RICE UNIVERSITY

Client-Server Component Architecture for Scientific Computing

by

Hala N. Dajani

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF ARTS

Approved, Thesis Committee:

William Symes , Chair
Professor of Computational and Applied Mathematics

Matthias Heinkenschloss
Associate Professor of Computational and Applied Mathematics

Mark Embree
Assistant Professor of Computational and Applied Mathematics

Houston, Texas

April, 2003

ABSTRACT

Client-Server Component Architecture for Scientific Computing

by

Hala N. Dajani

In a Distributed Computing Environment software components dispersed on a variety of computer platforms communicate transparently with each other to emulate a single computer platform. One distributed component framework model consists of two autonomous processes: the client and the server. The client-server model implemented in an object-oriented language shields low level platform complexities from the user and allows coupling of prefabricated components. These components must have a means of interfacing with each other in a distributed environment. To accommodate this need, while maintaining high performance, we propose a low level socket communication core. We employ the *proxy* design pattern and introduce new C++ classes to dynamically extend object behavior to a distributed environment. These classes also serve as component interfaces. Here, we describe general guidelines for partitioning objects into client and server components.

Acknowledgments

The past three years have been a time of both personal and intellectual growth. I have the faculty, students, and staff of the Computational and Applied Mathematics Department to thank. Everyone has contributed more than I can express. In particular, I thank Dr. William Symes for his unwavering belief in me, his inspiring creative intellect, and his liberal disposition. I could not have asked for a better advisor.

Contents

List of Figures

Chapter 1
Introduction

In scientific computing, many large-scale problems require the use of high-performance platforms such as parallel architectures. The use of object-orientated techniques enables abstraction to express scientific concepts and to reuse code. Such endeavors as the Toolkit for Advanced Optimization (TAO) succeed in developing scientific programs that utilize object-oriented techniques to allow the reuse of optimization software [3].

Although TAO brings sophisticated programing techniques to the development of scientific applications, it still admits low level details to infiltrate abstract concepts. TAO uses the Message Passing Interface (MPI) language embedded within the Portable Extensible Toolkit for Scientific Computation (PETSc) language to accommodate distributed computing [3]. The abstract types in TAO such as those that conceptualize mathematical vectors and matrices include environment details. For example, the creation of a vector type in TAO requires an MPI environment variable of the type MPI_Comm [3]. This results in limits to vector type abstraction. In addition, the PETSc environment must be available in order to codify and compile a driver. This leads to codifying the same driver separately for sequential and parallel platforms.

An algorithm that specifies mathematical operations is mostly independent of the data structures and environment of computations. Therefore, to achieve the most code reuse as possible, it is important to design software that keeps abstract concepts as separate as possible from low level environment details. The distributed client-server model allows this clean separation. It precludes computational details from permeating all levels of abstraction; allowing more code reuse, maintainability, and expansion.

In this thesis, I develop a process that extends object behavior to a distributed client-server component environment that is simple to use, interoperable, and maintains high performance. This mechanism entails the use of the *proxy* and *strategy* design patterns, the addition of new classes, and guidelines specifying the separation of objects into appropriate client and server domains. We demonstrate this process with the Standard Vector Library (SVL) software package.

Shannon Scott and William Symes propose the SVL software package as a refinement of the message forwarding (*visitor*) method Roscoe Bartlett initiates in *A Proposed Standard for User Defined Vector Reduction and Transformation Operators* [9, 1] . SVL, the next generation Hilbert Class Library (HCL), uses high levels of abstraction to emulate Hilbert Space Calculus.

SVL is a numerical software package written completely in C++ that defines a standard interface to vector objects. SVL defines two main class hierarchies; one for the elements being operated on, denoted appropriately as *DataContainers*, and the other for the *visitor* objects denoted as *FunctionObjects* [9]. True to form of *visitor* objects, *FunctionObjects* are passed as argument parameters to *DataContainer* methods. As *FunctionObjects* can define both mathematical and computational operations, SVL introduces a *Space* class that coupled with a *Vector* class models mathematical concepts.

In his Master's Thesis, Shannon Scott initiates the component client-server extension to SVL [8] . Each component is an autonomous process that has no persistent state. There are many advantages to adapting object-oriented software to components. Components allow further abstraction by encapsulating sets of objects. They use dynamic binding, are interchangeable, and can have multiple interfaces.

The complexity that arises form component based programming is the inclusion of communication middleware to couple components. Yet, components offer simplicity of use. A user need only familiarize herself with the component interface and not its internal structure. A client-server model naturally uses component design as client and server behaviors can be easily encapsulated within components.

According to Clemens Szyperski, there currently exist three major forces in the component middleware software arena: Common Object Request Broker Architecture (CORBA) based standard developed by Object Management Group (OMG) configured with the enterprise perspective, COM-based standards posed by Microsoft configured with the desktop perspective, and Java-based standards from Sun fashioned with the internet perspective [10] . Of the three forces, CORBA and Java are feasible for our component communication protocol.

CORBA defines an object-oriented Interface Definition Language (IDL) that offers encapsulation, polymorphism, and inheritance; all the abstractions of object oriented programming [5] . Such a highly evolved IDL requires some familiarity. IDL increases the complexity of formulating and passing messages; and the overall complexity of the entire application. Consequently, using CORBA only for message passing admits its inherent complexity while not taking full benefit of its resources.

The Java approach is one level of abstraction above base socket communication. It wraps communication protocol within objects and has a standard library already available. Rather than introduce another programming language into our package, we create our own objects by wrapping socket protocols as classes. Furthermore, by overloading C++ string operators, our socket abstractions mimic the C++ iostream, and as a result, require no extra familiarity with our socket library.

Chapter two describes the client-server component environment as well as the process design. In order to maintain efficiency, we formulate our own socket communication backbone. In chapter three, we present this socket library and its general use. Chapter four merges the design with the middleware library to produce a feasible implementation in SVL, and chapter five presents some examples to numerical applications.

Chapter 2
Client Server Behaviors

2.1 Introduction

In order to understand the process of extending single platform software to allow distributed behavior, we need first to look at client and server roles. By defining the component behaviors of client and server, we explain the general construction and use of this component system. We describe the behaviors of the two processes and outline the interaction between them. Then we go on to describe the process of splitting an application into the client server roles.

2.2 Behaviors

The characterization of any client-server model is the request/response relationship depicted in Figure 2.1 [2].

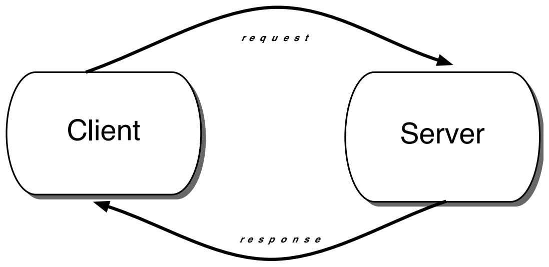

Figure 2.1 Client Server Interaction

Although in the client-server model we define two types of interaction behaviors, a component need not necessarily express only one behavior. For example, a component can both request and fulfill services from one or more components. For each behavior, we develop an interface that matches the counter behavior. These interfaces formulate the contractual agreement between the two processes.

The client-server model demonstrates the tradeoff between type complexity and environment complexity. The client domain has a larger object universe than the server. The client, tending to instantiate abstract objects, relies more on model conceptualization and less on data computation. Since the server does not interact with the user, it does not need to create an abstract conceptual model. Rather, its object universe consists of objects directly involved with computation. As a result, the server has a smaller object universe than the client.

2.2.1 Server Behavior

Although both client and server processes may draw from the same pool of objects, they typically instantiate a different set of objects. The server manages the distribution of and general handling of data in accordance with its specified environment. For example, a server specialized for a parallel data decomposition environment partitions the data and transports data blocks to various machines. A server process in this context listens at a static port, lying dormant until a connection is initiated by the client process. In this way, the server plays a more passive role in the client server relationship, waiting for invocation from the client.

The server listens for a connection, accepts it, and forks a child process to handle it, while the original process continues to listen for further client requests. The server provides an environment for the client to upload information, to instantiate objects on the server, and to execute object methods remotely. Figure 2.2 illustrates server actions.

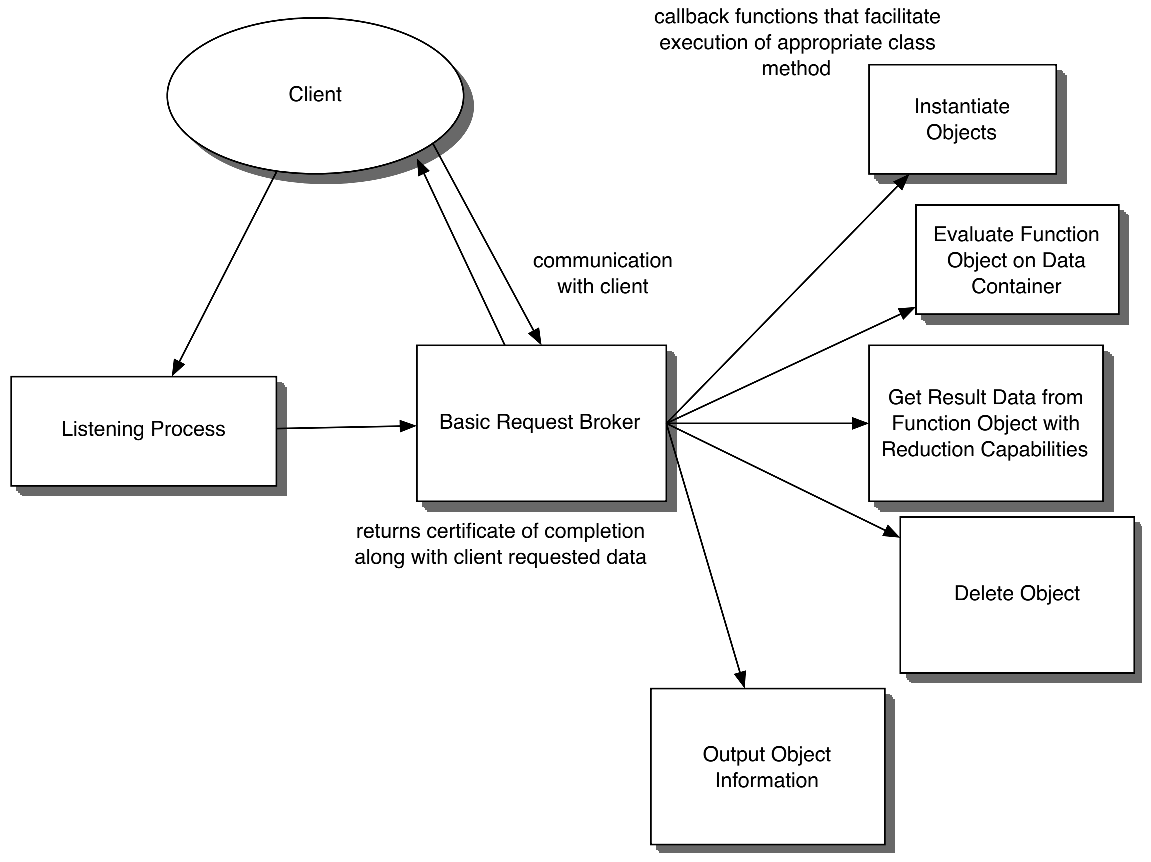
callback functions that facilitate execution of appropriate class method
Client
Instantiate Objects
Evaluate Function Object on Data Container
communication with client
Listening Process
Basic Request Broker
Get Result Data from Function Object with Reduction Capabilities
returns certificate of completion along with client requested data
Delete Object
Output Object Information

Figure 2.2 Server Behavior

The server request broker depicted in the figure is a registry of available requests that spawns events called "callbacks" that fulfill these requests. Each callback represents a functional unit, executing one service. The two base functional units create and delete objects in the server domain. To accommodate the *visitor* design pattern, we maintain a general functional unit for all evaluations of Function Objects upon data wrapped by Data Containers. The other two functional units included here, Output and Get Result, allow a complete set of output functions as specified by the class's Get Result and Output methods. In essence, these server functions convert the object identifier sent by the client into the appropriate object and call the requested method. In all cases a certificate of successful or unsuccessful completion is returned to the client. More functionality can be added to the server, but these represent a generic and minimal set [6] . In this project, we implement the server in an event driven procedural fashion. Regardless of how the server is implemented, these actions characterize the server.

2.2.2 Client Behavior

Although the server is specialized to a certain platform, the client process is not limited by platform requirements. It is the process by which users interact with the software package. As a result, the client instantiates objects that fit more naturally to the mathematical formulation much like a scripting language such as MatLab. This does not, however, preclude objects representing mathematical abstractions from existing on the server.

In the client behavior, we predominantly use the *proxy* design pattern to build a seamless distributed environment and the *visitor* design pattern to allow generic data manipulation. These types of classes model stand-in objects that represent objects created on the server [8, 4] . According to Design Patterns, "*proxy* objects are applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer [4]. " In some cases, we mimic the same interface as the counterpart server object. In other cases, the object only plays the *visitor* role and therefore does not require a user interface. In all cases, the role the object plays within the client environment mimics the role of its counterpart object within the server environment. Using this process, we can extend previously written applications without altering the kernel.

The client side behavior is expressed at an implicit level where the user has no interface with low level networking. Specific client side classes, through appropriate method invocation, send requests and receive data from the server.

In SVL, these *proxy* design pattern classes come in two types, *Remote* and *Proxy*. Yet, both provide a local existence to an object located in a different address space. These instances of the *(remote) proxy* design pattern differ in the extent of access to the object. The *Proxy* class allows more access to the object than the *Remote* class and consequently extends the *Remote* class. The main characteristic of *Proxy* objects in this package is that these objects provide the same interface as the objects they represent. While, *Remote* objects only verify existence. That is, the only *Remote* object interface we are concerned with on the client side is that of construction and deletion. All objects that require *Remote* representation and are not specialized as *Proxy* objects can employ the *visitor* design pattern. After construction of a strictly *Remote* object, we pass the existence certificate as an argument to other class methods. Any object in the server environment can have a *Remote* counterpart on the client. Figure 2.3 depicts remote extension for SVL base classes.

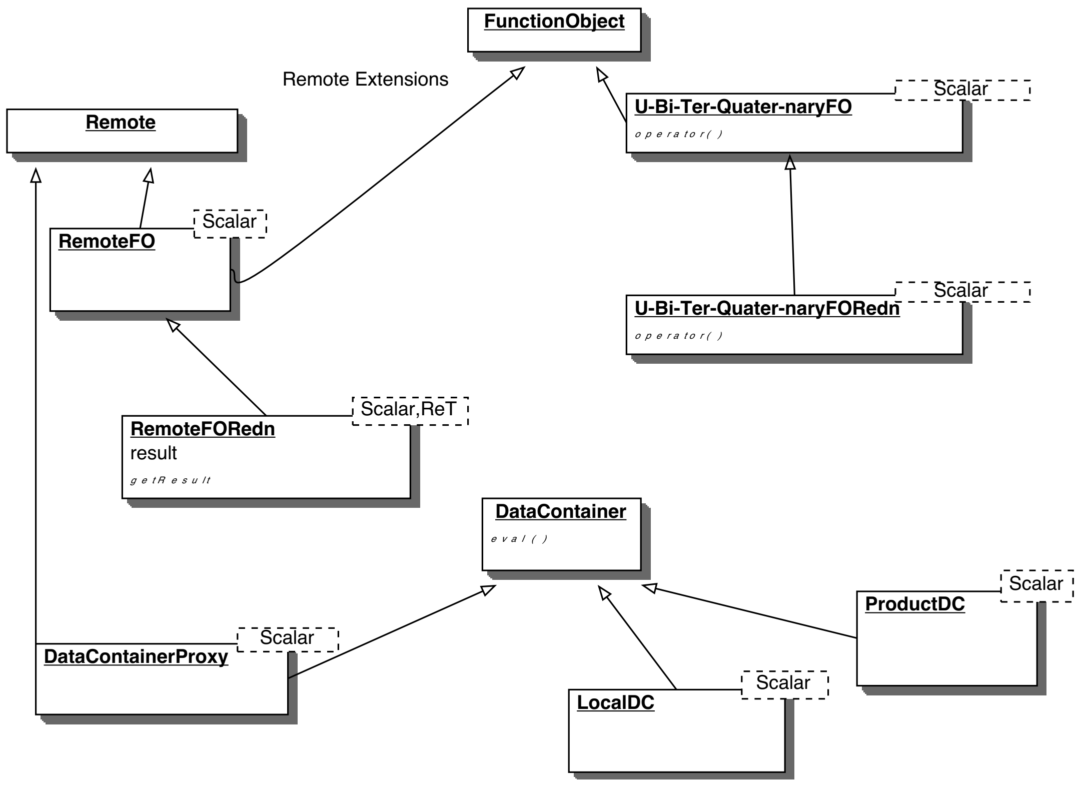

Figure 2.3 Base Class Remote Design: The *Remote* and *Proxy* classes (left) make use of double inheritance to extend the base classes (right).

2.3 Partitioning Object-Oriented Software into Components

This section describes the process by which we take SVL code, partition its class structure into client and server components, and introduce a middle-ware communication component to create a two tiered client server component system. The key point of component partitioning is to deduce the point of insertion of an object layer. The introduction of a new layer corresponds to new object domains which will later be refined and separated into components with differing address spaces. Our objective in specifying components is to find a balance between minimizing transportation of information and maximizing the reach of user interfaces.

The SVL design admits some natural extensions to client and server partitions. SVL's use of the *strategy* design pattern enables us to specify simple and general guidelines for splitting into client and server components. The *strategy* design pattern, graphically represented in Figure 2.4, uses composition and delegation to dynamically define the behavior of a class.

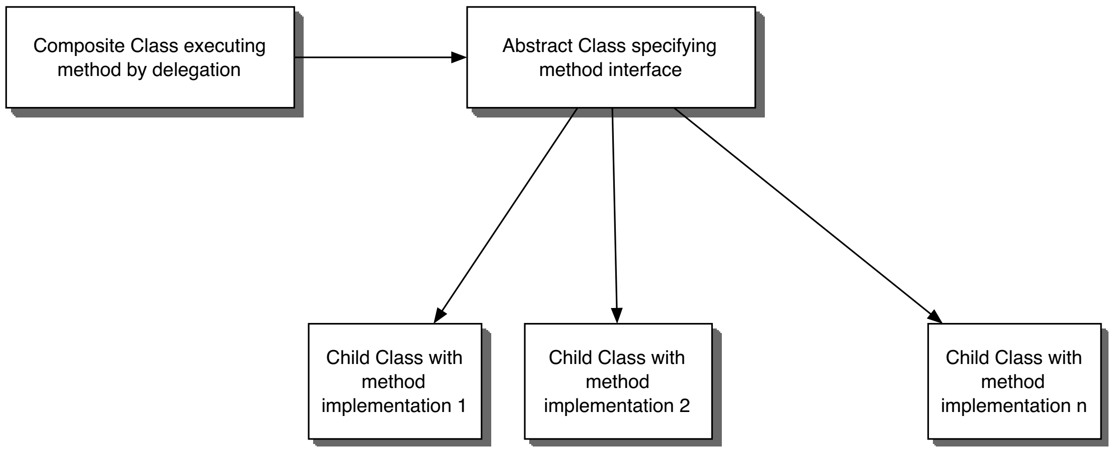
Composite Class executing method by delegation
Abstract Class specifying method interface
Child Class with method implementation 1
Child Class with method implementation 2
Child Class with method implementation n

Figure 2.4 *Strategy* Design Pattern

In general, a class that employs the *strategy* design pattern is an excellent candidate for the client domain, while the objects it is composed of are further evaluated to ascertain their appropriate component domain. Remote instances of composite objects relegated to the server must be created to introduce the new necessary object layer.

Light-weight objects are objects that do not explicitly manipulate large amounts of data, do not have resource constraints, and in some cases interact with the user directly. That is, they do not require a specific computer platform. Light-weight objects are developed to model concepts by way of object-oriented abstraction. All *strategy* classes are classified as light-weight objects, but not all light-weight objects are *strategy*. Heavy-weight objects, however, contain more resource constraints depending on the type of data operations they execute. They model computations and don't employ abstraction techniques. We classify objects into light-weight and heavy-weight groups based on common stratums of abstraction and subsequently, on their environment demands.

We have now established two guidelines for determining component domains and where to introduce the new object layer. The first guideline is to declare classes that have a user interface to the client. The objects remaining are heavy-weight objects and light-weight objects that do not express the highest level of abstraction. Our second guideline is that heavy-weight objects are assigned to the server. As we strive to minimize redundant communication, these objects are placed in the same domain as the objects with which they communicate. We substitute these heavy-weight and server-assigned light-weight objects with their remote counterparts on the client, complete the component specification, and split. We demonstrate this process first on high level SVL classes as this presents a general model of object interactions.

2.3.1 Basic Example

SVL has many layers of abstraction. At the highest level, we have the concrete class *Vector* with no subclasses. This class is a *strategy* class that calls upon other *strategy* classes. Figure 2.5 depicts the delegation chain in *Vector*. The *Vector* object only associates with three other object types; *Space, DataContainer*, and *FunctionObject.* It is composed of a *DataContainer* and a *Space*.

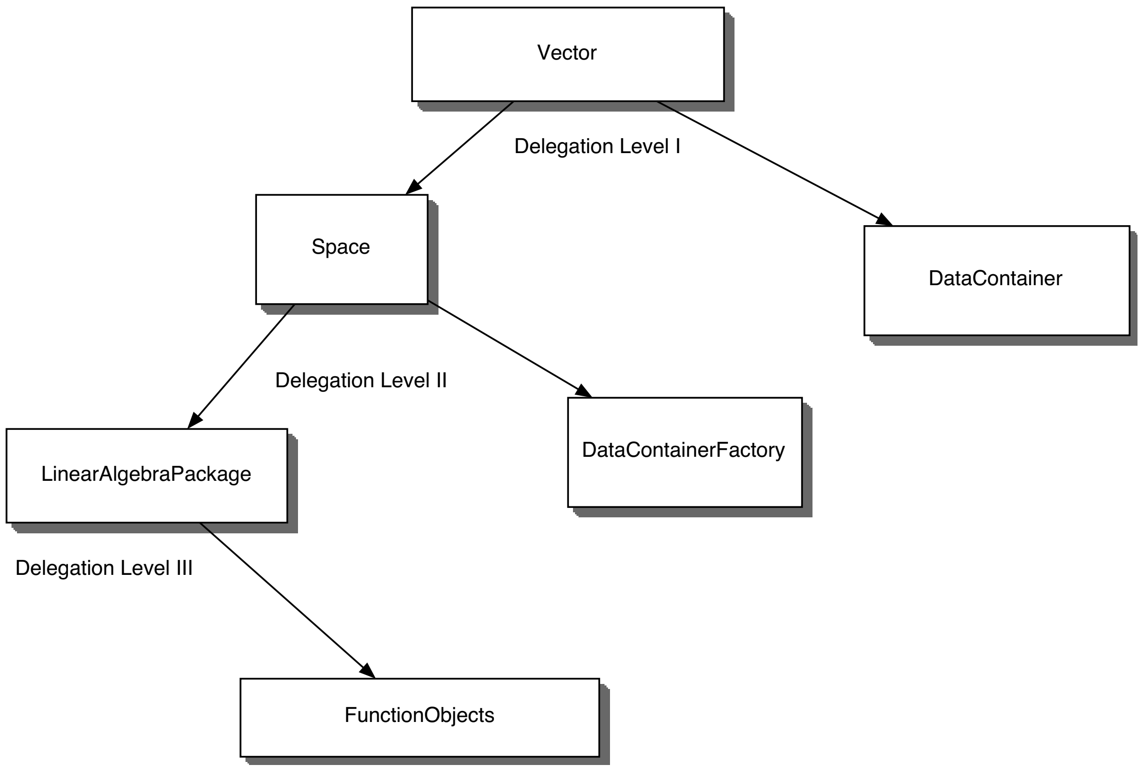

Figure 2.5 Delegation in class *Vector*

The *DataContainer* object encapsulates the data structure and in this manner is classified as a heavy-weight object. Like *Vector*, *Space* is also a *strategy* class that in turn is composed of light-weight objects (Figure 2.6).

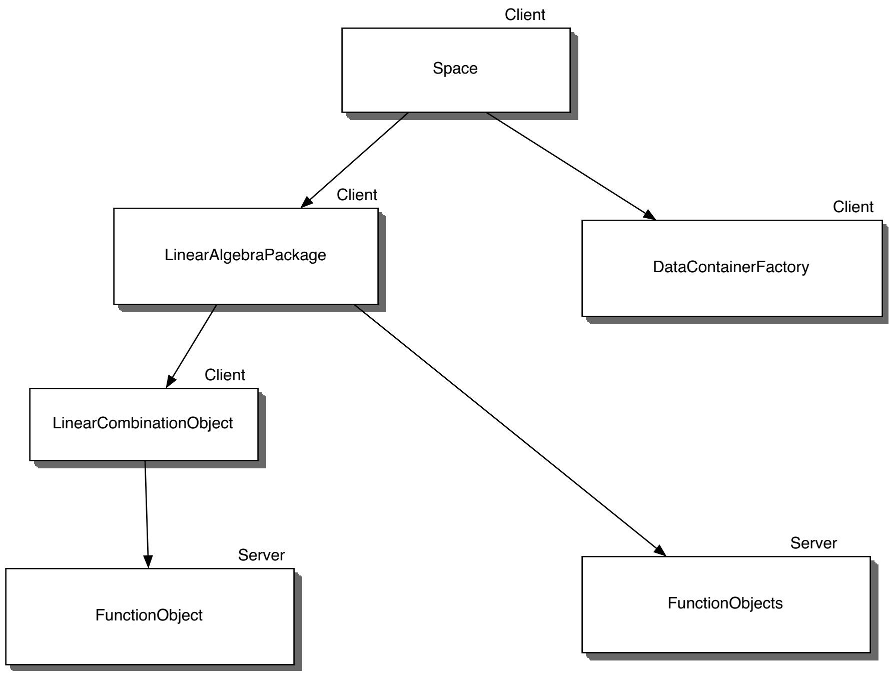

Figure 2.6 *Space Strategy* Design

Space's responsibility in creating *DataContainers* entails some provision in the class to distinguish between remote and local *DataContainer* creation. In order to provide for this flexibility without altering the user interface, we make *Space* a *strategy* class composed of a *DataContainerFactory* and *LinearAlgebraPackage*. According to our first guideline, *Space* and *Vector* are definitive client side objects. According to our second guideline, *DataContainer* is assigned to the server and we substitute a remote object on the client. The remaining objects to consider are *FunctionObjects*, *DataContainerFactory*, and *LinearAlgebraPackage*.

The next step is to analyze object relationship constraints. We seek to minimize dependencies between components. Consequently, we must localize objects that exhibit dependencies with each other. In our simple example, we have the following object dependencies:

- *Vector* relies on *Space* for *DataContainer* construction, hard-wired operations, and compatibility tests.
- *Space* in turn relies on *DataContainerFactory* for appropriate *DataContainer* creation, and *LinearAlgebraPackage* for appropriate operations.
- *DataContainerFactory* only passes (easily transmitted) construction information to a *DataContainer* constructor.
- *LinearAlgebraPackage* is another *strategy* class.
- *DataContainer* and *FunctionObject* work in unison to manipulate data.
- *Vector* and *Space* forward *FunctionObjects* as arguments to *DataContainer* methods.

DataContainers and *FunctionObjects* must be localized to the same component. We can also easily extend the message forwarding from *Vector* to *DataContainers* from a single object domain to distributed object domains by embedding stand-in objects and middle-ware communication calls within these actions. According to guideline one, *LinearAlgebraPackage* is relegated to the client. Because *DataContainerFactory* does not exhibit dependencies on any server side objects, we also assign it to the client.

The beauty of this design is that the distinction between remote (proxy) and local objects is not apparent to the front-end user level. We maintain this characteristic for all objects that interact with the user directly. Notice, that it is the use of the *strategy* design pattern, the process of composition, and subsequent delegation that allows us to hide the added complexity of the component client-server framework from the user. This allows the client component to plug into various servers without any change in code. That is, the user can develop an algorithm, plug it into a serial platform or a parallel platform without having to alter much of the code. This is the wonderful versatility that the client-server component framework admits. With such versatility, we can plug and play prefabricated scientific components.

2.3.2 Example: Partition of FDTD into Components

A more instructive example is the component partitioning of the Finite Difference Time Domain (FDTD) set of objects. The FDTD class employs the *strategy* design pattern as shown in Figure 2.7.

The FDTD object is a composite only of the FDTD associated function objects. We do not see the same *strategy* nesting behavior as that in the *Vector* class. This

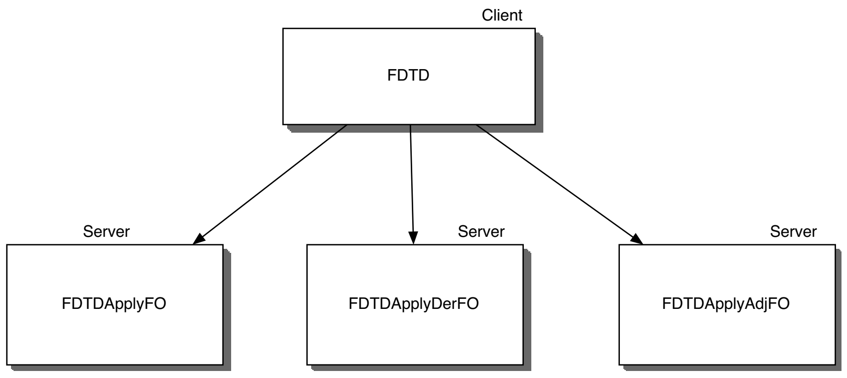

Figure 2.7 *Strategy* Design in FDTD

specifies a clean separation between client and server at the FDTD object level. That is, the FDTD class is relegated to the client, while its associated function objects and their dependencies are assigned to the server. We substitute remote instances of these function objects on the client, creating a new communication layer and extending the behavior of the FDTD object. Although, at lower levels, the FDTD algorithm is rather complex, we do not encounter such complexities when partitioning the FDTD class. The use of the *strategy* design pattern allows component partitioning at higher levels of abstraction, circumventing the need to consider intricate object dependencies.

Chapter 3
Low Level Socket Networking

3.1 Introduction

The essense of distributed, networked computing is the communication protocol. The convenience of high level middle-ware communication packages such as the Common Object Request Broker Architecture (CORBA) induce a cost in the form of processing overhead [5]. A well designed low level socket interface between remote components can outperform one based on such higher level facilities. This is a daunting task however, as socket programming is very primitive, and in some cases requires a well defined separation between interface and implementation.

The design of such a socket interface in SVL, is in a state of constant evolution. *Proxy* and *Remote* objects subsisting on the client delegate function calls to correspondent objects relegated to the server, and are exclusively responsible for inciting communication between client and server. It is the means and the structure of sending minimal amounts of data across the network that encapsulate the difficulty of designing a socket communication protocol.

In *Ideas in Proxy Design for a Client Server Architecture*, Tony Padula presents a coding scheme for communicating instructions and object types from the client to the server [6]. He packs this information as well as other object specific primitive data items into a character buffer and sends it across the network. If the instruction to the server refers to the creation of an object, the slightly unpacked buffer is sent to a specified skeleton. The skeleton's purpose is to unpack the buffer and instantiate the appropriate object. After delving into his design, we see no alternative at this time to such a coding scheme. The server socket interface can be simplified via the annihilation of the skeleton concept. With the implementation of a *Socket* class, we can transport user-defined objects such as a *Grid* over the network bypassing packing and unpacking a buffer.

3.2 The Socket Class

The *Socket* class provides a relatively simple procedure for applications to communicate with one another by wrapping function calls to the low level socket library. Wrapping these functions can miminize the impact of socket programming errors in code, and ensure that such socket calls are used properly. Furthermore, the internals of socket programming are hidden from developers who are not interested in communication protocol.

In his article *Guideline for Wrapping Sockets in Classes*, James Pee presents a suitable interface that follows the C++ iostream method of reading and writing [7]. That is, he reduces the socket reading and writing calls to **operator** << and **operator** >>.

More importantly, we can extend the interfaces to work with user-defined classes. To send an object across the network, the object's class is required to be derived from class *Streamable*. *Streamable* only defines two pure virtual functions, **Marshall** and **UnMarshall**. **Marshall** prepares an object for transport across the socket stream, while **UnMarshall** reassembles the object after transport. Implementing **Marshall** and **UnMarshall** is a rather straighforward procedure as the *Socket* class defines member functions to transport primitive data types.

For example, I derive class *Grid* from class *Streamable* and implement these two methods. Notice, we must serialize containers such as the std::vector object before transport. All objects that are referred to by pointers, must undergo some process of serialization.

```
template<class Scalar>
class Grid : public Streamable {

private:

  int naxes;
  vector<int> n;
  vector<Scalar> d;
  vector<Scalar> o;

  int dim;
  Scalar vol;
  Scalar tol;

  string fname;

public:

   void Marshall (Socket & s) const {

       vector<int>::const_iterator p1;
       typename vector<Scalar>::const_iterator  p2;
       s << naxes;
       for (p1 = n.begin(); p1 != n.end(); p1++)
```

```
      s << (*p1);
    for (p2 = d.begin(); p2 != d.end(); p2++)
      s << (*p2);
    for (p2 = o.begin(); p2 != o.end(); p2++)
      s << (*p2);
    s << dim;
    s << vol;
    s << tol;
}

 void UnMarshall (Socket & s) {
 //same order as Marshall

    vector<int>::iterator p1;
    typename vector<Scalar>::iterator p2;
    s >> naxes;
    if (n.size() != naxes) {  //allocate space
      n.resize(naxes,1);
      d.resize(naxes,1.0);
      o.resize(naxes,0.0);
    }

    for (p1 = n.begin(); p1 != n.end(); p1++)
      s >> (*p1);
    for (p2 = d.begin(); p2 != d.end(); p2++)
      s >> (*p2);
    for (p2 = o.begin(); p2 != o.end(); p2++)
      s >> (*p2);
    s >> dim;
    s >> vol;
    s >> tol;
 }
```

Member functions **Marshall** and **UnMarshall** suit the same purpose as packing elements into a buffer, sending the buffer, receiving the buffer and unpacking the buffer. With these two method calls, we can encapsulate the process of breaking down, sending, receiving, and assembling an object's data members as a behavior of the object. That is, the object is now *streamable*. The major difference between this communication protocol and the one that uses a buffer, is that here we are constantly sending and receiving. A socket connection is opened and information bypasses the process of being packed or unpacked, it is only sent and received. The advantage here is that this procedure is more efficient as we still only open one socket and so incur the same latency period.

3.3 Server Request Encoding

We now have a means of transporting *streamable* user-defined objects. The server, however, still requires some mechanism for deducing what the client requests and what information pertinent to the request the client is transporting. Tony Padula presents a clean-cut, organized encoding/decoding procedure [6]. Basically all calls to Buffer.pack and Buffer.unpack have been replaced with *Socket* << and *Socket* >>, respectively. The server executable consists of various switch statements thereby decoding the clients requests by way of enumerated typing.

3.4 Socket Error Handling Procedure

There are basically two categories of error handling that must be accounted for. The first deals exclusively with socket issues and communication errors. Reading and writing to a socket is not guaranteed. A call to recv/send (read/write) might return prior to reading or writing the requested number of bytes. Within the *Socket* method calls, we check the return codes for errors. This is another advantage of using a socket class.

The second category of error handling deals with verifying that the server is constructing objects and evaluating such objects appropriately. The error handling procedure Tony Padula utilizes is adopted here [6]. After a number of transactions between client and server, the server sends a bool status flag. If status is false, then the client expects to receive a string from the server describing the error in more detail. Another approach to this type of error handling is to make error objects streamable. The latter approach has not been implemented at this time.

Chapter 4
Remote Class Hierarchy

4.1 Introduction

In this chapter, we discuss the classes; *Handle*, *Remote*, *DataContainerProxy*, and *RemoteFunctionObject* generated to extend the behavior of the Standard Vector Library (SVL) to a distributed domain. We describe the process of codifying *Remote* and *DataContainerProxy* objects, enumerating the steps involved in expanding single domain classes to include distributed behavior. Finally, we look behind the scenes at *Remote* object instantiation.

4.2 The *Remote* Class Hierarchy

The *Handle* class suites two purposes. It wraps a unique identifier of an object, and allows object sharing between the client and server processes. We follow Shannon Scott's simple yet suitable identifier mechanism; the object's pointer variable in the server address space casted to an unsigned integer [8]. This *Handle* class encapsulates the unsigned integer identifier on the client, streams the object between the distributed processes, and provides convenience methods to convert the identifier from its client representation to its server representation and vice versa.

The *Remote* class extends distributive behavior to objects that function in a single computing domain. That is, it extends the interface of an object to allow message passing between client and server components through inheritance. The *Remote* class endows its subclass with a *Handle* data member and methods that send and receive this data given an open socket connection, thereby averting direct user access to the *Handle* data member.

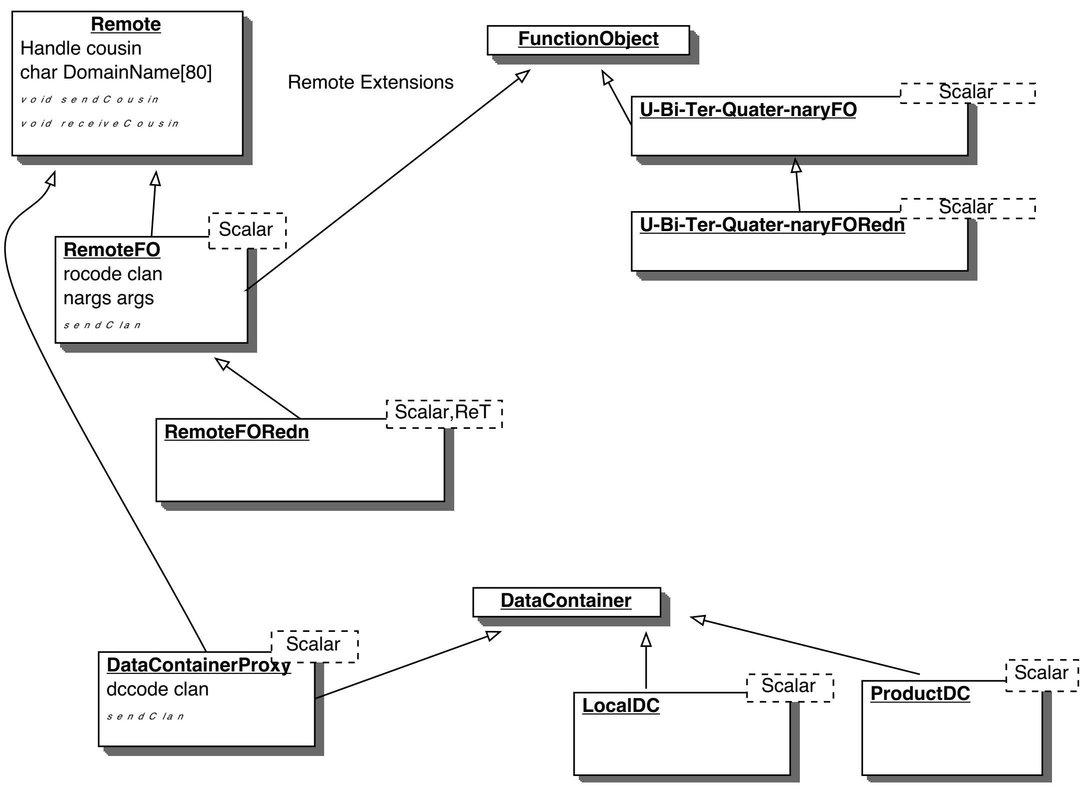

Figure 4.1 Base Class Remote Implementation

All objects that are to maintain dual residency, i.e. a concurrent existence on both client and server processes, are required to have counterpart objects that are subclasses of *class Remote.* In some situations this introduces the only multiple inheritance in the SVL design. Fortunately, the *Remote* class serves the unique purpose of streaming the object identifier between processes and does not conflict with other inherited behaviors.

In SVL, we define three specifications of surrogate objects (Figure 4.2). All such objects inherit from the *Remote* class and are endowed with a streamable identifier. *RemoteFunctionObjects* are *Remote* objects that are understood to be *FunctionObjects* on the server domain. Although on the server domain, *FunctionObjects* are subclassed according to the number of arguments for evaluation: unary, binary, ternary, or quaternary; their remote counterparts are not. Rather each *RemoteFunctionObject* owns an enumerated type data member indicating the number of arguments the corresponding server side object requires for evaluation in the *operator ()* method.

Proxy objects allow complete control of the remote object on the server from the client process. In addition to the streamable functionality endowed by the *Remote* superclass, *Proxy* objects include the same interface as the objects they represent. The SVL design only necessitates that *DataContainer* objects have *Proxy* counterparts on the client domain. These specifications also allow the client to maintain the same relationship between *RemoteFunctionObjects* and *DataContainerProxies* as that between *FunctionObjects* and *DataContainers*.

4.3 Forging the Client Server Contract

The client server contract specifies protocols for each unit of service; object creation, object deletion, function object evaluation, output, and get result. These protocols

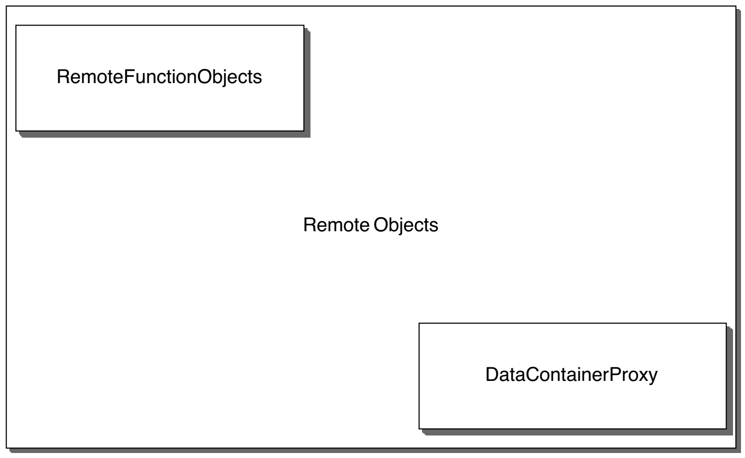

Figure 4.2 Remote Specifications

in turn depend on the type of object on which we execute the appropriate method. All remote objects have an enumerated type data member, known as *clan*, that is used for object registration on the server.

4.3.1 Creation

Depending on the remote object specification, creation of a *Remote* or *Proxy* object entails the following steps:

- Subclass of *Remote*
  - Object Registration: addition of object code name to *rocode* enumerated data type located in file **rocodes.H**
  - Constructor: socket stream fabrication and block of communication code that matches server construction block in file **createro.H**.
- Subclass of *RemoteFunctionObject*

  - Object Registration: addition of object code name to *rocode* enumerated data type located in file **rocodes.H**
  - Constructor: socket stream fabrication and block of communication code that matches server construction block in file **createro.H**. This step is required if the type of *RemoteFunctionObject* object requires data parameters to be shipped to server for counterpart object construction.

- Subclass of *class DataContainerProxy*
  - Object Registration: addition of object code name to *dccode* enumerated data type located in file **dccodes.H**
  - Constructor : socket stream fabrication and block of communication code that matches server construction block in file **createdc.H**. This step is required if the type of *DataContainerProxy* object requires data parameters to be shipped to server for counterpart object construction.

4.3.2 Deletion

All *Remote* objects follow the same protocol for deletion. After socket stream instantiation, we require communication codes that match the server deletion block in the case/switch statement in file **deleteRO.H**. The default case applies to generic deletion of *RemoteFunctionObjects*.

4.3.3 Function Object Evaluation

This service applies only to a *DataContainerProxy* calling one of its *eval* methods with a *RemoteFunctionObject* passed as an argument. Communication is completely the responsibility of the *DataContainerProxy*, which sends the appropriate *handles* to the server request broker. The server expects to receive the *handles*, converts them to generic *DataContainers* and *FunctionObjects*, and employs the *visitor* design pattern.

4.3.4 Output

The output function expects to receive a *Handle* from the client side object. It converts this *handle* to a generic object and executes its *write* method. This assumes, of course, that all objects in SVL have a *write* method.

4.3.5 Get Result

This service can only be initiated from a *RemoteFunctionObjectRedn*. This client side object streams its *Handle* and receives appropriate result information from the server.

4.4 Client Side Communication Templates

In the following, “s” is a *Socket* object, “clan” refers to type of *Remote*, “cousin” is a *Handle* object, and “status” is a *Bool* type. Note the Destuctor template and the getResult template are the same for all objects and reduction function objects, respectively.

Default Contructor:

s << CreateROCode (CreateDCCode);

s << clan;

s >> status;

receiveCousin(s);

Destructor:

s << DeleteObjectCode;

sendCousin(s);

s << CreateROCode; or s << CreateDCCode;

s >> status;

getResult:

s << GetResultCode;

sendCousin(s);

s >> status;

s >> result;

Output:

s << OutputCode;

sendCousin(s);

s >> status;

4.5 Behind the Scenes: *Remote* Object Construction

The intricacies of *Remote* object creation are completely hidden from the user. This process includes communication and execution of a server function. Figure 4.3, traces the procedure for *Remote* object construction.

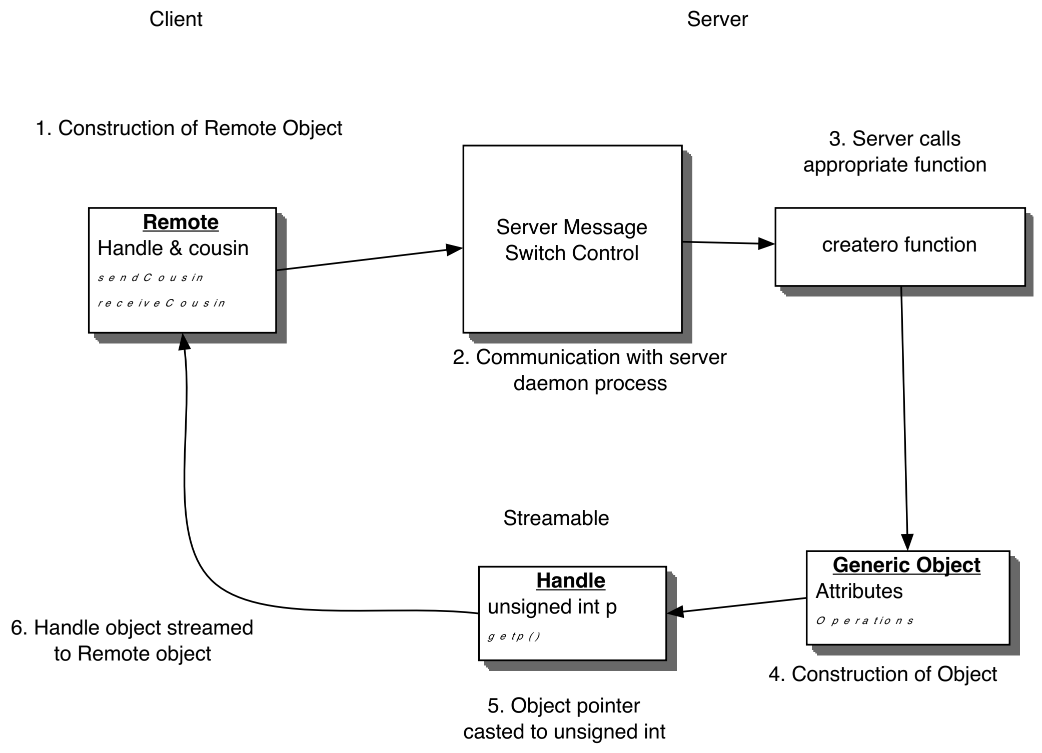
Client
Server
1. Construction of Remote Object
Remote
Handle & cousin
sendCousin
receiveCousin
Server Message Switch Control
2. Communication with server daemon process
3. Server calls appropriate function
createro function
Generic Object
Attributes
Operations
4. Construction of Object
Streamable
Handle
unsigned int p
getp()
5. Object pointer casted to unsigned int
6. Handle object streamed to Remote object

Figure 4.3 Distributed Object Construction

We see the following six steps:

- 1. Instantiation of *Remote* object on the client.

- 2. Within the object constructor, we open a connection between client and server, and stream the appropriate instruction command. In this case, we request object creation from the server object request broker.

- 3. The server request broker in turn invokes the appropriate callback function, passing the socket stream to this function.
- 4. The callback function receives the enumerated registry variable specifying what type of object to create. Using switch control, it calls the appropriate object constructor.
- 5. The callback function converts the object pointer to a *Handle* object.
- 6. The server request broker streams the *Handle* object back to the *Remote* object that initiated this procedure.

Chapter 5
Applications

5.1 Introduction

This chapter illustrates two examples of our client-server implementation: the Euler ODE Solver and the Acoustic PDE Solver. These examples are instances of the Finite Difference Time Domain (FDTD) set of objects in the Standard Vector Library (SVL). We outlined the client and server component partitions of FDTD in chapter two. Since FDTD function object construction requires the prequisite creation of objects, we encapsulate precursor object creation within one object, the *Environment*. Consequently, we require only one communication call for FDTD function object creation. In this way, we minimize communication between client and server components.

5.2 General Description of Euler FDTD ODE Solver

We consider the initial value problem:

$u'(t) = c. * u(t)$

$u(t_0) = u_0$

The Euler method for numerically approximating the solution to this O.D.E. is encapsulated by the right hand side class, *testrhs::operator ()* method.

$u(t_{i+1}) = u(t_i) + c. * u(t_i)dt$

This example demonstrates the general construction of an *FDTD* object. Upon each call to the *FDTDApplyFO* object, one of the objects that FDTD is composed of, we see the following system of delegation (Figure 5.1).

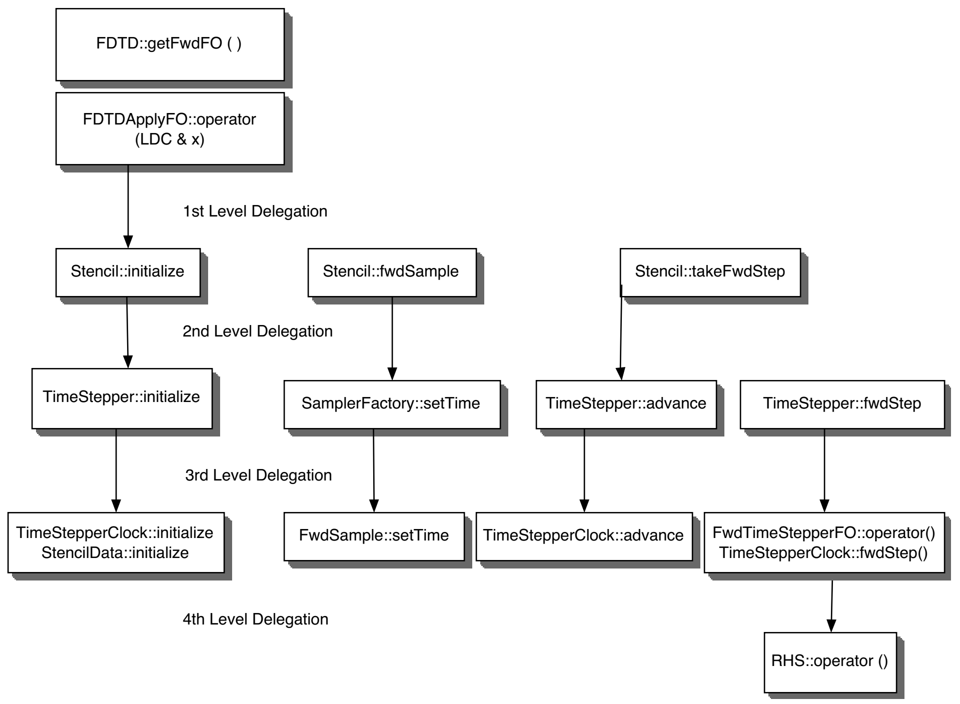

Figure 5.1 Delegation in Euler FDTD

We trace delegation starting at the FDTD function object. This is the only object in this scope to which the user has access. Subsequently, the remaining objects listed in the delegation system are server side only. Each object that uses delegation contains the object to which it delegates as a data member. In this fashion, all objects that delegate are structurally composites of other objects. With respect to construction of an FDTD object in the client-server domain, we must build these objects on the server in the opposite order of delegation.

The Figure shows four groups of objects based on the level of delegation. From bottom up, we have the following group members.

- I.
  - RHS (*testrhs*)
- II.
  - FwdSample
  - TimeStepperClock (*ConstTimeStepperClock*)
  - StencilData (*ODEStencilData*)
  - FwdTimeStepperFO (*EulerFwd*)
- III.
  - TimeStepper (*TimeStepper*)
  - SamplerFactory (*FVSamplerFactory*)
  - SamplingData (*TrivialSamplingData*)
- IV.
  - Stencil (*Stencil*)

On the server, the construction order of an *FDTDApplyFO* must follow the group order above, with all group I. objects created prior to group II., etc. This construction order is encapsulated in the appropriate *Environment* object. The client side *RemoteEnvironment* transmits the user supplied data parameters to the server side *Environment* for server side object construction, and creates client side FDTD object for user perusal.

```
template <class Scalar>
class EulerEnvironment {
private:

    Scalar tbeg, tend, dt;
    int n, itmax;


    testrhs<Scalar> f;
    EulerFwd<Scalar> fwd;
    ODEStencilData<Scalar> sd;
    ConstTimeStepperClock<Scalar> clk;
    TimeStepper<Scalar> ts;
    FVSamplerFactory<Scalar> samfac;
    TrivialSamplingData<Scalar> samdat;
    StencilRn<Scalar> sten;
    FDTDApplyFO<Scalar> fwdFO;
    FDTDApplyDerFO<Scalar> derFO;
    FDTDApplyAdjFO<Scalar> adjFO;


    EulerEnvironment();

public:

    EulerEnvironment(testrhs<Scalar> & _f,  Scalar _tbeg,
      Scalar _tend, Scalar _dt,
                                       int _n, RnArray<Scalar> & u0,
                                       int _itmax = 1000 )
```

```
        : tbeg(_tbeg), tend(_tend), dt(_dt), n(_n),
        f(_f), fwd(f,tbeg,dt), sd(n,n,u0), itmax(_itmax),
        clk(tbeg,tend,dt), ts(fwd,sd,clk),
        samfac(n,tbeg,tend),
        samdat(),
        sten(ts,samfac,samdat),
        fwdFO(sten,itmax),
        derFO(sten,itmax),
        adjFO(sten,itmax) {}



    FunctionObject *getFwd() {return &fwdFO;}
    FunctionObject *getDer() {return &derFO;}
    FunctionObject *getAdj() {return &adjFO;}

    virtual ~EulerEnvironment () {}

};


template <class Scalar>
class RemoteEulerEnvironment : public Remote {
private:

    RemoteEulerEnvironment();
    FDTDProxy<Scalar> meth;
    RnDataContainerProxyFactory<Scalar> dcfac;
    CSSpace<Scalar> sp;
    FDTDOp<Scalar> *fdtdop;
    RemoteFunctionObject<Scalar> rhs;



public:

        RemoteEulerEnvironment(RemoteFunctionObject<Scalar> & _rhs,
                               Scalar tbeg, Scalar tend,
                                Scalar dt, int n, int itmax = 1000 )
        : rhs(_rhs), dcfac(n), sp(dcfac)
    {
        Socket s;
        s.Connect();
        bool status;
        s << CreateROCode;
        s << EulerEnvironmentCode;
        rhs.sendCousin(s);
        s << tbeg;
        s << tend;
        s << dt;
        s << n;
        s << itmax;
        meth.fwd.receiveCousin(s);
        meth.der.receiveCousin(s);
        meth.adj.receiveCousin(s);
        s >> status;
        receiveCousin(s);
        s.Close();

        fdtdop = new FDTDOp<Scalar> (sp, sp, meth);

    }
s
    FDTD & getFDTD() {return meth;}
    FDTDOp<Scalar> & getOp() { return *fdtdop;}

    virtual ~RemoteEulerEnvironment(){
        Socket s;
        s.Connect();
        bool status;
        s << DeleteObjectCode;
        s << CreateROCode;
        sendCousin(s);
        s << EulerEnvironmentCode;
        s >> status;
        s.Close();
    }


};
```

5.3 General Description of the Acoustic FDTD PDE Solver

The Acoustic FDTD Solver has one level of delegation more than the Euler FDTD Solver. We create an *Environment* object in the same way.

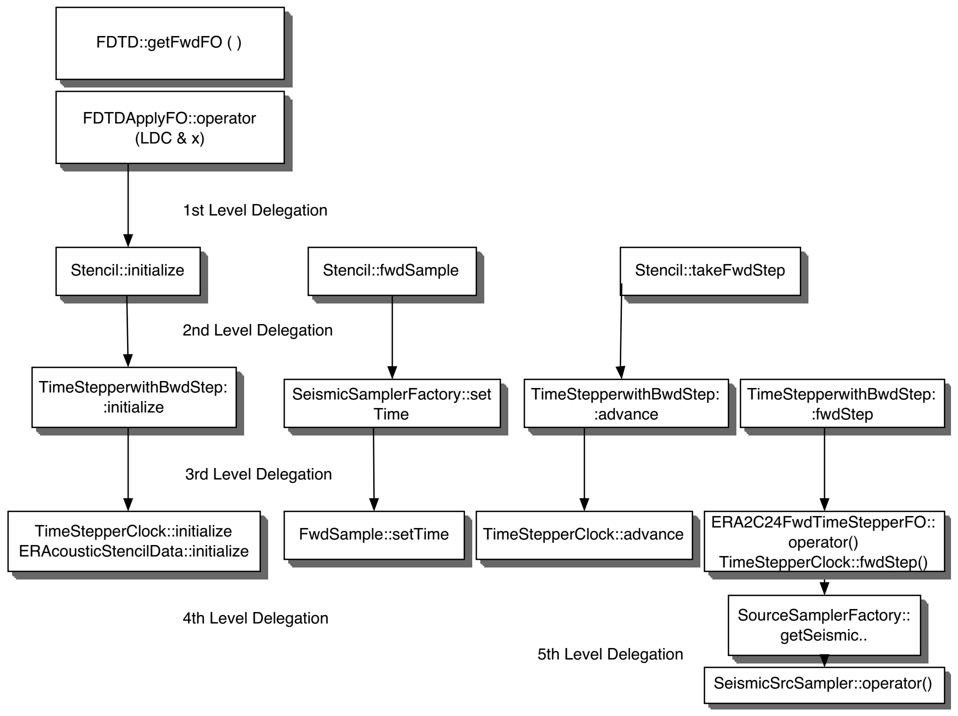
FDTD::getFwdFO ( )
FDTDApplyFO::operator (LDC & x)
1st Level Delegation
Stencil::initialize
Stencil::fwdSample
Stencil::takeFwdStep
2nd Level Delegation
TimeStepperwithBwdStep::initialize
SeismicSamplerFactory::setTime
TimeStepperwithBwdStep::advance
TimeStepperwithBwdStep::fwdStep
3rd Level Delegation
TimeStepperClock::initialize
ERAcousticStencilData::initialize
FwdSample::setTime
TimeStepperClock::advance
ERA2C24FwdTimeStepperFO::operator()
TimeStepperClock::fwdStep()
4th Level Delegation
SourceSamplerFactory::getSeismic..
5th Level Delegation
SeismicSrcSampler::operator()

Figure 5.2 Delegation in Acoustic FDTD

- I.
  - *SeismicSrcSampler*
  - Set of Fortran routines encapsulating the numerical algorithms
- II.
  - *SourceSamplerFactory*

- III.
  - FwdSample
  - *ConstTimeStepperClock*
  - *ERAcousticStencilData*)
  - *ERA2CC24FwdTimeStepperFO*
- IV.
  - *FwdTimeStepperWithBwdStep*
  - *SeismicSamplerFactory*
  - *AcousticSamplingData*)
- V.
  - Stencil (*Stencil*)

```
class A2C24Environment {

private:

    A2C24ParamHolder names;
    SeismicDataBase src_db;
    Grid<float> ctrlgrid;
    AcousticGridPar2D simpar;
    SeismicDataBase seisdb;
    float tinit;
    float tfinal;
    int nts;
    SeismicSrcSampler srcsam;

    /* srcfac, clock, stendat do not require
        any additional object creations for construction */

    SourceSamplerFactory srcfac;
    ConstTimeStepperClock<float> clock;
    AcousticStencilData stendat;

    A2C24FwdTimeStepperFO tsfo;
    A2C24BwdTimeStepperFO btsfo;
    A2C24DerTimeStepperFO dtsfo;
    A2C24AdjTimeStepperFO atsfo;

    TimeStepperWithBwdStep<float> ts;
    SeismicSamplerFactory samfac;
    AcousticSamplingData samdat;

    StencilRn<float> sten;
    FDTDApplyFO<float> fwdFO;
    FDTDApplyDerFO<float> derFO;
    FDTDApplyAdjFO<float> adjFO;

    A2C24Environment();
    A2C24Environment( const A2C24Environment & v );

public:

        A2C24Environment(string parfile)
        :  names(parfile),
        src_db((char*)(names.getSourceHeaders().c_str()),
               names.getAllKeys()),
        ctrlgrid((char*)(names.getVelocityGrid().c_str())),
        simpar(names.getCmin(),
               names.getCmax(),
               names.getFmax(),
               ctrlgrid.get_o(0),
               ctrlgrid.get_o(0) +
               (ctrlgrid.get_n(0) - 1)*
               ctrlgrid.get_d(0),
               ctrlgrid.get_o(1),
               (ctrlgrid.get_n(1) - 1)*
```

```
       ctrlgrid.get_d(1),
       names.getCFL(),
       names.getGridPointsPerWaveLength()),
seisdb((char*)(names.getSeismicHeaders().c_str()),
       names.getAllKeys(),
       names.getBinKey(),
       names.getNumberOfBins(),
       names.getFirstBinLocation(),
       names.getBinStep(),
       names.getBinStepTol()),
tinit((float) min(src_db.getTimeOfFirstSample(),
          seisdb.getTimeOfFirstSample())),
tfinal((seisdb.getNumberOfTimeSamples()-1)*
       seisdb.getTimeStep()+
       seisdb.getTimeOfFirstSample()),
nts(1+int((tfinal-tinit)*simpar.get_dtrecip()+0.01)),
srcsam(names.getSourceKeys(),
       src_db,
       nts,
       simpar.get_dt(),
       tinit,
       simpar.getGrid(),
       names.getSourceBin()),
srcfac(names.getSourceKeys(),
       seisdb,
       srcsam),
clock(tinit,
      tfinal,
      simpar.get_dt()),
stendat(simpar,
        names.getGridZeroLevel()),
tsfo(simpar,
     srcfac,
     samdat),
btsfo(tsfo),
dtsfo(simpar,
      srcfac,
      samdat),
atsfo(simpar),
ts(tsfo,
   btsfo,
   dtsfo,
          atsfo,
          stendat,
          clock),
       samfac(names.getReceiverKeys(),
              nts,
              simpar.get_dt(),
              tinit,
              simpar.getGrid(),
              seisdb),
       samdat(simpar),
       sten(ts,samfac,samdat),
       fwdFO(sten,1000),
       derFO(sten,1000),
       adjFO(sten,1000) {

    try {
       cerr<<"initialize internal source data in src sampler"<<endl;
       // set up source
       SeismicDataContainer srcdc(src_db);
       SeismicOpenFile of(names.getSource());
       srcdc.eval(of);
       AdjSamplerLoader<float> sld(srcsam);
       srcdc.eval(sld);

    }
    catch (SVLException & e) {
       e<<"\ncalled from A2C24Environment constructor\n";
       throw e;
    }
  }



    ~A2C24Environment() {}

    FunctionObject *getFwd() {return &fwdFO;}
    FunctionObject *getDer() {return &derFO;}
    FunctionObject *getAdj() {return &adjFO;}
    Grid<float> *getGrid() {return &ctrlgrid;}
    SeismicDataBase *getSeisdb() {return &seisdb;}
    virtual void write(SVLException & e)
    {
        e<<"A2C24Environment Object\n";
        e<<"Parameters:\n";
        names.write(e);
    }

    virtual ostream & write(ostream & e)
    {
        e<<"A2C24Environment Object\n";
        e<<"Parameters:\n";
        return e;
    }

};


class RemoteA2C24Environment : public Remote {
private:

    RemoteA2C24Environment();
    FDTDProxy<float> meth;
    Handle grid, seisdb;
    DataContainerFactory *gridfac, *seisfac;
    CSSpace<float> *ctrlsp, *datasp;
    FDTDOp<float> * fdtdop;

public:

        RemoteA2C24Environment( string parfile)
    {
            StreamString file(parfile);
            Socket s;
            s.Connect();
            bool status;
            s << CreateROCode;
            s << A2C24EnvironmentCode;
            s << file;
            meth.fwd.receiveCousin(s);
            meth.der.receiveCousin(s);
            meth.adj.receiveCousin(s);
            s >> grid;
            s >> seisdb;
            s >> status;
            receiveCousin(s);
            s.Close();

            gridfac = new GridDataContainerProxyFactory<float> (grid);
            seisfac = new SeismicDataContainerProxyFactory (seisdb);
            ctrlsp = new CSSpace<float> (*gridfac);
            datasp = new CSSpace<float> (*seisfac);
            fdtdop = new FDTDOp<float> (*ctrlsp, *datasp, meth);
    }


    virtual ~RemoteA2C24Environment(){

        if (gridfac) delete gridfac;
        if (seisfac) delete seisfac;
        if (ctrlsp) delete ctrlsp;
        if (datasp) delete datasp;
        if (fdtdop) delete fdtdop;

        Socket s;
        s.Connect();
        bool status;
        s << DeleteObjectCode;
        s << CreateROCode;
        sendCousin(s);
        s << A2C24EnvironmentCode;
        s >> status;
        s.Close();
    }

    FDTD & getFDTD() { return meth; }

    FDTDOp<float> & getOp() { return *fdtdop;}


};
```

Chapter 6
Closing Remarks

By using the process described here, we can extend previously written object-oriented software to a distributed client-server component architecture without altering the kernel. This mechanism entails the use of the *proxy*, *visitor*, and *strategy* design patterns. The client-server model defines complete separation between user interface and computational implementation. The separation of objects into predefined client and server roles allows platform complexities to exclusively remain with the server. In this way, low level details do not propagate to the abstract user level on the client. We achieve simplicity from the high level of abstraction in the SVL software design embedded within the component process. The client-server component framework serves as an environment for coupling prefabricated components. By developing a very simple communication backbone, we suffer very little loss in efficiency due to message passing between components. In essence, client and server represent a trade-off between type complexity and environment complexity. The client expresses the former, and the server expresses the latter. The use of the *strategy* design pattern allows component partitioning at higher levels of abstraction.

Chapter 7
An Appendix

7.1 SVL Client-Server Files

7.1.1 Communication Core

- socket.H: Defines class *Socket* which wraps all socket calls, and overloads iostream operators to allow for less strenuous socket programming.
- streamable.H: Defines class *Streamable* which supplies an interface for other classes to be "streamable" across the socket network.
- socketstring.H: Defines class *StreamString* which makes a string streamable.
- remote.H: These classes comprise the core mechanism for extending SVL kernel classes to the client-server environment. They specify the interfaces for remote objects. Classes defined here include *Remote, RemoteFunctionObject, RemoteFunctionObjectRedn*, and *DataContainerProxy*.
- handle.H: Defines "streamable" class *Handle* which wraps server side object identifiers (casted pointers on server side address space).
- rocodes.H: Enumerated list of all objects other than *DataContainer*s that are to have remote representation.
- dccodes.H: Enumerated list *DataContainer*s objects that are to have remote

representation.

- messages.H: Enumerated list of server services. Should never need to be expanded or modified.

- codes.H: Inclusion of all enumerated list files above.

7.1.2 Client Specific Files

- spaceproxy.H: Definition of class *SpaceProxy* which takes a *DataContainerProxyFactory* as a constructor argument. This class's *SVLLinearAlgebraPackage* data member is hard-wired for modular space proxy construction.

- rnproxy.H: Defines the remote counterpart class *RnDataContainerProxy* to class *RnDataContainer*.

- gridproxy.H: Defines the remote counterpart class *GridDataContainerProxy* to class *GridDataContainer*

- seismicproxy.H Defines the remote counterpart class *SeismicDataContainerProxy* to class *SeismicDataContainer*.

- remotefunctions.H: Defines remote counterparts to all function objects defined in file **functions.H**.

- remoteEuler.H: Defines classes *EulerEnivronment* and *RemoteEulerEnvironment* for a client server based FDTD Euler ODE solver.

- remoteAcoustic.H: Defines classes *A2C24Environment* and *RemoteA2C24Environment* for a client server based acoustic PDE solver.
- remoteExt_Acoustic.H: Same as above except we are using the external A2C24 acoustic application.

7.1.3 Server Specific Files

- server.C: Server executable that spawns client requests to either function *createro*, *createdc*, or *deleteRO*.
- exceptServ: Defines an exception class specific to socket calls and server requests.
- createro.H: Defines the function textitcreatero which includes a switch statement based on enumerated list defined in **rocodes.H** to construct appropriate kernel object.
- createdc.H: Defines the function textitcreatedc which includes a switch statement based on enumerated list defined in **dccodes.H** to construct appropriate kernel object.
- deleteRO.H: Defines the function *deleteRO* for appropriate kernel object deletion.

7.2 SVL Parallel Server Protocol

This comprises a general walk-through of the command sequences and logical threads of the parallel server. One of the guiding design principles of the parallel server is that we maintain all parallel specific, MPI, commands exclusively within the server component. In this manner, MPI calls are not pervasive in any of the client data container and function objects. The instruction switch control paths mimic that of the serial server.

The parallel server is a master/slave design embedded within one SPMD (single program, multiple processes) program with provisions to provide an independent master thread. This is by no means the final design. It is merely the first prototype and uses simplified logical calls. Later we will expand functionality with the addition of more classes and/or functions.

7.2.1 Breakdown of Services

In all parallel server function calls, the master server node packs creation and instantiation information into a character buffer, and then broadcasts this MPI_PACKED buffer to the remaining slave nodes. When creating an object, the master node gathers (MPI_Gather) the unsigned long casted pointers from the drone processes into an unsigned long integer array. The client receives from the master node a handle to this array. When evaluating a function object, getting the result from a function object, or deleting an object, the master node scatters (MPI_Scatter) these casted pointers to the appropriate processes. We are able to exploit the ordering properties of the MPI_Gather and MPI_Scatter routines, which makes for less complicated coding. Listed in Instruction Order:

1. Creation of Data Container Object
2. Creation of Function Object
3. Evaluation of Function Object
4. Get Result from Reduction Function Object
5. Deletion of Object

7.2.2 Creation of Data Containers and Remote Objects

We assume that parallelization is achieved by distribution of data across the processors. Auxilary parallel server functions: masterdc.H, slavedc.H, masterfo.H, and slavedo.H are employed for appropriate master/slave responsibilities of creation of objects. All function objects that the client requests for creation are basically serial. We create these identical function objects on all processes. The only "parallel" specific function object creation provision entails assignment of the "res" variable for all reduction function objects. Only the master node keeps an updated residual value, while the slave nodes maintain this value at zero. Parallel functions do exist, but the client has no knowledge of them. These truly parallel functions, such as the one to be discussed later in Part II of this report, are only employed by the server.

7.2.3 Evaluation of Function Objects

As described above, this procedure entails the scattering of objects' addresses to the appropriate processes and having each process serially evaluate the function objects.

7.2.4 Get Result from (Reduction) Function Object

The master node calls an MPI_Reduce function and sends the result to the client.

7.2.5 Delete an Object

Much like the evaluation protocol, this procedure involves scattering of the objects' addresses, and forwarding deletion request to all other drone nodes.

7.2.6 Example: Construction of Distributed Grid Object

We assume that the Grid object construction parameters file and binary Grid data file are located on the master server node only.

1. **Client**

    (a) Creates Grid Data Container Proxy using file name constructor. This instantiates socket request to master.

2. **Master**

(a) Accepts socket connection and receives instruction information from Client, which is to create a Grid Data Container.

(b) Receives construction information. In this instance, this is the file name that exists on master hard disk.

(c) Creates global Grid object from given file name.

(d) Loads data from binary file by instantiating a Grid Data object and evaluating a Grid Load function upon it.

(e) Extraction of global Grid parameters from global Grid object.

(f) Packs instruction information and global Grid construction parameters into character buffer using MPI_Pack

(g) Broadcasts this character buffer to drones.

(h) Creates SVLScatter function object. This is the first and so far only MPI function created. it is a Unary Function Object that scatters an RnArray to data container objects on the server. This function can be utilized by any data container. Upon construction, however, it requires the full array to be scattered. With our current assumption that all data to be distributed is relegated to the master server, this function is extremely straight forward.

(i) Conversion of global Grid parameters to local Grid parameters. For the moment this is hard coded, later we should invoke a function that allows partitioning diversity.

(j) Creation of Grid Data Container by passing local Grid object to constructor.

(k) Grid Data Container evaluates SVLScatter function for data distribution.

(l) Handle returned to main parallel server program.

(m) Gathers all Handles from drones.

3. **SLAVES**

   (a) Receives buffer broadcasted from master.

   (b) Unpacks buffer and extracts global Grid parameters.

   (c) Creates SVLScatter function, only supplying local dimension.

   (d) Conversts global Grid parameters to appropriate local Grid parameters.

   (e) Constructs local Grid object.

   (f) Constructs Grid Data Container from local Grid object.

   (g) Evaluates SVLScatter function on Grid Data Container.

   (h) Returns Handle to main parallel server program.

References

1. R. Bartlett. RTOp: A Proposed Specification for User Defined Vector Reduction and Transformation Operators. 2001.

2. B. Beej. Beej's Guide to Network Programming. www.ecst.csuchico.edu/ beej/guide/net/.

3. S. et all Benson. TAO Users Manual. Technical Report ANL/MCS-TM-242-Revision 1.5, Argonne National Laboratory, Argonne, Illinois 60439, January 2003.

4. Eric Gamma et all. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, New York, 1995.

5. Object Management Group. Documents and Technology. www.omg.org.

6. A. Padula. Ideas in Proxy Design for a Client-Server Architecture. May 2002.

7. J. Pee. Guidelines for Wrapping Sockets in Classes. *C/C++ Users Journal*, November 2001.

8. S. D. Scott. Software Components for Simulation and Optimization. Master's thesis, Rice University, Houston, TX 77006, 2001. Also available at Rice University, Department of Computational and Applied Mathematics.

9. S.D. Scott and W.W. Symes. Design of Vector Classes: A Counterproposal to Roscoe Bartlett's Proposed Standard. *TRIP Annual Report*, 2000.

10. C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley, New York, 1999.