYourself" to traceObj;
   send "deleteYourself" to obj;
   send "deleteYourself" to thisObject;
}

TestTrace:delete()
{
}

class Local . TestTraceObj {
   oid   traceObj;
} inherits from Object;

TestTraceObj:create()
{
   display("TestTraceObj created\n");
}

TestTraceObj:delete()
{
}

TestTraceObj:method1()
{
   display("TestTraceObj:method1, argv=", argv);
}
```

Meta objects can be used in a variety of ways. As illustrated by the example above, one very important use is to be able to intercept invocations against an object without requiring knowledge ahead of time of a method's name or argument list. Another is to extend the behavior of existing code to which one does not have access to the source code. In theory, extremely well designed object-oriented code would provide the opportunity for local extensions via inheritance. In practice, this level of design engineering is not achieved by application programmers. The use of a meta object enables the implementation of an object's method to be replaced/overridden. In environments that require non-stop operation, this same technique can be applied to deploy bug fixes to an already running application without requiring a restart[4].

[4] Support for non-stop operation is an often overlooked FARGOS/VISTA capability because the entire premise is completely foreign to most developers and system administrators. The capability to have multiple versions of a given class

HTTP Server Integration

The following demonstrates a comment page associated with a web server implemented by the **HTTPdaemon**. Normally, such a page would be implemented as a derived class of **HTTPcachedObject**; however, the example below implements all of the required methods on its own.

simultaneously in use is another cornerstone of FARGOS/VISTA's support for non-stop operation.

```
%include <OMEcore.o2h>

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

class Local . WebGuestBookForm {
   oid   urlDirectory;
   oid   logFileObj;
   string      pageName;
   array fieldName;
   assoc outputFieldPos;
   string      acknowledgementPage;
   string      dbFileName;
   int   appendMode;
} inherits from Object;

// serverName logicalPOSTpage fileName-of-ackTemplate addressFileName dbFileName
ouputFieldName1 inputFormfield1 ...

// DEFINED OUTPUT FIELDS:
// EmailAddress
// FirstName
// MidleInitial
// LastName
// CommentText

WebGuestBookForm:create(string serverName, string page, string ackPage,
   string addrLogFileName, string dbLogFileName, string fieldName1)
{
   int   i, count;
   string      fileSpec, key;
   assoc acl;

   dbFileName = dbLogFileName;

   // open address log file... (create/append)
   fileSpec = makeAsString("file:", addrLogFileName, ",ca");
   acl = makeDefaultACL();
   logFileObj = send "createObject"("IOobject", acl, fileSpec)
         to ObjectCreator;
   i = send "getID" to logFileObj;
   if (i == -1) {
         display("WebGuestBookForm:create: Could not open ", fileSpec, "\n");
   }

   urlDirectory = lookupLocalService(URL_DIR_PREFIX + serverName);
display("WebGuestBookForm:  Got URL directory oid as ", urlDirectory, "\n");
   pageName = page;
   acknowledgementPage = ackPage;
   // DEPENDS ON POSITION OF ARGUMENTS
   count = 0;
   for(i=5;i<argc;i+=2) {
         key = argv[i];
         outputFieldPos[key] = count;
         fieldName[count] = argv[i + 1];
         count += 1;
   }
   send "registerObject"(pageName, thisObject) to urlDirectory
         from nil;
}
```

```
WebGuestBookForm:delete()
{
   if (urlDirectory != nil) {
         send "removeFromCache"(pageName) to urlDirectory;
   }
   if (logFileObj != nil) {
         send "deleteYourself" to logFileObj;
   }
}


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

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


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

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

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


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

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

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

```
WebGuestBookForm:postRequest(array requestData, assoc options,
   string replyMethod, oid replyDest)
{
   array formInfo, dest;
   assoc acl, timeInfo;
   int   i, j, count, rc, t;
   assoc condensedData;
   string      key, f, fileSpec, tempFileName;
   set   writeArgs;
   string      body, hdr, text;
   string      fileName, comment, firstName, lastName, email;
   oid   newFileObj, dbFileObj;

   // FIRST:  parse form data...
   formInfo = parseHTTPformData(options["ENTITY_CONTENT"], array);
   for(i=0;indexExists(formInfo, i) != 0;i+=1) {
         j =  nextIndex(formInfo[i], 0);
         key = getKeyForIndex(formInfo[i], j);
         condensedData[key] = formInfo[i][key];
   }
display("Output field pos=", outputFieldPos);
display("Condensed data=", condensedData);
display("fieldName=", fieldName);
   // drop any malicious HTML tags...
   comment = condensedData[fieldName[outputFieldPos["CommentText"]]];
   i = findSubstring(comment, "<");
   if (i != -1) { // just get rid of everything...
         comment = midstr(comment, 0, i);
         comment += "[REMAINDER DELETED]";
   }
   firstName = condensedData[fieldName[outputFieldPos["FirstName"]]];
   i = findSubstring(firstName, "<");
   if (i != -1) { // just get rid of everything...
         firstName = midstr(firstName, 0, i);
         firstName += "[REMAINDER DELETED]";
   }
   lastName = condensedData[fieldName[outputFieldPos["LastName"]]];
   i = findSubstring(lastName, "<");
   if (i != -1) { // just get rid of everything...
         lastName = midstr(lastName, 0, i);
         lastName += "[REMAINDER DELETED]";
   }
   email = condensedData[fieldName[outputFieldPos["Email"]]];
   i = findSubstring(email, "<");
   if (i != -1) { // just get rid of everything...
         email = midstr(email, 0, i);
         email += "[REMAINDER DELETED]";
   }


   // Write email entries to CSV log file
   i = 0;
   writeArgs += i;   // skip offset = 0
   // Name
   writeArgs += "\"";
   writeArgs += firstName;
   writeArgs += "\",\"";
   writeArgs += condensedData[fieldName[outputFieldPos["MiddleInitial"]]];
   writeArgs += "\",\"";
   writeArgs += lastName;
   writeArgs += "\",\"";
   // Address
   writeArgs += email;
   writeArgs += "\"\n";    // add new line
   send "writeVectorOfBytes"(writeArgs) to logFileObj from nil;
   // CSV line added to address log file...
```

```
// NOW ADD COMMENTS TO GUEST BOOK DATA FILE
requestCount += 1;
// create temporary file in same directory (to permit rename)...
tempFileName = makeAsString(dbFileName, "_cpy", requestCount);

// create/truncate/write
fileSpec = makeAsString("file:", tempFileName, ",ctw");
acl = makeDefaultACL();

newFileObj = send "createObject"("IOobject", acl, fileSpec)
      to ObjectCreator;
i = send "getID" to newFileObj;

if (i == -1) {
      display("WebGuestBookForm:post could not open ", fileSpec, "\n");
      send "deleteYourself" to newFileObj;
      // abort with something like an appropriate message....
      call "getRequest"(arrayToSet(argv));
      exit;
}

// try to open DB file...
fileSpec = makeAsString("file:", dbFileName, ",r"); // read-only
acl = makeDefaultACL();

dbFileObj = send "createObject"("IOobject", acl, fileSpec)
      to ObjectCreator;
i = send "getID" to dbFileObj;
if (i == -1) {
      send "deleteYourself" to dbFileObj;
      dbFileObj = nil;
}

// now output to formatted DB file...
if (appendMode == 1) {  // copy original file first...
      call "_copyFile"(dbFileObj, newFileObj);
}


writeArgs = emptySet;
i = 0;
writeArgs += i;   // skip offset = 0
writeArgs += "<B>";
writeArgs += comment;
writeArgs += "</B><BR>\r\n";
writeArgs += firstName;
writeArgs += " ";
writeArgs += lastName;
writeArgs += " &lt;<A href=\"mailto:";
writeArgs += email;
writeArgs += "\">";
writeArgs += email;
writeArgs += "</A>&gt;<BR>\r\n";


t = getLocalRelativeTime();
timeInfo = convertLocalRelativeTimeToAbsolute(t, 0);
writeArgs += rfc1123Date(timeInfo);

writeArgs += "<HR>\r\n";
send "writeVectorOfBytes"(writeArgs) to newFileObj from nil;
```

```
   // now output to formatted DB file...
   if (appendMode == 0) {  // copy original file first...
         call "_copyFile"(dbFileObj, newFileObj);
   }
   if (dbFileObj != nil) {
         send "deleteYourself" to dbFileObj;
   }
   rc = send "closeForWrite" to newFileObj; // wait for close..
   send "deleteYourself" to newFileObj;
// unlinkFile(dbFileName);
   rc = renameFile(tempFileName, dbFileName);
display("Rename rc=", rc, "\n");


   text = "successfully recorded";
   body = makeAsString("<HTML><HEAD><TITLE>Thanks for your
Inquiry</TITLE></HEAD>\r\n<BODY><P><B>",
         "We have ", text," your information.</P><P><A href=\"",
         acknowledgementPage, "\">Continue</A></P></BODY></HTML>\r\n");


   hdr = makeAsString("Location: ", acknowledgementPage,
         "\r\nContent-type:  text/html\r\nContent-length: ",
         length(body), "\r\nConnection-close\r\n\r\n");
display("hdr=", hdr);
   send (replyMethod)(REDIRECT_CODE, REDIRECT_TYPE, hdr, body)
         to replyDest;
display("Method done\n");
}

WebGuestBookForm:_copyFile(oid srcFile, oid destFile)
{
   any   data;

   if (srcFile == nil) return (-1);
   data = send "readBytes"(0x1000) to srcFile;
   while (data != nil) {
         send "writeBytes"(data) to destFile from nil;
         data = send "readBytes"(0x1000) to srcFile;
   }
   return (0);
}
```

Remote Object Creation

Because the FARGOS/VISTA infrastructure is transparently distributed, the creation of an object on a remote system is trivial: rather than send a **createObject** request to the local **ObjectCreator** object, the request is sent to the **ObjectCreator** object of the target system. The object Ids of currently known systems can be obtained through the **listRemoteSystems()** function. This is illustrated by the code fragment below:

```
int   i, totalSystems;
assoc acl;
oid   obj;
array rmtSys;
set   args;

rmtSys = listRemoteSystems();
debugDisplay(debugLogLevel2, "Remote systems=", rmtSys);
// include ourselves as potential server
totalSystems = elementCount(rmtSys);
rmtSys[totalSystems] = ObjectCreator;
totalSystems += 1;
debugDisplay(debugLogLevel2, "total systems=", totalSystems, "\n");
acl = makeDefaultACL();
for(i=0;i < totalSystems;i += 1) {
      obj = send "createObject"("TestClass", acl, args) to rmtSys[i];
}
```

The example above, however, is useful only when the systems of interest represent a stable and already interconnected population. Robust applications should take advantage of the notifications of new peers and the loss of existing peers. An object of class **PeerRegistry** implements the "*/PeerRegistry*" service, which monitors the comings and goings of peers that are known to the local FARGOS/VISTA Object Management Environment. Interested applications can use the **addNotifyWhenPeerRegistered** method to request the delivery of **peerRegistered** and **peerUnregistered** messages.

The example below is a simplified version of the standard class **JobController**, which implements a facility for distributing jobs across a cluster of machines. The current buzzword of the day is “grid computing”, but the technique has been in used for decades supercomputing. Commentary appears between some of the methods.

```
%include <OMEcore.o2h>

global {
const int MAX_JOBS_PER_SERVER = 2;
};

class Local . JobController {
/*!
Provides a service that distributes tasks across a load-balanced pool
of servers.
!*/
   oid   peerRegistry;
   int   serverIdCounter;
   assoc serverList;
   array serverOID;
   array serverLoad;
   array serverForJob;
   array queuedWork;
   int   firstEntryIndex;
   int   lastEntryIndex;
   int   maxConcurrentJobs;
   int   jobsInProgress;
} inherits from Object;

JobController:create()
{
   peerRegistry = lookupLocalService("/PeerRegistry");
   if (peerRegistry == nil) {
         display("JobController: no /PeerRegistry\n");
         send "deleteYourself" to thisObject;
   }
   send "peerRegistered"(ObjectCreator) to thisObject;

   send "addNotifyWhenPeerRegistered"(thisObject, 1)
         to peerRegistry from nil;
   firstEntryIndex = 1;
   lastEntryIndex = 1;
}


JobController:delete()
{
   if (peerRegistry != nil) {
         send "removeNotifyWhenPeerRegistered"(thisObject)
               to peerRegistry from nil;
   }
}
```

The **peerRegistered** method is passed the object Id of the **ObjectCreator** object of the remote peer. Since the local system is not a peer, the **create** method shown above adds it to the list of available systems by performing the following:

```
   send "peerRegistered"(ObjectCreator) to thisObject;
```

When the connection to a peer is lost, the **PeerRegistry** object sends its clients **peerUnregistered** notifications.

```
JobController:peerRegistered(oid remotePeer)
{
   string      key;

   if (remotePeer == nil) exit;

   key = makeAsString(remotePeer);
   serverIdCounter += 1;
   serverList[key] = serverIdCounter;
   serverOID[serverIdCounter] = remotePeer;
   serverLoad[serverIdCounter] = 0;

   maxConcurrentJobs = elementCount(serverList) * MAX_JOBS_PER_SERVER;
   while (firstEntryIndex < lastEntryIndex) {
         if (jobsInProgress > maxConcurrentJobs) break;
         call "dequeueJob"();
   }
}

JobController:peerUnregistered(oid remotePeer)
{
   string      key;
   int   id;

   key = makeAsString(remotePeer);
   id = serverList[key];

   deleteIndex(serverList, key);
   deleteIndex(serverOID, id);
   deleteIndex(serverLoad, id);

   maxConcurrentJobs = elementCount(serverList) * MAX_JOBS_PER_SERVER;
}

JobController:selectServer()
{
   int   minId, minLoad, i, l;

   minId = -1;
   minLoad = 0x7f0000;
   for(i=nextIndex(serverLoad, 0);i != 0;i=nextIndex(serverLoad,i)) {
         l = serverLoad[i];
         if (l < minLoad) {
               minLoad = l;
               minId = i;
         }
   }
   return (minId);
}

JobController:queueJob(oid client, any clientInfo, string className, assoc acl, set
args)
{
/*!
Queue a work unit.  Result returned to <i>client</i> by sending it a
<b>jobComplete</b> message and passing it <i>clientInfo</i> and the
result of the job.
!*/
   queuedWork[lastEntryIndex] = argv;
   lastEntryIndex += 1;
   if (jobsInProgress < maxConcurrentJobs) {
         call "dequeueJob"();
   }
   if (fromObject != nil) return (0);
}
```

```
JobController:returnJobResult(int transactionId, any result)
{
/*!
Notes the completion of a queued task.
!*/
   array rec;
   int   server;
   oid   client;
   any   clientInfo;

   if (indexExists(serverForJob, transactionId) == 0) {
         debugDisplay(debugLogLevel3,
               "JobController:jobComplete no such transaction",
               transactionId, "\n");
         exit;
   }
   jobsInProgress -= 1;

   server = serverForJob[transactionId];
   serverLoad[server] -= 1;

   rec = queuedWork[transactionId];
   client = rec[0];
   clientInfo = rec[1];

   deleteIndex(queuedWork, transactionId);
   deleteIndex(serverForJob, transactionId);

   send "jobComplete"(clientInfo, result) to client;
   if (jobsInProgress < maxConcurrentJobs) {
         call "dequeueJob"();
   }
}

JobController:dequeueJob()
{
   array rec;
   string      className;
   assoc acl;
   set   args;
   oid   obj, rmtObj;
   int   server, transactionId;

   if (lastEntryIndex == firstEntryIndex) return (0);
   transactionId = firstEntryIndex;
   firstEntryIndex += 1;
   jobsInProgress += 1;

   server = call "selectServer"();
   serverForJob[transactionId] = server;
   rmtObj = serverOID[server];
   serverLoad[server] += 1;

   rec = queuedWork[transactionId];
   className = rec[2];
   acl = rec[3];
   args = rec[4];

   send "createObject"(className, acl, thisObject, transactionId,
         args) to rmtObj from nil;

   return (transactionId);
}
```

The usage of the **JobController** class described above is demonstrated by the following two classes. The **TestJobController** class creates a **JobController** object and issues a series of **queueJob** requests. Each unit of work is implemented by a **TestJob** object, which sends its results back to the **JobController** object using the **returnJobResult** method. Results are received by the **TestJobController** object via its **jobComplete** method.

```
class Local . TestJobController {
   oid   controllerObj;
   int   queuedJobs;
   int   completedJobs;
} inherits from Object;

TestJobController:create(int count)
{
   assoc acl;
   int   i;
   set   args;

   acl = makeDefaultACL();
   controllerObj = send "createObject"("JobController", acl, "TestPool")
         to ObjectCreator;

   queuedJobs = count;
   args += "hi";
   for(i=1;i<=count;i+=1) {
         send "queueJob"(thisObject, i, "TestJob", acl, emptySet + args)
               to controllerObj from nil;
   }
}

TestJobController:delete()
{
   send "deleteYourself" to controllerObj;
}

TestJobController:jobComplete(any jobInfo, any result)
{
   display("result from job ", jobInfo, " is ", result, "\n");
   completedJobs += 1;
   if (completedJobs == queuedJobs) send "deleteYourself" to thisObject;
}

class Local . TestJob {
   int   transactionId;
   oid   jobController;
} inherits from Object;

TestJob:create(oid jobCntrl, int transId, any arg1)
{
   string      result;

   jobController = jobCntrl;
   transactionId = transId;

   // Do lots of work...
   result = makeAsString("[", transId, ":", arg1, "]");

   // After lots of work completed, return result...
   send "returnJobResult"(transactionId, result) to jobController;

   send "deleteYourself" to thisObject;      // all done, delete
}

TestJob:delete()
{
}
```

The **RegisterPoolMembership** class provides a convenient (and suggested standard) mechanism for expressing the willingness of a particular FARGOS/VISTA Object Management Environment to participate as a member of a server pool. A given FARGOS/VISTA Object Management Environment can participate in as many distinct server pools as desired, be it zero, one or a hundred. By respecting the convention implemented by **RegisterPoolMembership**, applications can ensure that they only make use of spare cycles on systems that have intentionally expressed permission.

Persistent Objects

Just as in conventional environments, most FARGOS/VISTA objects are transient and have a lifetime that will not exceed that of the process into which the object was instantiated. When data needs to be retained between uses of an application, conventional environments store such data in a file or database. While accepted as normal practice, there is usually a clear dichotomy between the in-memory data and the corresponding copy on some stable media. While such techniques can be used by FARGOS/VISTA-based applications, it is also possible to use persistent objects. A persistent object survives the lifetime of the process in which it was initially created.

The FARGOS/VISTA Object Management Environment has several intrinsic capabilities that are used to implement persistent objects, but most applications programmers will not concern themselves with the details. Instead, the standard convenience class **PersistentObject** is the mechanism of choice. Persistence can be added to any class by inheriting from the class **PersistentObject**.

The **create** method of **PersistentObject** registers the new persistent object with the well-known service "*/ObjectPager*" by sending it a **makePersistent** message and passing the object's Id. This is illustrated below:

```
send "makePersistent"(thisObject) to "/ObjectPager";
```

By default, the "*/ObjectPager*" service is obtained by instantiating an object of the class **PersistenceService**; however, installations can implement the "*/ObjectPager*" service using a locally developed solution or other alternatives that are allomorphic to the class **PersistenceService**. As noted above, the **PersistentObject** class uses the **makePersistent** method. The remaining methods of the "*/ObjectPager*" service are invoked by the external object database daemon. The default version is provided by the **OMEpersistd** executable, but this too can easily be replaced by a locally developed solution. It is actually more probable to replace the **OMEpersistd** with an alternative that stores object data as binary large objects in a relational database than to replace the logic provided by the **PersistenceService**.

Persistent objects have to deal with some issues that are not faced by transient objects because a persistent will be restored into environments that have changed radically. Any transient objects of which it was previously aware will no longer exist.

An external persistence daemon notifies the "*/ObjectPager*" service of its existence by sending a **databaseConnected** message:

```
send "databaseConnected"(thisObject) to "/ObjectPager";
```

The **PersistenceService** sends a **listObjects** request to the external persistence daemon. The returned result is an array that provides the object Ids of all of the objects maintained by the external persistence daemon. The object Id information is actually provided as a string that must first be decoded by the **decodeData()** function. The object Ids are maintained as strings to permit their storage by applications that are not aware of FARGOS/VISTA data types. The process of decoding an object Id and setup of the */ObjectPager* service as its meta object is illustrated by the code fragment below:

```
key = list[i];
mgmtOID = decodeData(key);
OILsetExternalMetaObject("/ObjectPager", "objectFault", mgmtOID);
```

Email Mailbox Storage

As an illustration of the ease-of-use of the class **PersistentObject**, an email mailbox storage system is presented below. It accepts delivery of email received by an SMTP server and it can be accessed using any POP3-capable email client. The service is implemented using two classes. One, **SMTPmailboxService**, implements a directory of user mailboxes. It is used by creating a single, transient instance each time the FARGOS/VISTA Object Management Environment starts up. The second class, **SMTPmailbox**, is implemented as a derived class of **PersistentObject**. Each instance of **SMTPmailbox** corresponds to the mailbox of one user, thus there would be many instances of **SMTPmailbox**. When a new user is added to the system, an **SMTPmailbox** object would be created for them at that time. Because an **SMTPmailbox** object is persistent, the object would survive terminations of the FARGOS/VISTA Object Management Environment. If the user's email account eventually needs to be removed at some point, the specific **SMTPmailbox** object would be deleted by the administrator.

```
%include <OMEcore.o2h>

implicit {
   const string MAILBOX_SERVICE = "MailboxDirectory";
};

class Local . SMTPmailboxService {
   assoc userNameToMailbox;
   assoc mailboxAliases;
} inherits from Object;

SMTPmailboxService:create()
{
   registerService(MAILBOX_SERVICE, thisObject, 0);
}

SMTPmailboxService:delete()
{
   unregisterService(MAILBOX_SERVICE, thisObject);
}
```

```
SMTPmailboxService:registerMailbox(string mailboxName, oid mailbox)
{
   string      bracketKey;

   if (indexExists(userNameToMailbox, mailboxName) != 0) {
         return (-1);
   }
   userNameToMailbox[mailboxName] = mailbox;

   bracketKey = makeAsString("<", mailboxName, ">");
   mailboxAliases[bracketKey] = mailboxName;
   return (0);
}

SMTPmailboxService:registerAliases(string mailboxName, string alias1)
{
   int   i;

   for(i=1;i<argc;i+=1) {
         mailboxAliases[argv[i]] = mailboxName;
   }
   return (0);
}
```

```
SMTPmailboxService:unregisterAliases(string alias1)
{
   int   i;

   for(i=0;i<argc;i+=1) {
         mailboxAliases = deleteIndex(mailboxAliases, argv[i]);
   }
   return (0);
}

SMTPmailboxService:unregisterMailbox(string mailboxName, oid mailbox)
{
   string      bracketKey;

   if (indexExists(userNameToMailbox, mailboxName) == 0) {
display("mailbox name not known=", mailboxName, "\n");
         return (-1);
   }
   userNameToMailbox = deleteIndex(userNameToMailbox, mailboxName);

   bracketKey = makeAsString("<", mailboxName, ">");
   userNameToMailbox = deleteIndex(mailboxAliases, bracketKey);
   return (0);
}
```

```
SMTPmailboxService:lookupMailbox(string mailboxName)
{
   string      name;

   if (indexExists(userNameToMailbox, mailboxName) != 0) {
         return (userNameToMailbox[mailboxName]);
   }
   if (indexExists(mailboxAliases, mailboxName) != 0) {
         name = mailboxAliases[mailboxName];
         if (indexExists(userNameToMailbox, name) != 0) {
               return (userNameToMailbox[name]);
         }
   }
   return (nil);
}

SMTPmailboxService:listMailboxes()
{
   array result;
   int   count, i;

   i = nextIndex(userNameToMailbox, 0);
   while (i != 0) {
         result[count] = getKeyForIndex(userNameToMailbox, i);
         count += 1;
         i = nextIndex(userNameToMailbox, i);
   }
   return (result);
}
```

```
class Local . SMTPmailbox (1) {
   string      userName;
   string      fullName;
   string      fullyQualified;
   string      password;
   array       messages;
   int         messageTotal;
} inherits from PersistentObject;

SMTPmailbox:create(string forUser, string longName, string pw)
{
   if (argc != 0) {
         call "SMTPmailbox:initialize"(arrayToSet(argv));
   }
}

SMTPmailbox:initialize(string forUser, string longName, string pw)
{

   userName = forUser;
   if (typeOf(longName) == string) {
         fullName = longName;
   } else {
         fullName = userName;
   }
   if (findSubstring(userName, "@") != -1) {
         fullyQualified = userName;
   } else {
         fullyQualified = makeAsString(userName, "@",
               getSystemInfoAttribute("hostName"));
   }
   if (length(pw) > 0) {
         password = pw;
   } else {
         password = "changeme";
   }
   send "registerMailbox"(userName, thisObject)
         to MAILBOX_SERVICE from nil;
   send "registerAliases"(userName, fullyQualified, "<" + fullyQualified + ">")
         to MAILBOX_SERVICE from nil;
   // Initialize persistence, associate with "mailboxes" database
   call "PersistentObject:initialize"("mailboxes");
   return (0);
}

SMTPmailbox:delete()
{
   send "unregisterMailbox"(userName, thisObject)
         to MAILBOX_SERVICE from nil;
   send "unregisterAliases"(fullyQualified, "<" + fullyQualified + ">")
         to MAILBOX_SERVICE from nil;
}

SMTPmailbox:objectImported()
{
display("MAILBOX for ", userName, " imported\n");
   send "registerMailbox"(userName, thisObject)
         to MAILBOX_SERVICE from nil;
   send "registerAliases"(userName, fullyQualified, "<" + fullyQualified + ">")
         to MAILBOX_SERVICE from nil;
}
```

```
SMTPmailbox:authenticate(string checkPassword)
{
   if (checkPassword == password) return (1);
   return (0);
}

SMTPmailbox:changePassword(string newPassword, string checkPassword)
{
   if (checkPassword != password) {
         return (-1);
   }
   password = newPassword;
   return (0);
}
```

```
SMTPmailbox:getMailboxInfo()
{
   assoc attrs;

   attrs["userName"] = userName;
   attrs["fullName"] = fullName;
   attrs["fqName"] = fullyQualified;
   return (attrs);
}

SMTPmailbox:getMessageIds()
{
   array ids;
   int   i;

   i = nextIndex(messages, 0);
   while (i != 0) {
         ids[i] = length(messages[i]);
         i = nextIndex(messages, i);
   }
   return (ids);
}
```

```
SMTPmailbox:storeMail(string mailBody)
{
debugDisplay(debugLogLevel2, "Store email message:\n", mailBody);
   messageTotal += 1;
   messages[messageTotal] = mailBody;
   return (messageTotal);
}

SMTPmailbox:getMessage(int messageID)
{
   if (indexExists(messages, messageID) == 0) {
         return (nil);
   }
   return (messages[messageID]);
}
```

```
// Provided for POP3 UIDL command; permit hash to be done locally
// rather than transmit entire message body between systems
SMTPmailbox:getUniqueMessageId(int messageID)
{
   string      hash, text;

   if (indexExists(messages, messageID) == 0) {
         return ("doesNotExist");
   }
   hash = SHA1hash(messages[messageID]);
   text = makeAsHexString(hash);
   return (text);
}

// Provided for POP3 TOP command; do work locally rather than transmit
// entire message body between systems
SMTPmailbox:getMessageHeader(int messageID, int bodyLines)
{
   int   offset, nextOffset, i;
   string      mess, head;

   if (indexExists(messages, messageID) == 0) {
         return (nil);
   }
   mess = messages[messageID];
   offset = findSubstring(mess, "\r\n\r\n"); // end of MIME header
   if (offset == -1) {     // no end of MIME header, return everything
         return (mess);
   }
   offset += 4;      // length of CR LF CR LF
   for(i=0;i<bodyLines;i+=1) {
         nextOffset = findSubstringAfter(mess, "\r\n", offset);
         if (nextOffset == -1) { // ran out, return everything
               return (mess);
         }
         offset = nextOffset + 2;      // after CR LF
   }
   head = midstr(mess, 0, offset);
   return (head);
}
```

```
SMTPmailbox:deleteMessages(int messageID)
{
   int   i, id;

   for(i=0;i<argc;i+=1) {
         id = argv[i];
         if (indexExists(messages, id) == 0) {
               return (id);
         }
         messages = deleteIndex(messages, id);
   }
   return (0);
}
```

Byzantine Fault-Tolerant Transactions