Communication Between Agents

[0071] An agent of the present invention communicates with another agent by invoking interface methods of remote objects, for example in much the same way as Java's RMI. First, an interface that extends a Remote interface is defined with the signatures of instance methods that may be called from remote agents. Second, a class that implements the above interface is defined with the implementation of the methods. Third, an instance object created from the above class is recorded in either the agent's own registry or the global registry agent. Fourth, a remote agent looks up an object in the registry and receives the object reference to the actual object. Finally, the remote agent may call the instance method of the remote object in the RPC model, which is always applied to the remote method calls across agents.

[0072] Once the object reference is passed to the remote agent, the remote instance call may pass more remote object references via arguments to the remote agent and get another remote reference in a return value, so separate agents can be tightly coupled with many object references. The arguments and the return values are passed by reference when the objects implements a Remote interface. Otherwise, they are passed by value (a deep copy of the original). When objects are copied, the consistency of the field values in the objects is not maintained across agents.

Dynamic Linking

[0073] The present invention allows new class definitions (i.e. code) to be dynamically injected into a running program. This feature allows applications to incrementally enhance their functionality without requiring their reexecution. The structure of the class loader that provides this feature is similar to related mechanisms found in other languages that provide dynamic linking of new code structures (e.g. Scheme or Java). However, the introduction of a distributed object space raises issues not relevant in previous work.

[0074] Due to the distributed object semantics of the present invention, an agent has more than one class loader that control how and where to load classes. The first class loader that is created at the beginning of execution is preferably linked to the base where the agent starts to run, so that user-defined classes are loaded from the base by default. However, when an object migrates to a new base where the object's class has not yet been loaded, the class cannot be loaded from this new base but must be transferred from the source base (on which the object was originally created) to the destination base.

[0075] Class loaders also manage class objects. Though it is not necessary that all class objects reside on the same base as the class loader, the class loader must know if a class object is already created, or where the actual class objects are, so that it can observe the rule that each class has only one class object for a class loader. If a user-defined class file is located in a specific local disk that is different from the base on which an agent starts to run, and a programmer wishes to load the class, the programmer may use a base-dependent class loader.

[0076] FIGS. 8A-8C show three cases of class loading. FIGS. 8A and 8B show two cases of object migration to a base where the corresponding class file is not loaded, while FIG. 8C illustrates new class object creation.

[0077] FIG. 8A shows migration of an object in core library classes. Core class libraries may be considered as representing system classes not modifiable by the programmer. The core class files may always be loaded from a local disk. As shown in FIG. 8A, Base1 holds a class loader and a class object (Class1) and an instance of this class. When this instance moves to Base2, the class file containing the code for this object's methods must be loaded. To do so, the following steps are performed. After the object migrates (arrow A), a remote reference to the defining class found on Base1 is created on Base2 (arrow B). Similarly, a remote reference to the class loader is also created (arrow C). Since Class1 is a core class, it can be loaded from a disk local to the machine on which Base2 is found (arrow D) and linked to the instance of Class1 (arrow E).

[0078] FIG. 8B depicts object migration in a user-defined class. In this case, the class file must be loaded from the disk in which the class file exists, or from the source base of the migration, because the source base must have the class file. FIG. 8B illustrates the latter example. Base1 holds a class loader and a class object (Class2) and an instance of this class. When the Class2 instance migrates to Base2 (arrow A), remote references to the class object (arrow B) and the class loader (arrow C) are established. The remote reference to the class loader (arrow C) allows future dynamic linking of class files created on Base1 to be transparently loaded onto Base2. The remote reference to the class object (arrow B) is required because the class object may hold global state information. The class file containing method definitions is then copied (arrow D) from Base1 to Base2. The instance object is then linked to this class file (arrow E).

[0079] Finally, FIG. 8C depicts the creation of a new class object. Here, a computation on Base2 makes a reference to a new class. Base1 holds a class loader and a class object (Class3). The class loader on Base2 is simply a remote reference (arrow A) to the class loader on Base1. The class loader loads the class file from a file system owned by Base1 (arrow B) onto Base2's local file system (arrow C). A new instance of the Class3 object is then created on Base2. A new instance of the Class3 class itself is also created on Base2. Since there must be unique reference to a given class object in the system, a remote reference to the