guages greatly weakens the utility of process mobility in these languages.

Network-Centric, Object Based Languages

[0009] Recently, a shift to a new computational paradigm has occurred. Instead of regarding the locus of an executing program as a single address space physically resident on a single processor, or as a collection of independent programs distributed among a set of processors, the advent of concurrent, network-centric, object-based languages, such as Java, has offered a compelling alternative. *See* J. Gosling et al., *The Java Language Specification*, Sun Microsystems, Inc. (1995), which is expressly incorporated herein. By allowing concurrent threads of control to execute on top of a portable, distributed virtual machine, a network-aware language like Java presents a view of computation in which a single program can be seamlessly distributed among a collection of heterogeneous processors. Unlike distributed systems that require the same code to be resident on all machines prior to execution, code-mobile languages like Java allow new code to be transmitted and linked to an already executing process. This feature allows dynamic upload functionality in ways not possible in traditional distributed systems.

[0010] Java incorporates computational units known as "objects." An object includes a collection of data called instances variables, and a set of operations called methods which operate on the instance variables. Object state (i.e. the instance variables) is accessed and manipulated from outside an object through publicly visible methods. Because this object-oriented paradigm provides a natural form of encapsulation, it is generally well-suited for a distributed environment. Objects provide regulated access to shared resources and services. In contrast to distributed glue languages, distributed extensions of Java permit objects as well as base types to be communicated. Moreover, certain implementations, such as Java/RMI, also permit code to be dynamically linked into an address space on a remote site.

[0011] Since a primary goal of Java is to support code migration (note that code migration is conceptually distinct from process mobility, since code migration makes no assumption about the data to be operated by the instructions in the code being migrated) in a distributed environment, the language provides a socket mechanism through which processes on different machines in a distributed network may communicate. Sockets, however, are a low-level network communication abstraction. Applications using sockets must layer an application-level protocol on top of this network layer. The application-level protocol is responsible for encoding and decoding messages, performing type-checking and verification, and the like. This arrangement has been found to be error-prone and cumbersome. Moreover, Java only supports migration of whole programs. Threads of control cannot be transmitted among distinct machines.

[0012] RPC provides one way of abstracting low-level details necessary to use sockets. RPC is a poor fit, however, to an object-oriented system. In Java, for example, communication takes place among objects, not procedures per se. Java/RMI, described for example in A. Wollrath et al., "Java-Centric Distributed Computing," *IEEE Micro*, Vol. 2, No. 72, pp. 44-53 (May 1997), is a variant of RPC tailored for the object semantics defined by Java's sequential core. Instead of using procedure call as the basis for separating local and remote computation, Java/RMI uses objects. A remote computation is initiated by invoking a procedure on a remote object. Clients access remote objects through surrogate objects found on their own machines. These objects are generated automatically by the compiler, and compile to code that handles marshalling of arguments and the like. Like any other Java object, remote objects are first-class, and may be passed as arguments to, or returned as results from, a procedure call.

[0013] Java/RMI supports a number of features not available in distributed extensions of imperative languages or distributed glue languages. Most important among them is the ability to transfer behavior to and from clients and servers. Consider a remote interface I that defines some abstraction. A server may implement this interface, providing a specific behavior. When a client first requests this object, it gets the code defining the implementation. In other words, as long as clients and servers agree on a policy, the particular mechanism used to implement this policy can be altered dynamically. Clients can send behavior to servers by packaging them as tasks which can then be directly executed on the server. Again, if the procedure to be executed is not already found on the server, it is fetched from the client. Remote interfaces thus provide a powerful device to dynamically ship executable content with state among a distributed collection of machines. Java/RMI allows data as well as code to be communicated among machines in a Java ensemble. Such extensions permit Java programmers to view a computation not merely as a single monolithic unit moving from machine to machine (such as in the form of applets), but as a distributed entity, partitioned among a collection of machines. By using an architecture-independent virtual machine, information from one process can be sent to another without deep knowledge of the machines on which each process is executing or the underlying network infrastructure connecting these pieces together.

[0014] Java/RMI can be difficult to use, however. Remote objects are implicitly associated with global handles or uids, and thus are never copied across nodes. However, any argument which is not a remote object in a remote object procedure call is copied, in much the same way as in RPC. As a result, remote calls have different semantics from local calls even though they appear identical syntactically. The fact that Java is