ing to the object class. The agent instantiated in step 64 is distributed among the plurality of computer machines of the network, so the task instances and object instances, like the agent process itself, may similarly be distributed among the computer machines of the network. In a step 65, the object migrate method, task migrate method and agent migrate method are performed within the agent process. It should be noted that the methods performed during step 65 need not be performed in any particular order, and each may be performed multiple times, if desired. Moreover, only some of the migrate methods may be defined and performed, if desired.

Method Call Models

[0063] In an object-oriented language, an object defines a collection of data and operations called methods or procedures to manipulate that data. A method call invokes a method on some arguments. In a distributed object-oriented language like that used in the present invention, a method call may span machine boundaries: that is, the machine where the call is made may be different than the machine where the object containing the called method resides.

[0064] The present invention provides two different ways or protocols for executing methods. These two protocols derive from the fact that the caller of a method and the callee object may not be physically located on the same machine. Before describing these two calling protocols, it is useful to first explain fast and slow access modes to objects. In the fast access mode, an object field is accessed without checking and dereferencing the object's identity in the object space. Such an operation is only valid if the object is guaranteed to be present on the caller's base. In this case, the object is accessed through an ordinary addressing scheme. In the slow access mode, an object's global location must be checked via the object space and dereferenced every time one of its components is accessed.

[0065] In order to utilize these two access modes, the present invention defines the following two calling protocols.

(1) An "RPC Model" (remote procedure call model) utilizes the fast access mode. A method runs on the base where the *self* object that owns the method resides, so that field accesses of the object can always be done in fast mode. No dereferencing of the object's global identity is required.

(2) An "Invoker Model" realizes the slow access mode. A method runs on the caller base and does not require that the *self* object be on the same base, so field accesses require dereferencing before actually accessing. The invoker model allows code to be run at the calling or called location; that is, on different machines within the same agent.

[0066] Although the use of one or the other of these protocols impacts efficiency, neither of the two protocols influences program behavior.

[0067] The RPC Model and Invoker Model are illustrated in FIG. 7. A single agent spans two bases, Base1 and Base2. A method m0( ) is running on Base2. Method m0( ) calls a method x.m1( ) which is associated with an object x on Base1. Under the RPC model, computation in method m0( ) moves from Base2 to Base1 where object x and the associated method x.m1( ) resides. That is, the method x.m1( ) executes on Base1 though it was called by a method running on Base2. Field accesses, such as the statement "this.v1 = 0;", can be performed as a local computation requiring no communication between the bases. After method x.m1( ) is completed, control returns to method m0( ) on Base2. Next, a second method, x.m2( ), associated with object x on Base1 is called under the Invoker mode. A remote reference to object x is created on Base2, and method x.m2( ) is run on Base2 rather than on Base1. Field accesses, such as the statement "this.y = 0;", require initiation of a communication event between Base1 and Base2.

[0068] Special cases for method calls include constructor methods or instance methods of a class implementing either an Anchored or a Remote interface. A constructor method is always called in the RPC model, since it might have location-dependent initialization. The instance native method to anchored objects must always be called in the RPC model, since the semantics of the instances are dependent on their locations. The interface method to a remote object is called in one of the bases on which the agent that has the actual object resides.

[0069] Programmers may also explicitly specify a base where an invoker method call should be executed using a method name with an '@[base]' expression, although the invoker method is executed in a caller base by default. In this call, the caller base, the base on which the instance resides, and the base on which the method is executed might be different, but this does not raise an error since the instance methods and fields are always accessed using the slow access mode.

[0070] The language of the present invention preferably assumes the following default behavior:

(1) Unless specified otherwise, methods are always called in the invoker model.

(2) When a programmer specifies an RPC method modifier to a method, the method directly accesses instance fields of *self* objects in the fast mode, though realization of this protocol requires execution to move to the base where the associated object resides.

(3) RPC modifiers can also be applied to static methods. In this case, static methods are called at the location where the class object is, and then the methods access static fields in the fast mode.

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

Class3 object created on Base2 is established on Base1 (arrow D). Hence, future references to Class3 initiated on Base1 will refer to static fields and methods found in the Class3 class object resident on Base2.

Runtime System

[0059] The runtime system manages a data structure of a base and provides special functions described in this invention by the inventors. As FIG. 9 shows, each base is attached to a corresponding runtime system that provides certain management functions and communication support. A single communication system may be used to serve all of the runtime systems on a particular machine. An agent may comprise a plurality of subagents, each of which resides on separate bases. In this case, subagents in the separate bases are connected with the communication system supported by the runtime system or systems found on their respective bases.

[0060] FIG. 10 depicts subcomponents in a base and its runtime system in detail. A base includes a plurality of data blocks, including class files, object memory, task memory and subagent control storage. The object memory stores all objects in a subagent, including reference objects that refer to remote objects outside the subagent. The object memory is managed by an object manager in a runtime system and pointed to by an object table in the subagent control storage. The task memory stores thread frames, used by the task executer to manage task execution. Class files hold programming code that is accessed by the task executer. The subagent control storage stores management information for a subagent. An agent ID in the subagent control storage identifies the specific agent to which the subagent belongs (that is, the agent of which the subagent is a part). An object table in the subagent control storage points to an object memory in the subagent. A task stack in the subagent control storage points to a task memory to maintain the subagent's execution states.

[0062] An agent manager manages subagents in a base using subagent control storage and communicating with a task executer that instantiates agents, executes programs in the class files, instantiates objects in the object memory, and manages execution task states in the task memory. Since both tasks and objects can migrate freely within an agent and among subagents residing on different bases, some mechanism must be available to transmit object and task state among machines of potentially different types (i.e. heterogeneous machines). Serialization is a process wherein a complex data structure (such as a tree or graph) with internal pointers is transformed into a flat object (such as an array). Pointers in the original are replaced with indices in the flattened representation and reinstantiated as pointers on the receiving agent. An implementation of a serializer is straightforward, requiring only special care to ensure that cycles in the input structure are properly recognized.

[0063] The task executer also communicates with a task serializer, to which the executer makes requests to serialize task objects and a remote access controller, to which the executer makes requests to call remote methods. Details of one implementation of the remote access controller are described below (in the section entitled "Runtime Data Structure"). An object manager implements the object space discussed above by managing objects in the object memory, and in particular by instantiating objects, reclaiming garbage objects, and making requests for serializing objects to an object serializer. A communication system mediates interaction among bases in machines connected to the network.

[0064] While agent and object migration issues have been generally discussed above (see section entitled "Agent and Object Migration"), the participation of the runtime system in such migration is highlighted in the following additional discussion.

[0065] FIGS. 11A-11E show an example sequence of agent migration. As shown in FIG. 11A, an agent comprising a single Subagent A may reside on a Base A on a Machine A on the left side of the diagram. A Base A task executer is instructed to execute an agent migrate method on the agent comprising Subagent A to migrate Subagent A to Base B on Machine B. The Base A task executer requests a Base A agent manager to obtain agent control data for Subagent A and send it to a Machine A communication system. The agent control data comprises header information about the migrating agent, along with its tasks and objects. Next, the Base A task executer requests a Base A task serializer to serialize task objects within Subagent A in task memory, and the Base A task serializer sends the serialized tasks to the Machine A communication system. Similarly, objects are also serialized and sent to the Machine A communication system by the Base A object manager and object serializer. As shown in FIG. 11B, the Machine A communication system then sends the serialized objects, serialized tasks and agent control data for Subagent A over the network to the communication system for Machine B.

[0066] After the Machine B communication system receives the agent migration data for Subagent A (including the agent control data, serialized tasks and serialized objects), a Base B agent manager allocates a memory block for Subagent A on Base B (denoted Subagent A'), and creates a subagent control storage on Base B for Subagent A'. Machine B task executer and object manager also create task objects and data objects in Base B task memory and object memory, respectively. After Subagent A' is thus instantiated on Base B, a class request is sent from Base B to Base A over the network as shown in FIG. 11C. As shown in FIG. 11D, Base A responds to the class request by sending over the network to Base B class files for the agent which are necessary for resuming the agent on

Base B. After all migration steps are finished, the memory blocks for Subagent A on Base A is released, and the agent resumes as Subagent A' on the Base B as shown in FIG. 11E. In this example, Machine A and Machine B may be heterogeneous. Machine dependencies in the structure of tasks and objects are resolved by the runtime system, and in particular by the task and object serializers

[0087] FIGS. 12A and 12B depict an example of partial agent migration, in which a part of an agent residing on a base is sent to another base, and in which remaining parts of the agent continue to reside on their current bases. In this example, as shown in FIG. 12A, an Agent comprises two subagents, Subagent A1 and Subagent A2, which reside on two bases, Base A and Base B, respectively. Partial migration of the Agent is requested, by which only Subagent A1 is requested to migrate from Base A to a Base C, while Subagent A2 remains on Base B. A serialization process for Subagent A1 is performed in a manner similar to that shown in FIGS. 11A-11E and described above. After the partial migration as shown in FIG. 12B, Subagent A1 has migrated from Base A to Base C as Subagent A1', and the entire agent therefore resides on both Base B and Base C.

[0088] FIGS. 13A and 13B depict an example of whole agent migration, in which all parts of an agent migrate to a target base. In this example, as shown in FIG. 13A, an Agent comprises two subagents, Subagent A1 and Subagent A2, which reside on two bases, Base A and Base B, respectively. Subagent A1 residing on Base A executes a whole agent migrate method, requesting migration of the entire Agent to which Subagent A belongs to a Base C. Another portion of the Agent, namely Subagent A2, happens to reside on another base, namely Base B. As a result of whole agent migration, both Subagent A and Subagent B migrate to Base C and are merged into a single subagent Subagent A3, as shown in FIG. 13B.

[0089] FIGS. 14A and 14B show an example of object migration. An Agent comprising a single subagent, Subagent A1, resides on a Base A as shown in FIG. 14A. Subagent A1 includes an object memory containing an object, Object O1. A programmer requests that Object O1 migrate from Base A to Base B. Object O1 is serialized and sent to Base B using the Base A runtime system and communication system. The Base B communication system receives the serialized Object O1, and if there is no subagent associated with the Agent on Base B, then the Base B runtime system creates a new memory block for a new subagent, Subagent A2, as shown in FIG. 14B. In this example, the Agent resides on both Base A and Base B after the migration of Object O1, and a forwarding object "f" is created in Subagent A1' to Object O1 on Base B to maintain network-transparent references to Object O' even after the object migration.

[0090] FIGS. 15A and 15B depict one example of remote object access in the context of agent migration. In this example, as shown in FIG. 15A, a first agent, Agent A, resides on Base A and includes a Subagent A1 having a reference object R which refers to an Object O1, which is found within a second agent, Agent B, residing on Base B. Agent B migrates to a third base, Base C, after which Agent B comprises a Subagent B1 on Base B and a Subagent B2 on Base C as shown in FIG. 15B. A forwarding object "f" is created in Subagent B1 on Base B, so that Base A can access the Object O1 even after the Agent B migrates to Base C, as also shown in FIG. 15B.

[0091] FIGS. 16A and 16B depict another example of remote object access in the context of agent migration. In this example, as shown in FIG. 16A, a first agent, Agent A, resides on Base A and includes a subagent having an Object O1. A second agent, Agent B, resides on Base B and includes a subagent having a reference object R referring to the Object O1 on Base A. Agent B migrates to a third base, Base C, after which Agent B comprises a single subagent which now resides on Base C as shown in FIG. 16B. A new reference object R' is created within Agent B on Base C for maintaining consistent access to the Object O1 residing on Base A.

Runtime Data Structure

[0092] Instances in the present invention are preferably allocated from a heap. To keep preferable 64-bit values aligned properly, all objects are preferably maintained with 64-bit alignment. On byte-addressable machines this allows up to three low-order bits to be used as tags. For regularity, runtime data structures are implemented as instances of the invention whenever possible.

[0093] Both the garbage collector and the code that marshals messages need to distinguish pointers from data and to determine the sizes of objects in memory. The marshalling code also needs additional information. For example, it must be able to distinguish between interned and uninterned strings. Floating point numbers may need to be converted when moving data between dissimilar machines, so the marshalling code must be able to locate them as well.

(a) Instance

[0094] Besides its own fields, each instance preferably includes its class, an integer "hashCode" and a mutex. If the instance has ever been exported from the base on which it was created it also contains a global id. Most Java implementations derive an instance's hash value from the location of the object in memory. Because the present invention moves objects between bases, changing their locations in memory as it does so, the hash value needs to be stored within the instance itself. The hashCode, mutex, and global id are created as needed; a newly created instance has none of them. As shown in the "Instance" block 200 of FIG. 17, the layout

of an instance is preferably as follows:

- The instance's class
- (Integer hashCode)
- (Mutex)
- (Global id)
- The instance's fields

[0096] Most objects are not hashed, locked, imported or exported. Therefore, to reduce the size of these common objects these fields could be merged, at the cost of a slight increase in the cost of accessing them.

[0095] All information common to the instances of a particular class, including the data layout information needed by the garbage collector and marshalling code, is stored in the class.

(b) Arrays

[0097] For regularity, arrays are preferably represented as instances of a special array class. Each instance has fields containing the array's type, size, and elements. Array instances need to be handled specially by the garbage collector because, unlike other instances, the size of an array is not determined by the array's class.

(c) Class

[0098] As shown in FIG. 17, a class object 210 may be organized into five areas:

(1) Class-specific information for the garbage collector (GC);
(2) Instance-specific information for the garbage collector (GC);
(3) Data common to all classes;
(4) Instance and static method table; and
(5) Constants and static fields.

[0099] The following data is found in every class:

- Data layout information for this class,
- Data layout information for instances of this class, including whether this is an array class;
- This class's superclass;
- The instance of class "Class" for this class;
- The class loader used to load this class;
- Initialization status; and
- Interface method table index.

[0100] The method tables are sequences of pointers to code, one for each instance and static method in the class. An instance is invoked by jumping to the code found at the appropriate offset. Because instance method code offsets must be the same for a class and any subclasses, the instance method table begins at the same offset in every class.

[0101] The name of the class and the interfaces it implements are found in the Class instance. To speed up casts and run-time type checking, each class could also contain a succinct representation of its location in the class hierarchy.

[0102] Although not shown in FIG. 17, the code for a class's methods can contain pointers back to the class. Preferably class objects and their code are not in the heap. They are instead part of the class file, and are created when the class file is loaded.

(d) Thread

[0103] Threads express execution states of programs in runtime and may be instances of a thread class. In addition to the standard fields for that class, each thread contains a stack. This stack is a linked list of stack segments, each of which contain a sequence of stack frames. An implementation of a frame contains a pointer to size and type information for local variables and arguments. This information is used to properly handle routine type checking, and is also used by the garbage collector. It is possible to evaluate this information dynamically if garbage collection occurs infrequently.

(e) Subagents

[0104] A subagent is the portion of an agent that resides on a particular base. Instances within a subagent are "local" to that subagent; all other instances are "remote." Subagents are represented as instances of a subagent class. Their fields and methods are all related to the communication protocol and are detailed in that section.

(f) Remote References

[0105] References to instances that exist in other subagents have much the same representation as local instances. The class pointer does not point to the regular class, but instead to a copy of the class whose method pointers point to RPC stubs for the methods. Calling a method for a remote instance is identical to calling a method in a local instance. This avoids the need for testing the location of an object when doing a method dispatch. Such a test is required when doing a field reference for an instances other than *self*. Remote references have no fields; they have a non-null global id.

(g) Global Id

[0106] A global identifier or "global id" records the identity and current location of an instance that has been seen by more than one base. The global identity of the instance is determined by the base on which it was created along with an integer identifier assigned by that base.

[0107] Global identifiers are the mechanism by which

object spaces are implemented. Every object within an agent has a global id. The contents of the identifier are sufficient to locate the object regardless of where it resides in the system.

[0108] A global id preferably contains the following data:

- An integer identifier;
- The subagent to which the instance belongs;
- "null" or the instance if it currently resides on the local base; and
- A reference count and any other information needed by the global garbage collector.

[0109] Forwarding pointers are needed if the object migrates from its original home. References which touch a forwarding pointer are updated to reflect the object's new location.

(h) Communication Protocol

[0110] One example of an implementation of a communication protocol suitable for the invention is discussed below. It should be noted, however, that other suitable communication protocols may be devised which are suitable for use in connection with the present invention, and that the present invention is not limited by the particular communication protocol set forth below.

[0111] This protocol was originally designed and implemented for the Kali language, as described in H. Cejtin et al., "Higher-Order Distributed Objects," *ACM Transactions on Programming Languages and Systems* Vol. 17, No. 5, pp. 704-739 (1995), and as described in U.S. Patent No. 5,745,703 entitled "Transmission Of Higher-Order Objects Across A Network Of Heterogeneous Machines," issued April 28, 1998. Both of these references are expressly incorporated herein. Much of the Kali implementation can be used to implement the communication protocol for the present invention.

(1) Shared Data Structures

[0112] Most instances exist on only a single subagent. On all other subagents the instance is represented by a remote reference that contains no fields or other data. There are several exceptions to this rule: classes, interned strings, and subagents.

[0113] Every subagent has a local copy of the static data of any class. The values of any non-constant static fields of a class are located on a single subagent.

[0114] All literal strings and string-valued constant expressions have global identity. Each subagent has its own copy of every interned string that it references. Strings contain no mutable data, so no consistency issues.

[0115] The local representation of another subagent must contain the information needed to communicate with that subagent. Unlike classes and interned strings, this data is local; it is not a copy of information found on other subagents. The structure of a subagent is described in the next section.

(2) Subagent Data

[0116] Every subagent instance has a global id. All subagent instances preferably contain the following fields:

- A globally-unique identifier;
- decode: a vector mapping ids to instance; and
- pending: a vector of mapping ids to partially transmitted instances.

[0117] Fields in subagent instances that a particular subagent is communicating with:

- base: the base on which the subagent resides;
- wait queue: a queue of threads waiting to for a connection to be established; and
- in-port, out-port, ports for talking with the subagent.

(3) Communicating Instances

[0118] An instance is preferably transmitted as three ids: that of the class of the instance, that of the subagent that created the instance, and that of the instance itself. If the instance was created by the local subagent and has not been transmitted before, a global id must be created for it and the instance added to the local subagent's decode vector.

[0119] Note that all ids of subagent instances are those of the local instance representing the subagent. A particular subagent may be assigned different ids by every other subagent on which it is known. Preferably, by convention the id of the local subagent itself is zero.

[0120] The receiving subagent uses the three ids as follows: the subagent id is looked up in the decode vector for the transmitting agent, and the instance id is looked up in that subagent's decode vector. The class id is used only if the second lookup fails.

[0121] For example, consider three subagents A, B, and C, and that A has assigned id "5" to B. Further, B has an instance I to which it assigned id "2", that it has sent to A. When subagent A sends a reference to I to subagent C, it sends the ids "5" (for the subagent) and "2" (for the instance). Subagent C then uses its decode vector for subagent A to translate "2" into its subagent instance for B, and then uses the decode vector in that instance to translate the "3" into the local reference.

[0122] There are three problems that can arise with the receiver: it may not have a local entry for the subagent id; it may not have a local entry for the class id; and it may not have a local entry for the instance id. If it is missing the subagent id or the class id it can send a request back to the transmitter asking for the missing subagent's global identifier or the absolute name of the class. Once the global identifier is received, either the

receiver already has an instance for the subagent (it merely did not have the transmitter's id for it), or this is the first time the receiver has heard of this subagent and a new subagent instance is created.

[0123] If the receiver has no local entry for the instance, its next step depends on the class of the instance. If it is a subagent the receiver requests the global identifier as above. If it is an interned string, then the receiver asks the sender for the characters in the string, and either uses or creates a local copy. In all other cases the receiver can create a remote reference to the instance without any further communication.

(4) Delayed Messages and Pending Instances

[0124] As explained above, a message may be received which contains references to an unknown subagent or interned string. Such messages are preferably delayed until the relevant information arrives from the sender. Other messages that refer to the same unknown instance may arrive after a request for information has been sent and before the reply is received. These messages must also be delayed.

[0125] Information about received-but-unknown subagents and interned strings are stored in the "pending" vector in subagent instances. If a uic is not found in the decode vector it is looked up in the pending vector. If found there, a request for the instance's data has already been sent, but the current message must be delayed until the information arrives.

(i) Garbage Collection

[0126] Since objects within an agent are distributed among a collection of machines, a global, asynchronous garbage collection strategy is preferable. A scheme of distributed reference counts is preferably used to allow the identification of instances whose remote references have been garbage collected. Each global id contains a non-zero reference count. Sending an instance to another subagent requires sending one or more reference counts along with the three ids described above. These reference counts are added to those in the global id on the receiving subagent.

[0127] If an instance in a message has a global id whose reference count is one, the sending subagent must delay sending the message. It cannot send the instance without a reference count and it must keep the one that it has. Additional reference counts are requested from the subagent that currently contains the instance. Once they arrive, the message can be sent along with some of the newly arrived reference counts. When a global id is no longer referenced by a subagent, its reference counts are sent back to the subagent containing the instance. Once that subagent has received all extant counts for the instance, the instance can be reclaimed by the agent's local garbage collector.

Exemplary Uses For The Invention

[0128] As will be readily understood by one of ordinary skill in the art, the distributed agent system of the present invention clearly has wide applicability in the field of distributed computing, and can be implemented on a wide spectrum of network communication systems, from low-bandwidth, high latency communication over modems to high-bandwidth low-latency communications such as found in clusters of high performance computers. As particular examples of such utility, the present invention offers effective support for network-centric application in which mobility is important. Such applications may include mobile software assistants capable of automatically retrieving and updating data on a corporate intranet, and adaptable query engines that execute queries and migrate database state among machines in a network to optimize availability and bandwidth. In addition, distributed applications which require high performance, such as data mining, warehousing, and search applications, will also benefit from use of the present invention. The foregoing examples are to be understood as being merely exemplary and merely serve to illustrate but a few of the many and varied possible uses for the present invention.

[0129] While there has been described and illustrated herein a distributed agent system which provides an object-based encapsulation model (an agent) which allows the processes and state of the agent to be distributed over multiple potentially heterogeneous machines, enables transparent access of data resident on another machine, and allows easy and efficient process migration, in whole or in part, among distinct machines, it will be apparent to those skilled in the art that further variations and modifications are possible without deviating from the broad teachings of the invention.

Claims

1. A distributed software system for use with a plurality of computer machines connected as a network, the system comprising:

a plurality of bases, each base providing a local address space and computer resources on one of a plurality of computer machines;

at least one agent comprising a protection domain, wherein the protection domain of the at least one agent resides on at least one of the plurality of bases;

a plurality of objects contained within the protection domain of the at least one agent, a first object residing on a first base of the plurality of bases and a second object residing on a second base of the plurality of bases, wherein the first object on the first base may access the second object on the second base without

knowledge of the physical address of the second object on the second base; and

at least one runtime system connected to the first base and the second base, the at least one runtime system facilitating migration of agents and objects from at least the first base to at least the second base.

2. The distributed software system of claim 1, wherein each agent further comprises at least one subagent, each subagent residing on one base and comprising:

an object memory which stores objects in the subagent;
a task memory which stores task frames in the subagent;
program code for the agent to which the subagent belongs;
a subagent control storage comprising:

an agent identifier indicating the agent to which the subagent belongs;
an object table having a mapping which maps symbolic references of objects to corresponding physical addresses of said objects in the object memory;
a task stack which stores a plurality of task thread pointers in the task memory;

and wherein the at least one runtime system further comprises:

an agent manager for each base managing a plurality of subagent control storages of subagents residing on the corresponding base;
an object manager for each base managing a plurality of object memories for a plurality of subagents residing on the corresponding base;
an object serializer for each base serializing objects for transmitting the objects across the network to at least one base other than the corresponding base;
a task executer for each base reading program code, creating task stacks in task memories, and executing the program code;
a task serializer for each base serializing task stacks for transmitting the stacks across the network to at least one base other than the corresponding base;
a remote access controller for each base receiving remote object access messages from a task executer on at least one base other than the corresponding base and sending remote object access requests to at least one base other than the corresponding base; and
a communication system coordinating physical communication between the computer machine and the other computer machines.

3. The distributed software system of claim 2, wherein the program code is stored as class files in the subagent.

4. The distributed software system of claim 2, wherein the runtime system further facilitates migration of tasks from the first base to the second base.

5. The distributed software system of claim 1, wherein the first object is a task and the second object is a data object.

6. The distributed software system of claim 1, wherein the at least one agent further comprises a global object space, wherein the global object space includes a mapping of symbolic references of objects within the at least one agent to corresponding physical addresses of said objects, whereby the first object on the first base accesses the second object on the second base without knowledge of the physical address of the second object on the second base by obtaining a symbolic reference to the second object from the first object and obtaining the corresponding physical address of the second object using the mapping of the object space.

7. The distributed software system of claim 6, wherein the object space is implemented using global identifiers for addressing each object.

8. The distributed software system of claim 1, wherein the access by the first object of the second object is a method call specifying at least one of an argument and a return value, wherein a symbolic reference to the at least one argument or return value may be passed to or returned from the called method to identify the at least one argument or return value, and wherein the physical address of the at least one argument or return value need not be passed to or returned from the called method to identify the at least one argument or return value so as to render the method call network transparent.

9. The distributed software system of claim 8, wherein the at least one agent further comprises a global object space, wherein the global object space includes a mapping of symbolic references of objects within the at least one agent to corresponding physical addresses of said objects, whereby the system permits the symbolic reference to the at least one argument or return value of the method call to be passed to or returned from the called method to identify the at least one argument or return value by obtaining a symbolic reference to the second object from the first object and obtaining the corresponding physical address of the second

object using the mapping of the object space.

10. The distributed software system of claim 9, wherein the object space is implemented using global identifiers for addressing each object.

11. The distributed software system of claim 1, wherein:

the first object is a first task residing within the protection domain of the at least one agent and executing within the first base; and
the second object is a second task residing within the protection domain of the at least one agent and executing within the second base, wherein the first task and the second task execute concurrently on the first and second bases, respectively, and within the same protection domain of the at least one agent.

12. The distributed software system of claim 1, wherein each base may provide the local address space and computer resources to a plurality of agents simultaneously.

13. The distributed software system of claim 1, wherein the first base and the second base are located in heterogeneous computer machines.

14. A distributed software system for use with a plurality of computer machines connected as a network, the system comprising:

at least one agent comprising a protection domain, wherein the protection domain of the at least one agent resides on at least two of the plurality of computer machines; and
a plurality of objects contained within the protection domain of the at least one agent, the objects being selectively movable among the at least two computer machines by a programmer of the system, a first object residing on a first of the at least two computer machines and a second object residing on a second of the at least two computer machines, wherein the first object on the first computer machine may access the second object on the second computer machine without knowledge of the physical address of the second object on the second computer machine, and regardless of the selective movement of either the first object or the second object among the first and second computer machines.

15. The distributed software system of claim 14, wherein the first object is a task and the second object is a data object.

16. The distributed software system of claim 14, wherein the at least one agent further comprises a global object space, wherein the global object space includes a mapping of symbolic references of objects within the at least one agent to corresponding physical addresses of said objects, whereby the first object on the first computer machine accesses the second object on the second computer machine without knowledge of the physical address of the second object on the second computer machine by obtaining a symbolic reference to the second object from the first object and obtaining the corresponding physical address of the second object using the mapping of the object space.

17. The distributed software system of claim 16, wherein the object space is implemented using global identifiers for addressing each object.

18. The distributed software system of claim 14, wherein the access by the first object of the second object is a method call specifying at least one of an argument and a return value, wherein a symbolic reference to the at least one argument or return value may be passed to or returned from the called method to identify the at least one argument or return value, and wherein the physical address of the at least one argument or return value need not be passed to or returned from the called method to identify the at least one argument or return value so as to render the method call network transparent.

19. The distributed software system of claim 18, wherein the at least one agent further comprises a global object space, wherein the global object space includes a mapping of symbolic references of objects within the at least one agent to corresponding physical addresses of said objects, whereby the system permits the symbolic reference to the at least one argument or return value of the method call to be passed to or returned from the called method to identify the at least one argument or return value by obtaining a symbolic reference to the second object from the first object and obtaining the corresponding physical address of the second object using the mapping of the object space.

20. The distributed software system of claim 19, wherein the object space is implemented using global identifiers for addressing each object.

21. The distributed software system of claim 14, wherein:

the first object is a first task residing within the protection domain of the at least one agent and executing within the first computer machine;

and

the second object is a second task residing within the protection domain of the at least one agent and executing within the second computer machine, wherein the first task and the second task execute concurrently on the first and second computer machines, respectively, and within the same protection domain of the at least one agent.

22. The distributed software system of claim 14, further comprising:

a plurality of bases, each base providing a local address space and computer resources to at least one agent on one of the plurality of computer machines, wherein the at least one agent resides on at least one base.

23. The distributed software system of claim 22, wherein the at least one agent resides on at least a first base on the first computer machine and also on a second base on the second computer machine, and wherein the first object resides on the first base and the second object resides on the second base.

24. The distributed software system of claim 22, wherein each base may provide the local address space and computer resources to a plurality of agents simultaneously.

25. The distributed software system of claim 22, wherein each base is implemented as an operating system-level process.

26. The distributed software system of claim 14, wherein the first computer machine and the second computer machine are heterogeneous.

27. A distributed software system for use with a plurality of computer machines connected as a network, the system comprising:

a plurality of bases, each base providing a local address space and computer resources on one of a plurality of computer machines;

at least one agent comprising a protection domain, wherein the protection domain of the at least one agent resides on at least one of the plurality of bases;

at least one object residing within the protection domain of the at least one agent; and

at least one runtime system connected to the plurality of bases, the at least one runtime system including a communication system which facilitates migration of agents and objects among the plurality of bases.

28. The distributed software system of claim 27, wherein each agent further comprises at least one subagent, each subagent residing on one base and comprising:

an object memory which stores objects in the subagent;

a task memory which stores task frames in the subagent;

program code for the agent to which the subagent belongs;

a subagent control storage comprising:

an agent identifier indicating the agent to which the subagent belongs;

an object table having a mapping which maps symbolic references of objects to corresponding physical addresses of said objects in the object memory;

a task stack which stores a plurality of task thread pointers in the task memory,

and wherein the at least one runtime system further comprises:

an agent manager for each base managing a plurality of subagent control storages of subagents residing on the corresponding base;

an object manager for each base managing a plurality of object memories for a plurality of subagents residing on the corresponding base;

an object serializer for each base serializing objects for transmitting the objects across the network to at least one base other than the corresponding base;

a task executer for each base reading program code, creating task stacks in task memories, and executing the program code;

a task serializer for each base serializing task stacks for transmitting the stacks across the network to at least one base other than the corresponding base;

a remote access controller for each base receiving remote object access messages from a task executer on at least one base other than the corresponding base and sending remote object access requests to at least one base other than the corresponding base; and

a communication system coordinating physical communication between the computer machine and the other computer machines.

29. The distributed software system of claim 28, wherein the program code is stored as classfiles in the subagent.

30. The distributed software system of claim 28, wherein the runtime system further facilitates

migration of tasks among the plurality of bases.

31. The distributed software system of claim 27, wherein the at least one agent further comprises a first subagent residing on a first base and a second subagent simultaneously residing on a second base, and wherein the at least one agent migrates in part to a third base, whereby the first subagent remains on the first base and the second subagent migrates to the third base.

32. The distributed software system of claim 27, wherein the at least one agent further comprises a first subagent residing on a first base and a second subagent simultaneously residing on a second base, and wherein the at least one agent migrates in whole to a third base, whereby both the first and second subagent merge and migrate to the third base as one subagent.

33. The distributed software system of claim 27, wherein the at least one agent resides on a first base, the distributed software system further comprising at least one anchored object, the at least one anchored object being instantiated on the first base from a base-dependent class and which is permanently unable to be moved from the first base to any other base, the at least one anchored object residing in the at least one agent, and wherein the at least one agent maybe instructed to migrate from the first base to a second base by a first migrate method, whereby the at least one anchored object remains on the first base while the remainder of the at least one agent migrates to the second base.

34. The distributed software system of claim 33, wherein the at least one agent may be instructed to migrate from the first base to the second base by a second migrate method, whereby the at least one anchored object on the first base is abandoned while the remainder of the at least one agent migrates to the second base.

35. The distributed software system of claim 27, wherein the at least one agent resides on a first base, the distributed software system further comprising at least one anchored object, the at least one anchored object being instantiated on the first base from a base-dependent class and which is permanently unable to be moved from the first base to any other base, the at least one anchored object residing in the at least one agent, and wherein the at least one agent may be instructed to migrate from the first base to a second base by a first migrate method, whereby the at least one anchored object is abandoned while the remainder of the at least one agent migrates to the second base.

36. The distributed software system of claim 27, wherein the at least one agent resides on a first base, the distributed software system further comprising at least one pinned object which is temporarily unable to be moved from the first base to any other base, the at least one pinned object residing in the at least one agent, and wherein the at least one agent may be instructed to migrate from the first base to a second base by a first migrate method, whereby the at least one pinned object remains on the first base while the remainder of the at least one agent migrates to the second base.

37. The distributed software system of claim 36, wherein the at least one agent may be instructed to migrate from the first base to the second base by a second migrate method, whereby the at least one pinned object on the first base is abandoned while the remainder of the at least one agent migrates to the second base.

38. The distributed software system of claim 27, wherein the at least one agent resides on a first base, the distributed software system further comprising at least one pinned object which is temporarily unable to be moved from the first base to any other base, the at least one pinned object residing in the at least one agent, and wherein the at least one agent may be instructed to migrate from the first base to a second base by a first migrate method, whereby the at least one pinned object on the first base is abandoned while the remainder of the at least one agent migrates to the second base.

39. The distributed software system of claim 38, wherein the at least one pinned object may be unpinned so as to permit the unpinned object to be moved from the first base to any other base, whereby the unpinned object on the first base may migrate to the second base when the at least one agent is instructed to migrate from the first base to the second base.

40. The distributed software system of claim 27, wherein a first agent residing on a first base possesses a reference to an object in a second agent residing on a second base, and wherein after the first agent migrates to a third base with the first agent reference, the first agent reference remains valid.

41. The distributed software system of claim 27, wherein a first agent residing on a first base possesses a reference to a first object in a second agent residing on a second base, wherein after the second agent migrates to a third base with the first object, a second object is created for permitting forwarding access from the first base to the actual

location of the first object residing or the third base so that the first agent reference remains valid.

42. The distributed software system of claim 27, further comprising:

a first method calling protocol for calling, from a first base, a method to a first object residing on a second base, wherein the method is transmitted from the first base to the second base and wherein the method is executed on the second base where the first object resides; and

a second method calling protocol for calling, from the first base, a method to a second object residing on the second base wherein the method is executed on the first base using method code on the first base corresponding to the second object method and a remote reference to the second object on the second base.

43. The distributed software system of claim 42, wherein the method code on the first base corresponding to the second object method is stored in a class file on the first base.

44. A method for implementing a network-centric computer software programming system for a network comprising a plurality of computer machines, the method comprising the steps of:

defining a plurality of object-oriented classes including an object class, an agent class, a base class and a task class;

defining an object migrate method in the object class that migrates a selected object instance to a location specified with the base class;

defining an agent migrate method in the agent class that migrates a selected agent process to a location specified with the base class, including migration of all object instances and task instances within the agent;

instantiating a first agent process according to the agent class, the first agent process including a plurality of task instances and object instances and distributed among the plurality of computer machines; and

performing the object migrate method and the agent migrate method within the first agent process.

45. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the step of defining a task migrate method in the task class that migrates a selected task represented in a task instance to a location specified with the base class.

46. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the step of instantiating a plurality of base instances according to the base class, wherein the first agent process executes on at least two bases simultaneously, each base being specified with one of the plurality of base instances.

47. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the step of instantiating a plurality of base instances according to the base class, wherein object instances of the first agent process reside on at least two bases simultaneously, each base being specified with one of the plurality of base instances.

48. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the step of instantiating a plurality of base instances according to the base class, wherein task instances of the first agent process execute on at least two bases simultaneously, each base being specified with one of the plurality of base instances.

49. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the steps of:

instantiating a first base instance according to the base class; and

instantiating a second agent process according to the agent class, wherein the first and second agent processes execute simultaneously on a base specified with the first base instance.

50. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the step of defining a partial agent migrate method in the agent class that migrates a selected part of an agent process residing on a first base specified with a first base instance to a second base specified with a second base instance, including migration of all object instances and task instances on the first base within the selected part of the agent process.

51. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the step of defining a whole agent migrate method in the agent class that

migrates a selected whole agent process residing on a first base specified with a first base instance to a second base specified with a second base instance including migration of all object instances and task instances on the first base within the selected whole agent process.

52. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the steps of:

instantiating a first base according to the base class;
instantiating a second agent process according to the agent class, the second agent process residing at least in part on the first base;
instantiating an anchored object from a base-dependent object class, the anchored object being associated with a second agent process located on a first base specified with a base instance and being unable to be moved to other bases;
defining an agent migrate method in the agent class that migrates the second agent process to another base specified with a base instance, including migration of all task instances and object instances within the second agent process except for the anchored object.

53. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the steps of:

instantiating a first base according to the base class;
instantiating a second agent process according to the agent class, the second agent process residing at least in part on the first base;
instantiating a first object according to the object class the first object residing within the second agent process,
pinning the first object to the first base; and
defining an agent migrate method in the agent class that migrates the second agent process from the first base to another base specified with the base class, including migration of all task instances and object instances within the second agent process except for the pinned first object.

54. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the steps of:

instantiating a first base and a second base according to the base class;
instantiating a task according to the task class on the first base;
instantiating an object according to the object class on the second base, the object having a method; and
defining a method calling protocol wherein calling, from the task on the first base, the method of the object on the second base includes transmitting the method from the first base to the second base and executing the method on the second base.

55. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, further comprising the steps of:

instantiating a first base and a second base according to the base class;
instantiating a task according to the task class on the first base;
instantiating an object according to the object class on the second base, the object having a method; and
defining a method calling protocol wherein calling, from the task on the first base, the method of the object on the second base includes executing the method on the first base using method code on the first base corresponding to the method of the object and a remote reference to the object on the second base.

56. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 55, wherein the method code on the first base corresponding to the method of the object is stored in a class file on the first base.

57. The method for implementing a network-centric computer software programming system for a network of computer machines according to claim 44, wherein at least a subset of the plurality of computer machines are heterogeneous.

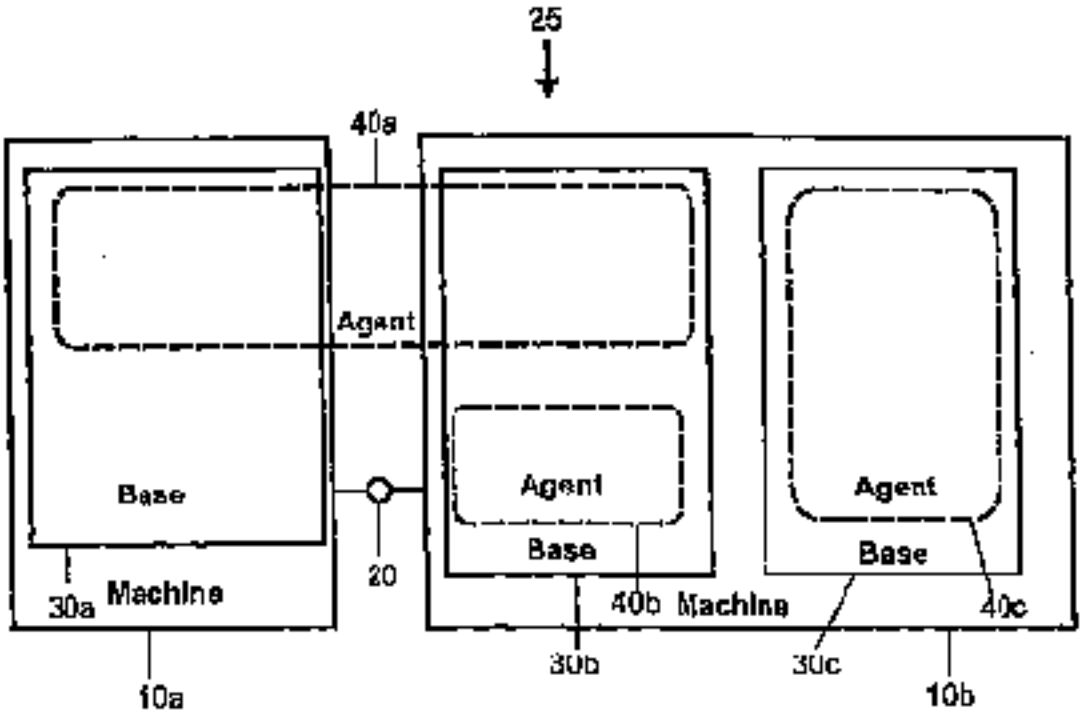
25
40a
Agent
Base
Agent
Base
Agent
Base
20
30a
Machine
40b
Machine
40c
10a
30b
30c
10b

FIG. 1

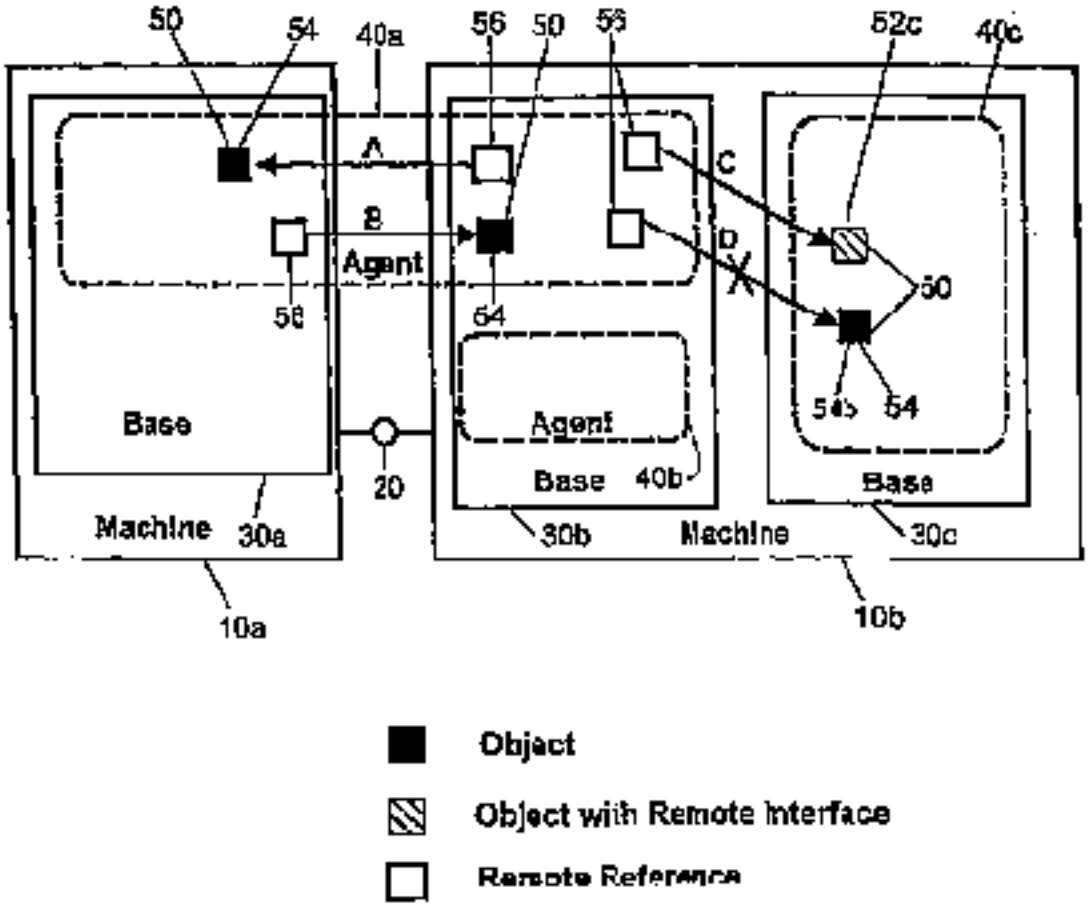
50
54
40a
56
50
58
52c
40c
A
B
C
D
Agent
56
64
50
54b
54
Base
Agent
Base
40b
Base
20
Machine
30a
30b
Machine
30c
10a
10b
Object
Object with Remote Interface
Remote Reference

FIG. 2

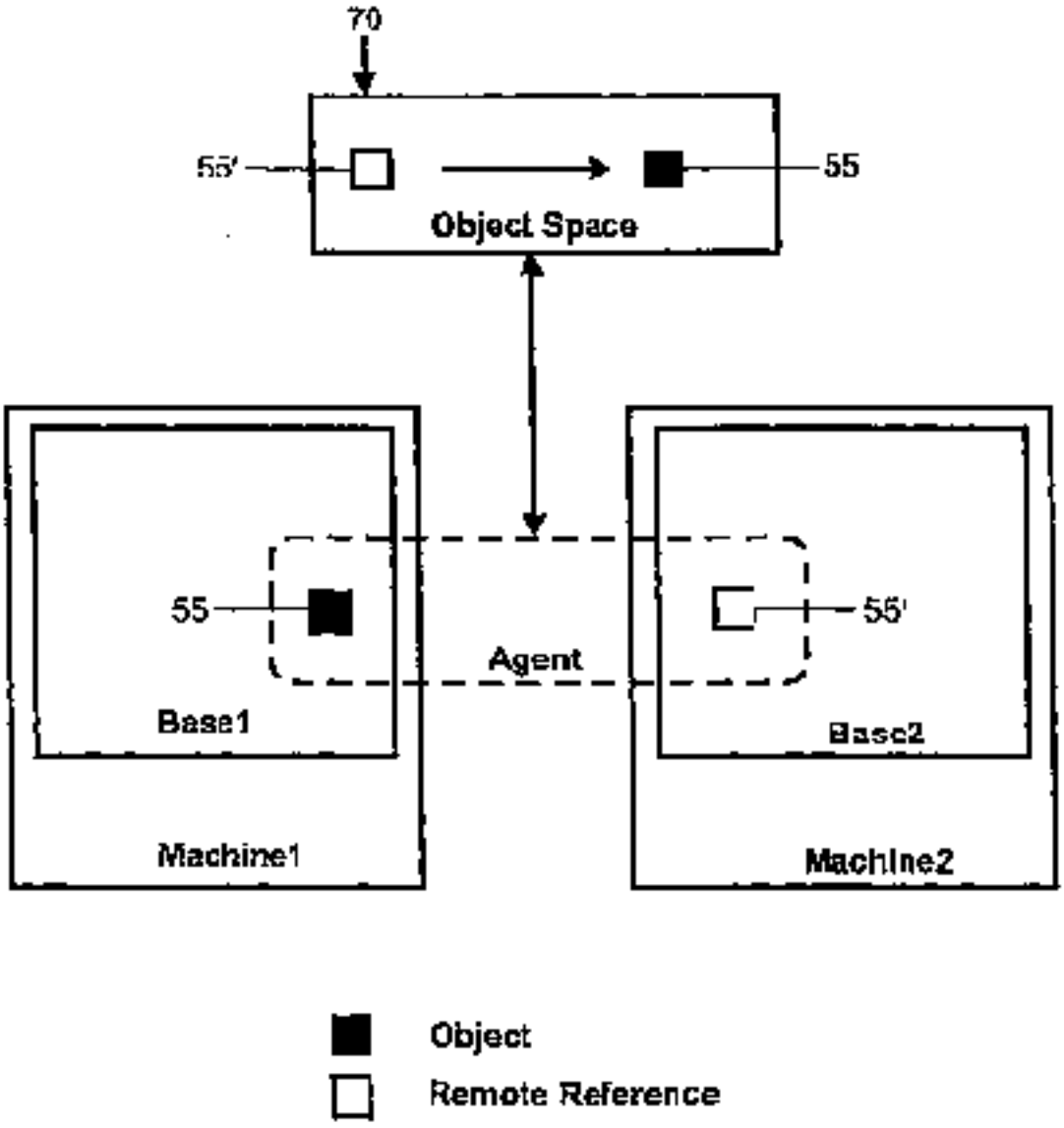
70
55'
55
Object Space
55
55'
Agent
Base1
Base2
Machine1
Machine2
Object
Remote Reference

FIG. 3

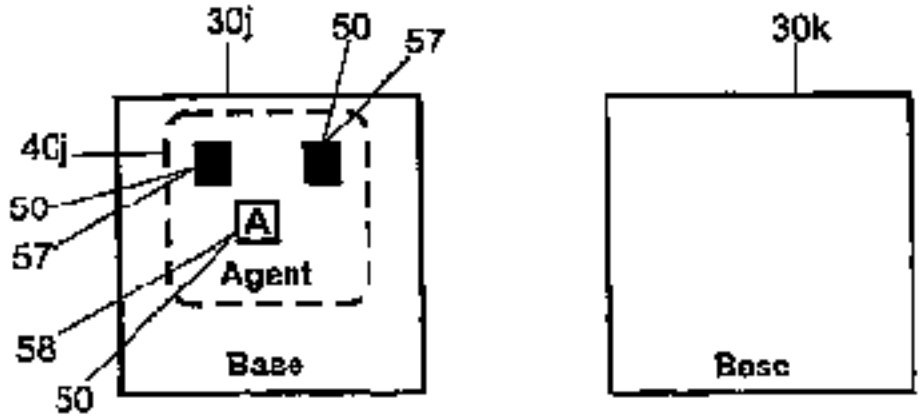

FIG. 4A

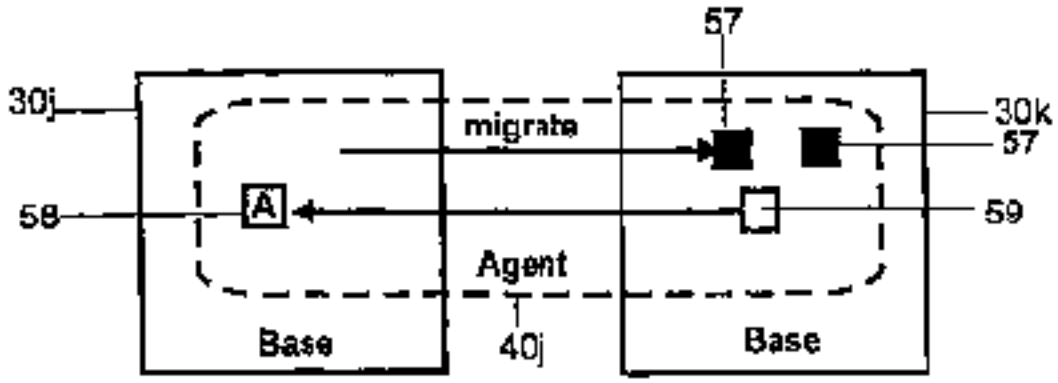

FIG. 4B

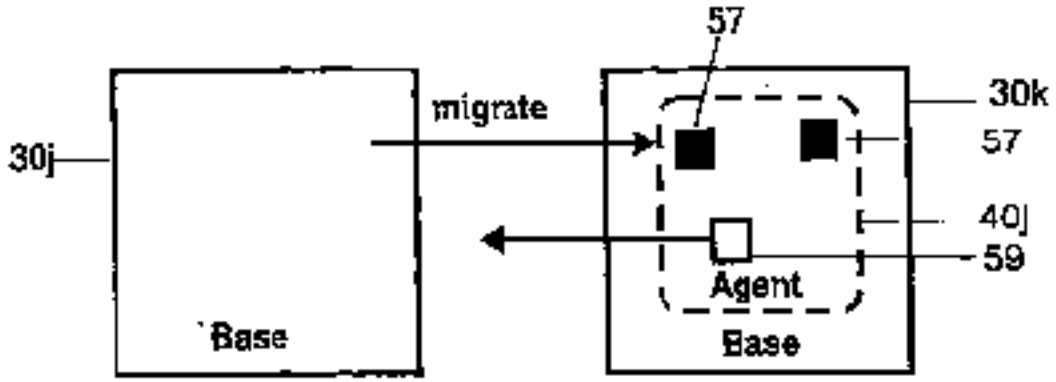

FIG. 4C

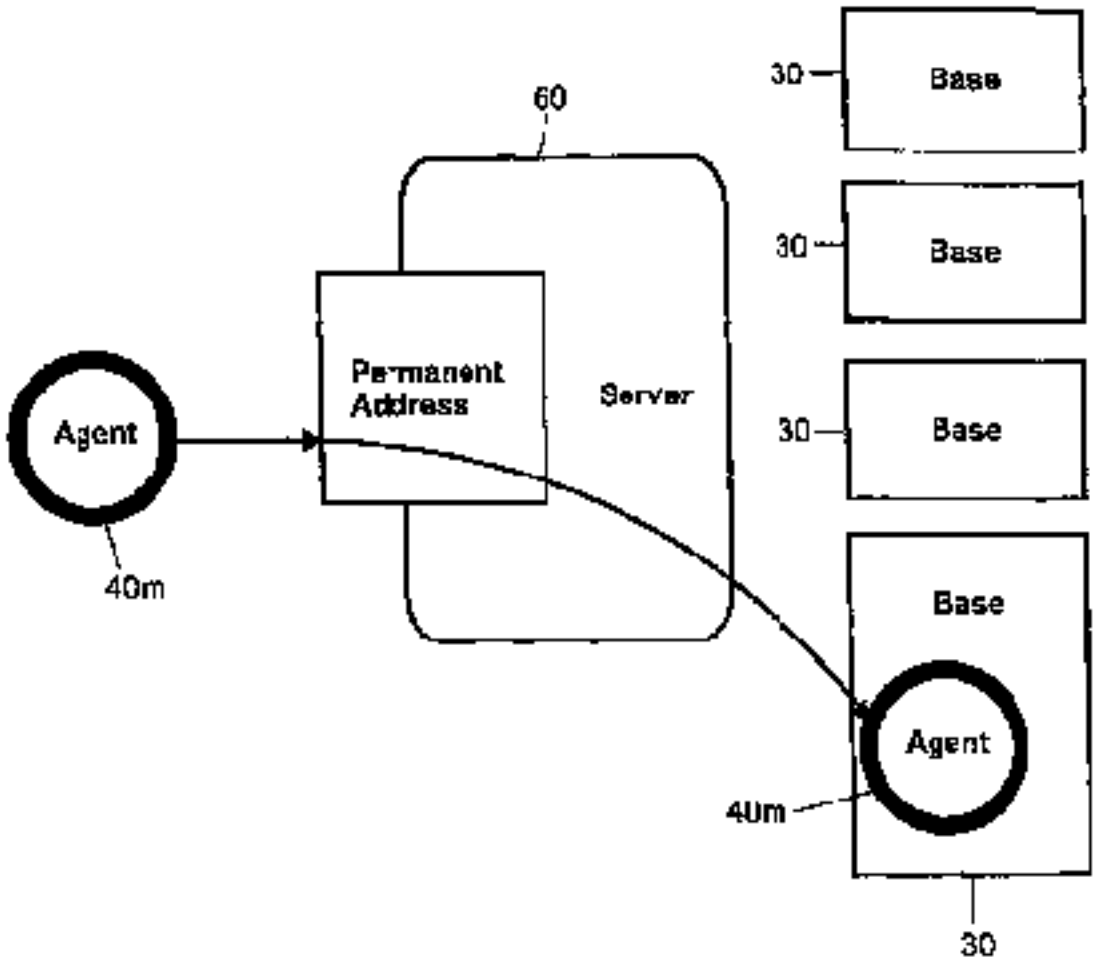
60
30
Base
30
Base
30
Base
Permanent Address
Server
Agent
40m
Base
Agent
40m
30

FIG. 5

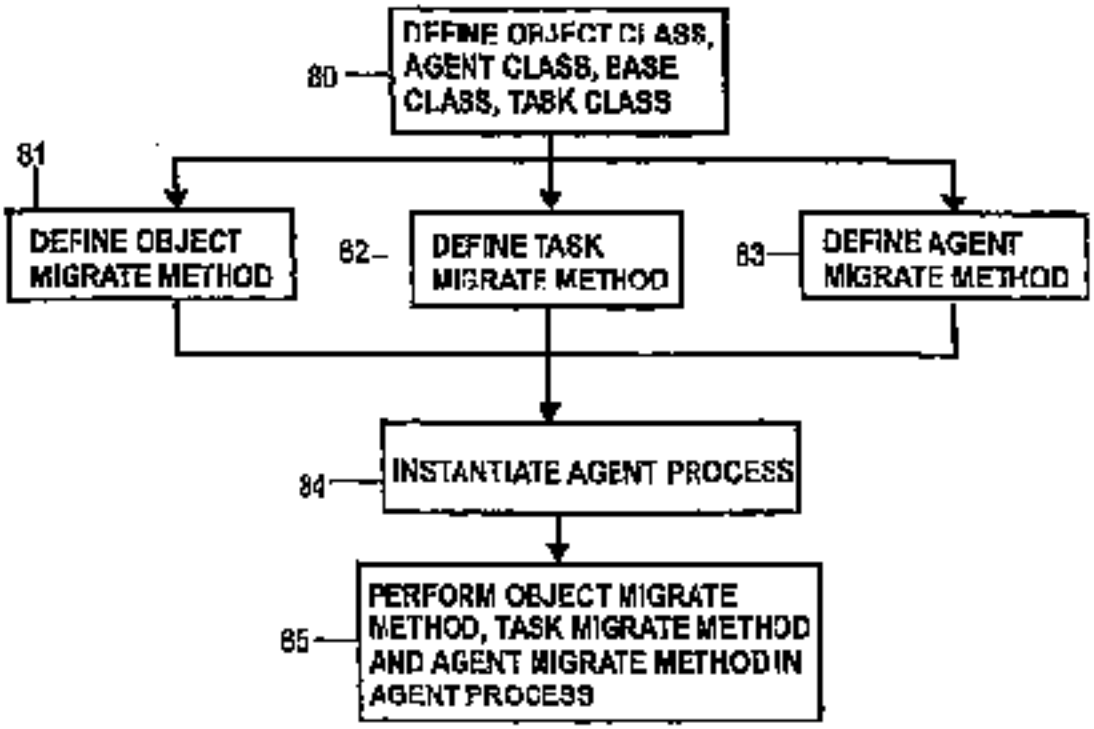
DEFINE OBJECT CLASS, AGENT CLASS, BASE CLASS, TASK CLASS
80
81
DEFINE OBJECT MIGRATE METHOD
82
DEFINE TASK MIGRATE METHOD
83
DEFINE AGENT MIGRATE METHOD
INSTANTIATE AGENT PROCESS
84
PERFORM OBJECT MIGRATE METHOD, TASK MIGRATE METHOD AND AGENT MIGRATE METHOD IN AGENT PROCESS
85

FIG. 6

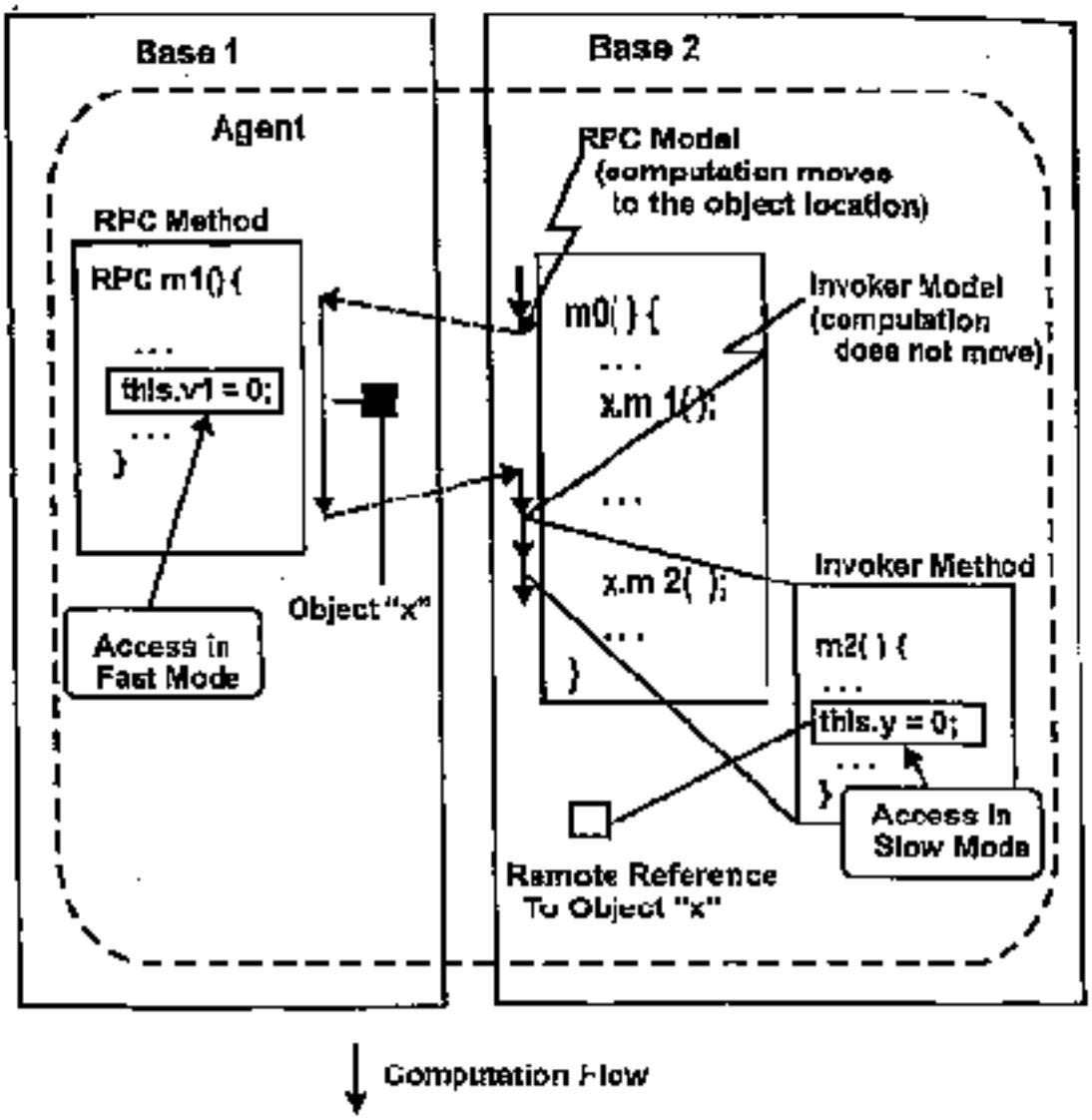
Base 1
Base 2
Agent
RPC Method
RPC m1() {
...
this.v1 = 0;
...
}
Access in
Fast Mode
Object "x"
RPC Model
(computation moves
to the object location)
m0( ) {
...
x.m 1( );
...
x.m 2( );
...
}
Invoker Model
(computation
does not move)
Invoker Method
m2( ) {
...
this.y = 0;
...
}
Access in
Slow Mode
Remote Reference
To Object "x"
Computation Flow

FIG. 7

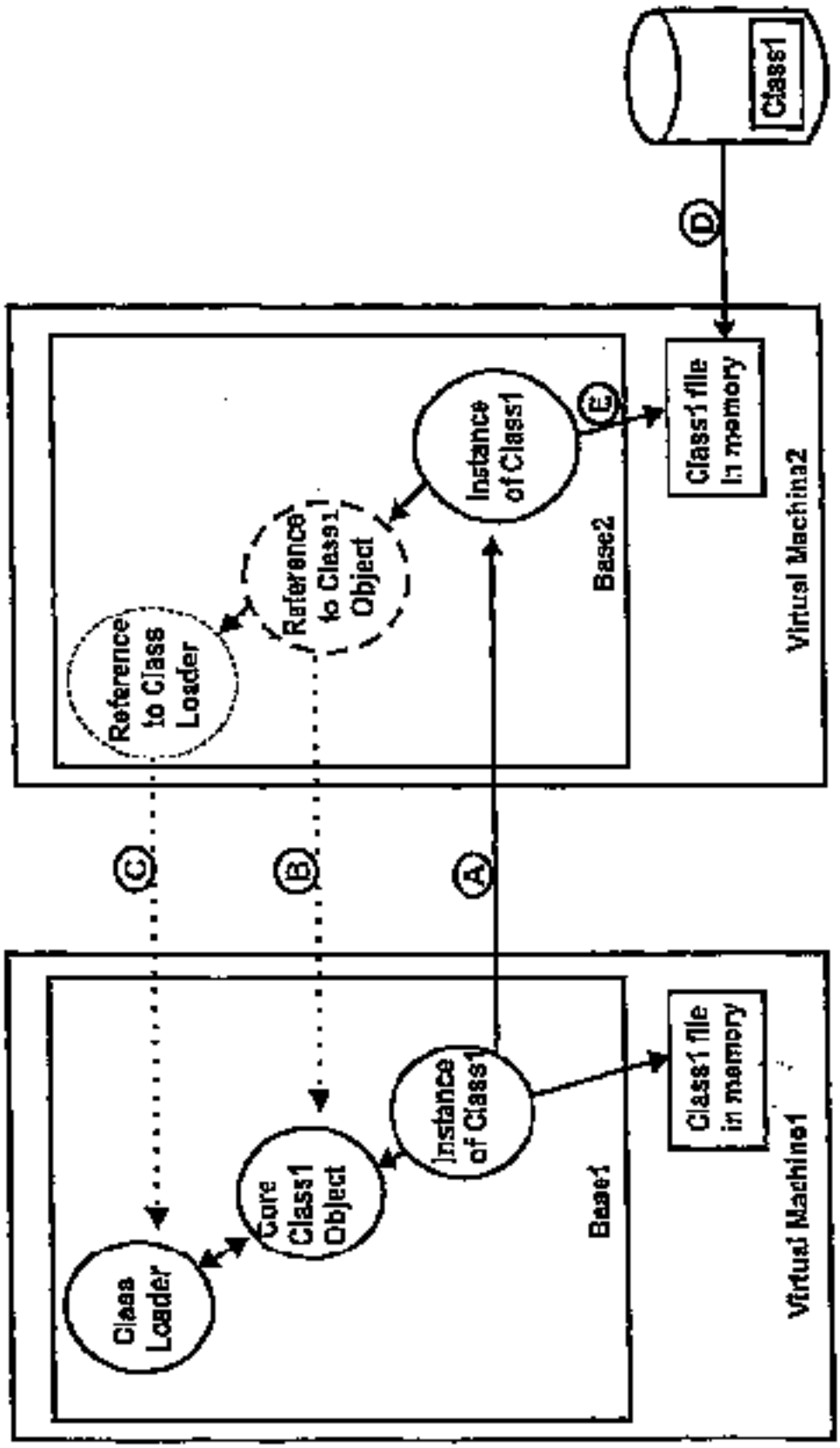
Class1
D
Virtual Machine2
Base2
Class1 file in memory
E
Instance of Class1
Reference to Class1 Object
Reference to Class Loader
C
B
A
Virtual Machine1
Base1
Class1 file in memory
Instance of Class1
Core Class1 Object
Class Loader

FIG 8A

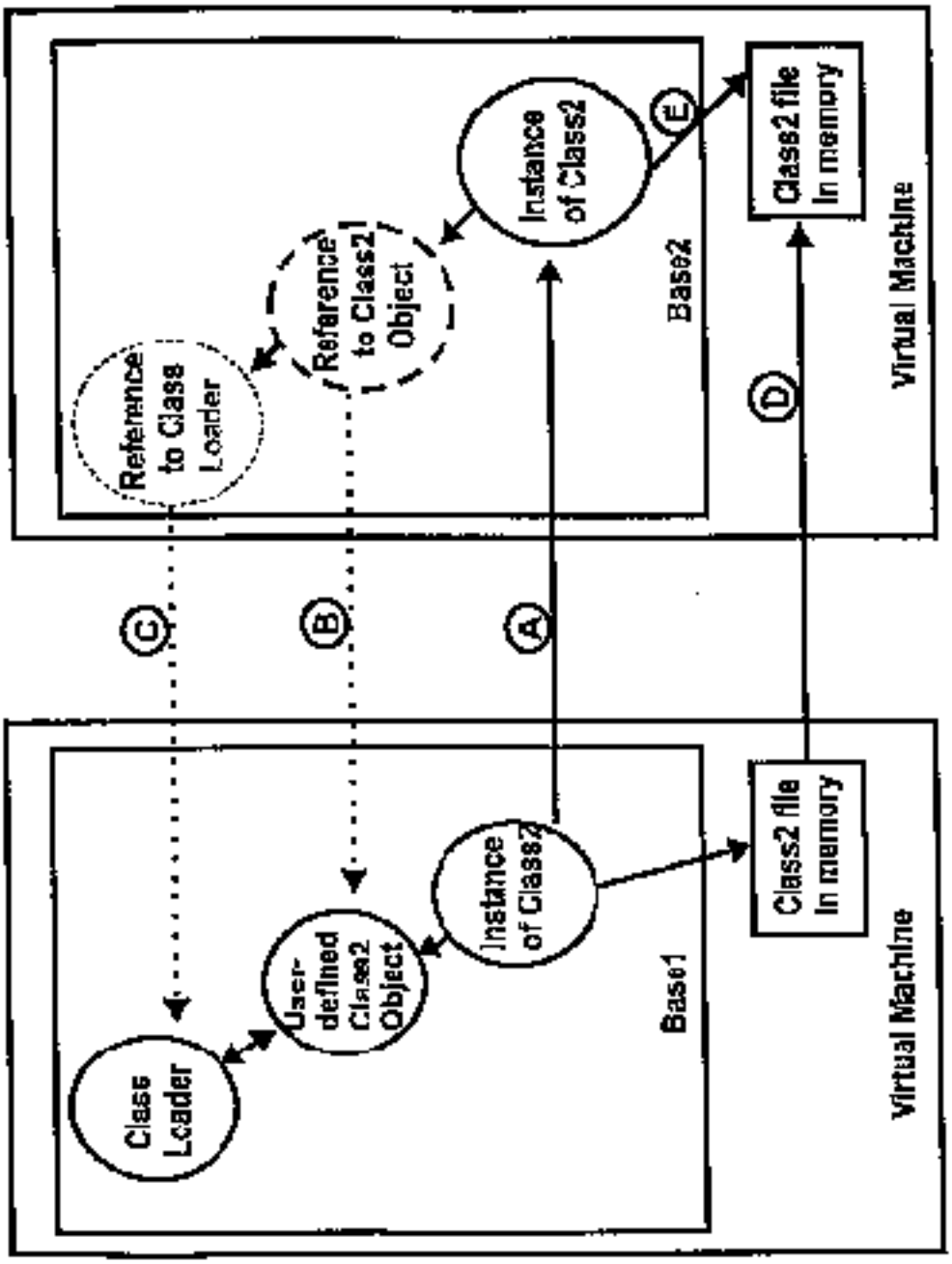
Virtual Machine
Base1
Class Loader
User-defined Class2 Object
Instance of Class2
Class2 file in memory
Virtual Machine
Base2
Reference to Class Loader
Reference to Class2 Object
Instance of Class2
Class2 file in memory
A
B
C
D
E

FIG. 8B

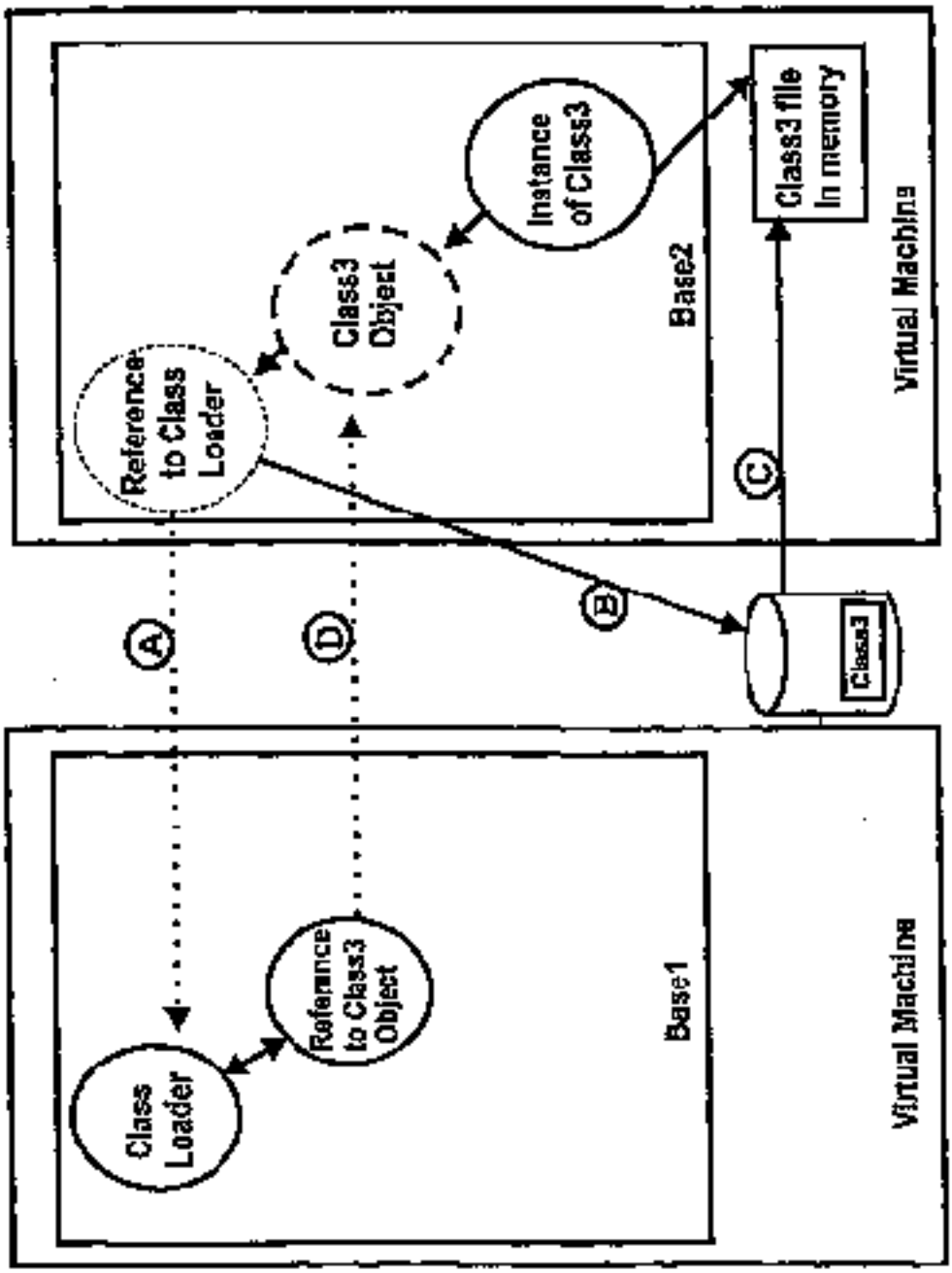
Virtual Machine
Base1
Class Loader
Reference to Class3 Object
A
D
Virtual Machine
Base2
Reference to Class Loader
Class3 Object
Instance of Class3
Class3 file in memory
B
C
Class3

FIG. 8C

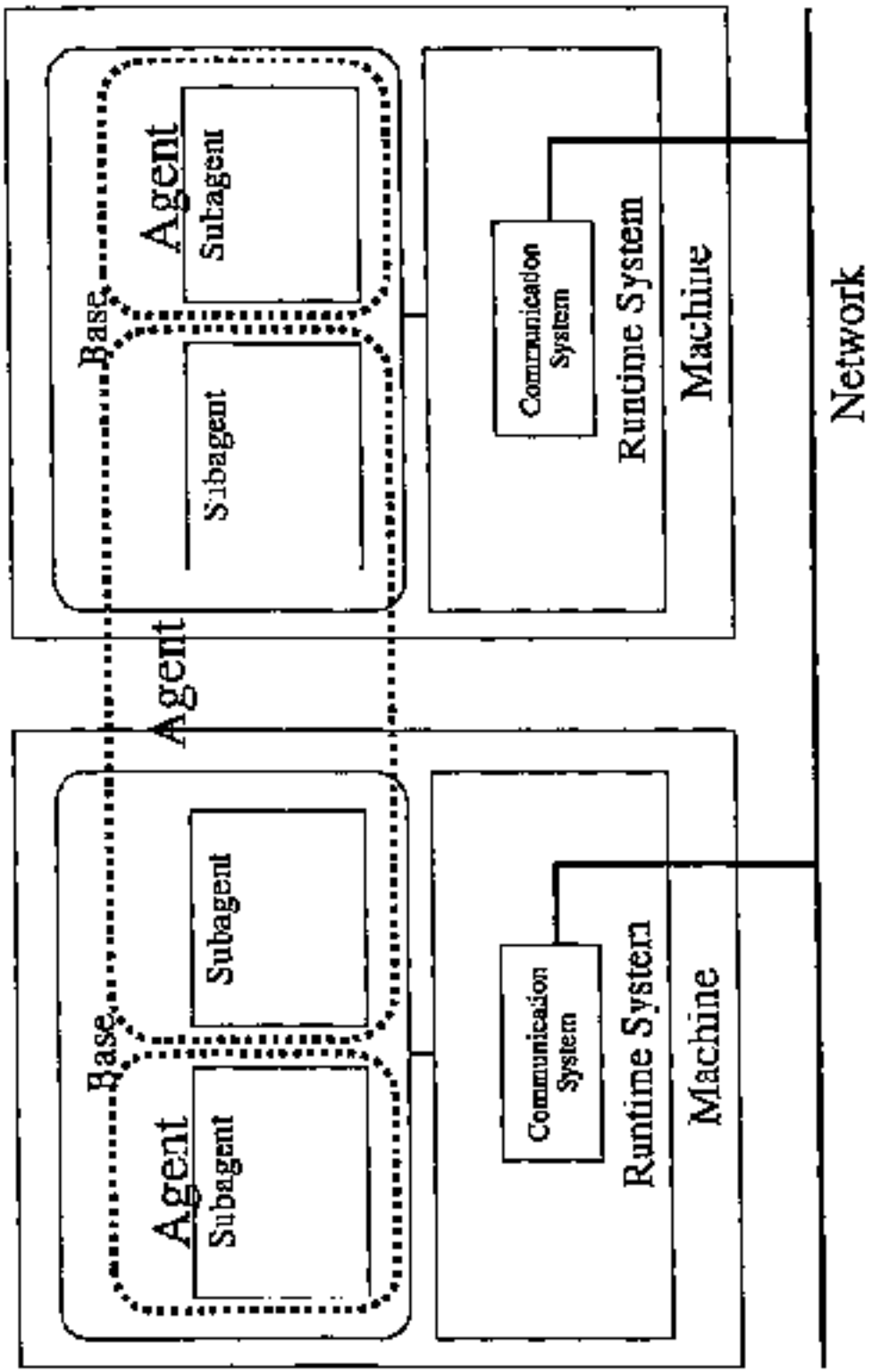
Network
Machine
Runtime System
Communication System
Agent
Base
Subagent
Subagent
Agent
Base
Subagent
Subagent
Machine
Runtime System
Communication System

FIG. 9

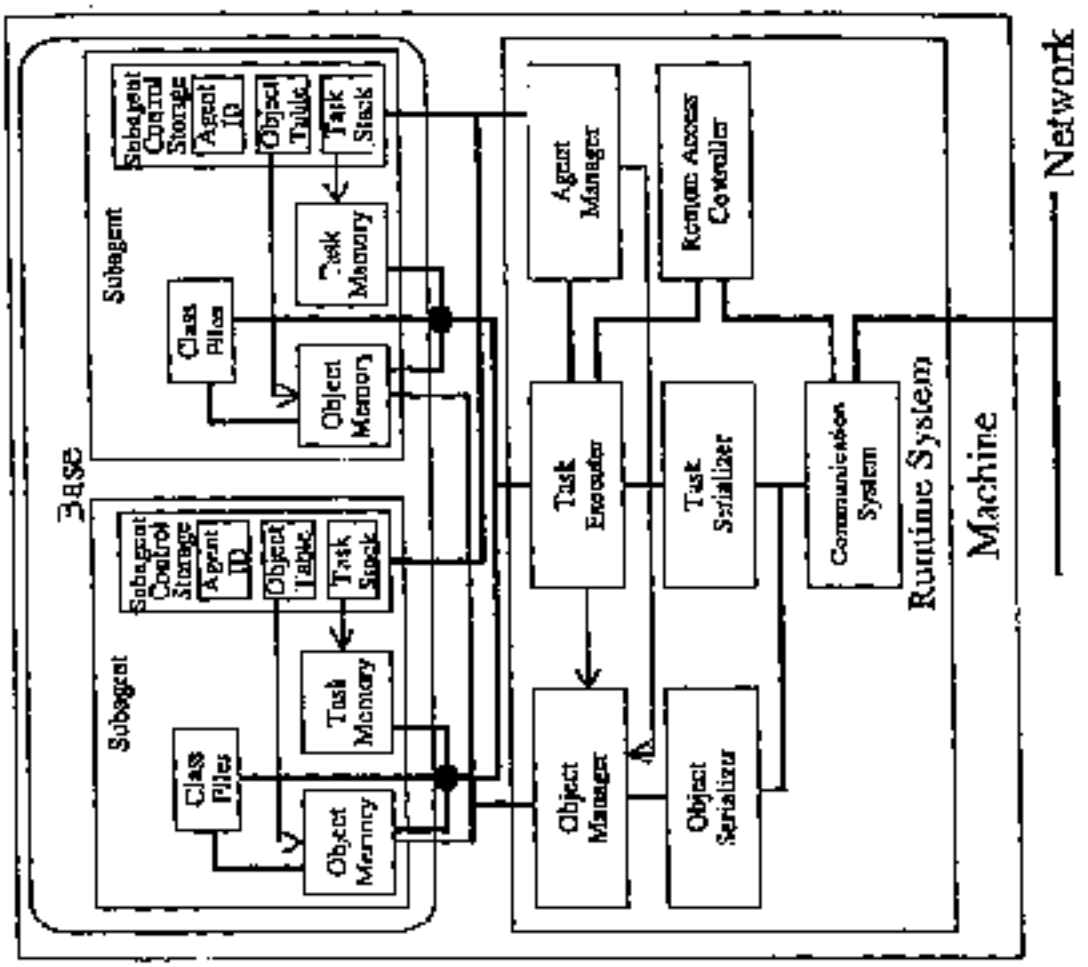
Base
Subagent
Subagent Control Storage
Agent ID
Object Table
Task Stack
Task Memory
Class Files
Object Memory
Subagent
Subagent Control Storage
Agent ID
Object Table
Task Stack
Task Memory
Class Files
Object Memory
Agent Manager
Remote Access Controller
Task Encoder
Task Serializer
Communication System
Object Manager
Object Serializer
Runtime System
Machine
Network

FIG. 10

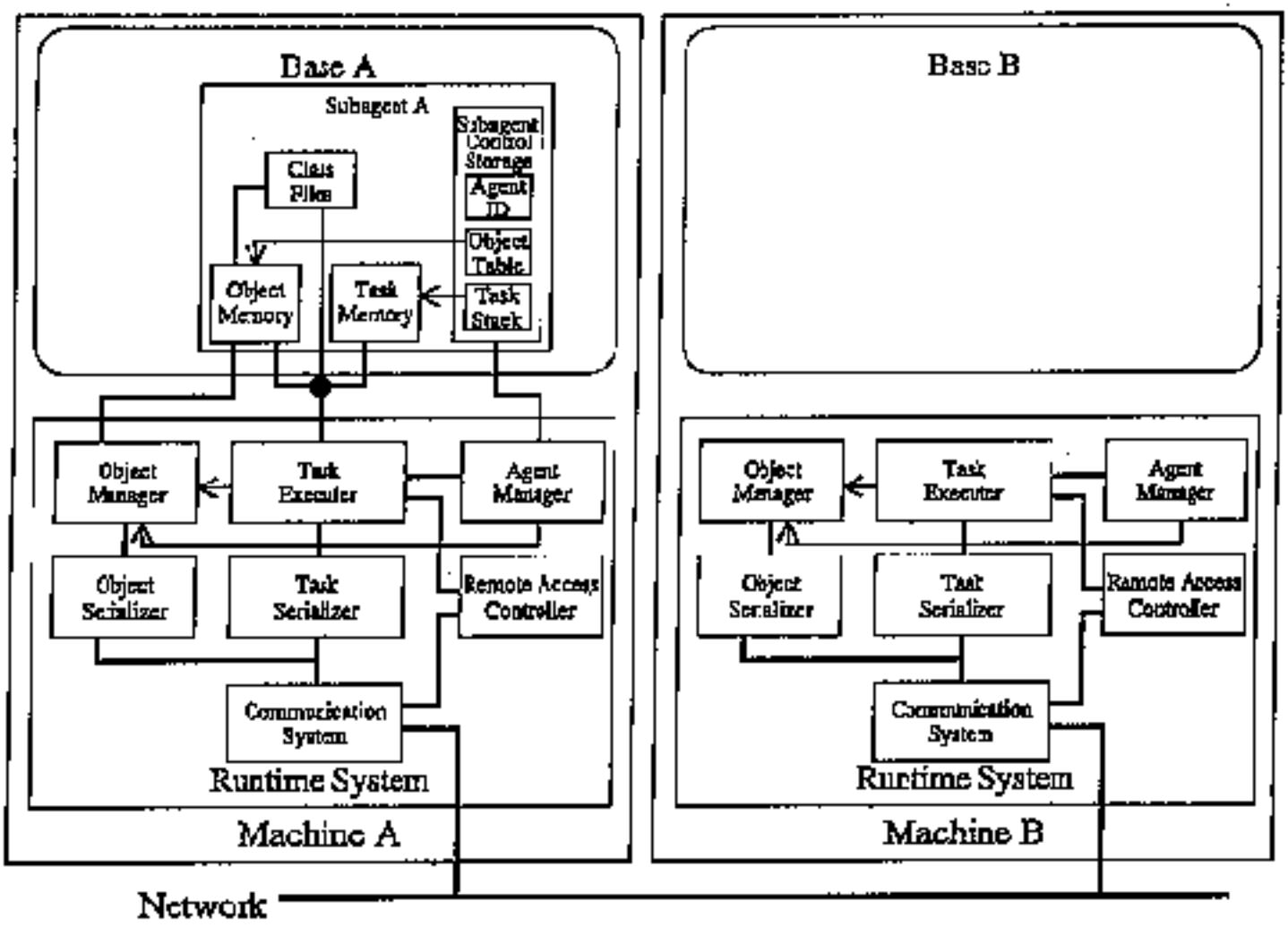
Base A
Subagent A
Subagent Control Storage
Class Files
Agent ID
Object Table
Object Memory
Task Memory
Task Stack
Object Manager
Task Executer
Agent Manager
Object Serializer
Task Serializer
Remote Access Controller
Communication System
Runtime System
Machine A
Base B
Object Manager
Task Executer
Agent Manager
Object Serializer
Task Serializer
Remote Access Controller
Communication System
Runtime System
Machine B
Network

FIG. 11A

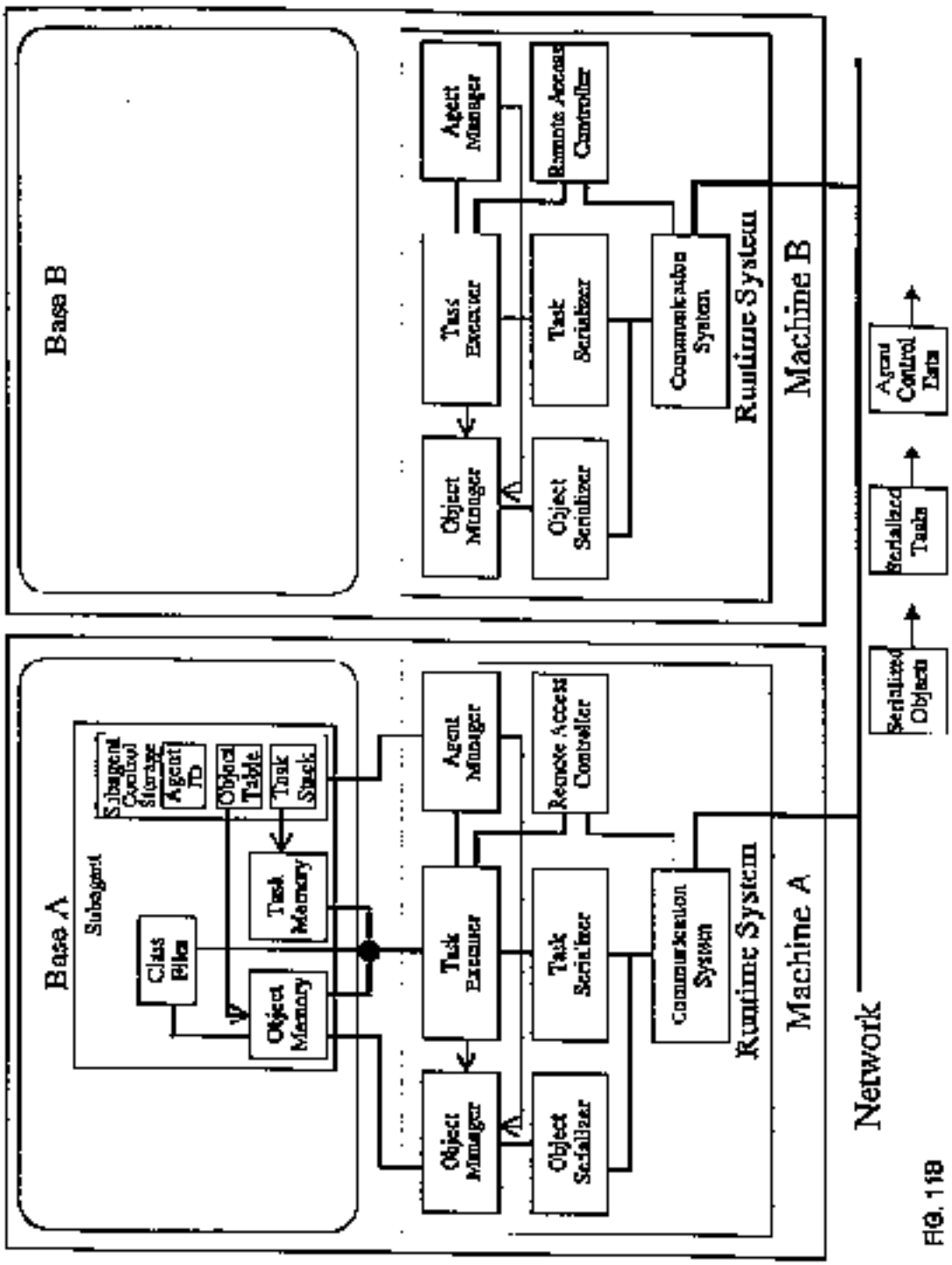
Base B
Agent Manager
Remote Access Controller
Task Executor
Task Serializer
Communication System
Runtime System
Machine B
Object Manager
Object Serializer
Agent Control Data
Serialized Tasks
Serialized Objects
Network
Base A
Subagent
Subagent Control Storage
Agent ID
Object Table
Task Stack
Class Files
Task Memory
Object Memory
Agent Manager
Remote Access Controller
Task Executor
Task Serializer
Communication System
Runtime System
Machine A
Object Manager
Object Serializer

FIG. 11B

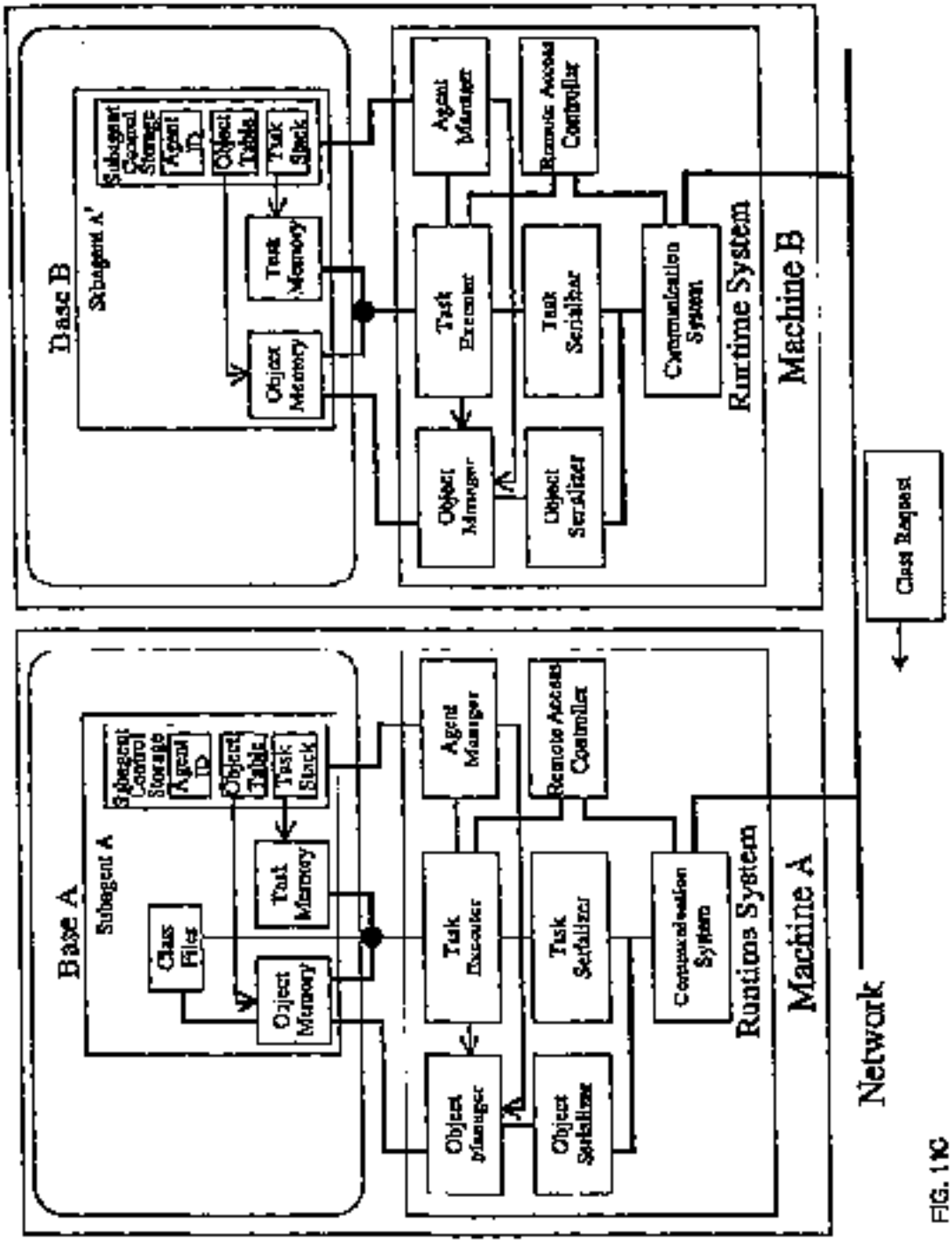
Base B
Subagent A'
Subagent Control Storage
Agent ID
Object Table
Task Stack
Task Memory
Object Memory
Agent Manager
Remote Access Controller
Task Executor
Task Serializer
Communication System
Runtime System
Machine B
Object Manager
Object Serializer
Class Request
Base A
Subagent A
Class Files
Network
Machine A

FIG. 1C

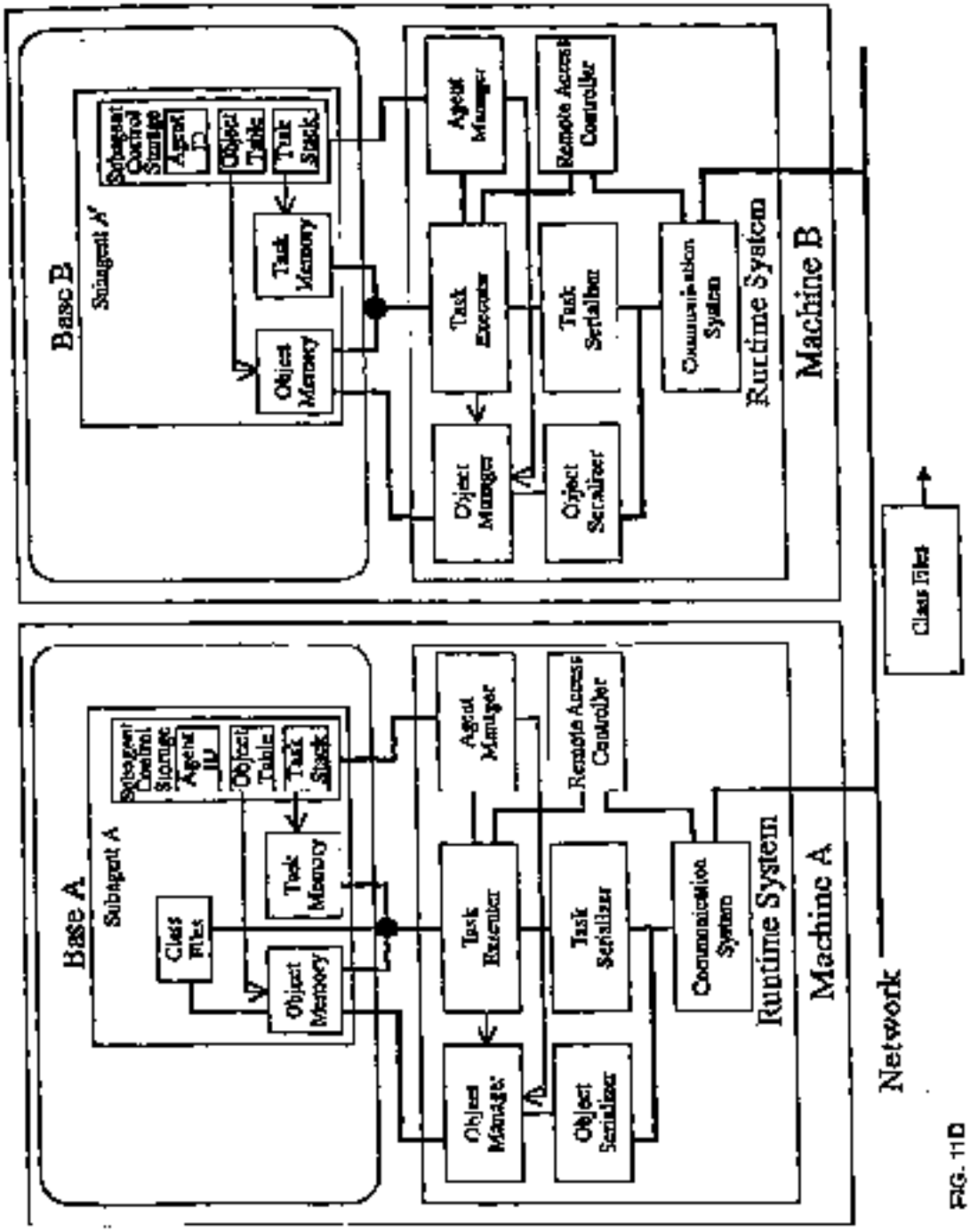
Base B
Subagent A'
Subagent Control Storage
Agent ID
Object Table
Task Stack
Task Memory
Object Memory
Agent Manager
Remote Access Controller
Task Executor
Task Serializer
Communication System
Runtime System
Machine B
Object Manager
Object Serializer
Class Files
Network
Base A
Subagent A
Subagent Control Storage
Agent ID
Object Table
Task Stack
Task Memory
Class Files
Object Memory
Agent Manager
Remote Access Controller
Task Executor
Task Serializer
Communication System
Runtime System
Machine A
Object Manager
Object Serializer

FIG. 11D

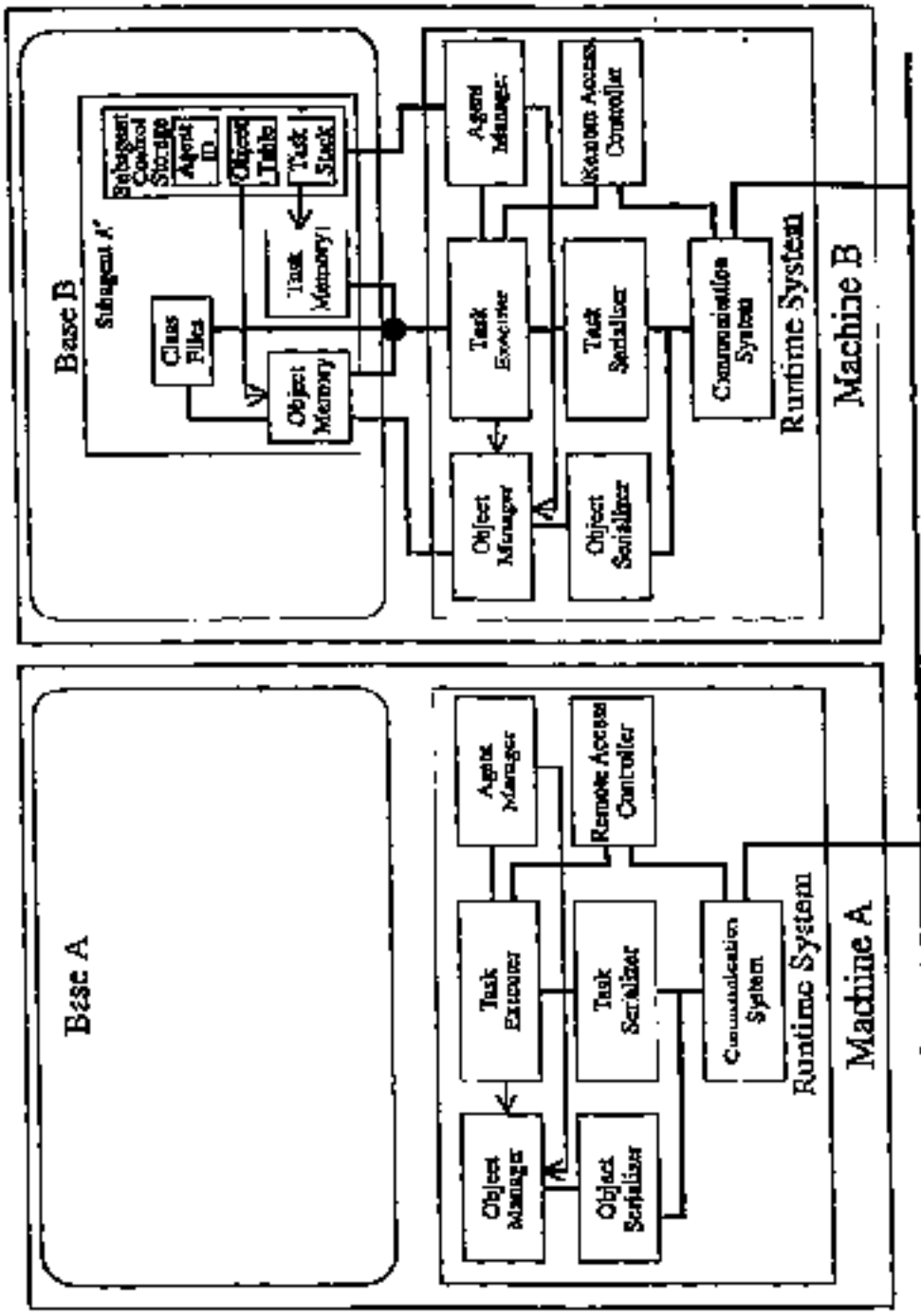
Base A
Machine A
Runtime System
Agent Manager
Remote Access Controller
Task Executor
Task Serializer
Communication System
Object Manager
Object Serializer
Network
Base B
Machine B
Runtime System
Agent Manager
Remote Access Controller
Task Executor
Task Serializer
Communication System
Object Manager
Object Serializer
Class Files
Object Memory
Task Memory
Delegent Control Storage
Agent ID
Object Table
Task Stack

FIG. 11E

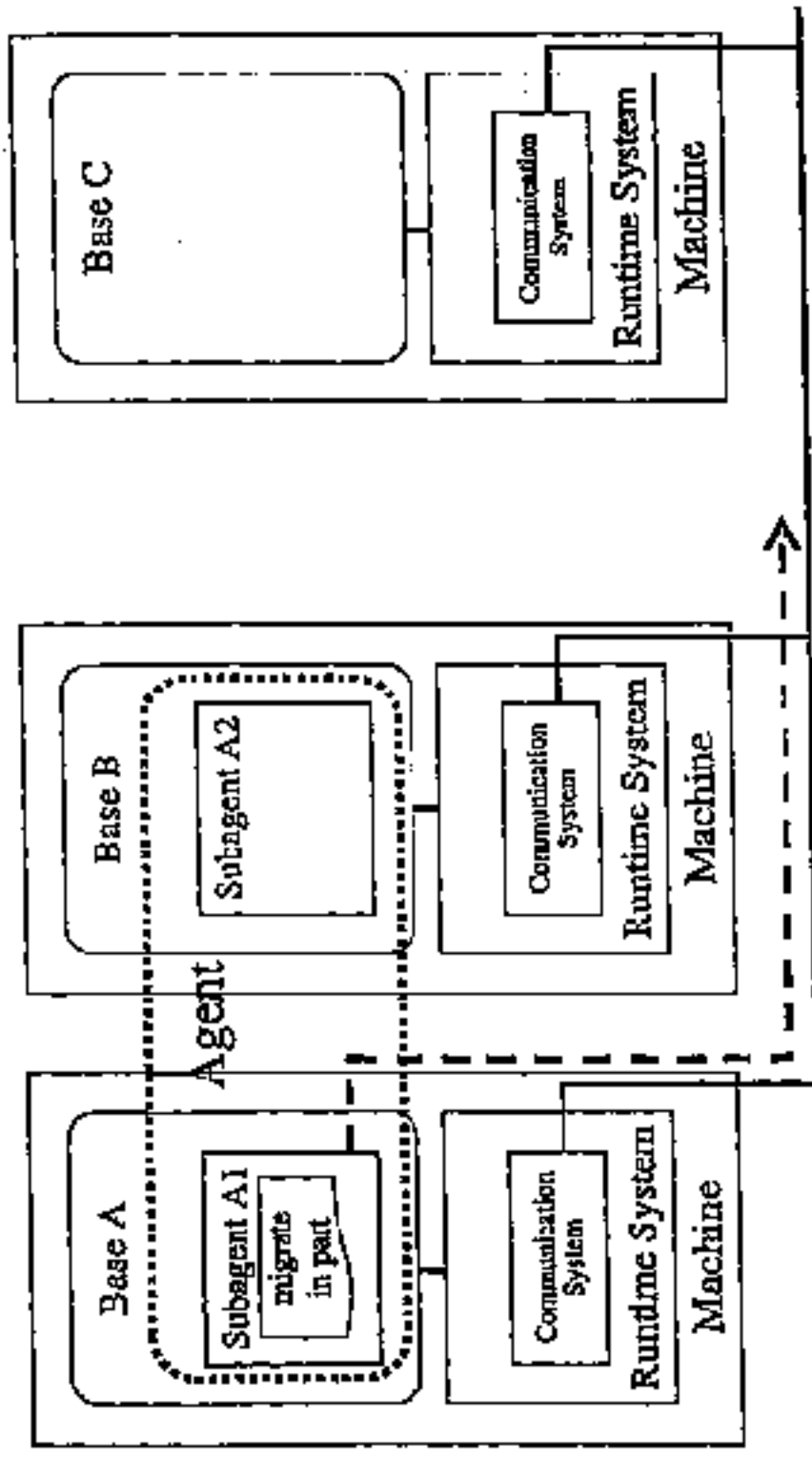
Base C
Communication System
Runtime System
Machine
Base B
Subagent A2
Communication System
Runtime System
Machine
Agent
Base A
Subagent A1
migrate in part
Communication System
Runtime System
Machine
Network

FIG. 12A

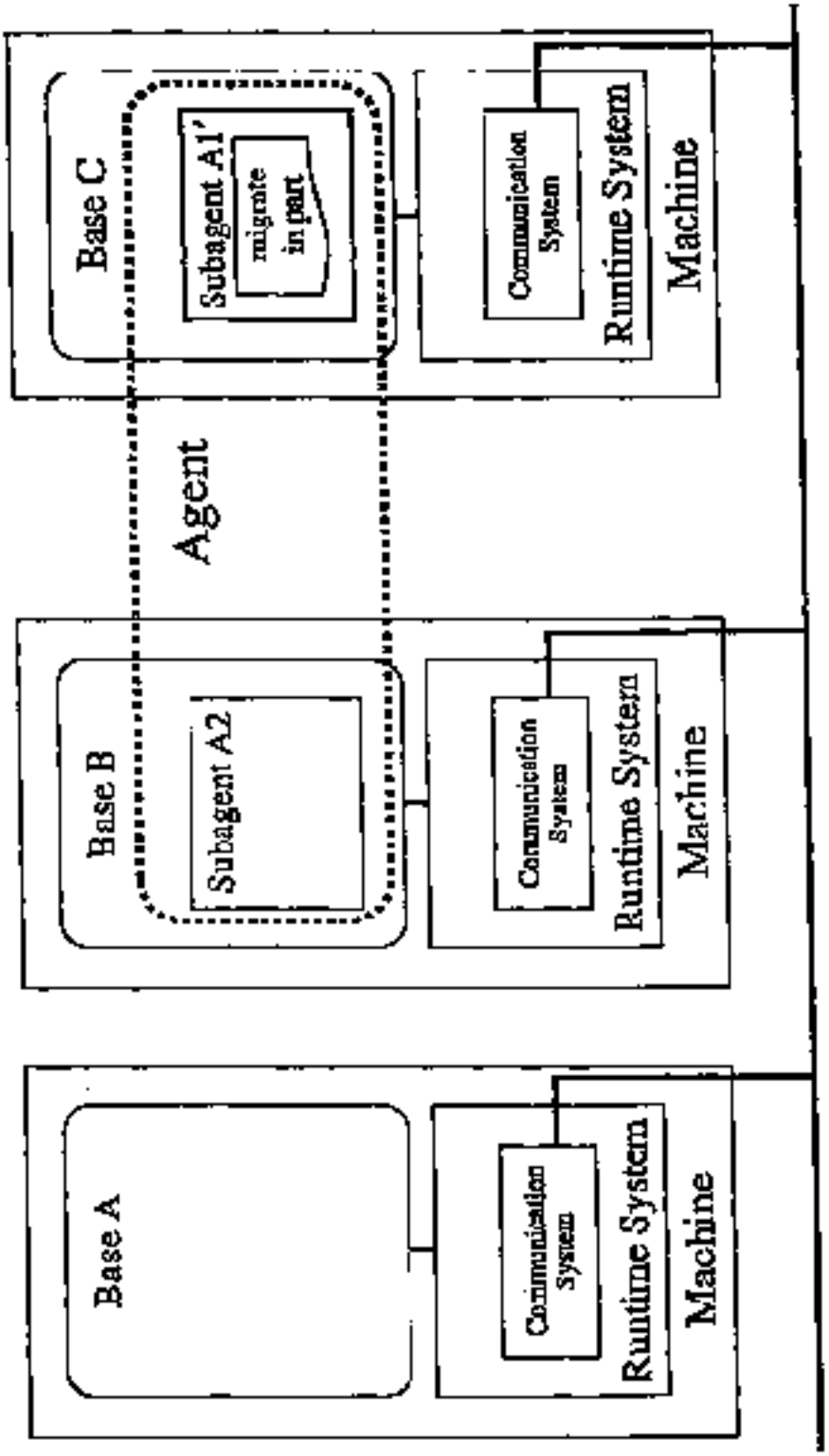
Base C
Subagent A1'
migrate in part
Communication System
Runtime System
Machine
Agent
Base B
Subagent A2
Communication System
Runtime System
Machine
Base A
Communication System
Runtime System
Machine
Network

FIG. 12B

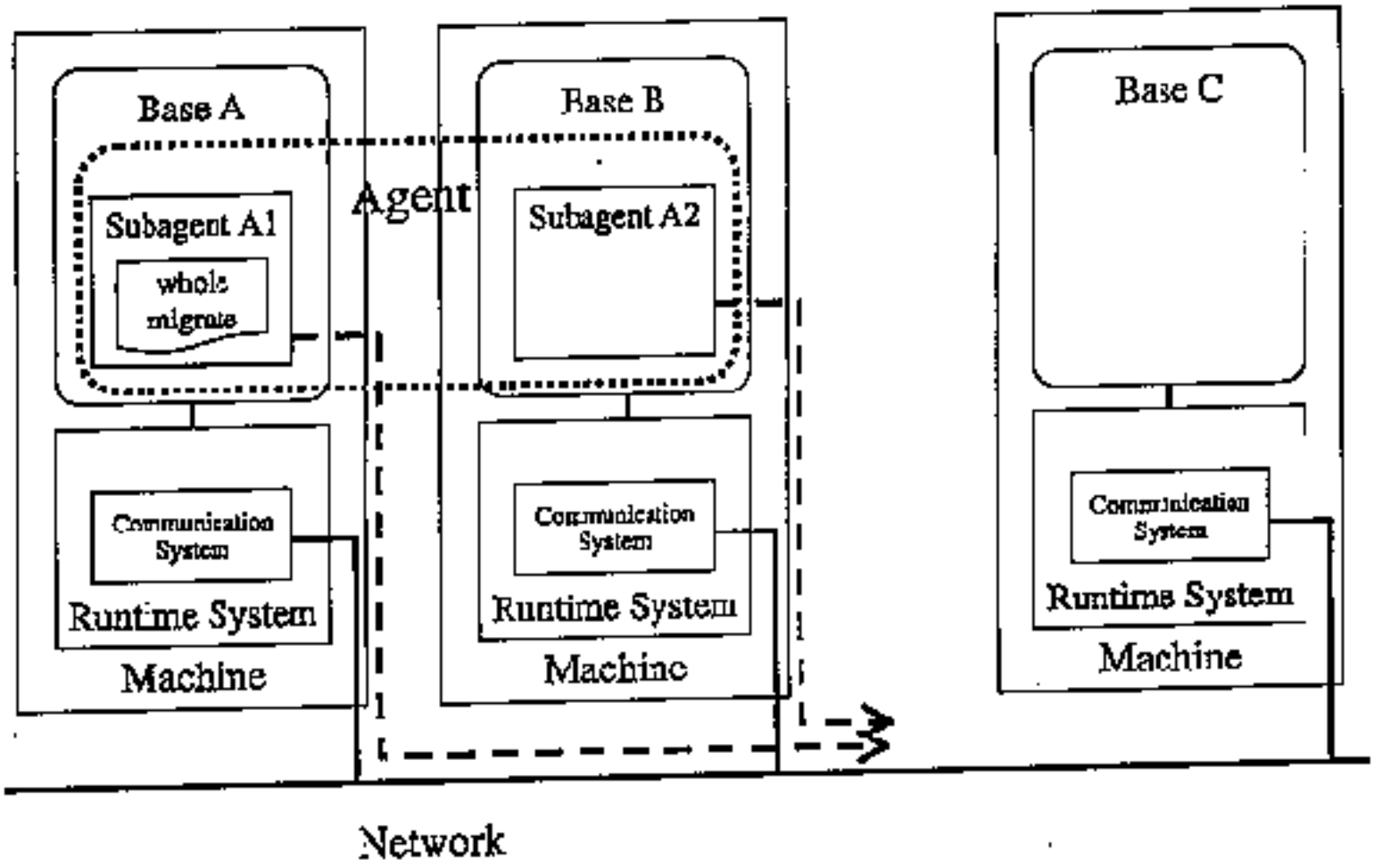
Base A
Base B
Base C
Agent
Subagent A1
whole migrate
Subagent A2
Communication System
Communication System
Communication System
Runtime System
Runtime System
Runtime System
Machine
Machine
Machine
Network

FIG. 13A

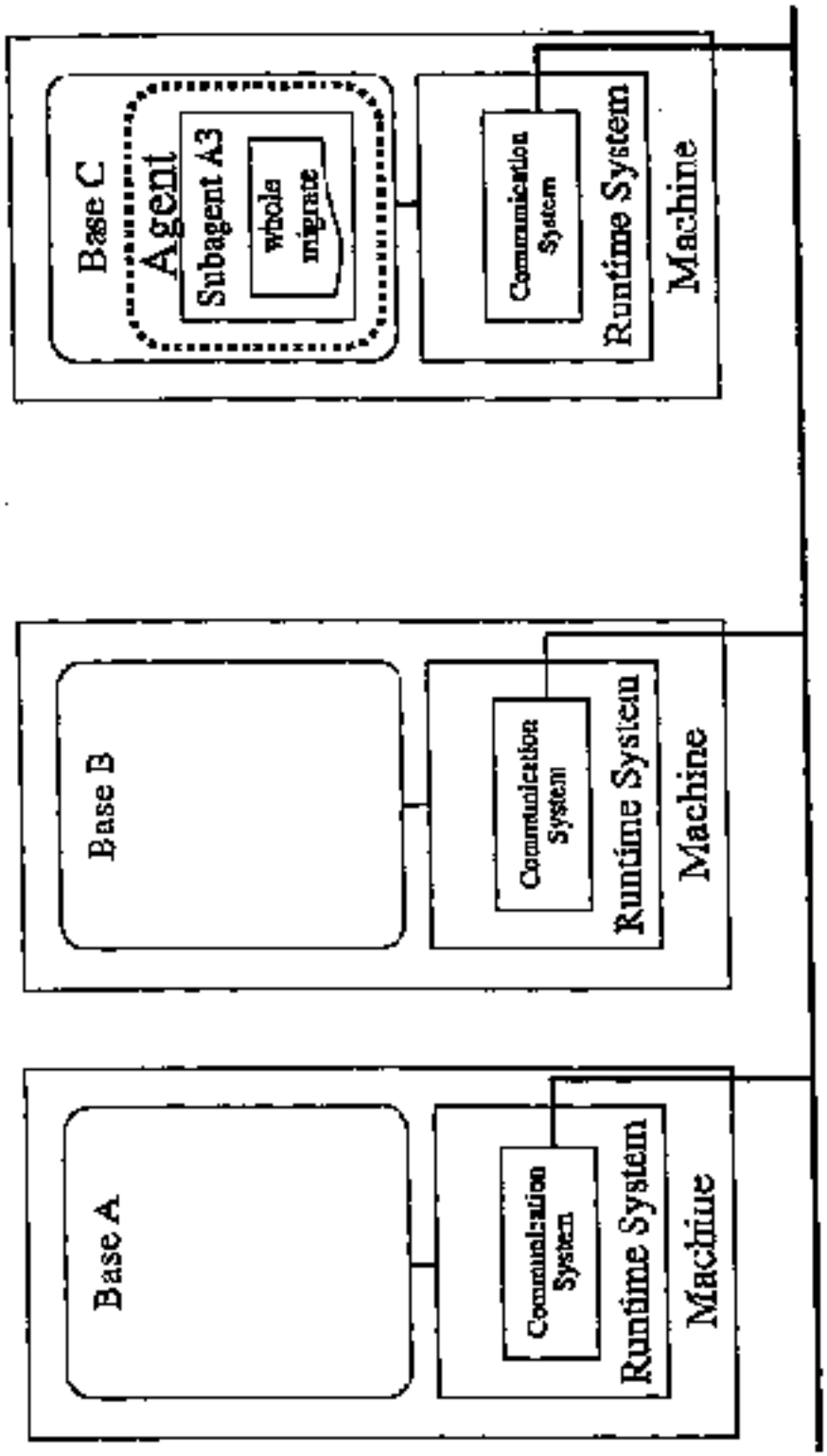
Base C
Agent
Subagent A3
whole migrate
Communication System
Runtime System
Machine
Base B
Communication System
Runtime System
Machine
Base A
Communication System
Runtime System
Machine
Network

FIG. 13B

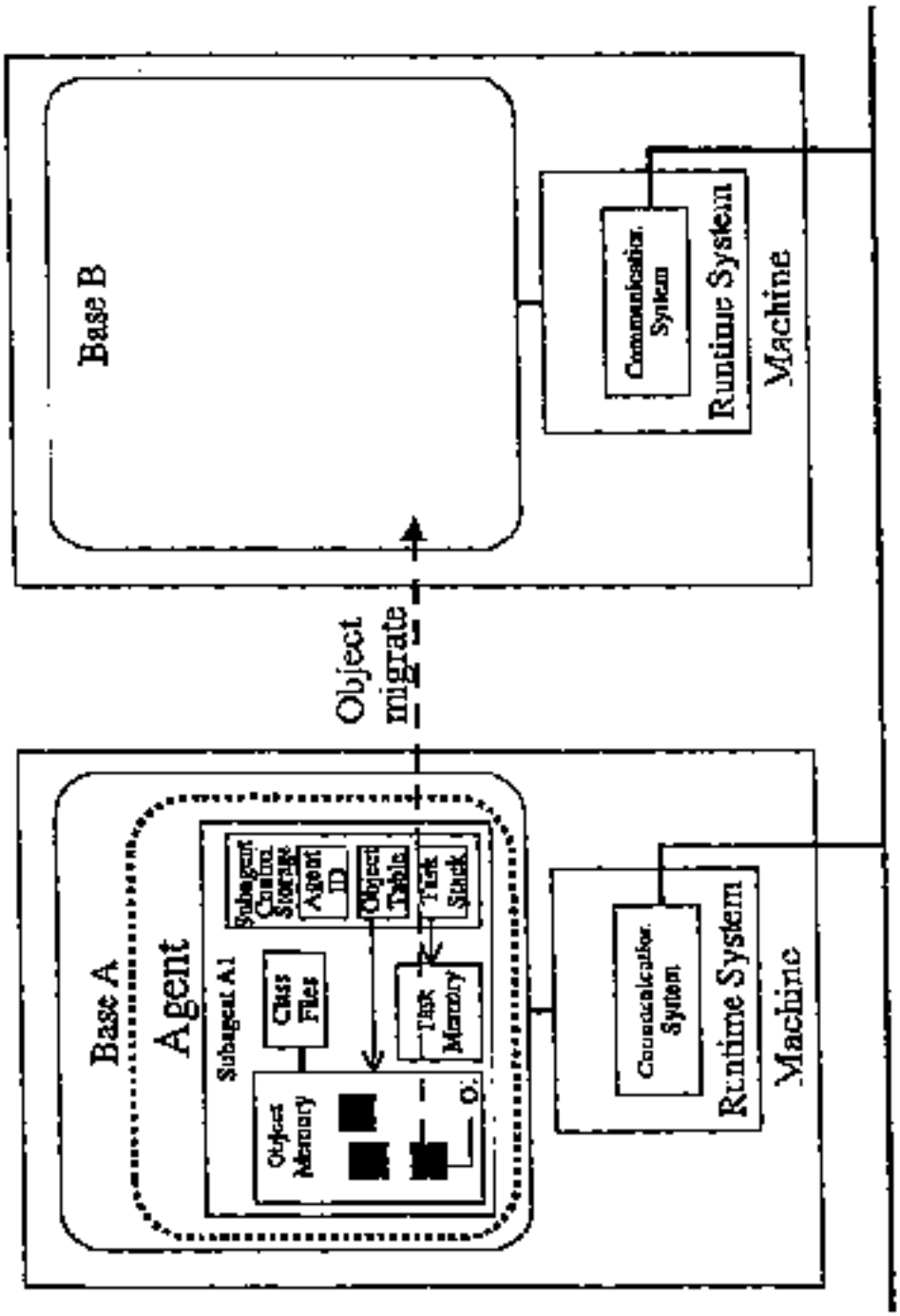
Base B
Communication System
Runtime System
Machine
Object migrate
Base A
Agent
Subagent A1
Subagent Creation Stack
Agent ID
Object Table
Task Stack
Class Files
Task Memory
Object Memory
O
Communication System
Runtime System
Machine
Network

FIG. 14A

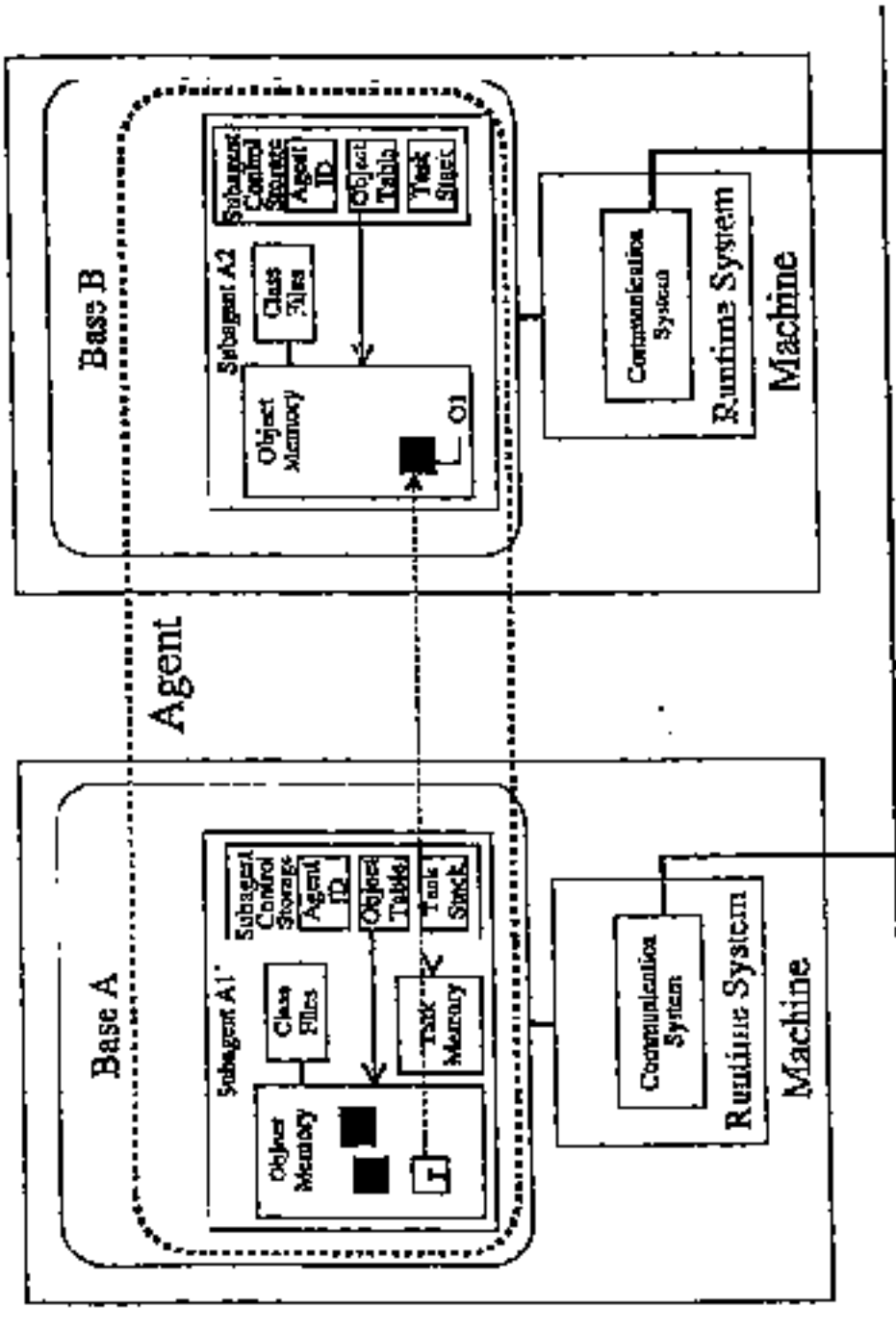
Base B
Subagent A2
Subagent Control Storage
Agent ID
Object Table
Task Stack
Class Files
Object Memory
O1
Communication System
Runtime System
Machine
Agent
Base A
Subagent A1'
Subagent Control Storage
Agent ID
Object Table
Task Stack
Class Files
Task Memory
Object Memory
T
Communication System
Runtime System
Machine
Network

FIG. 14B

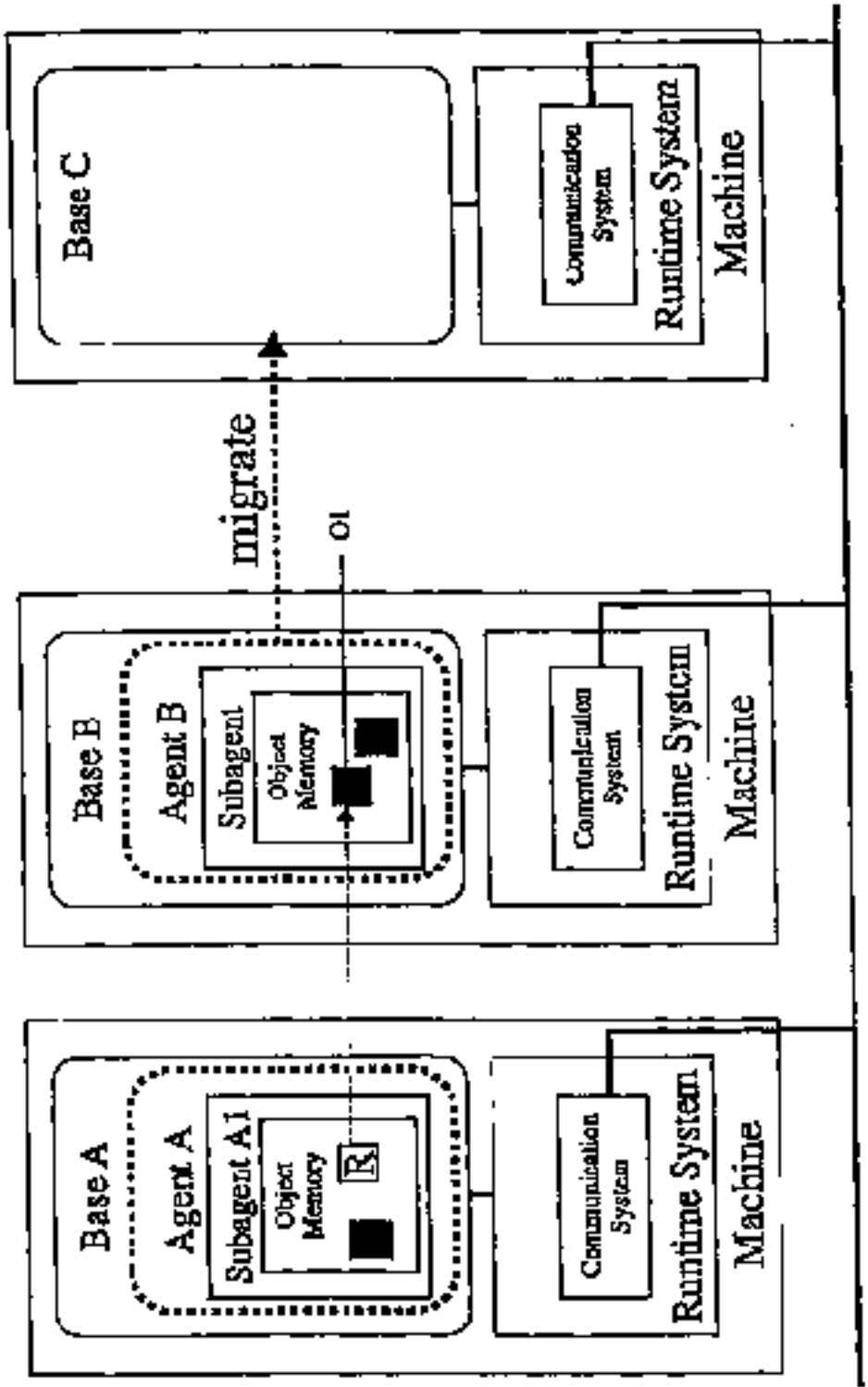
Base C
Communication System
Runtime System
Machine
migrate
OI
Base B
Agent B
Subagent
Object Memory
Communication System
Runtime System
Machine
Base A
Agent A
Subagent A1
Object Memory
R
Communication System
Runtime System
Machine
Network

FIG. 15A

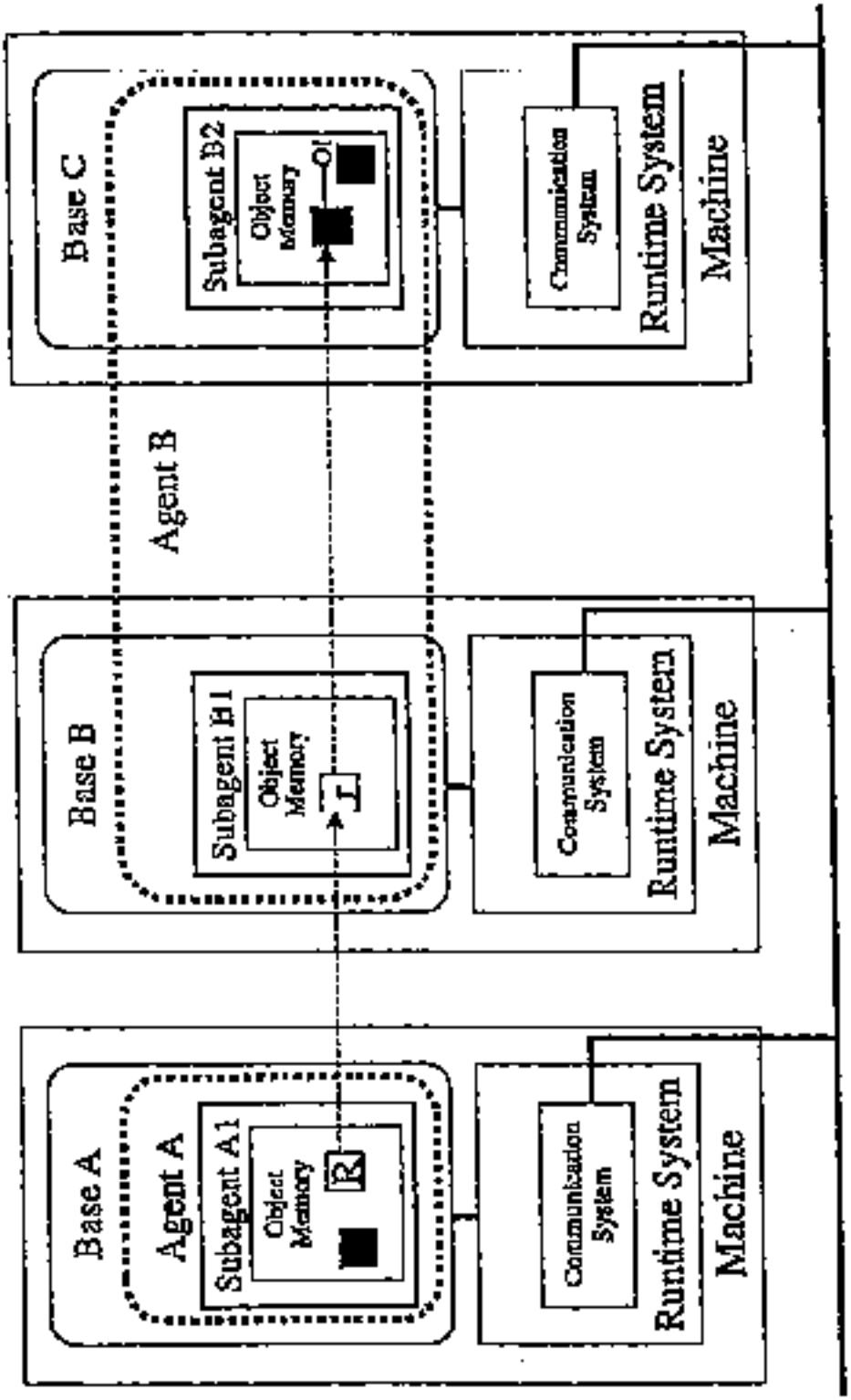
Base C
Subagent B2
Object Memory
O1
Communication System
Runtime System
Machine
Agent B
Base B
Subagent B1
Object Memory
I
Communication System
Runtime System
Machine
Base A
Agent A
Subagent A1
Object Memory
R
Communication System
Runtime System
Machine
Network

FIG. 15B

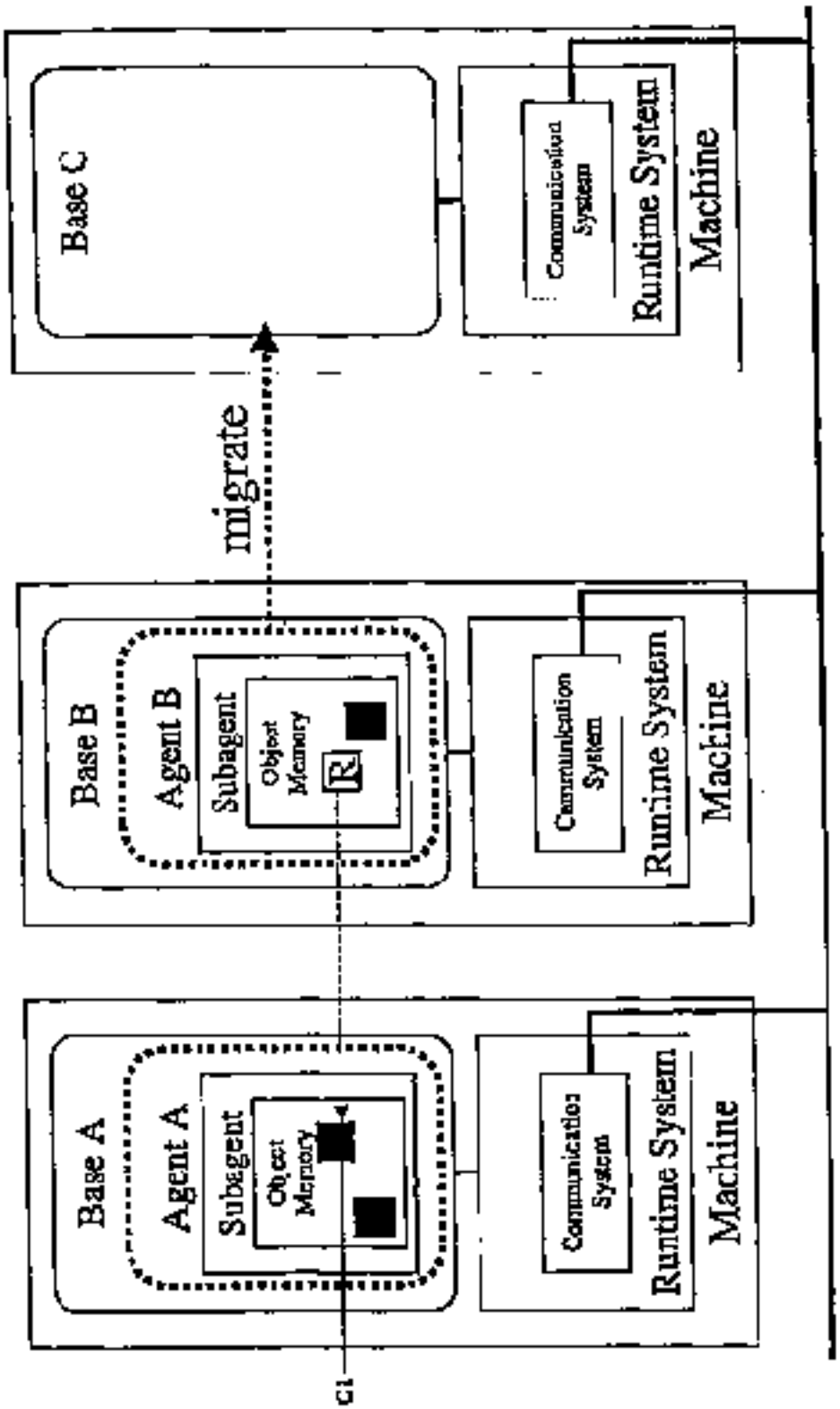
Base C
Communication System
Runtime System
Machine
migrate
Base B
Agent B
Subagent
Object Memory
R
Communication System
Runtime System
Machine
Network
Base A
Agent A
Subagent
Object Memory
Communication System
Runtime System
Machine
C1

FIG. 16A

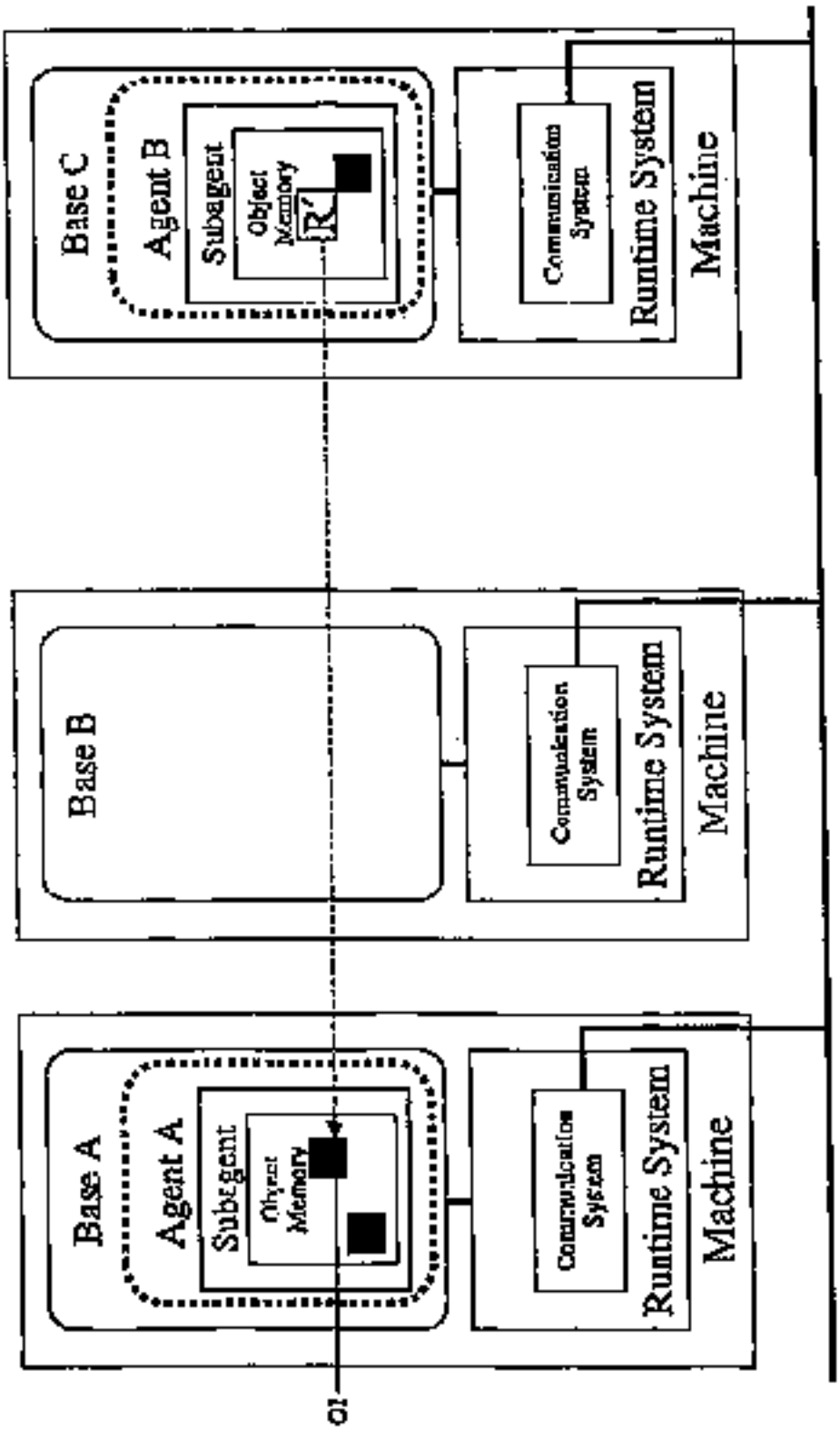
Base C
Agent B
Subagent
Object Memory
R'
Communication System
Runtime System
Machine
Base B
Communication System
Runtime System
Machine
Base A
Agent A
Subagent
Object Memory
Communication System
Runtime System
Machine
Network
C1

FIG. 16B

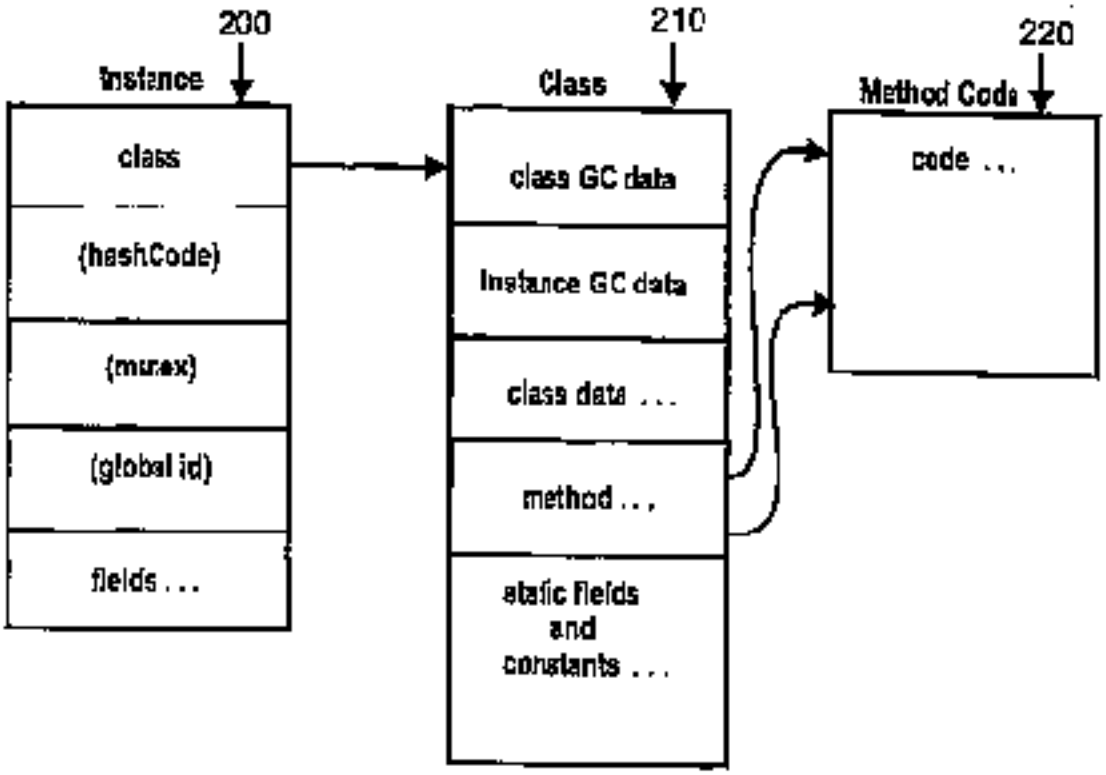
200
Instance
class
(hashCode)
(mutex)
(global id)
fields . . .
210
Class
class GC data
Instance GC data
class data . . .
method . . .
static fields
and
constants . . .
220
Method Code
code . . .

FIG. 17