Jay Almarode

Mechanisms for application partitioning

MY LAST COLUMN described the differences between client Smalltalk systems and server Smalltalk, and how server Smalltalk fits into the three-tier architecture that is emerging to meet the performance and business requirements of enterprise-wide applications. The key to balancing the processing load between clients and server, and sharing business objects in such architectures, is the ability to partition applications.

Application partitioning is the activity in which code written for the client can be moved to the server (or vice versa). When both the client and server can execute the same Smalltalk code, this movement of objects and code is much simpler and allows the application to be dynamically tuned in the face of changing hardware and software. Applications can be developed initially only on the client, and then portions can be moved to the server to share objects, enforce security policy, and gain fault tolerance of critical data as needed.

When the clients and server speak a different language, this partitioning should occur earlier in the design, because partitioning decisions are more difficult and costly to change later. Unfortunately, performance tuning often occurs late in the software process so, in many cases, the decision to repartition an application must balance the cost of reimplementing large sections of code against the expected performance gain. Making such changes is discouraged because the cost of repartitioning is higher. Also, developers must be proficient in two different languages, one for client development, and one for the server.

When Smalltalk is the language on both the client and server, what mechanisms are available to partition the application and to distribute objects and behavior between the client and server? In GemStone Smalltalk, there are several mechanisms available so that client applications can reference and manipulate objects located on the server. One mechanism is *forwarders*. A forwarder is a client object that covers for a server object. A forwarder does not contain any state of the server object, but maintains enough information to communicate with the server object when needed. When a message is sent to a forwarder, the execution of its behavior actually takes place on the server. The forwarder knows the identity of the server object and how to communicate with it.

Forwarders are implemented in such a way that no special code is required to check for the presence of a forwarder before sending it a message. Forwarders utilize Smalltalk's message-sending mechanism to automatically forward messages by special handling of the doesNotUnderstand: error. The Forwarder class does not inherit from class Object, so forwarders understand very few messages on the client side. Most messages to a forwarder are silently trapped by the execution thread on the client, forwarded to the server for execution, and the result returned to the client for continuation of its execution thread. If the message to a forwarder contains arguments, those arguments are transformed automatically into server objects if needed. This is implemented in such a way so that application code does not have to be written any differently, whether the receiver of a message is a forwarder or some other client object.

There are several ways a programmer can get a forwarder to a server object. One way is to send the message beForwarder to a replicate (discussed later). For example, a newly created client object could be copied to the server by sending it the message putInGS (thus, making it a replicate), then send the message beForwarder. At that point, the state of the object is only stored in the server Smalltalk, and any messages sent to the client object cause execution in the server. In some cases, a developer may design the application so that all instances of a particular client class are intended to be forwarders. Whenever an instance of such a class is fetched from the server, it should be instantiated as a forwarder. This is specified by implementing the method instancesAreForwarders in the client class to return true. Two other ways to specify that certain objects are to be manifested as forwarders in the client Smalltalk is by a *replication specification* (described later) or by a *connector*. A connector is a mechanism connecting certain client objects with certain server objects at the time the client logs into the server. There are many different kinds of connectors: some that connect classes and some that connect class variables; there are those that connect class instance variables; those that connect objects by name; and those that connect objects by identity. Each type of connectors allows a developer to specify that the client object is to be manifested as a forwarder.

Using Smalltalk since 1986, Jay Almarode has built CASE tools, interfaces to relational databases, multi-user classes, and query subsystems. He is currently a Senior Software Engineer at GemStone Systems Inc., and can be reached at almarode@slc.com.

Another mechanism to manipulate server objects on the client is with *replicates*. A replicate is a copy of a server object that resides on the client. Some or all of the state of the server object is copied in the replicate, and when a message is sent to a replicate, the execution of its behavior takes place on the client. Using replicates requires that a mapping be defined between classes on the client and classes on the server. At its simplest, this mapping can specify that a server class maps to a client class with the same name. This is the default mapping. You can also specify more complex mappings as objects are translated between client and server. This is done by reimplementing the instVarMap method in the class of the replicate. This method should return nested arrays, where each sub-array contains an instance variable name and a specification of how it should be mapped. This allows a developer to handle reordering, renaming, or omission of instance variables when an object moves from one domain to the other.

A key consideration when using replicates is the amount of synchronization that occurs between the client replicate and its corresponding server object. There are messages available to the application developer to explicitly manage keeping the two objects in sync. However, it is much easier to let the interface layer that manages replicates be responsible for keeping the state of client and server objects in sync. In this way, a replicate always accurately reflects the state of the server object (based on the current transaction's point of view), wherever the object is used in the application. This level of synchronization, called *full transparency*, is configurable by class. To enable full synchronization for a class, send it the message makeGSTransparent.

Modifications to replicates are flushed to the server before any server behavior is executed.

When replicates are used in full transparency mode, then modifications to replicates are managed automatically. When the application modifies a replicate, it is automatically marked dirty and changes are flushed to the server at the appropriate time. For example, modifications to replicates are flushed to the server before any server behavior is executed or when the transaction is committed. When other users modify and commit changes to server objects, those changes are not seen in the replicate until the current transaction is committed or aborted. At this time, the replicate is eligible to have its state updated from the server, a behavior called *faulting*. Ordinarily, the replicate will not be faulted until it is next accessed. However, this default behavior can be overridden by implementing a method called faultPolicy for the class of the replicate. This method should return #immediate if the replicate should be faulted immediately when the next transaction begins. It is also possible to cause additional application code to be executed before or after the replicate is faulted by implementing a preFault and postFault method.

An important consideration when programming with replicates is how to control the replication of composite objects (objects with nested subobjects). Some client applications may only need a portion of the state of the server object, so why send more to the client than is needed? When a replicate is being instantiated from a server object, an application wants to control which instance variables are retrieved and in what form the objects referenced by those instance variables are created (as forwarders or replicates). In addition, if the instance variable is assigned a replicate, the application may also want to specify how many levels deep to replicate. To exercise this control, a developer implements the replicationSpec method for the class of the replicate. This method returns nested arrays where each subarray contains the name of an instance variable and a specification of how it is to be instantiated. The developer has the option to specify whether the instance variable is to be instantiated as a replicate, a forwarder, or a stub (discussed later). If the instance variable is to be created as a replicate, the developer can specify a minimum or maximum number of levels to replicate as well. The following example shows the implementation of the replicationSpec method for class Employee, where the name instance variable is replicated, the address instance variable is replicated to at least level 2, and the department instance variable is created as a forwarder.

```
classmethod: Employee

replicationSpec

"Return nested arrays specifying how Employees are to be
 replicated."

^ super replicationSpec ,
   #( (name replicate)
   (address min 2)
   (department forwarder) )
```

In cases where not all of a composite object is copied into the client, some placeholder object must take the place of each object that remained on the server. This object, called a "stub," maintains information concerning its corresponding object on the server. When a stub is sent a message, it has the ability to create a replicate, replace the stub with the new replicate, and then resend the message to the replicate. This happens transparently to the end user, so application code does not have to test for the presence of a stub object. It is also possible to turn a replicate into a stub object. This is desirable if you want to free the space taken up by the replicate and its subobjects. You can do this by sending the message stubYourself to a replicate.

The mechanisms just described can be utilized in several ways to partition and fine-tune an application for maximum performance in a client/server environment. Developers can exercise greater control over where execution of object behavior takes place and how much data is transferred to the client.