Base B. After all migration steps are finished, the memory block for the 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 "r" is created in Subagent A1' to Object O1 on Base B to maintain network-transparent references to Object O1 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 "r" 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, run-time 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 convened 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 location 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