MODIFIED COMPUTER ARCHITECTURE FOR A COMPUTER TO OPERATE IN A MULTIPLE COMPUTER SYSTEM

Field of the Invention

The present invention relates to computers and, in particular, to a modified machine architecture which enables the execution of different portions of an application program written to operate only on a single computer, substantially simultaneous on each of a plurality of computers interconnected via a communications network.

Background Art

Ever since the advent of computers, and computing, software for computers has been written to be operated upon a single machine. As indicated in Fig. 1, that single prior art machine 1 is made up from a central processing unit, or CPU, 2 which is connected to a memory 3 via a bus 4. Also connected to the bus 4 are various other functional units of the single machine 1 such as a screen 5, keyboard 6 and mouse 7.

A fundamental limit to the performance of the machine 1 is that the data to be manipulated by the CPU 2, and the results of those manipulations, must be moved by the bus 4. The bus 4 suffers from a number of problems including so called bus "queues" formed by units wishing to gain an access to the bus, conflict or -+contention problems, and the like. These problems can, to some extent, be alleviated by various stratagems including cache memory, however, such stratagems invariably increase the administrative overhead of the machine 1.

Naturally, over the years various attempts have been made to increase machine performance. One approach is to use symmetric multiple processors. This prior art approach has been used in so called "super" computers and is schematically indicated in Fig. 2. Here a plurality of CPU's 12 are connected to global memory 13. Again, a bottleneck arises in the communications between the CPU's 12 and the memory 13. This process has been termed "Single System Image". There is only one application and one whole copy of the memory for the application which is distributed over the global memory. The single application can read from and write to, (ie share) any memory location completely transparently.

Where there are a number of such machines interconnected via a network, this is achieved by taking the single application written for a single machine and partitioning the required memory resources into parts. These parts are then distributed across a number of computers to form the global memory 13 accessible by all CPU's 12. This procedure relies on masking, or hiding, the memory partition from the single running application program. The performance degrades when one CPU on one machine must access (via a network) a memory location physically located in a different machine.

Although super computers have been technically successful in achieving high computational rates, they are not commercially successful in that their inherent complexity makes them extremely expensive not only to manufacture but to administer. In particular, the single system image concept has never been able to scale over "commodity" (or mass produced) computers and networks. Specifically, the Single

System Image concept has only found practical application on very fast (and hence very expensive) computers interconnected by very fast (and similarly expensive) networks.

A further possibility of increased computer power through the use of a plural number of machines arises from the prior art concept of distributed computing which is schematically illustrated in Fig. 3. In this known arrangement, a single application program (Ap) is partitioned by its author (or another programmer who has become familiar with the application program) into various discrete tasks so as to run upon, say, three machines in which case "n" in Fig. 3 is the integer 3. The intention here is that each of the machines M1...M3 runs a different third of the entire application and the intention is that the loads applied to the various machines be approximately equal. The machines communicate via a network 14 which can be provided in various forms such as a communications link, the internet, intranets, local area networks, and the like. Typically the speed of operation of such networks 14 is an order of magnitude slower than the speed of operation of the bus 4 in each of the individual machines M1, M2, etc.

Distributed computing suffers from a number of disadvantages. Firstly, it is a difficult job to partition the application and this must be done manually. Secondly, communicating data, partial results, results and the like over the network 14 is an administrative overhead. Thirdly, the need for partitioning makes it extremely difficult to scale upwardly by utilising more machines since the application having been partitioned into, say three, does not run well upon four machines. Fourthly, in the event that one of the machines should become disabled, the overall performance of the entire system is substantially degraded.

A further prior art arrangement is known as network computing via "clusters" as is schematically illustrated in Fig. 4. In this approach, the entire application is loaded onto each of the machines M1, M2 ....Mn. Each machine communicates with a common database but does not communicate directly with the other machines. Although each machine runs the same application, each machine is doing a different "job" and uses only its own memory. This is somewhat analogous to a number of windows each of which sell train tickets to the public. This approach does operate, is scalable and mainly suffers from the disadvantage that it is difficult to administer the network.

In computer languages such as for example JAVA and MICROSOFT.NET there are two major types of constructs with which programmers deal. In the JAVA language these are known as objects and classes. More generally they may be referred to as assets. Every time an object (or other asset) is created there is an initialization routine run known as an object initialization (e.g., "<init>") routine. Similarly, every time a class is loaded there is a class initialization routine known as "<clinit>". Other languages use different terms but utilize a similar concept. In either case, however, there is no equivalent "clean up" or deletion routine to delete an object or class (or other asset) once it is no longer required. Instead, this "clean up" happens unobtrusively in a background mode.

Furthermore, in any computer environment it is necessary to acquire and release a lock to enable the use of such objects, classes, assets, resources or structures to avoid different parts of the application program from attempting to use the same objects, classes, assets, resources or structures at the one time. In the JAVA environment this is

known as synchronization. Synchronization more generally refers to the exclusive use of an object, class, resource, structure, or other asset to avoid contention between and among computers or machines. This is achieved in JAVA by the "monitor enter" and "monitor exit" instructions or routines. Other languages use different terms but utilize a similar concept.

Unfortunately, conventional computing systems, architectures, and operating schemes do not provide for computing environments and methods in which an application program can operate simultaneously on an arbitrary plurality of computers where the environment and operating scheme ensure that the abovementioned memory management, initialization, clean up and synchronization procedures operate in a consistent and coordinated fashion across all the computing machines.

The genesis of the present invention is a desire to provide a multiple computer system (and related arrangements such as individual computers which can operate in such a system, and a method of operating such computers) which to some extent ameliorates the problems of prior art multiple computer systems.

SUMMARY OF THE INVENTION

The present invention discloses a computing environment in which an application program operates simultaneously on a plurality of computers. In such an environment it is advantageous to ensure that the abovementioned asset initialization, clean-up and synchronization procedures operate in a consistent and coordinated fashion across all the machines.

In accordance with a first aspect of the present invention there is disclosed a single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network, and
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s) whereby the corresponding replicated memory location of each said computer of said multiple system can be updated via said communicating network and all said replicated memory locations can remain substantially identical.

In accordance with a second aspect of the present invention there is disclosed a single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:

a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
initialization means which determine the initial content or value of said replicated memory location and which can be disabled.

In accordance with a third aspect of the present invention there is disclosed a A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
finalization means which deletes said replicated memory location when all said computers no longer need to refer thereto, said finalization means being connected to said communications port to receive therefrom data transmitted over said network relating to continued reference of other computers of said multiple computer system to said replicated memory location.

In accordance with a fourth aspect of the present invention there is disclosed a A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
lock acquisition and relinquishing means to respectively permit said replicated local memory location to be written to, and prevent said replicated local memory being written to, on command.

In accordance with a fifth aspect of the present invention there is disclosed a single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location

is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s) whereby the corresponding replicated memory location of each said computer of said multiple system can be updated via said communicating network and all said replicated memory locations can remain substantially identical,
initialization means which determine the initial content or value of said replicated memory location and which can be disabled,
finalization means which deletes said replicated memory location when all said computers no longer need to refer thereto, said finalization means being connected to said communications port to receive therefrom data transmitted over said network relating to continued reference of other computers of said multiple computer system to said replicated memory location, and
lock acquisition and relinquishing means to respectively permit said replicated local memory location to be written to, and prevent said replicated local memory being written to, on command.

In accordance with a sixth aspect of the present invention there is disclosed a multiple computer system having at least one application program each written to operate on only a single computer but running simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers, wherein each computer has an independent local memory accessible only by the corresponding portion of said application program(s) and wherein for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer.

In accordance with a seventh aspect of the present invention there is disclosed a plurality of computers interconnected via a communications link and each having an independent local memory and substantially simultaneously operating a different portion at least one application program each written to operate on only a single computer, each local memory being accessible only by the corresponding portion of said application program.

In accordance with a eighth aspect of the present invention there is disclosed a multiple computer system having at least one application program each written to operate on only a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and wherein the initial contents of each of said identically named objects is substantially the same.

In accordance with a ninth aspect of the present invention there is disclosed a plurality of computers interconnected via a communications link and substantially simultaneously operating at least one application program each written to operation on only a single computer wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion creates objects only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer are fundamentally similar but not, at each instant, identical, and every one of said computers has distribution update means to distribute to all other said computers objects created by said one computer.

In accordance with a tenth aspect of the present invention there is disclosed a multiple computer system having at least one application program each written to operate only on a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and wherein all said identical objects are collectively deleted when each one of said plurality of computers no longer needs to refer to their corresponding object.

In accordance with an eleventh aspect of the present invention there is disclosed a plurality of computers interconnected via a communications link and operating substantially simultaneously at least one application program each written to operate only on a single computer, wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion needs, or no longer needs to refer to an object only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has a finalization routine which deletes a non-referenced object only if each one of said plurality of computers no longer needs to refer to their corresponding object.

In accordance with a twelfth aspect of the present invention there is disclosed a multiple computer system having at least one application program each written to operate on only a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and said system including a lock means applicable to all said computers wherein any computer wishing to utilize a named object therein acquires an authorizing lock from said lock means which permits said utilization and which prevents all the other computers from utilizing their corresponding named object until said authorizing lock is relinquished.

In accordance with a thirteenth aspect of the present invention there is disclosed a plurality of computers interconnected via a communications link and operating

substantially simultaneously at least one application program each written to operate on only a single computer, wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion utilizes an object only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has an acquire lock routine and a release lock routine which permit utilization of the local object only by one computer and each of the remainder of said plurality of computers is locked out of utilization of their corresponding object.

In accordance with a fourteenth aspect of the present invention there is disclosed a method of running simultaneously on a plurality of computers at least one application program each written to operate on only a single computer, said computers being interconnected by means of a communications network, said method comprising the step of,

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each accessible only by the corresponding portion of said application program.

In accordance with a fifteenth aspect of the present invention there is disclosed a method of loading an application program written to operate only on a single computer onto each of a plurality of computers, the computers being interconnected via a communications link, and different portions of said application program(s) being substantially simultaneously executable on different computers with each computer having an independent local memory accessible only by the corresponding portion of said application program(s), the method comprising the step of modifying the application before, during, or after loading and before execution of the relevant portion of the application program.

In accordance with a sixteenth aspect of the present invention there is disclosed a method of operating simultaneously on a plurality of computers all interconnected via a communications link at least one application program each written to operate on only a single computer, each of said computers having at least a minimum predetermined local memory capacity, different portions of said application program(s) being substantially simultaneously executed on different ones of said computers with the local memory of each computer being only accessible by the corresponding portion of said application program(s), said method comprising the steps of:

(i) initially providing each local memory in substantially identical condition,

(ii) satisfying all memory reads and writes generated by each said application program portion from said corresponding local memory, and

(iii) communicating via said communications link all said memory writes at each said computer which take place locally to all the remainder of said plurality of computers whereby the contents of the local memory utilised by each said computer, subject to an updating data transmission delay, remains substantially identical.

In accordance with a seventeenth aspect of the present invention there is disclosed a method of compiling or modifying an application program written to operate on only a

single computer but to run simultaneously on a plurality of computers interconnected via a communications link, with different portions of said application program(s) executing substantially simultaneously on different ones of said computers each of which has an independent local memory accessible only by the corresponding portion of said application program, said method comprising the steps of:

(i) detecting instructions which share memory records utilizing one of said computers,

(ii) listing all such shared memory records and providing a naming tag for each listed memory record,

(iii) detecting those instructions which write to, or manipulate the contents of, any of said listed memory records, and

(iv) activating an updating propagation routine following each said detected write or manipulate instruction, said updating propagation routine forwarding the re-written or manipulated contents and name tag of each said re-written or manipulated listed memory record to the remainder of said computers.

In accordance with an eighteenth aspect of the present invention there is disclosed a multiple thread processing computer operation in which individual threads of a single application program written to operate on only a single computer are simultaneously being processed each on a different corresponding one of a plurality of computers each having an independent local memory accessible only by the corresponding thread and each being interconnected via a communications link, the improvement comprising communicating changes in the contents of local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

In accordance with a nineteenth aspect of the present invention there is disclosed a method of running substantially simultaneously on a plurality of computers at least one application program each written to operate on only a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) creating the initial contents of each of said identically named objects substantially the same.

In accordance with a twentieth aspect of the present invention there is disclosed a method of compiling or modifying an application program written to operate on only a single computer to have different portions thereof to execute substantially simultaneously on different ones of a plurality of computers interconnected via a communications link, said method comprising the steps of:

(i) detecting instructions which create objects utilizing one of said computers,

(ii) activating an initialization routine following each said detected object creation instruction, said initialization routine forwarding each created object to the remainder of said computers.

In accordance with a twenty first aspect of the present invention there is disclosed a multiple thread processing computer operation in which individual threads of a single application program written to operate on only a single computer are substantially simultaneously being processed each on a different corresponding one of a plurality of computers interconnected via a communications link, the improvement comprising communicating objects created in local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

In accordance with a twenty second aspect of the present invention there is disclosed a method of ensuring consistent initialization of an application program written to operate on only a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:
(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an initialization routine, and
(ii) modifying said initialization routine to ensure consistent operation of all said computers.

In accordance with a twenty third aspect of the present invention there is disclosed a method of running substantially simultaneously on a plurality of computers at least one application program each written to operate only on a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:
(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and
(ii) deleting all said identical objects collectively when all of said plurality of computers no longer need to refer to their corresponding object.

In accordance with a twenty fourth aspect of the present invention there is disclosed a method of ensuring consistent finalization of an application program written to operate only on a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:
(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an finalization routine, and
(ii) modifying said finalization routine to ensure collective deletion of corresponding objects in all said computers only when each one of said computers no longer needs to refer to their corresponding object.

In accordance with a twenty fifth aspect of the present invention there is disclosed a multiple thread processing computer operation in which individual threads of a single application program written to operate only on a single computer are substantially simultaneously being processed each on a corresponding different one of a plurality of computers interconnected via a communications link, and in which objects in local memory physically associated with the computer processing each thread have

corresponding objects in the local memory of each other said computer, the improvement comprising collectively deleting all said corresponding objects when each one of said plurality of computers no longer needs to refer to their corresponding object.

In accordance with a twenty sixth aspect of the present invention there is disclosed a method of running substantially simultaneously on a plurality of computers at least one application program each written to operate only on a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) requiring any of said computers wishing to utilize a named object therein to acquire an authorizing lock which permits said utilization and which prevents all the other computers from utilizing their corresponding named object until said authorizing lock is relinquished.

In accordance with a twenty seventh aspect of the present invention there is disclosed a method of ensuring consistent synchronization of an application program written to operate only on a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:

(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an synchronization routine, and

(ii) modifying said synchronization routine to ensure utilization of an object by only one computer and preventing all the remaining computers from simultaneously utilizing their corresponding objects.

In accordance with a twenty eighth aspect of the present invention there is disclosed a multiple thread processing computer operation in which individual threads of a single application program written to operate only on a single computer are substantially simultaneously being processed each on a corresponding different one of a plurality of computers interconnected via a communications link, and in which objects in local memory physically associated with the computer processing each thread have corresponding objects in the local memory of each other said computer, the improvement comprising permitting only one of said computers to utilize an object and preventing all the remaining computers from simultaneously utilizing their corresponding object.

In accordance with a twenty ninth aspect of the present invention there is disclosed a computer program product comprising a set of program instructions stored in a storage medium and operable to permit one or a plurality of computers to carry out the abovementioned methods.

In accordance with a thirtieth aspect of the invention there is disclosed a distributed run time and distributed run time system adapted to enable communications between a plurality of computers, computing machines, or information appliances.

In accordance with a thirty first aspect of the invention there is disclosed a modifier, modifier means, and modifier routine for modifying an application program written to execute on a single computer or computing machine whereby the modified application program executes substantially simultaneously on a plurality of networked computers or computing machines.

In accordance with a thirty second aspect of the present invention there is disclosed a computer program and computer program product written to operate on only a single computer but product comprising a set of program instructions stored in a storage medium and operable to permit a plurality of computers to carry out the abovementioned procedures, routines, and methods.

Brief Description of the Drawings

Embodiments of the present invention will now be described with reference to the drawings in which:

Fig. 1 is a schematic view of the internal architecture of a conventional computer,

Fig. 2 is a schematic illustration showing the internal architecture of known symmetric multiple processors,

Fig. 3 is a schematic representation of prior art distributed computing,

Fig. 4 is a schematic representation of a prior art network computing using clusters,

Fig. 5 is a schematic block diagram of a plurality of machines operating the same application program in accordance with a first embodiment of the present invention,

Fig. 6 is a schematic illustration of a prior art computer arranged to operate JAVA code and thereby constitute a JAVA virtual machine,

Fig. 7 is a drawing similar to Fig. 6 but illustrating the initial loading of code in accordance with the preferred embodiment,

Fig. 8 is a drawing similar to Fig. 5 but illustrating the interconnection of a plurality of computers each operating JAVA code in the manner illustrated in Fig. 7,

Fig. 9 is a flow chart of the procedure followed during loading of the same application on each machine in the network,

Fig. 10 is a flow chart showing a modified procedure similar to that of Fig. 9,

Fig. 11 is a schematic representation of multiple thread processing carried out on the machines of Fig. 8 utilizing a first embodiment of memory updating,

Fig. 12 is a schematic representation similar to Fig. 11 but illustrating an alternative embodiment,

Fig. 13 illustrates multi-thread memory updating for the computers of Fig. 8,

Fig. 14 is a schematic illustration of a prior art computer arranged to operate in JAVA code and thereby constitute a JAVA virtual machine,

Fig. 15 is a schematic representation of n machines running the application program and serviced by an additional server machine X,

Fig. 16 is a flow chart of illustrating the modification of initialization routines,

Fig. 17 is a flow chart illustrating the continuation or abortion of initialization routines,

Fig. 18 is a flow chart illustrating the enquiry sent to the server machine X,

Fig. 19 is a flow chart of the response of the server machine X to the request of Fig. 18,

Fig. 20 is a flowchart illustrating a modified initialization routine for the <clinit>

instruction,

Fig. 21 is a flowchart illustrating a modified initialization routine for the <init> instruction,

Fig. 22 is a flow chart of illustrating the modification of "clean up" or finalization routines,

Fig. 23 is a flow chart illustrating the continuation or abortion of finalization routines,

Fig. 24 is a flow chart illustrating the enquiry sent to the server machine X,

Fig. 25 is a flow chart of the response of the server machine X to the request of Fig. 24,

Fig. 26 is a flow chart of illustrating the modification of the monitor enter and exit routines,

Fig. 27 is a flow chart illustrating the process followed by processing machine in requesting the acquisition of a lock,

Fig. 28 is a flow chart illustrating the requesting of the release of a lock,

Fig. 29 is a flow chart of the response of the server machine X to the request of Fig. 27,

Fig. 30 is a flow chart illustrating the response of the server machine X to the request of Fig. 28,

Fig. 31 is a schematic representation of two laptop computers interconnected to simultaneously run a plurality of applications, with both applications running on a single computer,

Fig. 32 is a view similar to Fig. 31 but showing the Fig. 31 apparatus with one application operating on each computer, and

Fig. 33 is a view similar to Figs. 31 and 32 but showing the Fig. 31 apparatus with both applications operating simultaneously on both computers.

REFERENCE TO ANNEXES

Although the specification provides a complete and detailed description of the several embodiments of the invention such that the invention may be understood and implemented without reference to other materials, the specification does includes Annexures A, B, C and D which provide exemplary actual program or code fragments which implement various aspects of the described embodiments. Although aspects of the invention are described throughout the specification including the Annexes, drawings, and claims, it may be appreciated that Annexure A relates primarily to fields, Annexure B relates primarily to initialization, Annexure C relates primarily to finalization, and Annexure D relates primarily to synchronization. More particularly, the accompanying Annexures are provided in which:

Annexures A1-A10 illustrate exemplary code to illustrate embodiments of the invention in relation to fields.

Annexure B1 is an exemplary typical code fragment from an unmodified class initialization <clinit> instruction, Annexure B2 is an equivalent in respect of a modified class initialization <clinit> instruction. Annexure B3 is a typical code fragment from an unmodified object initialization <init> instruction. Annexure B4 is an equivalent in respect of a modified object initialization <init> instruction. In addition, Annexure B5 is an alternative to the code of Annexure B2 for an unmodified class initialization

instruction, and Annexure B6 is an alternative to the code of Annexure B4 for a modified object initialization <init> instruction. Furthermore, Annexure B7 is exemplary computer program source-code of InitClient, which queries an "initialization server" for the initialization status of the relevant class or object. Annexure B8 is the computer program source-code of InitServer, which receives an initialization status query by InitClient and in response returns the corresponding status. Similarly, Annexure B9 is the computer program source-code of the example application used in the before/after examples of Annexure B1-B6.

It will be appreciated in light of the description provided here that the categorization of the Annexures as well as the use of other headings and subheadings in this description is intended as an aid to the reader and is not to be used to limit the scope of the invention in any way.

Detailed Description

The present invention discloses a modified computer architecture which enables an applications program to be run simultaneously on a plurality of computers in a manner that overcomes the limitations of the aforedescribed conventional architectures, systems, methods, and computer programs.

In one aspect, shared memory at each computer may be updated with amendments and/or overwrites so that all memory read requests are satisfied locally. Before, during or after program loading, but before execution of relevant portions of the program code are executed, or similar, instructions which result in memory being re-written or manipulated are identified. Additional instructions are inserted into the program code (or other modification made) to cause the equivalent memory locations at all computers to be updated. While the invention is not limited to JAVA language or virtual machines, exemplary embodiments are described relative to the JAVA language and standards. In another aspect, the initialization of JAVA language classes and objects (or other assets) are provided for so all memory locations for all computers are initialized in the same manner. In another aspect, the finalization of JAVA language classes and objects is also provide so finalization only occurs when the last class or object present on all machines is no longer required. In still another aspect, synchronization is provided such that instructions which result in the application program acquiring (or releasing) a lock on a particular asset (synchronization) are identified. Additional instructions are inserted (or other code modifications performed) to result in a modified synchronization routine with which all computers are updated.

The present invention also discloses a computing environment and computing method in which an application program operates simultaneously on a plurality of computers. In such an environment it is advantageous to ensure that the abovementioned initialization, clean-up and synchronization procedures operate in a consistent and coordinated fashion across all the machines. These memory replication, object or other asset initialization, finalization, and synchronization may be used and applied separately in a variety of computing and information processing environments. Furthermore, they may advantageously be implemented and applied in any combination so as to provide synergistic effects for multi-computer processing, such as network based distributed computing.

As each of the architectural, system, procedural, method and computer program aspects of the invention (e.g., memory management and replication, initialization, finalization, and synchronization) may be applied separately, they are thus first described without specific reference to the other aspects. It will however be appreciated in light of the descriptions provided that the object, class, or other asset creation or initialization may generally precede finalization of such objects, classes, or other assets.

In addition, during the loading of, or at any time preceding the execution of, the application code 50 (or relevant portion thereof) on each machine M1, M2...Mn, each application code 50 has been modified by the corresponding modifier 51 according to the same rules (or substantially the same rules since minor optimizing changes are permitted within each modifier 51/1, 51/2, ..., 51/n). Where separate modifications are required on any particular machine, such as to machine M2, to effect the memory management, initialization, finalization, and/or synchronization for that machine, then each machine may in fact have and be modified according to a plurality of separate modifiers (such as 51/2-M (e.g., M2 memory management modifier), 51/2-I (e.g., M2 initialization modifier), 51/2-F (e.g., M2 finalization modifier), and/or 51/2-S (e.g., M2 synchronization modifier); or alternatively any one or more of these modifiers may be combined into a combined modifier for that computer or machine. In at least some embodiments, efficiencies will result from performing the steps required to identify the modification required, in performing the actual modification, and in coordinating the operation of the plurality or constellation of computers or machines in an organized, consistent, and coherent manner. These modifications may be performed in accordance with aspects of the invention by the distributed run time means 71 described in greater detail hereinafter. In analogous manner those workers having ordinary skill in the art in light of the description provided herein will appreciate that the structural and methodological aspects of the distributed run time, distributed run time system, and distributed run time means as they are described herein specifically to memory management, initialization, finalization, and/or synchronization may be combined so any of the modifications required to an application program or code may be made separately or in combination to achieve any required memory management, initialization, finalization, and/or synchronization on any particular machine and across the plurality of machines M1, M2, ..., Mn.

With specific reference to any memory management modifier that may be provided, such memory management modifier 51-M or DRT 71-M or other code modifying means component of the overall modifier or distributed run time means is responsible for creating or replicating a memory structure and contents on each of the individual machines M1, M2...Mn that permits the plurality of machines to interoperate. In some embodiments this replicated memory structure will be identical, in other embodiments this memory structure will have portions that are identical and other portions that are not, and in still other embodiments the memory structures are or may not be identical.

With reference to any initialisation modifier that may be present, such initialisation modifier 51-I or DRT 71-I or other code modifying means component of the overall modifier or distributed run time means is responsible for modifying the

application code 50 so that it may execute initialisation routines or other initialization operations, such as for example class and object initialization methods or routines in the JAVA language and virtual machine environment, in a coordinated, coherent, and consistent manner across the plurality of individual machines M1, M2...Mn.

With reference to the finalization modifier that may be present, such finalization modifier 51-F or DRT 71-F or other code modifying means is responsible for modifying the application code 50 so that the code may execute finalization clean-up, or other memory reclamation, recycling, deletion or finalization operations, such as for example finalization methods in the JAVA language and virtual machine environment, in a coordinated, coherent and consistent manner across the plurality of individual machines M1, M2, ..., Mn.

Furthermore, with reference to any synchronization modifier that may be present, such synchronization modifier 51-S or DRT 71-S or other code modifying means is responsible for ensuring that when a part (such as a thread or process) of the modified application program 50 running on one or more of the machines exclusively utilizes (e.g., by means of a synchronization routine or similar or equivalent mutual exclusion operator or operation) a particular local asset, such as an objects 50X-50Z or class 50A, no other different and potentially concurrently executing part on machines M2...Mn exclusively utilizes the similar equivalent corresponding asset in its local memory at once or at the same time.

These structures and procedures when applied in combination when required, maintain a computing environment where memory locations, address ranges, objects, classes, assets, resources, or any other procedural or structural aspect of a computer or computing environment are where required created, maintained, operated, and deactivated or deleted in a coordinated, coherent, and consistent manner across the plurality of individual machines M1, M2...Mn.

The embodiments will be described with reference to the JAVA language, however, it will be apparent to those skilled in the art that the invention is not limited to this language and, in particular can be used with the similar languages (including procedural, declarative and object oriented languages) including the MICROSOFT.NET platform and architecture (Visual Basic, Visual C, and Visual C++, and Visual C#), FORTRAN, C, C++, COBOL, BASIC and the like.

In connection with Fig. 5, in accordance with a preferred embodiment of the present invention a single application program 50 can be operated simultaneously on a number of computers or machines M1, M2...Mn communicating via network 53. As it will become apparent hereafter, each of the machines M1, M2...Mn operates with the same application program 50 on each machine M1, M2...Mn and thus all of the machines M1, M2...Mn have the same, or substantially the same, application code and data 50. Similarly, each of the machines M1, M2...Mn operates with the same (or substantially the same) modifier 51 on each machine M1, M2...Mn and thus all of the machines M1, M2...Mn have the same (or substantially the same) modifier 51 with the modifier of machine M2 being designated 51/2. In addition, during the loading of, or preceding the execution of, the application 50 on each machine M1, M2...Mn, each application 50 has

been modified by the corresponding modifier 51 according to the same rules (or substantially the same rules since minor optimising changes are permitted within each modifier 51/1 ...51/n).

As a consequence of the above described arrangement, if each of the machines M1, M2...Mn has, say, a shared memory capability of 10MB, then the total shared memory available to each application 50 is not, as one might expect, 10n MB. However, how this results in improved operation will become apparent hereafter. Naturally, each machine M1, M2...Mn has an unshared memory capability. The unshared memory capability of the machines M1, M2...Mn are normally approximately equal but need not be.

It is known in the prior art to provide a single computer or machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) utilizing the particular language of the application by creating a virtual machine as illustrated in Fig 6.

The code and data and virtual machine configuration or arrangement of Fig 6 takes the form of the application code 50 written in the JAVA language and executing within the JAVA virtual machine 61. Thus where the intended language of the application is the language JAVA, a JAVA virtual machine is used which is able to operate code in JAVA irrespective of the machine manufacturer and internal details of the computer or machine.

For further details, see "The JAVA Virtual Machine Specification" 2nd Edition by T. Lindholm and F. Yellin of Sun Microsystems Inc of the USA which is incorporated by reference herein.

This conventional art arrangement of Fig 6 is modified in accordance with embodiments of the present invention by the provision of an additional facility which is conveniently termed a "distributed run time" or a "distributed run time system" DRT 71 and as seen in Fig 7.

In Figs. 7 and 8, the application code 50 is loaded onto the Java Virtual Machine(s) M1, M2,...Mn in cooperation with the distributed runtime system 71, through the loading procedure indicated by arrow 75 or 75A or 75B. As used herein the terms "distributed runtime" and the "distributed run time system" are essentially synonymous, and by means of illustration but not limitation are generally understood to include library code and processes which support software written in a particular language running on a particular platform. Additionally, a distributed runtime system may also include library code and processes which support software written in a particular language running within a particular distributed computing environment. The runtime system typically deals with the details of the interface between the program and the operating system such as system calls, program start-up and termination, and memory management. For purposes of background, a conventional Distributed Computing Environment (DCE) (that does not provide the capabilities of the inventive distributed run time or distributed run time system 71 used in the preferred embodiments of the present invention) is available from the Open Software Foundation. This Distributed Computing Environment (DCE)

performs a form of computer-to-computer communication for software running on the machines, but among its many limitations, it is not able to implement the desired modification or communication operations. Among its functions and operations the preferred DRT 71 coordinates the particular communications between the plurality of machines M1, M2,...Mn. Moreover, the preferred distributed runtime 71 comes into operation during the loading procedure indicated by arrow 75A or 75B of the JAVA application 50 on each JAVA virtual machine 72 or machines JVM#1, JVKMJ#2,...JVM#n of Fig. 8. It will be appreciated in light of the description provided herein that although many examples and descriptions are provided relative to the JAVA language and JAVA virtual machines so that the reader may get the benefit of specific examples, the invention is not restricted to either the JAVA language or JAVA virtual machines, or to any other language, virtual machine, machine or operating environment.

Fig 8 shows in modified form the arrangement of the JAVA virtual machines, each as illustrated in Fig 7. It will be apparent that again the same application code 50 is loaded onto each machine M1, M2...Mn. However, the communications between each machine M1, M2...Mn are as indicated by arrows 83, and although physically routed through the machine hardware, are advantageously controlled by the individual DRT's 71/1...71/n within each machine. Thus, in practice this may be conceptionalised as the DRT's 71/1, ...71/n communicating with each other via the network or other communications link 53 rather than the machines M1, M2...Mn communicating directly themselves or with each other. Contemplated and included are either this direct communication between machines M1, M2...Mn or DRT's 71/1, 71/2...71/n or a combination of such communications. The preferred DRT 71 provides communication that is transport, protocol, and link independent.

The one common application program or application code 50 and its executable version (with likely modification) is simultaneously or concurrently executing across the plurality of computers or machines M1, M2...Mn. The common application program 5. The application program 5 is written with the intention that it only operate on a single machine or computer. Essentially the modified structure is to replicate and identical memory structure and contents on each of the individual machines

The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on each one of the plurality of computers or machines M1, M2...Mn, or optionally on each one of some subset of the plurality of computers or machines M1, M2...Mn. Put somewhat differently, there is a common application program represented in application code 50. This is either a single copy or a plurality of identical copies each individually modified to generate a modified copy or version of the application program or program code. Each copy or instance is then prepared for execution on the corresponding machine. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing embodiments of the invention may optionally be connected to or coupled with other computers, machines, information appliances, or the like that do not implement embodiments of the invention.

The same application program 50 (such as for example a parallel merge sort, or a computational fluid dynamics application or a data mining application)is run on each machine, but the executable code of that application program is modified on each machine as necessary such that each executing instance (copy or replica) on each machine coordinates its local operations on that particular machine with the operations of the respective instances (or copies or replicas) on the other machines such that they function together in a consistent, coherent and coordinated manner and give the appearance of being one global instance of the application (i.e. a "meta-application").

The copies or replicas of the same or substantially the same application codes, are each loaded onto a corresponding one of the interoperating and connected machines or computers. As the characteristics of each machine or computer may differ, the application code 50 may be modified before loading, during the loading process, and with some disadvantages after the loading process, to provide a customization or modification of the code on each machine. Some dissimilarity between the programs may be permitted so long as the other requirements for interoperability, consistency, and coherency as described herein can be maintained. As it will become apparent hereafter, each of the machines M1, M2...Mn and thus all of the machines M1, M2...Mn have the same or substantially the same application code 50, usually with a modification that may be machine specific.

Before the loading of, during the loading of, or at any time preceding the execution of, the application code 50 (or the relevant portion thereof) on each machine M1, M2...Mn, each application code 50 is modified by a corresponding modifier 51 according to the same rules (or substantially the same rules since minor optimizing changes are permitted within each modifier 51/1, 51/2...51/n).

Each of the machines M1, M2...Mn operates with the same (or substantially the same or similar) modifier 51 (in some embodiments implemented as a distributed run time or DRT71 and in other embodiments implemented as an adjunct to the code and data 50, and also able to be implemented either to the JAVA virtual machine itself). Thus all of the machines M1, M2...Mn have the same (or substantially the same or similar) modifier 51 for each modification required. A different modification, for example, may be required for memory management and replication, for initialization, for finalization, and/or for synchronization (though not all of these modification types may be required for all embodiments).

There are alternative implementations of the modifier 51 and the distributed run time 71. For example as indicated by broken lines in Fig. 8, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71 such as within the code and data 50, or within the JAVA virtual machine itself. In one embodiment, both the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. In this case the modifier function and structure is, in practice, subsumed into the DRT. Independent of how it is implemented, the modifier function and structure is responsible

for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the DRT on each machine. The DRT can, for example, implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures, is not crucial.

However, in the arrangement illustrated in Fig 8, a plurality of individual computers or machines M1, M2...Mn are provided, each of which are interconnected via a communications network 53 or other communications link. Each individual computer or machine is provided with a corresponding modifier 51. Each individual computer is also provided with a communications port which connects to the communications network. The communications network 53 or path can be any electronic signalling, data, or digital communications network or path and is preferably slow speed, and thus low cost, communications path, such as a network connection over the Internet or any common networking configurations including communication ports known or available as of the date of this application such as ETHERNET or INFINIBAND and extensions and improvements, thereto.

As a consequence of the above described arrangement, if each of the machines M1, M2, ..., Mn has say an internal or local memory capability of 10MB, then the total memory available to the application code 50 in its entirety is not, as one might expect, the number of machines (n) times 10MB. Nor is it the additive combination of the internal memory capability of all n machines. Instead it is either 10MB, or some number greater than 10MB but less than n x 10MB. In the situation where the internal memory capacities of the machines are different, which is permissible, then in the case where the internal memory in one machine is smaller than the internal memory capability of at least one other of the machines, then the size of the smallest memory of any of the machines may be used as the maximum memory capacity of the machines when such memory (or a portion thereof) is to be treated as 'common' memory (i.e. similar equivalent memory on each of the machines M1...Mn) or otherwise used to execute the common application code.

However, even though the manner that the internal memory of each machine is treated may initially appear to be a possible constraint on performance, how this results in improved operation and performance will become apparent hereafter. Naturally, each machine M1, M2...Mn has a private (i.e. 'non-common') internal memory capability. The private internal memory capability of the machines M1, M2, ..., Mn are normally approximately equal but need not be. It may also be advantageous to select the amounts of internal memory in each machine to achieve a desired performance level in each machine and across a constellation or network of connected or coupled plurality of machines, computers, or information appliances M1, M2, ..., Mn. Having described these internal and common memory considerations, it will be apparent in light of the

description provided herein that the amount of memory that can be common between machines is not a limitation.

In some embodiments, some or all of the plurality of individual computers or machines can be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated that the platform and/or runtime system can include virtual machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the Power PC computer architecture manufactured by International Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and Boolean data types), structured data types (such a arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, reference and unions. These structures and procedures when applied in combination when required, maintain a computing environment where memory locations, address ranges, objects, classes, assets, resources, or any other procedural or structural aspect of a computer or computing environment are where required created, maintained, operated, and deactivated or deleted in a coordinated, coherent, and consistent manner across the plurality of individual machines M1, M2...Mn.

This analysis or scrutiny of the application code 50 can take place either prior to loading the application program code 50, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application code can be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as for example from source-code language or intermediate-code language to object-code language or machine-code language). In this connection it is understood that the term compilation normally or conventionally involves a change in code or language, for example, from source code to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object code), and compilation from source-code to source-code, as well as compilation from object-code to object code, and any altered combinations therein. It is also inclusive of so-called "intermediary-code languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 takes place during the loading of the application program code such as by the operating system reading the application code 50 from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader.loadClass method (e.g. "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the relevant corresponding portion of the application program code has started, such as for example after the JAVA virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

Persons skilled in the computing arts will be aware of various possible techniques that may be used in the modification of computer code, including but not limited to instrumentation, program transformation, translation, or compilation means.

One such technique is to make the modification(s) to the application code, without a preceding or consequential change of the language of the application code. Another such technique is to convert the original code (for example, JAVA language source-code) into an intermediate representation (or intermediate-code language, or pseudo code), such as JAVA byte code. Once this conversion takes place the modification is made to the byte code and then the conversion may be reversed. This gives the desired result of modified JAVA code.

A further possible technique is to convert the application program to machine code, either directly from source-code or via the abovementioned intermediate language or through some other intermediate means. Then the machine code is modified before being loaded and executed. A still further such technique is to convert the original code to an intermediate representation, which is thus modified and subsequently converted into machine code.

The present invention encompasses all such modification routes and also a combination of two, three or even more, of such routes.

The DRT or other code modifying means is responsible for creating or replication a memory structure and contents on each of the individual machines M1, M2...Mn that permits the plurality of machines to interoperate. In some embodiments this replicated memory structure will be identical. Whilst in other embodiments this memory structure will have portions that are identical and other portions that are not. In still other embodiments the memory structures are different only in format or storage conventions such as Big Endian or Little Endian formats or conventions.

These structures and procedures when applied in combination when required, maintain a computing environment where the memory locations, address ranges, objects, classes, assets, resources, or any other procedural or structural aspect of a computer or computing environment are where required created, maintained, operated, and deactivated or deleted in a coordinated, coherent, and consistent manner across the plurality of individual machines M1, M2...Mn.

Therefore the terminology "one", "single", and "common" application code or program includes the situation where all machines M1, M2...Mn are operating or executing the same program or code and not different (and unrelated) programs, in other words copies or replicas of same or substantially the same application code are loaded onto each of the interoperating and connected machines or computers.

In conventional arrangements utilising distributed software, memory access from one machine's software to memory physically located on another machine takes place via the network interconnecting the machines. However, because the read and/or write memory access to memory physically located on another computer require the use of the slow network interconnecting the computers, in these configurations such memory accesses can result in substantial delays in memory read/write processing operations, potentially of the order of $10^6 - 10^7$ cycles of the central processing unit of the machine. Ultimately this delay is dependent upon numerous factors, such as for example, the speed, bandwidth, and/or latency of the communication network. This in large part accounts for the diminished performance of the multiple interconnected machines in the prior art arrangement.

However, in the present arrangement all reading of memory locations or data is satisfied locally because a current value of all (or some subset of all) memory locations is stored on the machine carrying out the processing which generates the demand to read memory.

Similarly, all writing of memory locations or data is satisfied locally because a current value of all (or some subset of all) memory locations is stored on the machine carrying out the processing which generates the demand to write to memory.

Such local memory read and write processing operation can typically be satisfied with 10^2 – 10^3 cycles of the central processing unit. Thus, in practice there is substantially less waiting for memory accesses which involves and/or writes.

The invention is transport, network, and communications path independent, and does not depend on how the communication between machines or DRTs takes place. In one embodiment, even electronic mail (email) exchanges between machines or DRTs may suffice for the communications.

Turning now to Fig. 9, during the loading procedure 75, the program 50 being loaded to create each JAVA virtual machine M1, M2,...Mn is modified. This modification commences at 90 in Fig. 9 and involves the initial step 91 of detecting all memory locations (termed fields in JAVA - but equivalent terms are used in other languages) in the application 50 being loaded. Such memory locations need to be identified for subsequent processing at steps 92 and 93. The DRT 71/1,...DRT71/n during the loading procedure 75 creates a list of all the memory locations thus identified, the JAVA fields being listed by object and class. Both volatile and synchronous fields are listed.

The next phase (designated 92 in Fig. 9) of the modification procedure is to search through the executable application code in order to locate every processing activity that manipulates or changes field values corresponding to the list generated at step 91 and thus writes to fields so the value at the corresponding memory location is changed. When such an operation (typically putstatic or putfield in the JAVA language) is detected which changes the field value, then an "updating propagation routine" is inserted by step 93 at this place in the program to ensure that all other machines are notified that the value of the field has changed. Thereafter, the loading procedure continues in a normal way as indicated by step 94 in Fig. 9.

An alternative form of initial modification during loading is illustrated in Fig. 10. Here the start and listing steps 90 and 91 and the searching step 92 are the same as in Fig. 9. However, rather than insert the "updating propagation routine" as in step 93 in which the processing thread carries out the updating, instead an "alert routine" is inserted at step 103. The "alert routine" instructs a thread or threads not used in processing and allocated to the DRT, to carry out the necessary propagation. This step 103 is a quicker alternative which results in lower overhead.

Once this initial modification during the loading procedure has taken place, then either one of the multiple thread processing operations illustrated in Figs. 11 and 12 takes place. As seen in Fig. 11, multiple thread processing 110 on the machines consisting of threads 111/1...111/4 is occurring and the processing of the second thread 111/2 (in this example) results in that thread 111/2 becoming aware at step 113 of a change of field value. At this stage the normal processing of that thread 111/2 is halted at step 114, and

the same thread 111/2 notifies all other machines M2...Mn via the network 53 of the identity of the changed field and the changed value which occurred at step 113. At the end of that communication procedure, the thread 111/2 then resumes the processing at step 115 until the next instance where there is a change of field value.

In the alternative arrangement illustrated in Fig. 12, once a thread 121/2 has become aware of a change of field value at step 113, it instructs DRT processing 120 (as indicated by step 125 and arrow 127) that another thread(s) 121/1 allocated to the DRT processing 120 is to propagate in accordance with step 128 via the network 53 to all other machines M2...Mn the identity of the changed field and the changed value detected at step 113. This is an operation which can be carried out quickly and thus the processing of the initial thread 111/2 is only interrupted momentarily as indicated in step 125 before the thread 111/2 resumes processing in step 115. The other thread 121/1 which has been notified of the change (as indicated by arrow 127) then communicates that change as indicated in step 128 via the network 53 to each of the other machines M2...Mn.

This second arrangement of Fig. 12 makes better utilisation of the processing power of the various threads 111/1...111/3 and 121/1 (which are not, in general, subject to equal demands) and gives better scaling with increasing size of "n", (n being an integer greater than or equal to 2 which represents the total number of machines which are connected to the network 53 and which run the application program 50 simultaneously). Irrespective of which arrangement is used, the changed field and identities and values detected at step 113 are propagated to all the other machines M2...Mn on the network.

This is illustrated in Fig. 13 where the DRT 71/1 and its thread 121/1 of Fig. 12 (represented by step 128 in Fig. 13) sends via the network 53 the identity and changed value of the listed memory location generated at step 113 of Fig. 12 by processing in machine M1, to each of the other machines M2...Mn.

Each of the other machines M2...Mn carries out the action indicated by steps 135 and 136 in Fig. 13 for machine Mn by receiving the identity and value pair from the network 53 and writing the new value into the local corresponding memory location.

In the prior art arrangement in Fig. 3 utilising distributed software, memory accesses from one machine's software to memory physically located on another machine are permitted by the network interconnecting the machines. However, such memory accesses can result in delays in processing of the order of $10^6 - 10^7$ cycles of the central processing unit of the machine. This in large part accounts for the diminished performance of the multiple interconnected machines.

However, in the present arrangement as described above in connection with Fig. 8, it will be appreciated that all reading of data is satisfied locally because the current value of all fields is stored on the machine carrying out the processing which generates the demand to read memory. Such local processing can be satisfied within $10^2 - 10^3$ cycles of the central processing unit. Thus, in practice, there is substantially no waiting for memory accesses which involves reads.

However, most application software reads memory frequently but writes to memory relatively infrequently. As a consequence, the rate at which memory is being written or re-written is relatively slow compared to the rate at which memory is being read. Because of this slow demand for writing or re-writing of memory, the fields can be continually updated at a relatively low speed via the inexpensive commodity network 53, yet this low speed is sufficient to meet the application program's demand for writing to memory. The result is that the performance of the Fig. 8 arrangement is vastly superior to that of Fig. 3.

In a further modification in relation to the above, the identities and values of changed fields can be grouped into batches so as to further reduce the demands on the communication speed of the network 53 interconnecting the various machines.

It will also be apparent to those skilled in the art that in a table created by each DRT 71 when initially recording the fields, for each field there is a name or identity which is common throughout the network and which the network recognises. However, in the individual machines the memory location corresponding to a given named field will vary over time since each machine will progressively store changed field values at different locations according to its own internal processes. Thus the table in each of the DRTs will have, in general, different memory locations but each global "field name" will have the same "field value" stored in the different memory locations.

It will also be apparent to those skilled in the art that the abovementioned modification of the application program during loading can be accomplished in up to five ways by:

(i) re-compilation at loading,
(ii) by a pre-compilation procedure prior to loading,
(iii) compilation prior to loading,
(iv) a "just-in-time" compilation, or
(v) re-compilation after loading (but, or for example, before execution of the relevant or corresponding application code in a distributed environment).

Traditionally the term "compilation" implies a change in code or language, eg from source to object code or one language to another. Clearly the use of the term "compilation" (and its grammatical equivalents) in the present specification is not so restricted and can also include or embrace modifications within the same code or language.

In the first embodiment, a particular machine, say machine M2, loads the application code on itself, modifies it, and then loads each of the other machines M1, M3 ... Mn (either sequentially or simultaneously) with the modified code. In this arrangement, which may be termed "master/slave", each of machines M1, M3, ... Mn loads what it is given by machine M2.

In a still further embodiment, each machine receives the application code, but modifies it and loads the modified code on that machine. This enables the modification carried out by each machine to be slightly different being optimized based upon its architecture and operating system, yet still coherent with all other similar modifications.

In a further arrangement, a particular machine, say M1, loads the unmodified code and all other machines M2, M3 ... Mn do a modification to delete the original application code and load the modified version.

In all instances, the supply can be branched (ie M2 supplies each of M1, M3, M4, etc directly) or cascaded or sequential (ie M2 applies M1 which then supplies M3 which then supplies M4, and so on).

In a still further arrangement, the machines M1 to Mn, can send all load requests to an additional machine (not illustrated) which is not running the application program, which performs the modification via any of the aforementioned methods, and returns the modified routine to each of the machines M1 to Mn which then load the modified routine locally. In this arrangement, machines M1 to Mn forward all load requests to this additional machine which returns a modified routine to each machine. The modifications performed by this additional machine can include any of the modifications covered under the scope of the present invention.

Persons skilled in the computing arts will be aware of at least four techniques used in creating modifications in computer code. The first is to make the modification in the original (source) language. The second is to convert the original code (in say JAVA) into an intermediate representation (or intermediate language). Once this conversion takes place the modification is made and then the conversion is reversed. This gives the desired result of modified JAVA code.

The third possibility is to convert to machine code (either directly or via the abovementioned intermediate language). Then the machine code is modified before being loaded and executed. The fourth possibility is to convert the original code to an intermediate representation, which is then modified and subsequently converted into machine code.

The present invention encompasses all four modification routes and also a combination of two, three or even all four, of such routes.

MEMORY MANAGEMENT AND REPLICATION

In connection with FIG. 5, in accordance with a preferred embodiment of the present invention a single application code 50 (sometimes more informally referred to as the application or the application program) can be operated simultaneously on a number of machines M1, M2...Mn interconnected via a communications network or other communications link or path 53. By way of example but not limitation, one application code or program 50 would be a single common application program on the machines, such as Microsoft Word, as opposed to different applications on each machine, such as Microsoft Word on machine M1, and Microsoft PowerPoint on machine M2, and Netscape Navigator on machine M3 and so forth. Therefore the terminology "one", "single", and "common" application code or program is used to try and capture this situation where all machines M1, ..., Mn are operating or executing the same program or code and not different (and unrelated) programs. In other words copies or replicas of same or substantially the same application code is loaded onto each of the interoperating

and connected machines or computers. As the characteristics of each machine or computer may differ, the application code 50 may be modified before loading, during the loading process, or after the loading process to provide a customization or modification of the code on each machine. Some dissimilarity between the programs may be permitted so long as the other requirements for interoperability, consistency, and coherency as described herein can be maintain. As it will become apparent hereafter, each of the machines M1, M2...Mn operates with the same application code 50 on each machine M1, M2...Mn and thus all of the machines M1, M2, ..., Mn have the same or substantially the same application code 50 usually with a modification that may be machine specific.

Similarly, each of the machines M1, M2, ..., Mn operates with the same (or substantially the same or similar) modifier 51 on each machine M1, M2, ...,Mn and thus all of the machines M1, M2...Mn have the same (or substantially the same or similar) modifier 51 with the modifier of machine M1 being designated 51/1 and the modifier of machine M2 being designated 51/2, etc. In addition, before or during the loading of, or preceding the execution of, or even after execution has commenced, the application code 50 on each machine M1, M2...Mn is modified by the corresponding modifier 51 according to the same rules (or substantially the same rules since minor optimizing changes are permitted within each modifier 51/1, 51/2, ..., 51/n).

As will become more apparent in light of the further description provided herein, one of the features of the invention is to make it appear that one application program instance of application code 50 is executing simultaneously across all of the plurality of machines M1, M2, ..., Mn. As will be described in considerable detail hereinafter, the instant invention achieves this by running the same application program code (for example, Microsoft Word or Adobe Photoshop CS2) on each machine, but modifying the executable code of that application program on each machine such that each executing occurrence (or 'local instance') on each one of the machines M1...Mn coordinates its local operations with the operations of the respective occurrences on each one of the other machines such that each occurrence on each one of the plurality of machines function together in a consistent, coherent and coordinated manner so as to give the appearance of being one global instance (or occurrence) of the application program and program code(i.e., a "meta-application").

As a consequence of the above described arrangement, if each of the machines M1, M2, ..., Mn has, say, an internal memory capability of 10MB, then the total memory available to each application code 50 is not necessarily, as one might expect the number of machines (n) times 10MB, or alternatively the additive combination of the internal memory capability of all n machines, but rather or still may only be 10MB. In the situation where the internal memory capacities of the machines are different, which is permissible, then in the case where the internal memory in one machine is smaller than the internal memory capability of at least one other of the machines, then the size of the smallest memory of any of the machines may be used as the maximum memory capacity of the machines when such memory (or a portion thereof) is to be treated as a 'common' memory (i.e. similar equivalent memory on each of the machines M1...Mn) or otherwise used to execute the common application code.

However, even though the manner that the internal memory of each machine is treated may initially appear to be a possible constraint on performance, how this results in

improved operation and performance will become apparent hereafter. Naturally, each machine M1, M2...Mn has a private (i.e. 'non-common') internal memory capability. The private internal memory capability of the machines M1, M2, ..., Mn are normally approximately equal but need not be. It may also be advantageous to select the amounts of internal memory in each machine to achieve a desired performance level in each machine and across a constellation or network of connected or coupled plurality of machines, computers, or information appliances M1, M2, ..., Mn. Having described these internal and common memory considerations, it will be apparent in light of the description provided herein that the amount of memory that can be common between machines is not a limitation of the invention.

It is known from the prior art to operate a single computer or machine (produced by one of various manufacturers and having an operating system operating in one of various different languages) in a particular language of the application, by creating a virtual machine as schematically illustrated in FIG. 6. The code and data and virtual machine configuration or arrangement of FIG. 6 takes the form of the application code 50 written in the Java language and executing within a Java Virtual Machine 61. Thus, where the intended language of the application is the language JAVA, a JAVA virtual machine is used which is able to operate code in JAVA irrespective of the machine manufacturer and internal details of the machine. For further details see "The JAVA Virtual Machine Specification" 2nd Edition by T. Lindholm & F. Yellin of Sun Microsystems Inc. of the USA, which is incorporated by reference herein.

This conventional art arrangement of FIG. 6 is modified in accordance with embodiments of the present invention by the provision of an additional facility which is conveniently termed "distributed run time" or "distributed run time system" DRT 71 and as seen in FIG. 7.

In FIG. 7, the application code 50 is loaded onto the Java Virtual Machine 72 in cooperation with the distributed runtime system 71, through the loading procedure indicated by arrow 75. As used herein the terms distributed runtime and the distributed run time system are essentially synonymous, and by means of illustration but not limitation are generally understood to include library code and processes which support software written in a particular language running on a particular platform. Additionally, a distributed runtime system may also include library code and processes which support software written in a particular language running within a particular distributed computing environment. The runtime system typically deals with the details of the interface between the program and the operation system such as system calls, program start-up and termination, and memory management. For purposes of background, a conventional Distributed Computing Environment (DCE) that does not provide the capabilities of the inventive distributed run time or distributed run time system 71 required in the invention is available from the Open Software Foundation. This Distributed Computing Environment (DCE) performs a form of computer-to-computer communication for software running on the machines, but among its many limitations, it is not able to implement the modification or communication operations of this invention. Among its functions and operations, the inventive DRT 71 coordinates the particular communications between the plurality of machines M1, M2, ..., Mn. Moreover, the inventive distributed runtime 71 comes into operation during the loading procedure

indicated by arrow 75 of the JAVA application 50 on each JAVA virtual machine 72 of machines JVM#1, JVM#2,...JVM#n. The sequence of operations during loading will be described hereafter in relation to FIG. 9. It will be appreciated in light of the description provided herein that although many examples and descriptions are provided relative to the JAVA language and JAVA virtual machines so that the reader may get the benefit of specific examples, the invention is not restricted to either the JAVA language or JAVA virtual machines, or to any other language, virtual machine, machine, or operating environment.

FIG. 8 shows in modified form the arrangement of FIG. 5 utilising JAVA virtual machines, each as illustrated in FIG. 7. It will be apparent that again the same application code 50 is loaded onto each machine M1, M2...Mn. However, the communications between each machine M1, M2, ..., Mn, and indicated by arrows 83, although physically routed through the machine hardware, are advantageously controlled by the individual DRT's 71/1...71/n within each machine. Thus, in practice this may be conceptionalised as the DRT's 71/1, ..., 71/n communicating with each other via the network or other communications link 73 rather than the machines M1, M2, ..., Mn communicating directly with themselves or each other. Actually, the invention contemplates and included either this direct communication between machines M1, M2, ..., Mn or DRTs 71/1, 71/2, ..., 71/n or a combination of such communications. The inventive DRT 71 provides communication that is transport, protocol, and link independent.

It will be appreciated in light of the description provided herein that there are alternative implementations of the modifier 51 and the distributed run time 71. For example, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71. In one embodiment, the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. The modifier function and structure therefore maybe subsumed into the DRT and considered to be an optional component. Independent of how implemented, the modifier function and structure is responsible for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the DRT on each machine. The DRT may for example implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures within the invention are less important than that they are provided.

However, in the arrangement illustrated in FIG. 8, (and also in FIGS. 31-32), a plurality of individual computers or machines M1, M2, ..., Mn are provided, each of which are interconnected via a communications network 53 or other communications link and each of which individual computers or machines provided with a modifier 51 (See in

FIG. 5) and realised by or in for example the distributed run time (DRT) 71 (See FIG. 8) and loaded with a common application code 50. The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on each one of the plurality of computers or machines M1, M2...Mn, or optionally on each one of some subset of the plurality of computers or machines M1,M2...Mn. Put somewhat differently, there is a common application program represented in application code 50, and this single copy or perhaps a plurality of identical copies are modified to generate a modified copy or version of the application program or program code, each copy or instance prepared for execution on the plurality of machines. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing the features of the invention may optionally be connected to or coupled with other computers, machines, information appliances, or the like that do not implement the features of the invention.

Essentially in at least one embodiment the modifier 51 or DRT 71 or other code modifying means is responsible for modifying the application code 50 so that it may execute memory manipulation operations, such as memory putstatic and putfield instructions in the JAVA language and virtual machine environment, in a coordinated, consistent, and coherent manner across and between the plurality of individual machines M1...Mn. It follows therefore that in such a computing environment it is necessary to ensure that each of memory location is manipulated in a consistent fashion (with respect to the others).

In some embodiments, some or all of the plurality of individual computers or machines may be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

A machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) can operate in the particular language of the application program code 50, in this instance the JAVA language. That is, a JAVA virtual machine 72 is able to operate application code 50 in the JAVA language, and utilize the JAVA architecture irrespective of the machine manufacturer and the internal details of the machine.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform, and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated in light of the description provided herein that platform and/or runtime system may include virtual

machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method, and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the PowerPC computer architecture manufactured by International Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and Boolean data types), structured data types (such as arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, references and unions.

Turning now to FIGS. 7 and 9, during the loading procedure 75, the application code 50 being loaded onto or into each JAVA virtual machine 72 is modified by DRT 71. This modification commences at Step 90 in FIG. 9 and involves the initial step 91 of preferably scrutinizing or analysing the code and detecting all memory locations addressable by the application code 50, or optionally some subset of all memory locations addressable by the application code 50; such as for example named and unnamed memory locations, variables (such as local variables, global variables, and formal arguments to subroutines or functions), fields, registers, or any other address space or range of addresses which application code 50 may access. Such memory locations in some instances need to be identified for subsequent processing at steps 92 and 93. In some embodiments, where a list of detected memory locations is required for further processing, the DRT 71 during the loading procedure 75 creates a list of all the memory locations thus identified. In one embodiment, the memory locations in the form of JAVA fields are listed by object and class, however, the memory locations, fields, or the like may be listed or organized in any manner so long as they comport with the architectural and programming requirements of the system on which the program is to be used and the principles of the invention described herein. This detection is optional and not required in all embodiments of the invention. It may be noted that the DRT is at least in part fulfilling the roll of the modifier 51.

The next phase (designated Step 92 in FIG. 9) [Step 92] of the modification procedure is to search through the application code 50 in order to locate processing activity or activities that manipulate or change values or contents of any listed memory location (for example, but not limited to JAVA fields) corresponding to the list generated at step 91 when required. Preferably, all processing activities that manipulate or change

any one or more values or contents of any one or more listed memory locations, are located.

When such a processing activity or operation (typically "putstatic" or "putfield" in the JAVA language, or for example, a memory assignment operation, or a memory write operation, or a memory manipulation operation, or more generally operations that otherwise manipulate or change value(s) or content(s) of memory or other addressable areas), is detected which changes the value or content of a listed or detected memory location , then an "updating propagation routine" is inserted by step 93 in the application code 50 corresponding to the detected memory manipulation operation, to communicate with all other machines in order to notify all other machines of the identity of the manipulated memory location, and the updated, manipulated or changed value(s) or content(s) of the manipulated memory location. The inserted "updating propagation routine" preferably takes the form of a method, function, procedure, or similar subroutine call or operation to a network communications library of DRT 71. Alternatively, the "updating propagation routine" may take the optional form of a code-block (or other inline code form) inserted into the application code instruction stream at, after, before, or otherwise corresponding to the detected manipulation instruction or operation. And preferably, in a multi-tasking or parallel processing machine environment (and in some embodiments inclusive or exclusive of operating system), such as a machine environment capable of potentially simultaneous or concurrent execution of multiple or different threads or processes, the "updating propagation routine" may execute on the same thread or process or processor as the detected memory manipulation operation of step 92. Thereafter, the loading procedure continues, by loading the modified application code 50 on the machine 72 in place of the unmodified application code 50, as indicated by step 94 in FIG. 9.

An alternative form of modification during loading is illustrated in the illustration of FIG. 10. Here the start and listing steps 90 and 91 and the searching step 92 are the same as in FIG. 9. However, rather than insert the "updating propagation routine" into the application code 50 corresponding to the detected memory manipulation operation identified in step 92, as is indicated in step 93, in which the application code 50, or network communications library code 71 of the DRT executing on the same thread or process or processor as the detected memory manipulation operation, carries out the updating, instead an "alert routine" is inserted corresponding to the detected memory manipulation operation, at step 103. The "alert routine" instructs, notifies or otherwise requests a different and potentially simultaneously or concurrently executing thread or process or processor not used to perform the memory manipulation operation (that is, a different thread or process or processor than the thread or process or processor which manipulated the memory location), such as a different thread or process allocated to the DRT 71, to carry out the notification , propagation, or communication of all other machines of the identity of the manipulated memory location, and the updated, manipulated or changed value(s) or content(s) of the manipulated memory location.

Once this modification during the loading procedure has taken place and execution begins of the modified application code 50, then either the steps of FIG. 11 or FIG. 12 take place. FIG. 11 (and the steps 112, 113, 114, and 115 therein) correspond to the execution and operation of the modified application code 50 when modified in

accordance with the procedures set forth in and described relative to FIG. 9. FIG. 12 on the other hand (and the steps 112, 113, 125, 127, and 115 therein) set forth therein correspond to the execution and operation of the modified application code 50 when modified in accordance with FIG. 10.

This analysis or scrutiny of the application code 50 can take place either prior to loading the application program code 50, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application code may be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as for example from source-code language or intermediate-code language to object-code language or machine-code language), and with the understanding that the term compilation normally or conventionally involves a change in code or language, for example, from source code to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object-code), and compilation from source-code to source-code, as well as compilation from object-code to object-code, and any altered combinations therein. It is also inclusive of so-called "intermediary-code languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the application program code such as by the operating system reading the application code from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader loadClass method (e.g., "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the relevant corresponding portion of the application program code has started, such as for example after the JAVA virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

As seen in FIG. 11, a multiple thread processing machine environment 110, on each one of the machines M1, ..., Mn and consisting of threads 111/1...111/4 exists. The processing and execution of the second thread 111/2 (in this example) results in that thread 111/2 manipulating a memory location at step 113, by writing to a listed memory location. In accordance with the modifications made to the application code 50 in the steps 90-94 of FIG. 9, the application code 50 is modified at a point corresponding to the write to the memory location of step 113, so that it propagates, notifies, or communicates

the identity and changed value of the manipulated memory location of step 113 to the other machines M2, ..., Mn via network 53 or other communication link or path, as indicated at step 114. At this stage the processing of the application code 50 of that thread 111/2 is or may be altered and in some instances interrupted at step 114 by the executing of the inserted "updating propagation routine", and the same thread 111/2 notifies, or propagates, or communicates to all other machines M2, ..., Mn via the network 53 or other communications link or path of the identity and changed value of the manipulated memory location of step 113. At the end of that notification, or propagation, or communication procedure 114, the thread 111/2 then resumes or continues the processing or the execution of the modified application code 50 at step 115.

In the alternative arrangement illustrated in FIG. 12, a multiple thread processing machine environment 110 comprising or consisting of threads 111/1, ..., 111/3, and a simultaneously or concurrently executing DRT processing environment 120 consisting of the thread 121/1 as illustrated, or optionally a plurality of threads, is executing on each one of the machines M1,...Mn. The processing and execution of the modified application code 50 on thread 111/2 results in a memory manipulation operation of step 113, which in this instance is a write to a listed memory location. In accordance with the modifications made to the application code 50 in the steps 90, 91, 92, 103, and 94 of FIG. 9, the application code 50 is modified at a point corresponding to the write to the memory location of step 113, so that it requests or otherwise notifies the threads of the DRT processing environment 120 to notify, or propagate, or communicate to the other machines M2, ..., Mn of the identity and changed value of the manipulated memory location of step 113, as indicated at steps 125 and 128 and arrow 127. In accordance with this modification, the thread 111/2 processing and executing the modified application code 50 requests a different and potentially simultaneously or concurrently executing thread or process (such as thread 121/1) of the DRT processing environment 120 to notify the machines M2, ..., Mn via network 53 or other communications link or path of the identity and changed value of the manipulated memory location of step 113, as indicated in step 125 and arrow 127. In response to this request of step 125 and arrow 127, a different and potentially simultaneously or concurrently executing thread or process 121/1 of the DRT processing environment 120 notifies the machines M2, ..., Mn via network 53 or other communications link or path of the identity and changed value of the manipulated memory location of step 113, as requested of it by the modified application code 50 executing on thread 111/2 of step 125 and arrow 127.

When compared to the earlier described step 114 of thread 111/2 of FIG. 11, step 125 of thread 111/2 of FIG. 12 can be carried out quickly, because step 114 of thread 111/2 must notify and communicate with machines M2, ..., Mn via the relatively slow network 53 (relatively slow for example when compared to the internal memory bus 4 of FIG. 1 or the global memory 13 of FIG. 2) of the identity and changed value of the manipulated memory location of step 113, whereas step 125 of thread 111/2 does not communicate with machines M2, ..., Mn via the relatively slow network 53. Instead, step 125 of thread 111/2 requests or otherwise notifies a different and potentially simultaneously or concurrently executing thread 121/1 of the DRT processing environment 120 to perform the notification and communication with machines M2, ..., Mn via the relatively slow network 53 of the identify and changed value of the manipulated memory location of step 113, as indicated by arrow 127. Thus thread 111/2

carrying out step 125 is only interrupted momentarily before the thread 111/2 resumes or continues processing or execution of modified application code in step 115. The other thread 121/1 of the DRT processing environment 120 then communicates the identity and changed value of the manipulated memory location of step 113 to machines M2, ..., Mn via the relatively slow network 53 or other relatively slow communications link or path.

This second arrangement of FIG. 12 makes better utilisation of the processing power of the various threads 111/1...111/3 and 121/1 (which are not, in general, subject to equal demands). Irrespective of which arrangement is used, the identity and change value of the manipulated memory location(s) of step 113 is (are) propagated to all the other machines M2...Mn on the network 53 or other communications link or path.

This is illustrated in FIG. 13 where step 114 of Fig 11, or the DRT 71/1 (corresponding to the DRT processing environment 120 of Fig 12) and its thread 121/1 of FIG. 12 (represented by step 128 in FIG. 13), send, via the network 53 or other communications link or path, the identity and changed value of the manipulated memory location of step 113 of FIGS. 11 and 12, to each of the other machines M2, ..., Mn.

With reference to FIG. 13, each of the other machines M2, ..., Mn carries out the action of receiving from the network 53 the identity and changed value of, for example, the manipulated memory location of step 113 from machine M1, indicated by step 135, and writes the value received at step 135 to the local memory location corresponding to the identified memory location received at step 135, indicated by step 136.

In the conventional arrangement in FIG. 3 utilising distributed software, memory access from one machine's software to memory physically located on another machine is permitted by the network interconnecting the machines. However, because the read and/or write memory access to memory physically located on another computer require the use of the slow network 14, in these configurations such memory accesses can result in substantial delays in memory read/write processing operation, potentially of the order of $10^6 - 10^7$ cycles of the central processing unit of the machine, but ultimately being dependant upon numerous factors, such as for example, the speed, bandwidth, and/or latency of the network 14. This in large part accounts for the diminished performance of the multiple interconnected machines in the prior art arrangement of FIG. 3.

However, in the present arrangement as described above in connection with FIG. 8, it will be appreciated that all reading of memory locations or data is satisfied locally because a current value of all (or some subset of all) memory locations is stored on the machine carrying out the processing which generates the demand to read memory.

Similarly, in the present arrangement as described above in connection with FIG. 8, it will be appreciated that all writing of memory locations or data may be satisfied locally because a current value of all (or some subset of all) memory locations is stored on the machine carrying out the processing which generates the demand to write to memory.

Such local memory read and write processing operation as performed according to the invention can typically be satisfied within $10^2 - 10^3$ cycles of the central processing

unit. Thus, in practice, there is substantially less waiting for memory accesses which involves reads than the arrangement shown and described relative to FIG. 3. Additionally, in practice, there may be less waiting for memory accesses which involve writes than the arrangement shown and described relative to FIG. 3.

It may be appreciated that most application software reads memory frequently but writes to memory relatively infrequently. As a consequence, the rate at which memory is being written or re-written is relatively slow compared to the rate at which memory is being read. Because of this slow demand for writing or re-writing of memory, the memory locations or fields can be continually updated at a relatively low speed via the possibly relatively slow and inexpensive commodity network 53, yet this possibly relatively slow speed is sufficient to meet the application program's demand for writing to memory. The result is that the performance of the FIG. 8 arrangement is superior to that of FIG. 3. It may be appreciated in light of the description provided herein that while a relatively slow network communication link or path 53 may advantageously be used because it provides the desired performance and low cost, the invention is not limited to a relatively low speed network connection and may be used with any communication link or path. The invention is transport, network, and communications path independent, and does not depend on how the communication between machines or DRTs takes place. In one embodiment, even electronic mail (email) exchanges between machines or DRTs may suffice for the communications.

In a further optional modification in relation to the above, the identity and changed value pair of a manipulated memory location sent over network 53, each pair typically sent as the sole contents of a single packet, frame or cell for example, can be grouped into batches of multiple pairs of identities and changed values corresponding to multiple manipulated memory locations, and sent together over network 53 or other communications link or path in a single packet, frame, or cell. This further modification further reduces the demands on the communication speed of the network 53 or other communications link or path interconnecting the various machines, as each packet, cell or frame may contain multiple identity and changed value pairs, and therefore fewer packets, frames, or cells require to be sent.

It may be apparent that in an environment where the application program code writes repeatedly to a single memory location, the embodiment illustrated of FIG. 11 of step 114 sends an updating and propagation message to all machines corresponding to every performed memory manipulation operation. In a still further optimal modification in relation to the above, the DRT thread 121/1 of FIG. 12 does not need to perform an updating and propagation operation corresponding to every local memory manipulation operation, but instead may send fewer updating and propagation messages than memory manipulation operations, each message containing the last or latest changed value or content of the manipulated memory location, or optionally may only send a single updating and propagation message corresponding to the last memory manipulation operation. This further improvement reduces the demands on the network 53 or other communications link or path, as fewer packets, frames, or cells require to be sent.

It will also be apparent to those skilled in the art in light of the detailed description provided herein that in a table or list or other data structure created by each DRT 71 when initially recording or creating the list of all, or some subset of all, memory

locations (or fields), for each such recorded memory location on each machine M1, ..., Mn there is a name or identity which is common or similar on each of the machines M1, ..., Mn. However, in the individual machines the local memory location corresponding to a given name or identity (listed for example, during step 91 of FIG. 9) will or may vary over time since each machine may and generally will store changed memory values or contents at different memory locations according to its own internal processes. Thus the table, or list, or other data structure in each of the DRTs will have, in general, different local memory locations corresponding to a single memory name or identity, but each global "memory name" or identity will have the same "memory value" stored in the different local memory locations.

It will also be apparent to those skilled in the art in light of the description provided herein that the abovementioned modification of the application program code 50 during loading can be accomplished in many ways or by a variety of means. These ways or means include, but are not limited to at least the following five ways and variations or combinations of these five, including by:

(i) re-compilation at loading,
(ii) by a pre-compilation procedure prior to loading,
(iii) compilation prior to loading,
(iv) a "just-in-time" compilation, or
(v) re-compilation after loading (but, or for example, before execution of the relevant or corresponding application code in a distributed environment).

Traditionally the term "compilation" implies a change in code or language, for example, from source to object code or one language to another. Clearly the use of the term "compilation" (and its grammatical equivalents) in the present specification is not so restricted and can also include or embrace modifications within the same code or language.

Given the fundamental concept of modifying memory manipulation operations to coordinate operation between and amongst a plurality of machines M1....Mn, there are several different ways or embodiments in which this coordinated, coherent and consistent memory state and manipulation operation concept, method, and procedure may be carried out or implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (such as class or object) inclusive of memory manipulation operation(s), modifies it, and then loads each of the other machines M1, M3, ..., Mn (either sequentially or simultaneously or according to any other order, routine or procedure) with the modified object (or class or other asset or resource) inclusive of the new modified memory manipulation operation. Note that there may be one or a plurality of memory manipulation operations corresponding to only one object in the application code, or there may be a plurality of memory manipulation operations corresponding to a plurality of objects in the application code. Note that in one embodiment, the memory manipulation operation(s) that is (are) loaded is binary executable object code. Alternatively, the memory manipulation operation(s) that is (are) loaded is executable intermediary code.

In this arrangement, which may be termed "master/slave" each of the slave (or secondary) machines M1, M3, ..., Mn loads the modified object (or class), and inclusive of the new modified memory manipulation operation(s), that was sent to it over the computer communications network or other communications link or path by the master (or primary) machine, such as machine M2, or some other machine such as a machine X of FIG. 15. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer communications network can be replaced by a shared storage device such as a shared file system, or a shared document/file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that in accordance with the inventive principles described herein, each of the plurality of machines behaves consistently and coherently relative to the other machines to accomplish the operations and objectives described herein. Furthermore, it will be appreciated in light of the description provided herein that there are a myriad of ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that embodiments of the invention may be implemented within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software, in firmware, or in any combination of these.

In a still further embodiment, each machine M1, ..., Mn receives the unmodified asset (such as class or object) inclusive of one or more memory manipulation operation(s), but modifies the operations and then loads the asset (such as class or object) consisting of the now modified operations. Although one machine, such as the master or primary machine may customize or perform a different modification to the memory manipulation operation(s) sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different and to be enhanced, customized, and/or optimized based upon its particular machine architecture, hardware, processor, memory, configuration, operating system, or other factors, yet still similar, coherent and consistent with other machines with all other similar modifications and characteristics that may not need to be similar or identical.

In all of the described instances or embodiments, the supply or the communication of the asset code (such as class code or object code) to the machines M1, ..., Mn, and optionally inclusive of a machine X of FIG. 15, can be branched, distributed or communicated among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4, etc. directly), or by providing or using cascaded or sequential communication (for example, M2 supplies M1 which then supplies M3 which then supplies M4, and so on), or a combination of the direct and cascaded and/or sequential.

Reference is made to the accompanying Annexure A in which: Annexure A5 is a typical code fragment from a memory manipulation operation prior to modification (e.g., an exemplary unmodified routine with a memory manipulation operation), and Annexure A6 is the same routine with a memory manipulation operation after modification (e.g., an

exemplary modified routine with a memory manipulation operation). These code fragments are exemplary only and identify one software code means for performing the modification in an exemplary language. It will be appreciated that other software/firmware or computer program code may be used to accomplish the same or analogous function or operation without departing from the invention.

Annexures A5 and A6 (also reproduced in part in Table VI and Table VII below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of a routine with a memory manipulation operation of application program code 50 and a post-modification excerpt of the same routine such as may be used in embodiments of the present invention having multiple machines. The modified code that is added to the routine is highlighted in **bold** text.

Table I. Summary Listing of Contents of Annexure A

Annexure A includes exemplary program listings in the JAVA language to further illustrate features, aspects, methods, and procedures of described in the detailed description
A1. This first excerpt is part of an illustration of the modification code of the modifier 51 in accordance with steps 92 and 103 of FIG. 10. It searches through the code array of the application program code 50, and when it detects a memory manipulation instruction (i.e. a putstatic instruction (opcode 178) in the JAVA language and virtual machine environment) it modifies the application program code by the insertion of an "alert" routine.
A2. This second excerpt is part of the DRT.alert() method and implements the step of 125 and arrow of 127 of FIG. 12. This DRT.alert() method requests one or more threads of the DRT processing environment of FIG. 12 to update and propagate the value and identity of the changed memory location corresponding to the operation of Annexure A1.
A3. This third excerpt is part of the DRT 71, and corresponds to step 128 of FIG. 12. This code fragment shows the DRT in a separate thread, such as thread 121/1 of FIG. 12, after being notified or requested by step 125 and array 127, and sending the changed value and changed value location/identity across the network 53 to the other of the plurality of machines M1...Mn.
A4. The fourth excerpt is part of the DRT 71, and corresponds to steps 135 and 136 of FIG. 13. This is a fragment of code to receive a propagated identity and value pair sent by another DRT 71 over the network, and write the changed value to the identified memory location.
A5. The fifth excerpt is an disassembled compiled form of the example.java application of Annexure A7, which performs a memory manipulation operation (putstatic and putfield).

Annexure A includes exemplary program listings in the JAVA language to further illustrate features, aspects, methods, and procedures of described in the detailed description

A6. The sixth excerpt is the disassembled compiled form of the same example application in Annexure A5 after modification has been performed by FieldLoader.java of Annexure A11, in accordance with FIG. 9 of this invention. The modifications are highlighted in bold.

A7. The seventh excerpt is the source-code of the example.java application used in excerpt A5 and A6. This example application has two memory locations (staticValue and instanceValue) and performs two memory manipulation operations.

A8. The eighth excerpt is the source-code of FieldAlert.java which corresponds to step 125 and arrow 127 of FIG. 12, and which requests a thread 121/1 executing FieldSend.java of the "distributed run-time" 71 to propagate a changed value and identity pair to the other machines M1...Mn.

A9. The ninth excerpt is the source-code of FieldSend.java which corresponds to step 128 of FIG. 12, and waits for a request/notification generated by FieldAlert.java of A8 corresponding to step 125 and arrow 127, and which propagates a changed value/identity pair requested of it by FieldAlert.java, via network 53.

A10. The tenth excerpt is the source-code of FieldReceive.java, which corresponds to steps 135 and 136 of FIG. 13, and which receives a propagated changed value and identity pair sent to it over the network 53 via FieldSend.java of annexure A9.

A11. FieldLoader.java. This excerpt is the source-code of FieldLoader.java, which modifies an application program code, such as the example.java application code of Annexure A7, as it is being loaded into a JAVA virtual machine in accordance with steps 90, 91, 92, 103, and 94 of FIG. 10. FieldLoader.java makes use of the convenience classes of Annexures A12 through to A36 during the modification of a compiled JAVA

A12. Attribute_info.java
Convience class for representing attribute_info structures within ClassFiles.

A13. ClassFile.java
Convience class for representing ClassFile structures.

A14. Code_attribute.java
Convience class for representing Code_attribute structures within ClassFiles.

A15. CONSTANT_Class_info.java
Convience class for representing CONSTANT_Class_info structures within ClassFiles.

Annexure A includes exemplary program listings in the JAVA language to further illustrate features, aspects, methods, and procedures of described in the detailed description

A16. CONSTANT_Double_info.java
Convience class for representing CONSTANT_Double_info structures within ClassFiles.

A17. CONSTANT_Fieldref_info.java
Convience class for representing CONSTANT_Fieldref_info structures within ClassFiles.

A18. CONSTANT_Float_info.java
Convience class for representing CONSTANT_Float_info structures within ClassFiles.

A19. CONSTANT_Integer_info.java
Convience class for representing CONSTANT_Integer_info structures within ClassFiles.

A20. CONSTANT_InterfaceMethodref_info.java
Convience class for representing CONSTANT_InterfaceMethodref_info structures within ClassFiles.

A21. CONSTANT_Long_info.java
Convience class for representing CONSTANT_Long_info structures within ClassFiles.

A22. CONSTANT_Methodref_info.java
Convience class for representing CONSTANT_Methodref_info structures within ClassFiles.

A23. CONSTANT_NameAndType_info.java
Convience class for representing CONSTANT_NameAndType_info structures within ClassFiles.

A24. CONSTANT_String_info.java
Convience class for representing CONSTANT_String_info structures within ClassFiles.

A25. CONSTANT_Utf8_info.java
Convience class for representing CONSTANT_Utf8_info structures within ClassFiles.

A26. ConstantValue_attribute.java
Convience class for representing ConstantValue_attribute structures within ClassFiles.

A27. cp_info.java
Convience class for representing cp_info structures within ClassFiles.

Annexure A includes exemplary program listings in the JAVA language to further illustrate features, aspects, methods, and procedures of described in the detailed description

A28. Deprecated_attribute.java
Convience class for representing Deprecated_attribute structures within ClassFiles.

A29. Exceptions_attribute.java
Convience class for representing Exceptions_attribute structures within ClassFiles.

A30. field_info.java
Convience class for representing field_info structures within ClassFiles.

A31. InnerClasses_attribute.java
Convience class for representing InnerClasses_attribute structures within ClassFiles.

A32. LineNumberTable_attribute.java
Convience class for representing LineNumberTable_attribute structures within ClassFiles.

A33. LocalVariableTable_attribute.java
Convience class for representing LocalVariableTable_attribute structures within ClassFiles.

A34. method_info.java
Convience class for representing method_info structures within ClassFiles.

A35. SourceFile_attribute.java
Convience class for representing SourceFile_attribute structures within ClassFiles.

A36. Synthetic_attribute.java
Convience class for representing Synthetic_attribute structures within ClassFiles.

Table II. Exemplary code listing showing embodiment of modified code.

A1. This first excerpt is part of an illustration of the modification code of the modifier 51 in accordance with steps 92 and 103 of FIG. 10. It searches through the code array of the application program code 50, and when it detects a memory manipulation instruction (i.e. a putstatic instruction (opcode 178) in the JAVA language and virtual machine environment) it modifies the application program code by the insertion of an "alert" routine.

```
// START
byte[] code =  Code_attribute.code;      // Bytecode of a given method in a
                                         // given classfile.

int code_length = Code_attribute.code_length;

int DRT = 99;        // Location of the CONSTANT_Methodref_info for the
```

```
                    // DRT.alert() method.

for (int i=0; i<code_length; i++){

   if ((code[i] & 0xff) == 179){ // Putstatic instruction.

      System.arraycopy(code, i+3, code, i+6, code_length-(i+3));

      code[i+3] = (byte) 184;     // Invokestatic instruction for the
                                  // DRT.alert() method.

      code[i+4] = (byte) ((DRT >>> 8) & 0xff);

      code[i+5] = (byte) (DRT & 0xff);

   }

}
// END
```

Table III. Exemplary code listing showing embodiment of code for alert method

A2. This second excerpt is part of the DRT.alert() method and implements the step of 125 and arrow of 127 of FIG. 12. This DRT.alert() method requests one or more threads of the DRT processing environment of FIG. 12 to update and propagate the value and identity of the changed memory location corresponding to the operation of Annexure A1.

```
// START
public static void alert(){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.notify(); // Alerts a waiting DRT thread in the background.

   }

}
// END
```

Table IV. Exemplary code listing showing embodiment of code for DRT

A3. This third excerpt is part of the DRT 71, and corresponds to step 128 of FIG. 12. This code fragment shows the DRT in a separate thread, such as thread 121/1 of FIG. 12, after being notified or requested by step 125 and array 127, and sending the changed value and changed value location/identity across the network 53 to the other of the plurality of machines M1...Mn.

```
// START
MulticastSocket ms = DRT.getMulticastSocket();        // The multicast socket
                                                      // used by the DRT for
                                                      // communication.

byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.

Field field = modifiedClass.getDeclaredField("myField1");    // Stores
                                                             // the field
                                                             // from the
```

```
                                                                    // modified
                                                                    // class.

// In this example, the field is a byte field.
while (DRT.isRunning()){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.wait();   // The DRT thread is waiting for the alert
                           // method to be called.

      byte[] b = new byte[]{nameTag, field.getByte(null)};  // Stores
                                                            // the
                                                            // nameTag
                                                            // and the
                                                            // value
                                                            // of the
                                                            // field from
                                                            // the modified
                                                            // class in a
                                                            // buffer.

      DatagramPacket dp = new DatagramPacket(b, 0, b.length);

      ms.send(dp);  // Send the buffer out across the network.

   }

}
// END
```

Table V. Exemplary code listing showing embodiment of code for DRT receiving.

A4. The fourth excerpt is part of the DRT 71, and corresponds to steps 135 and 136 of FIG. 13. This is a fragment of code to receive a propagated identity and value pair sent by another DRT 71 over the network, and write the changed value to the identified memory location.

```
// START
MulticastSocket ms = DRT.getMulticastSocket(); // The multicast socket
                                               // used by the DRT for
                                               // communication.

DatagramPacket dp = new DatagramPacket(new byte[2], 0, 2);

byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.

Field field = modifiedClass.getDeclaredField("myField1");   // Stores the
                                                            // field from
                                                            // the modified
                                                            // class.

// In this example, the field is a byte field.
while (DRT.isRunning){

   ms.receive(dp);  // Receive the previously sent buffer from the network.

   byte[] b = dp.getData();

   if (b[0] == nameTag){   // Check the nametags match.
```

```
        field.setByte(null, b[1]);  // Write the value from the network packet
                                    // into the field location in memory.

    }

}
// END
```

Table VI. Exemplary code listing showing embodiment of application before modification is made.

A5. The fifth excerpt is an disassembled compiled form of the example.java application of Annexure A7, which performs a memory manipulation operation (putstatic and putfield).
Method void setValues(int, int) 0 iload_1 1 putstatic #3 <Field int staticValue> 4 aload_0 5 iload_2 6 putfield #2 <Field int instanceValue> 9 return

Table VII. Exemplary code listing showing embodiment of application after modification is made.

A6. The sixth excerpt is the disassembled compiled form of the same example application in Annexure A5 after modification has been performed by FieldLoader.java of Annexure A11, in accordance with FIG. 9 of this invention. The modifications are highlighted in **bold.**
Method void setValues(int, int) 0 iload_1 1 putstatic #3 <Field int staticValue> **4 ldc #4 <String "example">** **6 iconst_0** **7 invokestatic #5 <Method void alert(java.lang.Object, int)>** 10 aload_0 11 iload_2 12 putfield #2 <Field int instanceValue> **15 aload_0** **16 iconst_1** **17 invokestatic #5 <Method void alert(java.lang.Object, int)>** 20 return

Table VIII. Exemplary code listing showing embodiment of source-code of the example application.

A7. The seventh excerpt is the source-code of the example.java application used in excerpt A5 and A6. This example application has two memory locations (staticValue and instanceValue) and performs two memory manipulation operations.
`import java.lang.*;`

```
public class example{

   /** Shared static field. */
   public static int staticValue = 0;

   /** Shared instance field. */
   public int instanceValue = 0;

   /** Example method that writes to memory (instance field). */
   public void setValues(int a, int b){

      staticValue = a;

      instanceValue = b;

   }


}
```

Table IX. Exemplary code listing showing embodiment of the source-code of FieldAlert.

A8. The eighth excerpt is the source-code of FieldAlert.java which corresponds to step 125 and arrow 127 of FIG. 12, and which requests a thread 121/1 executing FieldSend.java of the "distributed run-time" 71 to propagate a changed value and identity pair to the other machines M1...Mn.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;
public class FieldAlert{

   /** Table of alerts. */
   public final static Hashtable alerts = new Hashtable();

   /** Object handle. */
   public Object reference = null;


   /** Table of field alerts for this object. */
   public boolean[] fieldAlerts = null;

   /** Constructor. */
   public FieldAlert(Object o, int initialFieldCount){
      reference = o;
      fieldAlerts = new boolean[initialFieldCount];
   }

   /** Called when an application modifies a value. (Both objects and
       classes) */
   public static void alert(Object o, int fieldID){

      // Lock the alerts table.
      synchronized (alerts){

         FieldAlert alert = (FieldAlert) alerts.get(o);

         if (alert == null){      // This object hasn't been alerted already,
                                  // so add to alerts table.
            alert = new FieldAlert(o, fieldID + 1);
```

```
            alerts.put(o, alert);
         }

         if (fieldID >= alert.fieldAlerts.length){
            // Ok, enlarge fieldAlerts array.
            boolean[] b = new boolean[fieldID+1];
            System.arraycopy(alert.fieldAlerts, 0, b, 0,
               alert.fieldAlerts.length);
            alert.fieldAlerts = b;
         }

         // Record the alert.
         alert.fieldAlerts[fieldID] = true;

         // Mark as pending.
         FieldSend.pending = true;        // Signal that there is one or more
                                          // propagations waiting.

         // Finally, notify the waiting FieldSend thread(s)
         if (FieldSend.waiting){
            FieldSend.waiting = false;
            alerts.notify();
         }

      }

   }


}
```

It is noted that the compiled code in the annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure A7 (Table VIII). In the procedure of Annexure A5 and Table VI, the procedure name "Method void setValues(int, int)" of Step 001 is the name of the displayed disassembled output of the setValues method of the compiled application code of "example.java". The name "Method void setValues(int, int)" is arbitrary and selected for this example to indicate a typical JAVA method inclusive of a memory manipulation operation. Overall the method is responsible for writing two values to two different memory locations through the use of a memory manipulation assignment statement (being "putstatic" and "putfield" in this example) and the steps to accomplish this are described in turn.

First (Step 002), the Java Virtual Machine instruction "iload_1" causes the Java Virtual Machine to load the integer value in the local variable array at index 1 of the current method frame and store this item on the top of the stack of the current method frame and results in the integer value passed to this method as the first argument and stored in the local variable array at index 1 being pushed onto the stack.

The Java Virtual Machine instruction "putstatic #3 <Field int staticValue>" (Step 003) causes the Java Virtual Machine to pop the topmost value off the stack of the current method frame and store the value in the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example setValues() method and

results in the topmost integer value of the stack of the current method frame being stored in the integer field named "staticValue".

The Java Virtual Machine instruction "aload_0" (Step 004) causes the Java Virtual Machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame and results in the 'this' object reference stored in the local variable array at index 0 being pushed onto the stack.

First (Step 005), the Java Virtual Machine instruction "iload_2" causes the Java Virtual Machine to load the integer value in the local variable array at index 2 of the current method frame and store this item on the top of the stack of the current method frame and results in the integer value passed to this method as the first argument and stored in the local variable array at index 2 being pushed onto the stack.

The Java Virtual Machine instruction "putfield #2 <Field int instanceValue>" (Step 006) causes the Java Virtual Machine to pop the two topmost values off the stack of the current method frame and store the topmost value in the object instance field of the second popped value, indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 2nd index of the classfile structure of the application program containing this example setValues method and results in the integer value on the top of the stack of the current method frame being stored in the instance field named "instanceValue" of the object reference below the integer value on the stack.

Finally, the JAVA virtual machine instruction "return" (Step 007) causes the JAVA virtual machine to cease executing this setValues() method by returning control to the previous method frame and results in termination of execution of this setValues() method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the JAVA virtual machine manipulates (i.e. writes to) the staticValue and instanceValue memory locations, and in executing the setValues() method containing the memory manipulation operation(s) is able to ensure that memory is and remains consistent between multiple threads of a single application instance, and therefore ensure that unwanted behaviour, such as for example inconsistent or incoherent memory between multiple threads of a single application instance (such inconsistent or incoherent memory being for example incorrect or different values or contents with respect to a single memory location) does not occur. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 by concurrently executing the application program code 50 on each one of the plurality of machines M1..Mn, the memory manipulation operations of each concurrently executing application program occurrence on each one of the machines would be performed without coordination between any other machine(s), such coordination being for example updating of corresponding memory locations on each machine such that they each report a same content or value. Given the desirable result of consistent, coordinated and coherent memory state and manipulation and updating operation across a plurality of a machines, this prior art arrangement would fail to perform such consistent, coherent, and coordinated memory state and manipulation and updating operation across the plurality of

machines, as each machine performs memory manipulation only locally and without any attempt to coordinate or update their local memory state and manipulation operation with any other similar memory state on any one or more other machines. Such an arrangement would therefore be susceptible to inconsistent and incoherent memory state amongst machines M1..Mn due to uncoordinated, inconsistent and/or incoherent memory manipulation and updating operation. Therefore it is desirable to overcome this limitation of the prior art arrangement.

In the exemplary code in Table VII (Annexure A6), the code has been modified so that it solves the problem of consistent, coordinated memory manipulation and updating operation for a plurality of machines M1...Mn, that was not solved in the code example from Table VI (Annexure A5). In this modified setValues() method code, an "ldc #4 <String "example">" instruction is inserted after the "putstatic #3" instruction in order to be the first instruction following the execution of the "putstatic #3" instruction. This causes the JAVA virtual machine to load the String value "example" onto the stack of the current method frame and results in the String value of "example" loaded onto the top of the stack of the current method frame. This change is significant because it modifies the setValues() method to load a String identifier corresponding to the classname of the class containing the static field location written to by the "putstatic #3" instruction onto the stack.

Furthermore, the JAVA virtual machine instruction "iconst_0" is inserted after the "ldc #4" instruction so that the JAVA virtual machine loads an integer value of "0" onto the stack of the current method frame and results in the integer value of "0" loaded onto the top of the stack of the current method frame. This change is significant because it modifies the setValues() method to load an integer value, which in this example is "0", which represents the identity of the memory location (field) manipulated by the preceding "putstatic #3" operation. It is to be noted that the choice or particular form of the memory identifier used for the implementation of this invention is for illustration purposes only. In this example, the integer value of "0" is the identifier used of the manipulated memory location, and corresponds to the "staticValue" field as the first field of the "example.java" application, as shown in Annexure A7. Therefore, corresponding to the "putstatic #3" instruction, the "iconst_0" instruction loads the integer value "0" corresponding to the index of the manipulated field of the "putstatic #3" instruction, and which in this case is the first field of "example.java" hence the "0" integer index value, onto the stack.

Additionally, the JAVA virtual machine instruction "invokestatic #5 <Method boolean alert(java.lang.Object, int)>" is inserted after the "iconst_0" instruction so that the JAVA virtual machine pops the two topmost items off the stack of the current method frame (which in accordance with the preceding "ldc #4" instruction is a reference to the String object with the value "example" corresponding to the name of the class to which manipulated field belongs, and the integer "0" corresponding to the index of the manipulated field in the example.java application) and invokes the "alert" method, passing the two topmost items popped off the stack to the new method frame as its first two arguments. This change is significant because it modifies the setValues() method to execute the "alert" method and associated operations, corresponding to the preceding

memory manipulation operation (that is, the "putstatic #3" instruction) of the setValues() method.

Likewise, in this modified setValues() method code, an "aload_0" instruction is inserted after the "putfield #2" instruction in order to be the first instruction following the execution of the "putfield #2" instruction. This causes the JAVA virtual machine to load the instance object of the example class to which the manipulated field of the preceding "putfield #2" instruction belongs, onto the stack of the current method frame and results in the object reference corresponding to the instance field written to by the "putfield #2" instruction, loaded onto the top of the stack of the current method frame. This change is significant because it modifies the setValues() method to load a reference to the object corresponding to the manipulated field onto the stack..

Furthermore, the JAVA virtual machine instruction "iconst_1" is inserted after the "aload_0" instruction so that the JAVA virtual machine loads an integer value of "1" onto the stack of the current method frame and results in the integer value of "1" loaded onto the top of the stack of the current method frame. This change is significant because it modifies the setValues() method to load an integer value, which in this example is "1", which represents the identity of the memory location (field) manipulated by the preceding "putfield #2" operation. It is to be noted that the choice or particular form of the identifier used for the implementation of this invention is for illustration purposes only. In this example, the integer value of "1" corresponds to the "instanceValue" field as the second field of the "example.java" application, as shown in Annexure A7. Therefore, corresponding to the "putfield #2" instruction, the "iconst_1" instruction loads the integer value "1" corresponding to the index of the manipulated field of the "putfield #2" instruction, and which in this case is the second field of "example.java" hence the "1" integer index value, onto the stack.

Additionally, the JAVA virtual machine instruction "invokestatic #5 <Method boolean alert(java.lang.Object, int)>" is inserted after the "iconst_1" instruction so that the JAVA virtual machine pops the two topmost item off the stack of the current method frame (which in accordance with the preceding "aload_0" instruction is a reference to the object corresponding to the object to which the manipulated instance field belongs, and the integer "1" corresponding to the index of the manipulated field in the example.java application) and invokes the "alert" method, passing the two topmost items popped off the stack to the new method frame as its first two arguments. This change is significant because it modifies the setValues() method to execute the "alert" method and associated operations, corresponding to the preceding memory manipulation operation (that is, the "putfield #2" instruction) of the setValues() method.

The method void alert(java.lang.Object, int), part of the FieldAlert code of Annexure A8 and part of the distributed runtime system (DRT) 71, requests or otherwise notifies a DRT thread 121/1 executing the FieldSend.java code of Annexure A9 to update and propagate the changed identity and value of the manipulated memory location to the plurality of machines M1...Mn.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated

operation of memory manipulation operations so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as for example inconsistent and incoherent memory state and manipulation and updating operation) does not occur when applying the modified code or procedure.

INITIALIZATION

Returning again to FIG. 14, there is illustrated a schematic representation of a single prior art computer operated as a JAVA virtual machine. In this way, a machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) can operate in the particular language of the application program code 50, in this instance the JAVA language. That is, a JAVA virtual machine 72 is able to operate application code 50 in the JAVA language, and utilize the JAVA architecture irrespective of the machine manufacturer and the internal details of the machine.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform, and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated in light of the description provided herein that the platform and/or runtime system may include virtual machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

Returning to the example of the JAVA language virtual machine environment, in the JAVA language, the class initialization routine <clinit> happens only once when a given class file 50A is loaded. However, the object initialization routine <init> typically happens frequently, for example the object initialization routine may usually occur every time a new object (such as an object 50X, 50Y or 50Z) is created. In addition, within the JAVA environment and other machine or other runtime system environments using classes and object constructs, classes (generally being a broader category than objects) are loaded prior to objects (which are the narrower category and wherein the objects belong to or are identified with a particular class) so that in the application code 50 illustrated in FIG. 14, having a single class 50A and three objects 50X, 50Y, and 50Z, the first class 50A is loaded first, then first object 50X is loaded, then second object 50Y is loaded and finally third object 50Z is loaded.

Where, as in the embodiment illustrated relative to FIG. 14, there is only a single computer or machine 72 (and not a plurality of connected or coupled computers or machines), then no conflict or inconsistency arises in the running of the initialization routines (such as class and object initialization routines) intended to operate during the loading procedure because for conventional operation each initialization routine is executed only once by the single virtual machine or machine or runtime system or language environment as needed for each of the one or more classes and one or more

objects belonging to or identified with the classes, or equivalent where the terms classes and object are not used.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method, and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the PowerPC computer architecture manufactured by International Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, the terms 'class' and 'object' may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and boolean data types), structured data types (such as arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, references and unions.

Returning to the example of the JAVA language virtual machine environment, in the JAVA language, the class initialization routine <clinit> happens only once when a given class file 50A is loaded. However, the object initialization routine <init> typically happens frequently, for example the object initialisation routine will occur every time a new object (such as an object 50X, 50Y and 50Z) is created. In addition, within the JAVA environment and other machine or other runtime system environments using classes and object constructures, classes (being the broader category) are loaded prior to objects (which are the narrower category and wherein the objects belong to or are identified with a particular class) so that in the application code 50 illustrated in FIG. 14, having a single class 50A and three objects 50X-50Z, the first class 50A is loaded first, then the first object 50X is loaded, then second object 50Y is loaded and finally third object 50Z is loaded.

Where, as in the embodiment illustrated relative to FIG. 14, there is only a single computer or machine 72 (not a plurality of connected or coupled machines), then no conflict or inconsistency arises in the running of the initialization routines (i.e. the class initialization routine <clinit> and the object initialisation routine <init>) intended to operate during the loading procedure because for conventional operation each initialisation routine is executed only once by the single virtual machine or machine or runtime system or language environment as needed for each of the one or more classes and one or more objects belonging to or identified with the classes.

However, in the arrangement illustrated in FIG. 8, (and also in FIGS. 31-33), a plurality of individual computers or machines M1, M2, ..., Mn are provided, each of which are interconnected via a communications network 53 or other communications link

and each of which individual computers or machines provided with a modifier 51 (See in FIG. 5) and realised by or in for example the distributed runtime system(DRT) 71 (See FIG. 8) and loaded with a common application code 50. The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on each one of the plurality of computers or machines M1, M2...Mn, or optionally on each one of some subset of the plurality of computers or machines M1, M2...Mn. Put somewhat differently, there is a common application program represented in application code 50, and this single copy or perhaps a plurality of identical copies are modified to generate a modified copy or version of the application program or program code, each copy or instance prepared for execution on the plurality of machines. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing the features of the invention may optionally be connected to or coupled with other computers, machines, information appliances, or the like that do not implement the features of the invention.

In some embodiments, some or all of the plurality of individual computers or machines may be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

Essentially the modifier 51 or DRT 71 or other code modifying means is responsible for modifying the application code 50 so that it may execute initialisation routines or other initialization operations, such as for example class and object initialization methods or routines in the JAVA language and virtual machine environment, in a coordinated, coherent, and consistent manner across and between the plurality of individual machines M1, M2...Mn. It follows therefore that in such a computing environment it is necessary to ensure that the local objects and classes on each of the individual machines M1, M2...Mn is initialized in a consistent fashion (with respect to the others).

It will be appreciated in light of the description provided herein that there are alternative implementations of the modifier 51 and the distributed run time 71. For example, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71. In one embodiment, the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. The modifier function and structure therefore maybe subsumed into the DRT and considered to be an optional component. Independent of how implemented, the modifier function and structure is responsible for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the

DRT on each machine. The DRT may for example implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures within the invention are less important than that they are provided.

In order to ensure consistent class and object (or equivalent) initialisation status and initialisation operation between and amongst machines M1, M2,..., Mn, the application code 50 is analysed or scrutinized by searching through the executable application code 50 in order to detect program steps (such as particular instructions or instruction types) in the application code 50 which define or constitute or otherwise represent an initialization operation or routine (or other similar memory, resource, data, or code initialization routine or operation). In the JAVA language, such program steps may for example comprise or consist of some part of, or all of, a "<init>" or "<clinit>" method of an object or class, and optionally any other code, routine, or method related to a "<init>" or "<clinit>" method, for example by means of a method invocation from the body of the "<init>" of "<clinit>" method to a different method.

This analysis or scrutiny of the application code 50 may take place either prior to loading the application program code 50, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application code may be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as for example from source-code language or intermediate-code language to object-code language or machine-code language), and with the understanding that the term compilation normally or conventionally involves a change in code or language, for example, from source code to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object-code), and compilation from source-code to source-code, as well as compilation from object-code to object-code, and any altered combinations therein. It is also inclusive of so-called "intermediary-code languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the application program code such as by the operating system reading the application code from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader loadClass method (e.g., "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the application program code has started or commenced, such as for example after the JAVA virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

As a consequence, of the above described analysis or scrutiny, initialization routines (for example <clinit> class initialisation methods and <init> object initialization methods) are initially looked for, and when found or identified a modifying code is inserted, so as to give rise to a modified initialization routine. This modified routine is adapted and written to initialize the class 50A on one of the machines, for example JVM#1, and tell, notify, or otherwise communicate to all the other machines M2, ..., Mn that such a class 50A exists and optionally its initialized state. There are several different alternative modes wherein this modification and loading can be carried out.

Thus, in one mode, the DRT 71/1 on the loading machine, in this example Java Virtual Machine M1 (JVM#1), asks the DRT's 71/2...71/n of all the other machines M1, ..., Mn if the similar equivalent first class 50A is initialized (i.e. has already been initialized) on any other machine. If the answer to this question is yes (that is, a similar equivalent class 50A has already been initialized on another machine), then the execution of the initialization procedure is aborted, paused, terminated, turned off or otherwise disabled for the class 50A on machine JVM#1. If the answer is no (that is, a similar equivalent class 50A has not already been initialised on another machine), then the initialization operation is continued (or resumed, or started, or commenced and the class 50A is initialized and optionally the consequential changes (such as for example initialized code and data-structures in memory) brought about during that initialization procedure are transferred to each similar equivalent local class on each one of the other machines as indicated by arrows 83 in FIG. 8.

A similar procedure happens on each occasion that an object, say 50X, 50Y or 50Z is to be loaded and initialized. Where the DRT 71/1 of the loading machine, in this example Java Machine M1 (JVM#1), does not discern, as a result of interrogation of the other machines M2...Mn that, a similar equivalent object to the particular object to be initialized on machine M1, say object 50Y, has already been initialised by another machine, then the DRT 71/1 on machine M1 may execute the object initialization routine corresponding to object 50Y, and optionally each of the other machines M2...Mn may load a similar equivalent local object (which may conveniently be termed a peer object) and associated consequential changes (such as for example initialized data, initialized code, and/or initialized system or resources structures) brought about by the execution of the initialization operation on machine M1. However, if the DRT 71/1 of machine M1 determines that a similar equivalent object to the object 50Y in question has already been initialization on another machine of the plurality of machines (say for example machine M2), then the execution by machine M1 of the initialization function, procedure, or routine corresponding to object 50Y is not started or commenced, or is otherwise aborted, terminated, turned off or otherwise disabled, and object 50Y on machine M1 is loaded, and preferably but optionally the consequential changes (such as for example initialized data, initialized code, and/or other initialized system or resource structures) brought about

by the execution of the initialization routine by machine M2, is loaded on machine M1 corresponding to object 50Y. Again there are various ways of bringing about the desired result.

Preferably, execution of the initialization routine is allocated to one machine, such as the first machine M1 to load (and optionally seek to initialize) the object or class. The execution of the initialization routine corresponding to the determination that a particular class or object (and any similar equivalent local classes or objects on each of the machines M1...Mn) is not already initialized, is to execute only once with respect to all machines M1...Mn, and preferably by only one machine, on behalf of all machines M1...Mn. Corresponding to, and preferably following, the execution of the initialization routine by one machine (say machine M1), all other machines may then each load a similar equivalent local object (or class) and optionally load the consequential changes (such as for example initialized data, initialized code, and/or other initialized system or resource structures) brought about by the execution of the initialization operation by machine M1.

As seen in FIG. 15 a modification to the general arrangement of FIG. 8 is provided in that machines M1, M2...Mn are as before and run the same application code 50 (or codes) on all machines M1, M2...Mn simultaneously or concurrently. However, the previous arrangement is modified by the provision of a server machine X which is conveniently able to supply housekeeping functions, for example, and especially the initialisation of structures, assets, and resources. Such a server machine X can be a low value commodity computer such as a PC since its computational load is low. As indicated by broken lines in FIG. 15, two server machines X and X+1 can be provided for redundancy purposes to increase the overall reliability of the system. Where two such server machines X and X+1 are provided, they are preferably but optionally operated as redundant machines in a failover arrangement.

It is not necessary to provide a server machine X as its computational load can be distributed over machines M1, M2...Mn. Alternatively, a database operated by one machine (in a master/slave type operation) can be used for the housekeeping function(s).

FIG. 16 shows a preferred general procedure to be followed. After a loading step 161 has been commenced, the instructions to be executed are considered in sequence and all initialization routines are detected as indicated in step 162. In the JAVA language these are the object initialisation methods (e.g. "<init>") and class initialisation methods (e.g. "<clinit>"). Other languages use different terms.

Where an initialization routine is detected in step 162, it is modified in step 163 in order to perform consistent, coordinated, and coherent initialization operation (such as for example initialization of data structures and code structures) across and between the plurality of machines M1,M2...Mn, typically by inserting further instructions into the initialisation routine to, for example, determine if a similar equivalent object or class (or other asset) on machines M1...Mn corresponding to the object or class (or asset) to which this initialisation routine corresponds, has already been initialised, and if so, aborting, pausing, terminating, turning off, or otherwise disabling the execution of this initialization routine (and/or initialization operation(s)), or if not then starting, continuing,

or resuming the executing the initialization routine (and/or initialization operation(s)), and optionally instructing the other machines M1...Mn to load a similar equivalent object or class and consequential changes brought about by the execution of the initialization routine. Alternatively, the modifying instructions may be inserted prior to the routine, such as for example prior to the instruction(s) or operation(s) which commence initialization of the corresponding class or object. Once the modification step 163 has been completed the loading procedure continues by loading the modified application code in place of the unmodified application code, as indicated in step 164. Altogether, the initialization routine is to be executed only once, and preferably by only one machine, on behalf of all machines M1...Mn corresponding to the determination by all machines M1...Mn that the particular object or class (i.e. the similar equivalent local object or class on each machine M1...Mn corresponding to the particular object or class to which this initialization routine relates) has not been initialized.

FIG. 17 illustrates a particular form of modification. After commencing the routine in step 171, the structures, assets or resources (in JAVA termed classes or objects) to be initialised are, in step 172, allocated a name or tag (for example a global name or tag) which can be used to identify corresponding similar equivalent local objects on each of the machines M1,..., Mn. This is most conveniently done via a table (or similar data or record structure) maintained by server machine X of Fig 15. This table may also include an initialization status of the similar equivalent classes or object to be initialised. It will be understood that this table or other data structure may store only the initialization status, or it may store other status or information as well.

As indicated in FIG. 17, if steps 173 and 174 determine by means of the communication between machines M1...Mn by DRT 71 that the similar equivalent local objects on each other machine corresponding to the global name or tag is not already initialised (i.e., not initialized on a machine other than the machine carrying out the loading and seeking to perform initialization), then this means that the object or class can be initialised, preferably but optionally in the normal fashion, by starting, commencing, continuing, or resuming the execution of, or otherwise executing, the initialization routine, as indicated in step 176, since it is the first of the plurality of similar equivalent local objects or classes of machines M1...Mn to be initialized.

In one embodiment, the initialization routine is stopped from initiating or commencing or beginning execution; however, in some implementations it is difficult or practically impossible to stop the initialization routine from initiating or beginning or commencing execution. Therefore, in an alternative embodiment, the execution of the initialization routine that has already started or commenced is aborted such that it does not complete or does not complete in its normal manner. This alternative abortion is understood to include an actual abortion, or a suspend, or postpone, or pause of the execution of a initialization routine that has started to execute (regardless of the stage of execution before completion) and therefore to make sure that the initialization routine does not get the chance to execute to completion the initialization of the object (or class or other asset) – and therefore the object (or class or other asset) remains "un-initialized" (i.e., "not initialized").

However or alternatively, if steps 173 and 174 determine that the global name corresponding to the plurality of similar equivalent local objects or classes, each on a one of the plurality of machines M1...Mn, is already initialised on another machine, then this means that the object or class is considered to be initialized on behalf of, and for the purposes of, the plurality of machines M1....Mn. As a consequence, the execution of the initialisation routine is aborted, terminated, turned off, or otherwise disabled, by carrying out step 175.

FIG. 18, illustrative of one embodiment of step 173 of FIG. 17, shows the inquiry made by the loading machine (one of M1, M2...Mn) to the server machine X of FIG. 15, to enquire as to the initialisation status of the plurality of similar equivalent local objects (or classes) corresponding to the global name. The operation of the loading machine is temporarily interrupted as indicated by step 181, and corresponding to step 173 of FIG. 17, until a reply to this preceding request is received from machine X, as indicated by step 182. In step 181 the loading machine sends an inquiry message to machine X to request the initialization status of the object (or class or other asset) to be initialized. Next, the loading machine awaits a reply from machine X corresponding to the inquiry message sent by the proposing machine at step 181, indicated by step 182.

FIG. 19 shows the activity carried out by machine X of FIG. 15 in response to such an initialization enquiry of step 181 of FIG. 18. The initialization status is determined in steps 192 and 193, which determines if a similar equivalent object (or class or other asset) corresponding to the initialization status request of global name, as received at step 191, is initialized on another machine (i.e. a machine other than the enquiring machine 181 from which the initialization status request of step 191 originates), where a table of initialisation states is consulted corresponding to the record for the global name and, if the initialisation status record indicates that a similar equivalent local object (or class) on another machine (such as on a one of the machines M1...Mn) and corresponding to global name is already initialised, the response to that effect is sent to the enquiring machine by carrying out step 194. Alternatively, if the initialisation status record indicates that a similar equivalent local object (or class) on another machine (such as on a one of the plurality of machines M1...Mn) and corresponding to global name is uninitialized, a corresponding reply is sent to the enquiring machine by carrying out steps 195 and 196. The singular term object or class as used here (or the equivalent term of asset, or resource used in step 192) are to be understood to be inclusive of all similar equivalent objects (or classes, or assets, or resources) corresponding to the same global name on each one of the plurality of machines M1...Mn. The waiting enquiring machine of step 182 is then able to respond and/or operate accordingly, such as for example by (i) aborting (or pausing, or postponing) execution of the initialization routine when the reply from machine X of step 182 indicated that a similar equivalent local object on another machine (such as a one of the plurality of machines M1...Mn) corresponding to the global name of the object proposed to be initialized of step 172 is already initialized elsewhere (i.e. is initialized on a machine other than the machine proposing to carry out the initialization); or (ii) by continuing (or resuming, or starting, or commencing) execution of the initialization routine when the reply from machine X of step 182 indicated that a similar equivalent local object on the plurality of machines M1...Mn corresponding to the global name of the object proposing to be initialized of step 172 is

not initialized elsewhere (i.e. not initialized on a machine other than the machine proposing to carry out the initialization).

Reference is made to the accompanying Annexures in which: Annexures A1-A10 illustrate actual code in relation to fields, Annexure B1 is a typical code fragment from an unmodified <clinit> instruction, Annexure B2 is an equivalent in respect of a modified <clinit> instruction, Annexure B3 is a typical code fragment from an unmodified <init> instruction, Annexure B4 is an equivalent in respect of a modified <init> instruction, In addition, Annexure B5 is an alternative to the code of Annexure B2, and Annexure B6 is an alternative to the code of Annexure B4.

Furthermore, Annexure B7 is the source-code of InitClient which carries out one embodiment of the steps of FIGS. 17 and 18, which queries an "initialization server" (for example a machine X) for the initialization status of the specified class or object with respect to the plurality of similar equivalent classes or objects on the plurality of machines M1...Mn. Annexure B8 is the source-code of InitServer which carries out one embodiment of the steps of FIG. 19, which receives an initialization status query sent by InitClient and in response returns the corresponding initialization status of the specified class or object. Similarly, Annexure B9 is the source-code of the example application used in the before/after examples of Annexure B1-B6 (Repeated as Tables X through XV). And, Annexure B10 is the source-code of InitLoader which carries out one embodiment of the steps of FIGS. 16, 20, and 21, which modifies the example application program code of Annexure B9 in accordance with one mode of this invention.

Annexures B1 and B2 (also reproduced in part in Tables X and XI below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of an initialization routine of application program 50 and a post-modification excerpt of the same initialization routine such as may be used in embodiments of the present invention having multiple machines. The modified code that is added to the initialization routine is highlighted in **bold** text.

It is noted that the disassembled compiled code in the annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure B4 (Table XIII). In the procedure of Annexure B1 and Table X, the procedure name "Method <clinit>" of Step 001 is the name of the displayed disassembled output of the clinit method of the compiled application code "example.java". The method name "<clinit>" is the name of a class' initialization method in accordance with the JAVA platform specification, and selected for this example to indicate a typical mode of operation of a JAVA initialization method. Overall the method is responsible for initializing the class 'example' so that it may be used, and the steps the "example.java" code performs are described in turn.

First (Step 002) the JAVA virtual machine instruction "new #2 <Class example>" causes the JAVA virtual machine to instantiate a new class instance of the example class indicated by the CONSTANT_Classref_info constant_pool item stored in the 2nd index of the classfile structure of the application program containing this example <clinit> method

and results in a reference to an newly created object of type 'example' being placed (pushed) on the stack of the current method frame of the currently executing thread.

Next (Step 003), the Java Virtual Machine instruction "dup" causes the Java Virtual Machine to duplicate the topmost item of the stack and push the duplicated item onto the topmost position of the stack of the current method frame and results in the reference to the new created 'example' object at the top of the stack being duplicated and pushed onto the stack.

Next (Step 004), the JAVA virtual machine instruction "invokespecial #3 <Method example()>" causes the JAVA virtual machine to pop the topmost item off the stack of the current method frame and invoke the instance initialization method "<init>" on the popped object and results in the "<init>" constructor of the newly created 'example' object being invoked.

The Java Virtual Machine instruction "putstatic #3 <Field example currentExample>" (Step 005) causes the Java Virtual Machine to pop the topmost value off the stack of the current method frame and store the value in the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example <clinit> method and results in the reference to the newly created and initialized 'example' object on the top of the stack of the current method frame being stored in the static reference field named "currentExample" of class 'example'.

Finally, the Java Virtual Machine instruction "return" (Step 006) causes the Java Virtual Machine to cease executing this <clinit> method by returning control to the previous method frame and results in termination of execution of this <clinit> method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the JAVA virtual machine can keep track of the initialization status of a class in a consistent, coherent and coordinated manner, and in executing the <clinit> method containing the initialization operations is able to ensure that unwanted behaviour (for example execution of the <init> method of class 'example.java' more than once) such as may be caused by inconsistent and/or incoherent initialization operation, does not occur. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 with the memory update and propagation replication means of FIGS. 9, 10, 11, 12, and 13, and concurrently executing the application program code 50 on each one of the plurality of machines M1...Mn, the initialization operations of each concurrently executing application program occurrence on each one of the machines would be performed without coordination between any other of the occurrences on any other of the machine(s). Given the desirable result of consistent, coordinated and coherent initialization operation across a plurality of a machines, this prior art arrangement would fail to perform such consistent coordinated initialization operation across the plurality of machines, as each machine performs initialization only locally and without any attempt to coordinate their local initialization operation with any other similar initialization operation on any one or more other machines. Such an arrangement would therefore be susceptible to unwanted or other anomalous behaviour due to uncoordinated, inconsistent and/or incoherent initialization

states, and associated initialization operation. Therefore it is desirable to overcome this limitation of the prior art arrangement.

In the exemplary code in Table XIV (Annexure B5), the code has been modified so that it solves the problem of consistent, coordinated initialization operation for a plurality of machines M1...Mn, that was not solved in the code example from Table X (Annexure B1). In this modified <clinit> method code, an "ldc #2 <String "example">" instruction is inserted before the "new #5" instruction in order to be the first instruction of the <clinit> method. This causes the JAVA virtual machine to load the item in the constant_pool at index 2 of the current classfile and store this item on the top of the stack of the current method frame, and results in the reference to a String object of value "example" being pushed onto the stack.

Furthermore, the JAVA virtual machine instruction "invokestatic #3 <Method Boolean isAlreadyLoaded(java.lang.String)>" is inserted after the "0 ldc #2" instruction so that the JAVA virtual machine pops the topmost item off the stack of the current method frame (which in accordance with the preceding "ldc #2" instruction is a reference to the String object with the value "example" which corresponds to the name of the class to which this <clinit> method belongs) and invokes the "isAlreadyLoaded" method, passing the popped item to the new method frame as its first argument, and returning a boolean value onto the stack upon return from this "invokestatic" instruction. This change is significant because it modifies the <clinit> method to execute the "isAlreadyLoaded" method and associated operations, corresponding to the start of execution of the <clinit> method, and returns a boolean argument (indicating whether the class corresponding to this <clinit> method is initialized on another machine amongst the plurality of machines M1...Mn) onto the stack of the executing method frame of the <clinit> method.

Next, two JAVA virtual machine instructions "ifeq 9" and "return" are inserted into the code stream after the "2 invokestatic #3" instruction and before the "new #5" instruction. The first of these two instructions, the "ifeq 9" instruction, causes the JAVA virtual machine to pop the topmost item off the stack and performs a comparison between the popped value and zero. If the performed comparison succeeds (i.e. if and only if the popped value is equal to zero), then execution continues at the "9 new #5" instruction. If however the performed comparison fails (i.e. if and only if the popped value is not equal to zero), then execution continues at the next instruction in the code stream, which is the "8 return" instruction. This change is particularly significant because it modifies the <clinit> method to either continue execution of the <clinit> method (i.e. instructions 9-19) if the returned value of the "isAlreadyLoaded" method was negative (i.e. "false"), or discontinue execution of the <clinit> method (i.e. the "8 return" instruction causing a return of control to the invoker of this <clinit> method) if the returned value of the "isAlreadyLoaded" method was positive (i.e. "true").

The method void isAlreadyLoaded(java.lang.String), part of the InitClient code of Annexure B7, and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1...Mn to coordinate the execution of the <clinit> method amongst the machines M1...Mn. The isAlreadyLoaded method of this example communicates with the InitServer code of Annexure B8 executing on a

machine X of FIG. 15, by means of sending an "initialization status request" to machine X corresponding to the class being "initialized" (i.e. the class to which this <clinit> method belongs). With reference to FIG. 19 and Annexure B8, machine X receives the "initialization status request" corresponding to the class to which the <clinit> method belongs, and consults a table of initialization states or records to determine the initialization state for the class to which the request corresponds.

If the class corresponding to the initialization status request is not initialized on another machine other than the requesting machine, then machine X will send a response indicating that the class was not already initialized, and update a record entry corresponding to the specified class to indicate the class is now initialized. Alternatively, if the class corresponding to the initialization status request is initialized on another machine other than the requesting machine, then machine X will send a response indicating that the class is already initialized. Corresponding to the determination that the class to which this initialization status request pertains is not initialized on another machine other than the requesting machine, a reply is generated and sent to the requesting machine indicating that the class is not initialized. Additionally, machine X preferably updates the entry corresponding to the class to which the initialization status request pertained to indicate the class is now initialized. Following a receipt of such a message from machine X indicating that the class is not initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a 'false' value to the previous method frame, which is the executing method frame of the <clinit> method. Alternatively, following a receipt of a message from machine X indicating that the class is already initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a "true" value to the previous method frame, which is the executing method frame of the <clinit> method. Following this return operation, the execution of the <clinit> method frame then resumes as indicated in the code sequence of Annexure B5 at step 004.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of initialization routines or other initialization operations between and amongst machines M1...Mn so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as for example multiple initialization operation, or re-initialization operation) does not occur when applying the modified code or procedure.

Similarly, the procedure followed to modify an <init> method relating to objects so as to convert from the code fragment of Annexure B3 (See Table XII) to the code fragment of Annexure B6 (See Table XV) is indicated.

Annexures B3 and B6 (also reproduced in part in Tables XII and XV below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of an initialization routine of application program 50 and a post-modification excerpt of the same initialization routine such as may be used in embodiments of the present invention having multiple machines. The modified code that is added to the initialization routine is highlighted in **bold** text.

It is noted that the disassembled compiled code in the annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure B4. In the procedure of Annexure B1 and Table XI, the procedure name "Method <init>" of Step 001 is the name of the displayed disassembled output of the init method of the compiled application code "example.java". The method name "<init>" is the name of an object's initialization method (or methods, as there may be more than one) in accordance with the JAVA platform specification, and selected for this example to indicate a typical mode of operation of a JAVA initialization method. Overall the method is responsible for initializing an 'example' object so that it may be used, and the steps the "example.java" code performs are described in turn.

The Java Virtual Machine instruction "aload_0" (Step 002) causes the Java Virtual Machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame and results in the 'this' object reference stored in the local variable array at index 0 being pushed onto the stack.

Next (Step 003), the JAVA virtual machine instruction "invokespecial #1 <Method java.lang.Object()>" causes the JAVA virtual machine to pop the topmost item off the stack of the current method frame and invoke the instance initialization method "<init>" on the popped object and results in the "<init>" constructor (or method) of the 'example' object's superclass being invoked.

The Java Virtual Machine instruction "aload_0" (Step 004) causes the Java Virtual Machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame and results in the 'this' object reference stored in the local variable array at index 0 being pushed onto the stack.

Next (Step 005), the JAVA virtual machine instruction "invokestatic #2 <Method long currentTimeMillis()>" causes the JAVA virtual machine to invoke the "currentTimeMillis()" method of the java.lang.System class, and results in a long value pushed onto the top of the stack corresponding to the return value from the currentTimeMillis() method invocation.

The Java Virtual Machine instruction "putfield #3 <Field long timestamp>" (Step 006) causes the Java Virtual Machine to pop the two topmost values off the stack of the current method frame and store the topmost value in the object instance field of the second popped value, indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example <init> method, and results in the long value on the top of the stack of the current method frame being stored in the instance field named "timestamp" of the object reference below the long value on the stack.

Finally, the Java Virtual Machine instruction "return" (Step 007) causes the Java Virtual Machine to cease executing this <init> method by returning control to the previous method frame and results in termination of execution of this <init> method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the JAVA virtual machine can keep track of the initialization status of an object in a consistent, coherent and coordinated manner, and in executing the <init> method containing the initialization operations is able to ensure that unwanted behaviour (for example execution of the <init> method of a single 'example.java' object more than once, or re-initialization of the same object) such as may be caused by inconsistent and/or incoherent initialization operation, does not occur. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 with the memory update and propagation replication means of FIGS. 9, 10, 11, 12, and 13, and concurrently executing the application program code 50 on each one of the plurality of machines M1...Mn, the initialization operations of each concurrently executing application program occurrence on each one of the machines would be performed without coordination between any other of the occurrences on any other of the machine(s). Given the desirable result of consistent, coordinated and coherent initialization operation across a plurality of a machines, this prior art arrangement would fail to perform such consistent coordinated initialization operation across the plurality of machines, as each machine performs initialization only locally and without any attempt to coordinate their local initialization operation with any other similar initialization operation on any one or more other machines. Such an arrangement would therefore be susceptible to unwanted or other anomalous behaviour due to uncoordinated, inconsistent and/or incoherent initialization states, and associated initialization operation. Therefore it is desirable to overcome this limitation of the prior art arrangement.

In the exemplary code in Table XV (Annexure B6), the code has been modified so that it solves the problem of consistent, coordinated initialization operation for a plurality of machines M1...Mn, that was not solved in the code example from Table XII(Annexure B3). In this modified <init> method code, an "aload_0" instruction is inserted after the "1 invokespecial #1" instruction, as the "invokespecial #1" instruction must execute before the object may be further used. This inserted "aload_0" instruction causes the JAVA virtual machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame, and results in the object reference to the 'this' object at index 0 being pushed onto the stack.

Furthermore, the JAVA virtual machine instruction "invokestatic #3 <Method Boolean isAlreadyLoaded(java.lang.Object)>" is inserted after the "4 aload_0" instruction so that the JAVA virtual machine pops the topmost item off the stack of the current method frame (which in accordance with the preceding "aload_0" instruction is a reference to the object to which this <init> method belongs) and invokes the "isAlreadyLoaded" method, passing the popped item to the new method frame as its first argument, and returning a boolean value onto the stack upon return from this "invokestatic" instruction. This change is significant because it modifies the <init> method to execute the "isAlreadyLoaded" method and associated operations, corresponding to the start of execution of the <init> method, and returns a boolean argument (indicating whether the object corresponding to this <init> method is initialized on another machine amongst the plurality of machines M1...Mn) onto the stack of the executing method frame of the <init> method.

Next, two JAVA virtual machine instructions "ifeq 13" and "return" are inserted into the code stream after the "5 invokestatic #2" instruction and before the "12 aload_0" instruction. The first of these two instructions, the "ifeq 13" instruction, causes the JAVA virtual machine to pop the topmost item off the stack and performs a comparison between the popped value and zero. If the performed comparison succeeds (i.e. if and only if the popped value is equal to zero), then execution continues at the "12 aload_0" instruction. If however the performed comparison fails (i.e. if and only if the popped value is not equal to zero), then execution continues at the next instruction in the code stream, which is the "11 return" instruction. This change is particularly significant because it modifies the <init> method to either continue execution of the <init> method (i.e. instructions 12-19) if the returned value of the "isAlreadyLoaded" method was negative (i.e. "false"), or discontinue execution of the <init> method (i.e. the "11 return" instruction causing a return of control to the invoker of this <init> method) if the returned value of the "isAlreadyLoaded" method was positive (i.e. "true").

The method void isAlreadyLoaded(java.lang.Object), part of the InitClient code of Annexure B7, and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1...Mn to coordinate the execution of the <init> method amongst the machines M1...Mn. The isAlreadyLoaded method of this example communicates with the InitServer code of Annexure B8 executing on a machine X of FIG. 15, by means of sending an "initialization status request" to machine X corresponding to the object being "initialized" (i.e. the object to which this <clinit> method belongs). With reference to FIG. 19 and Annexure B8, machine X receives the "initialization status request" corresponding to the object to which the <clinit> method belongs, and consults a table of initialization states or records to determine the initialization state for the object to which the request corresponds.

If the object corresponding to the initialization status request is not initialized on another machine other than the requesting machine, then machine X will send a response indicating that the object was not already initialized, and update a record entry corresponding to the specified object to indicate the object is now initialized. Alternatively, if the object corresponding to the initialization status request is initialized on another machine other than the requesting machine, then machine X will send a response indicating that the object is already initialized. Corresponding to the determination that the object to which this initialization status request pertains is not initialized on another machine other than the requesting machine, a reply is generated and sent to the requesting machine indicating that the object is not initialized. Additionally, machine X preferably updates the entry corresponding to the object to which the initialization status request pertained to indicate the object is now initialized. Following a receipt of such a message from machine X indicating that the object is not initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a 'false' value to the previous method frame, which is the executing method frame of the <init> method. Alternatively, following a receipt of a message from machine X indicating that the object is already initialized on another machine, the isAlreadyLoaded() method and operations terminate execution and return a "true" value to the previous method frame, which is the executing method frame of the <init> method. Following this return operation, the execution of the <init> method frame then resumes as indicated in the code sequence of Annexure B5 at step 006.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of initialization routines or other initialization operations so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as for example multiple initialization, or re-initialization operation) does not occur when applying the modified code or procedure.

Annexure B1 is a before-modification excerpt of the disassembled compiled form of the <clinit> method of the example.java application of Annexure B9. Annexure B2 is an after-modification form of Annexure B1, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 20. Annexure B3 is a before-modification excerpt of the disassembled compiled form of the <init> method of the example.java application of Annexure B9. Annexure B4 is an after-modification form of Annexure B3, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 21. Annexure B5 is an alternative after-modification form of Annexure B1, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 20. And Annexure B6 is an alternative after-modification form of Annexure B3, modified by InitLoader.java of Annexure B10 in accordance with the steps of FIG. 21. The modifications are highlighted in **bold.**

Table X. Annexure B1

B1
Method <clinit>
0 new #2 <Class example>
3 dup
4 invokespecial #3 <Method example()>
7 putstatic #4 <Field example currentExample>
10 return

Table XI. Annexure B2

B2
Method <clinit>
0 invokestatic #3 <Method boolean isAlreadyLoaded()>
3 ifeq 7
6 return
7 new #5 <Class example>
10 dup
11 invokespecial #6 <Method example()>
14 putstatic #7 <Field example example>
17 return

Table XII. Annexure B3

B3
Method <init>
0 aload_0

```
 1 invokespecial #1 <Method java.lang.Object()>
 4 aload_0
 5 invokestatic #2 <Method long currentTimeMillis()>
 8 putfield #3 <Field long timestamp>
11 return
```

Table XIII. Annexure B4

```
B4
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 invokestatic #2 <Method boolean isAlreadyLoaded()>
  7 ifeq 11
 10 return
 11 aload_0
 12 invokestatic #4 <Method long currentTimeMillis()>
 15 putfield #5 <Field long timestamp>
 18 return
```

Table XIV. Annexure B5

```
B5
Method <clinit>
  0 ldc #2 <String "example">
  2 invokestatic #3 <Method boolean isAlreadyLoaded(java.lang.String)>
  5 ifeq 9
  8 return
  9 new #5 <Class example>
 12 dup
 13 invokespecial #6 <Method example()>
 16 putstatic #7 <Field example currentExample>
 19 return
```

Table XV. Annexure B6

```
B6
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 aload_0
  5 invokestatic #2 <Method boolean isAlreadyLoaded(java.lang.Object)>
  8 ifeq 12
 11 return
 12 aload_0
 13 invokestatic #4 <Method long currentTimeMillis()>
 16 putfield #5 <Field long timestamp>
 19 return
```

Turning now to FIGS. 20 and 21, the procedure followed to modify class initialisation routines (i.e., the "<clinit>"method) and object initialization routines (i.e. the "<init>" method) is presented. The procedure followed to modify a <clinit> method relating to classes so as to convert from the code fragment of Annexure B1 (See Table X) to the code fragment of Annexure B5 (See Table XIV) is indicated.

Similarly, the procedure followed to modify an object initialization <init> method relating to objects so as to convert from the code fragment of Annexure B3 (See Table XII) to the code fragment of Annexure B6 (See Table XV) is indicated.

The initial loading of the application code 50 (an illustrative example in source-code form of which is displayed in Annexure B9, and a corresponding partially disassembled form of which is displayed in Annexure B1 (See also Table X) and Annexure B3 (See also Table XII)) onto the JAVA virtual machine 72 is commenced at step 201, and the code is analysed or scrutinized in order to detect one or more class initialization instructions, code-blocks or methods (i.e. "<clinit>" methods) by carrying out step 202, and/or one or more object initialization instructions, code-blocks, or methods (i.e. "<init>" methods) by carrying out step 212. Once so detected, an <clinit> method is modified by carrying out step 203, and an <init> method is modified by carrying out step 213. One example illustration for a modified class initialisation routine is indicated in Annexure B2 (See also Table XI), and a further illustration of which is indicated in Annexure B5 (See also Table XIV). One example illustration for a modified object initialisation routine is indicated in Annexure B4 (See also Table XIII), and a further illustration of which is indicated in Annexure B6 (See also Table XV). As indicated by step 204 and 214, after the modification is completed the loading procedure is then continued such that the modified application code is loaded into or onto each of the machines instead of the unmodified application code.

Annexure B1 (See also Table X) and Annexure B2 (See also Table XI) are the before (or pre-modification or unmodified code) and after (or post-modification or modified code) excerpt of a class initialisation routine (i.e. a "<clinit>" method) respectively. Additionally, a further example of an alternative modified <clinit> method is illustrated in Annexure B5 (See also Table XIV). The modified code that is added to the method is highlighted in bold. In the unmodified partially disassembled code sample of Annexure B1, the "new #2" and "invokespecial #3" instructions of the <clinit> method creates a new object (of the type 'example'), and the following instruction "putstatic #4" writes the reference of this newly created object to the memory location (field) called "currentExample". Thus, without management of coordinated class initialisation in a distributed environment of a plurality of machines M1, ..., Mn, and each with a memory updating and propagation means of FIG. 9, 10, 11, 12, and 13, whereby the application program code 50 is to operate as a single co-ordinated, consistent, and coherent instance across the plurality of machines M1...Mn, each computer or computing machine would re-initialise (and optionally alternatively re-write or over-write) the "currentExample" memory location (field) with multiple and different objects corresponding to the multiple executions of the <clinit> method, leading to potentially incoherent or inconsistent

memory between and amongst the occurrences of the application program code 50 on each of the machines M1, ..., Mn. Clearly this is not what the programmer or user of a single application program code 50 instance expects to happen.

So, taking advantage of the DRT, the application code 50 is modified as it is loaded into the machine by changing the class initialisation routine (i.e., the <clinit> method). The changes made (highlighted in bold) are the initial instructions that the modified <clinit> method executes. These added instructions determine the initialization status of this particular class by checking if a similar equivalent local class on another machine corresponding to this particular class, has already been initialized and optionally loaded, by calling a routine or procedure to determine the initialization status of the plurality of similar equivalent classes, such as the "is already loaded" (e.g., "isAlreadyLoaded()") procedure or method. The "isAlreadyLoaded()" method of InitClient of Annexure B7 of DRT 71 performing the steps of 172-176 of FIG. 17 determines the initialization status of the similar equivalent local classes each on a one of the machines M1, ..., Mn corresponding to the particular class being loaded, the result of which is either a true result or a false result corresponding to whether or not another one (or more) of the machines M1...Mn have already initialized, and optionally loaded, a similar equivalent class.

The initialisation determination procedure or method "isAlreadyLoaded()" of InitClient of Annexure B7 of the DRT 71 can optionally take an argument which represents a unique identifier for this class (See Annexure B5 and Table XIV). For example, the name of the class that is being considered for initialisation, a reference to the class or class-object representing this class being considered for initialization, or a unique number or identifier representing this class across all machines (that is, a unique identifier corresponding to the plurality of similar equivalent local classes each on a one of the plurality of machines M1...Mn), to be used in the determination of the initialisation status of the plurality of similar equivalent local classes on each of the machines M1...Mn. This way, the DRT can support the initialization of multiple classes at the same time without becoming confused as to which of the multiple classes are already loaded and which are not, by using the unique identifier of each class.

The DRT 71 can determine the initialization status of the class in a number of possible ways. Preferably, the requesting machine can ask each other requested machine in turn (such as by using a computer communications network to exchange query and response messages between the requesting machine and the requested machine(s)) if the requested machine's similar equivalent local class corresponding to the unique identifier is initialized, and if any requested machine replies true indicating that the similar equivalent local class has already been initialized, then return a true result at return from the isAlreadyLoaded() method indicating that the local class should not be initialized, otherwise return a false result at return from the isAlreadyLoaded() method indicating that the local class should be initialized. Of course different logic schemes for true or false results may alternatively be implemented with the same effect. Alternatively, the DRT on the local machine can consult a shared record table (perhaps on a separate machine (eg machine X), or a coherent shared record table on each local machine and updated to remain substantially identical, or in a database) to determine if one of the plurality of similar equivalent classes on other machines has been initialised.

If the isAlreadyLoaded() method of the DRT 71 returns false, then this means that this class (of the plurality of similar equivalent local classes on the plurality of machines M1...Mn) has not been initialized before on any other machine in the distributed computing environment of the plurality of machines M1...Mn, and hence, the execution of the class initialisation method is to take place or proceed as this is considered the first and original initialization of a class of the plurality of similar equivalent classes on each machine. As a result, when a shared record table of initialisation states exists, the DRT must update the initialisation status record corresponding to this class in the shared record table to true or other value indicating that this class is initialized, such that subsequent consultations of the shared record table of initialisation states (such as performed by all subsequent invocations of isAlreadyLoaded method) by all machines, and optionally including the current machine, will now return a true value indicating that this class is already initialized. Thus, if isAlreadyLoaded() returns false, the modified class initialisation routine resumes or continues (or otherwise optionally begins or starts) execution.

On the other hand, if the isAlreadyLoaded method of the DRT 71 returns true, then this means that this class (of the plurality of similar equivalent local classes each on one of the plurality of machines M1...Mn) has already been initialised in the distributed environment, as recorded in the shared record table on machine X of the initialisation states of classes. In such a case, the class initialisation method is not to be executed (or alternatively resumed, or continued, or started, or executed to completion), as it will potentially cause unwanted interactions or conflicts, such as re-initialization of memory, data structures or other machine resources or devices. Thus, when the DRT returns true, the inserted instructions at the start of the <clinit> method prevent execution of the initialization routine (optionally in whole or in part) by aborting the start or continued execution of the <clinit> method through the use of the return instruction, and consequently aborting the JAVA Virtual Machine's initialization operation for this class.

An equivalent procedure for the initialization routines of object (for example "<init>" methods) is illustrated in FIG. 21 where steps 212 and 213 are equivalent to steps 202 and 203 of FIG. 20. This results in the code of Annexure B3 being converted into the code of Annexure B4 (See also Table XIII) or Annexure B6 (See also Table XV).

Annexure B3 (See also Table XII) and Annexure B4 (See also Table XIV) are the before (or pre-modification or unmodified code) and after (or post-modification or modified code) excerpt of a object initialisation routine (i.e. a "<init>" method) respectively. Additionally, a further example of an alternative modified <init> method is illustrated in Annexure B6 (See also Table XV). The modified code that is added to the method is highlighted in bold. In the unmodified partially disassembled code sample of Annexure B4, the "aload_0" and "invokespecial #3" instructions of the <init> method invokes the <init> of the java.lang.Object superclass. Next, the following instructions "aload_0" loads a reference to the 'this' object onto the stack to be one of the arguments to the "8 putfield #3" instruction. Next, the following instruction "invokestatic #2" invokes the method java.lang.System.currentTimeMillis() and returns a long value on the stack. Next the following instruction "putfield #3" writes the long value placed on the stack be the preceding "invokestatic #2" instruction to the memory location (field) called

"timestamp" corresponding to the object instance loaded on the stack by the "4 aload_0" instruction. Thus, without management of coordinated object initialisation in a distributed environment of a plurality of machines M1, ..., Mn, and each with a memory updating and propagation means of FIG. 9, 10, 11, 12, and 13, whereby the application program code 50 is to operate as a single co-ordinated, consistent, and coherent instance across the plurality of machines M1...Mn, each computer or computing machine would re-initialise (and optionally alternatively re-write or over-write) the "timestamp" memory location (field) with multiple and different values corresponding to the multiple executions of the <init> method, leading to potentially incoherent or inconsistent memory between and amongst the occurrences of application program code 50 on each of the machines M1, ..., Mn. Clearly this is not what the programmer or user of a single application program code 50 instance expects to happen.

So, taking advantage of the DRT, the application code 50 is modified as it is loaded into the machine by changing the object initialisation routine (i.e. the <init> method). The changes made (highlighted in bold) are the initial instructions that the modified <init> method executes. These added instructions determine the initialisation status of this particular object by checking if a similar equivalent local object on another machine corresponding to this particular object, has already been initialized and optionally loaded, by calling a routine or procedure to determine the initialisation status of the object to be initialised, such as the "is already loaded" (e.g., "isAlreadyLoaded()") procedure or method of Annexure B7. The "isAlreadyLoaded()" method of DRT 71 performing the steps of 172-176 of FIG. 17 determines the initialization status of the similar equivalent local objects each on a one of the machines M1, ..., Mn corresponding to the particular object being loaded, the result of which is either a true result or a false result corresponding to whether or not another one (or more) of the machines M1...Mn have already initialized, and optionally loaded, this object.

The initialisation determination procedure or method "isAlreadyLoaded()" of the DRT 71 can optionally take an argument which represents a unique identifier for this object (See Annexure B6 and Table XV). For example, the name of the object that is being considered for initialisation, a reference to the object being considered for initialization, or a unique number or identifier representing this object across all machines (that is, a unique identifier corresponding to the plurality of similar equivalent local objects each on a one of the plurality of machines M1...Mn), to be used in the determination of the initialisation status of this object in the plurality of similar equivalent local objects on each of the machines M1...Mn. This way, the DRT can support the initialization of multiple objects at the same time without becoming confused as to which of the multiple objects are already loaded and which are not, by using the unique identifier of each object.

The DRT 71 can determine the initialization status of the object in a number of possible ways. Preferably, the requesting machine can ask each other requested machine in turn (such as by using a computer communications network to exchange query and response messages between the requesting machine and the requested machine(s)) if the requested machine's similar equivalent local object corresponding to the unique identifier is initialized, and if any requested machine replies true indicating that the similar equivalent local object has already been initialized, then return a true result at return from

the isAlreadyLoaded() method indicating that the local object should not be initialized, otherwise return a false result at return from the isAlreadyLoaded() method indicating that the local object should be initialized. Of course different logic schemes for true or false results may alternatively be implemented with the same effect. Alternatively, the DRT on the local machine can consult a shared record table (perhaps on a separate machine (eg machine X), or a coherent shared record table on each local machine and updated to remain substantially identical, or in a database) to determine if this particular object (or any one of the plurality of similar equivalent objects on other machines) has been initialised by one of the requested machines.

If the isAlreadyLoaded() method of the DRT 71 returns false, then this means that this object (of the plurality of similar equivalent local objects on the plurality of machines M1...Mn) has not been initialized before on any other machine in the distributed computing environment of the plurality of machines M1...Mn, and hence, the execution of the object initialisation method is to take place or proceed as this is considered the first and original initialization. As a result, when a shared record table of initialisation states exists, the DRT must update the initialisation status record corresponding to this object in the shared record table to true or other value indicating that this object is initialized, such that subsequent consultations of the shared record table of initialisation states (such as performed by all subsequent invocations of isAlreadyLoaded method) by all machines, and including the current machine, will now return a true value indicating that this object is already initialized. Thus, if isAlreadyLoaded() returns false, the modified object initialisation routine resumes or continues (or otherwise optionally begins or starts) execution..

On the other hand, if the isAlreadyLoaded method of the DRT 71 returns true, then this means that this object (of the plurality of similar equivalent local objects each on one of the plurality of machines M1...Mn) has already been initialised in the distributed environment, as recorded in the shared record table on machine X of the initialisation states of objects. In such a case, the object initialisation method is not to be executed (or alternatively resumed, or continued, or started, or executed to completion), as it will potentially cause unwanted interactions or conflicts, such as re-initialization of memory, data structures or other machine resources or devices. Thus, when the DRT returns true, the inserted instructions near the start of the <init> method prevent execution of the initialization routine (optionally in whole or in part) by aborting the start or continued execution of the <init> method through the use of the return instruction, and consequently aborting the JAVA Virtual Machine's initialization operation for this object.

A similar modification as used for <clinit> is used for <init>. The application program's <init> method (or methods, as there may be multiple) is or are detected as shown by step 212 and modified as shown by step 213 to behave coherently across the distributed environment.

The disassembled instruction sequence after modification has taken place is set out in Annexure B4 (and an alternative similar arrangement is provided in Annexure B6) and the modified/inserted instructions are highlighted in bold. For the <init> modification, unlike the <clinit> modification, the modifying instructions are often required to be placed after the "invokespecial" instruction, instead of at the very

beginning. The reasons for this are driven by the JAVA Virtual Machine specification. Other languages often have similar subtle design nuances.

Given the fundamental concept of testing to determine if initialization has already been carried out on a one of a plurality of similar equivalent classes or object or other asset each on a one of the machines M1...Mn, and if not carrying out the initialization, and if so, not carrying out the initialization; there are several different ways or embodiments in which this coordinated and coherent initialization concept, method, and procedure may be carried out or implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (such as class or object) inclusive of an initialisation routine, modifies it, and then loads each of the other machines M1, M3, ..., Mn (either sequentially or simultaneously or according to any other order, routine or procedure) with the modified object (or class or other asset or resource) inclusive of the new modified initialization routine(s). Note that there may be one or a plurality of routines corresponding to only one object in the application code, or there may be a plurality of routines corresponding to a plurality of objects in the application code. Note that in one embodiment, the initialization routine(s) that is (are) loaded is binary executable object code. Alternatively, the initialization routine(s) that is (are) loaded is executable intermediary code.

In this arrangement, which may be termed "master/slave" each of the slave (or secondary) machines M1, M3, ..., Mn loads the modified object (or class), and inclusive of the new modified initialisation routine(s), that was sent to it over the computer communications network or other communications link or path by the master (or primary) machine, such as machine M2, or some other machine such as a machine X of FIG. 15. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer communications network can be replaced by a shared storage device such as a shared file system, or a shared document/file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that in accordance with the inventive principles described herein, each of the plurality of machines behaves consistently and coherently relative to the other machines to accomplish the operations and objectives described herein. Furthermore, it will be appreciated in light of the description provided herein that there are a myriad of ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that embodiments of the invention may be implemented within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software, in firmware, or in any combination of these.

In a further variation of this "master/slave" or "primary/secondary" arrangement, machine M2 loads asset (such as class or object) inclusive of an (or even one or more) initialization routine in unmodified form on machine M2, and then (for example, machine M2 or each local machine) modifies the class (or object or asset) by deleting the initialization routine in whole or part from the asset (or class or object) and loads by means of a computer communications network or other communications link or path the

modified code for the asset with the now modified or deleted initialization routine on the other machines. Thus in this instance the modification is not a transformation, instrumentation, translation or compilation of the asset initialization routine but a deletion of the initialization routine on all machines except one.

The process of deleting the initialization routine in its entirety can either be performed by the "master" machine (such as machine M2 or some other machine such as machine X of FIG. 15) or alternatively by each other machine M1, M3, ..., Mn upon receipt of the unmodified asset. An additional variation of this "master/slave" or "primary/secondary" arrangement is to use a shared storage device such as a shared file system, or a shared document/file repository such as a shared database as means of exchanging the code (including for example, the modified code) for the asset, class or object between machines M1, M2,...,Mn and optionally a machine X of FIG. 15.

In a still further embodiment, each machine M1, ..., Mn receives the unmodified asset (such as class or object) inclusive of one or more initialization routines, but modifies the routines and then loads the asset (such as class or object) consisting of the now modified routines. Although one machine, such as the master or primary machine may customize or perform a different modification to the initialization routine sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different and to be enhanced, customized, and/or optimized based upon its particular machine architecture, hardware, processor, memory, configuration, operating system, or other factors, yet still similar, coherent and consistent with other machines with all other similar modifications and characteristics that may not need to be similar or identical.

In a further arrangement, a particular machine, say M1, loads the unmodified asset (such as class or object) inclusive of one or more initialisation routine and all other machines M2, M3, ..., Mn perform a modification to delete the initialization routine of the asset (such as class or object) and load the modified version.

In all of the described instances or embodiments, the supply or the communication of the asset code (such as class code or object code) to the machines M1, ..., Mn, and optionally inclusive of a machine X of FIG. 15, can be branched, distributed or communicated among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4, etc. directly), or by providing or using cascaded or sequential communication (for example, M2 supplies M1 which then supplies M3 which then supplies M4, and so on), or a combination of the direct and cascaded and/or sequential.

In a still further arrangement, the initial machine, say M2, can carry out the initial loading of the application code 50, modify it in accordance with this invention, and then generate a class/object loaded and initialised table which lists all or at least all the pertinent classes and/or objects loaded and initialised by machine M2. This table is then sent or communicated (or at least its contents are sent or communicated) to all other machines (including for example in branched or cascade fashion). Then if a machine, other than M2, needs to load and therefore initialise a class listed in the table, it sends a

request to M2 to provide the necessary information, optionally consisting of either the unmodified application code 50 of the class or object to be loaded, or the modified application code of the class or object to be loaded, and optionally a copy of the previously initialised (or optionally and if available, the latest or even the current) values or contents of the previously loaded and initialised class or object on machine M2. An alternative arrangement of this mode may be to send the request for necessary information not to machine M2, but some other, or even more than one of, machine M1, ..., Mn or machine X. Thus the information provided to machine Mn is, in general, different from the initial state loaded and initialise by machine M2.

Under the above circumstances it is preferable and advantageous for each entry in the table to be accompanied by a counter which is incremented on each occasion that a class or object is loaded and initialised on one of the machines M1, ..., Mn. Thus, when data or other content is demanded, both the class or object contents and the count of the corresponding counter, and optionally in addition the modified or unmodified application code, are transferred in response to the demand. This "on demand" mode may somewhat increase the overhead of the execution of this invention for one or more machines M1, ..., Mn, but it also reduces the volume of traffic on the communications network which interconnects the computers and therefore provides an overall advantage.

In a still further arrangement, the machines M1 to Mn, may send some or all load requests to an additional machine X (see for example the embodiment of FIG. 15), which performs the modification to the application code 50 inclusive of an (and possibly a plurality of) initialisation routine(s) via any of the afore mentioned methods, and returns the modified application code inclusive of the now modified initialization routine(s) to each of the machines M1 to Mn, and these machines in turn load the modified application code inclusive of the modified routines locally. In this arrangement, machines M1 to Mn forward all load requests to machine X, which returns a modified application program code 50 inclusive of modified initialization routine(s) to each machine. The modifications performed by machine X can include any of the modifications covered under the scope of the present invention. This arrangement may of course be applied to some of the machines and other arrangements described herein before applied to other of the machines.

Persons skilled in the computing arts will be aware of various possible techniques that may be used in the modification of computer code, including but not limited to instrumentation, program transformation, translation, or compilation means.

One such technique is to make the modification(s) to the application code, without a preceding or consequential change of the language of the application code. Another such technique is to convert the original code (for example, JAVA language source-code) into an intermediate representation (or intermediate-code language, or pseudo code), such as JAVA byte code. Once this conversion takes place the modification is made to the byte code and then the conversion may be reversed. This gives the desired result of modified JAVA code.

A further possible technique is to convert the application program to machine code, either directly from source-code or via the abovementioned intermediate language

or through some other intermediate means. Then the machine code is modified before being loaded and executed. A still further such technique is to convert the original code to an intermediate representation, which is thus modified and subsequently converted into machine code.

The present invention encompasses all such modification routes and also a combination of two, three or even more, of such routes.

FINALIZATION

Turning again to FIG. 14, there is illustrated a schematic representation of a single prior art computer operated as a JAVA virtual machine. In this way, a machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) can operate in the particular language of the application program code 50, in this instance the JAVA language. That is, a JAVA virtual machine 72 is able to operate application code 50 in the JAVA language, and utilize the JAVA architecture irrespective of the machine manufacturer and the internal details of the machine.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform, and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated in light of the description provided herein that the platform and/or runtime system may include virtual machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

Furthermore, when there is only a single computer or machine 72, the single machine of FIG. 14 is able to easily keep track of whether the specific objects 50X, 50Y, and/or 50Z are, liable to be required by the application code 50 at a later point of execution of the application code 50. This may typically be done by maintaining a "handle count" or similar count or index for each object and/or class. This count may typically keep track of the number of places or times in the executing application code 50 where reference is made to a specific object (or class). For a handle count (or other count or index based) implementation that increments the handle count (or index) upward when a new reference to the object or class is created or assigned, and decrements the handle count (or index) downward when a reference to the object or class is destroyed or lost, when the object handle count for a specific object reaches zero, there is nowhere in the executing application code 50 which makes reference to the specific object (or class) for which the zero object handle count (or class handle count) pertains. For example, in the JAVA language and virtual machine environment, a "zero object handle count" correlates to the lack of the existence of any references (zero reference count) which point to the specific object. The object is then said to be "finalizable" or exist in a finalizable state. Object handle counts (and handle counters) may be maintained for each object in an

analogous manner so that finalizable or non-finalizable state of each particular or specific object may be known. Class handle counts (and class handle counters) may be maintained for each class in an analogous manner to that for objects so that finalizable or non-finalizable state of each particular or specific class may be known. Furthermore, asset handle counts or indexes and counters may be maintained for each asset in an analogous manner to that for classes and objects so that finalizable or non-finalizable state of each particular or specific asset may be known.

Once this finalizable state has been achieved for an object (or class), the object (or class) can be safely finalized. This finalization may typically include object (or class) deletion, removal, clean-up, reclamation, recycling, finalization or other memory freeing operation because the object (or class) is no longer needed.

Therefore, in light of the availability of these reference, pointer, handle count or other class and object type tracking means, the computer programmer (or other automated or nonautomated program generator or generation means) when writing a program such as the application code 50 using the JAVA language and architecture, need not write any specific code in order to provide for this class or object removal, clean up, deletion, reclamation, recycling, finalization or other memory freeing operation. As there is only a single JAVA virtual machine 72, the single JAVA virtual machine 72 can keep track of the class and object handle counts in a consistent, coherent and coordinated manner, and clean up (or carry out finalization) as necessary in an automated and unobtrusive fashion, and without unwanted behaviour for example erroneous, premature, supernumerary, or re-finalization operation such as may be caused by inconsistent and/or incoherent finalization states or handle counts. In analogous manner, a single generalized virtual machine or machine or runtime system can keep track of the class and object handle counts (or equivalent if the machine does not specifically use "object" and "class" designations) and clean up (or carry out finalization) as necessary in an automated and unobtrusive fashion.

The automated handle counting system described above is used to indicate when an object (or class) of an executing application program 50 is no longer needed and may be 'deleted' (or cleaned up, or finalized, or reclaimed, or recycled, or other otherwise freed). It is to be understood that when implemented in 'non-automated memory management' languages and architectures (such as for example 'non-garbage collected' programming languages such as C, C++, FORTRAN, COBOL, and machine-CODE languages such as x86, SPARC, PowerPC, or intermediate-code languages), the application program code 50 or programmer (or other automated or non-automated program generator or generation means) may be able to make the determination at what point a specific object (or class) is no longer needed, and consequently may be 'deleted' (or cleaned up, or finalized, or reclaimed, or recycled). Thus, 'deletion' in the context of this invention is to be understood to be inclusive of the deletion (or cleaning up, or finalization, or reclamation, or recycling, or freeing) of objects (or classes) on 'non-automated memory management' languages and architectures corresponding to deletion, finalization, clean up, recycling, or reclamation operations on those 'non-automated memory management' languages and architectures.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method, and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the PowerPC computer architecture manufactured by International Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, the terms 'class' and 'object' may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and Boolean data types), structured data types (such as arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, references and unions.

However, in the arrangement illustrated in FIG. 8, (and also in FIGS. 31-33), a plurality of individual computers or machines M1, M2 .... Mn are provided, each of which are interconnected via a communications network 53 or other communications link and each of which individual computers or machines is provided with a modifier 51 (See FIG. 5) and realised or implemented by or in for example the distributed run-time system (DRT) 71 (See FIG. 8) and loaded with a common application code 50. The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on the plurality of computers or machines M1, M2...Mn. Put somewhat differently, there is a common application program represented in application code 50, and this single copy or perhaps a plurality of identical copies are modified to generate a modified copy or version of the application program or program code, each copy or instance prepared for execution on the plurality of machines. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing the features of the invention may optionally be connected to or coupled with other computers, machines, information appliances, or the like that do not implement the features of the invention.

In some embodiments, some or all of the plurality of individual computers or machines may be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

Essentially the modifier 51 or DRT 71, or other code modifying means is responsible for modifying the application code 50 so that it may execute clean up or other

memory reclamation, recycling, deletion or finalization operations, such as for example finalization methods in the JAVA language and virtual machine environment, in a coordinated, coherent and consistent manner across and between the plurality of individual machines M1, M2, ..., Mn. It follows therefore that in such a computing environment it is necessary to ensure that the local objects and classes on each of the individual machines is finalized in a consistent fashion (with respect to the others).

It will be appreciated in light of the description provided herein that there are alternative implementations of the modifier 51 and the distributed run time 71. For example, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71. In one embodiment, the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. The modifier function and structure therefore maybe subsumed into the DRT and considered to be an optional component. Independent of how implemented, the modifier function and structure is responsible for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the DRT on each machine. The DRT may for example implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures within the invention are less important than that they are provided.

In particular, whilst the application program code executing on one particular machine (say, for example machine M3) may have no active handle, reference, or pointer to a specific local object or class (i.e. a "zero handle count"), the same application program code executing on another machine (say for example machine M5) may have an active handle, reference, or pointer to the local similar equivalent object or class corresponding to the 'un-referenced' local object or class of machine M3, and therefore this other machine (machine M5) may still need to refer to or use that object or class in future. Thus if the corresponding similar equivalent local object or class on each machine M3 and M5 were to be finalized (or otherwise cleaned-up by some other memory clean-up operation) in an independent and uncoordinated manner relative to other machine(s), the behaviour of the object and application as a whole is undefined – that is, in the absence of coordinated, coherent, and consistent finalization or memory clean-up operations between machines M1...Mn, conflict, unwanted interactions, or other anomalous behaviour such as permanent inconsistency between local similar equivalent corresponding objects on machine M5 and machine M3 is likely to result. For example, if the local similar equivalent object or class on machine M3 were to be finalized, such as by being deleted, or cleaned up, or reclaimed, or recycled, from machine M3, in an uncoordinated and inconsistent manner with respect to machine M5, then if machine M5 were to perform an operation on or otherwise use the local object or class corresponding

to the now finalized similar equivalent local object on machine M3 (such operation being for example, in an environment with a memory updating and propagation means of FIGS. 9, 10, 11, 12, and 13, a write to (or try to write to) the similar equivalent local object on machine M5 or amendment to that particular object's value), then that operation (the change or attempted change in value) could not be performed (propagated from machine M5) throughout all the other machines M1, M2...Mn since at least the machine M3 would not include the relevant similar equivalent corresponding particular object in its local memory, the object and its data, contents and value(s) having been deleted by the prior object clean-up or finalization or reclamation or recycling operation. Therefore, even though one may contemplate machine M5 being able to write to the object (or class) the fact that it has already been finalized on machine M3 means that likely such a write operation is not possible, or at the very least not possible on machine M3.

Additionally, if an object of class on machine M3 were to be marked finalizable and subsequently finalized (such as by being deleted, or cleaned up, or reclaimed, or recycled) whilst the same object on the other machines M1, M2...Mn were not also marked as finalizable, then the execution of the finalization (or deletion, or clean up, or reclamation, or recycling) operation of that object on machine M3 would be premature with respect to coordinated finalization operation between all machines M1, M2...Mn, as machines other than M3 are not yet ready to finalize their local similar equivalent object corresponding to the particular object now finalized or finalizable by machine M3. Therefore were machine M3 to execute the cleanup or other finalization routine on a given particular object (or class), the cleanup or other finalization routine would preform the clean-up or finalization not just for that local object (or class) on machine M3, but also for all similar equivalent local objects or classes (i.e. corresponding to the particular object or class to be cleaned-up or otherwise finalized) on all other machines as well.

Were such either these circumstance to happen, the behaviour of the equivalent object on the other machines M1, M2...Mn is undefined and likely to result in permanent and irrecoverable inconsistency between machine M3 and machines M1, M2...Mn. Therefore, though machine M3 may independently determine an object (or class) is ready for finalization and proceed to finalize the specified object (or class), machine M5 may not have made the same determination as to the same similar equivalent local object (or class) being ready to be finalized, and therefore inconsistent behaviour will likely result due to the deletion of one of the plurality of similar equivalent objects on one machine (eg, machine M3) but not on the other machine (eg, machine M5) or machines, and the premature execution of the finalization routine of the specified object (or class) by machine M3 and on behalf of all other machines M1,M2...Mn. At the very least operation of machine M5 as well as other machines in such as an above circumstance is unpredictable and would likely lead to inconsistent results, such inconsistency potentially arising for example from, uncoordinated premature execution of the finalization routine and/or deletion of the object on one, or a subset of, machines but not others. Thus, the desirable result of achieving or providing consistent coordinated finalization operation (or other memory clean-up operation) as required for the simultaneous operation of the same application program code on each of the plurality of machines M1, M2...Mn would not be achieved. Any attempt therefore to maintain identical memory contents with a memory updating and propagation means of FIGS. 9, 10, 11, 12, and 13, or even identical memory contents as to a particular or defined set of classes, objects, values, or other data, for each

of the machines M1, M2, ..., Mn, as required for simultaneous operation of the same application program, would not be achieved given conventional schemes.

In order to ensure consistent class and object (or equivalent) finalizable status and finalization or clean up between and amongst machines M1, M2, ..., Mn, the application code 50 is analysed or scrutinized by searching through the executable application code 50 in order to detect program steps (such as particular instructions or instruction types) in the application code 50 which define or constitute or otherwise represent a finalization operation or routine (or other memory, data, or code clean up routine, or other similar reclamation, recycling, or deletion operation). In the JAVA language, such program steps may for example comprise or consist of some part of, or all of, a "finalize()" method of an object, and optionally any other code, routine, or method related to a 'finalize()' method, for example by means of a method invocation from the body of the 'finalize()' method to a different method.

This analysis or scrutiny may take place either prior to loading the application program, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application program may be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as from source-code or intermediate-code language to machine language), and with the understanding that the term compilation normally involves a change in code or language, for example, from source to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object-code), and compilation from source-code to source-code, as well as compilation from object-code to object-code, and any altered combinations therein. It is also inclusive of so-called "intermediary languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the application program code such as by the operating system reading the application code from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader loadClass method (e.g., "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the application program code has started, such as for example after the JAVA virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

As a consequence, of the above described analysis or scrutiny, clean up routines are initially looked for, and when found or identified a modifying code is inserted so as to give rise to a modified clean up routine. This modified routine is adapted and written to abort the clean up routine on any specific machine unless the class or object (or in the more general case to be 'asset') to be deleted, cleaned up, reclaimed, recycled, freed, or otherwise finalized is marked for deletion by all other machines. There are several different alternative modes wherein this modification and loading can be carried out.

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the application program code such as by the operating system reading the application code from the hard disk or other storage device and copying it into memory whilst preparing to begin execution of the application program. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the execution of the java.lang.ClassLoader loadClass (e.g., "java.lang.ClassLoader.loadClass()") method.

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure such as after the operating system has loaded the application code into memory and even started execution, or after the java virtual machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method. In other words, in the case of the JAVA virtual machine, after the execution of "java.lang.ClassLoader.loadclass()" has concluded.

Thus, in one mode, the DRT 71/1 on the loading machine, in this example Java Machine M1 (JVM#1), asks the DRT's 71/2, ..., 71/n of all the other machines M2, ..., Mn if the similar equivalent first object 50X on all machines, say, is utilized, referenced, or in-use (i.e. not marked as finalizable) by any other machine M2, ..., Mn. If the answer to this question is yes (that is, a similar equivalent object is being utilized by another one or more of the machines, and is not marked as finalizable and therefore not liable to be deleted, cleaned up, finalized, reclaimed, recycled, or freed), then the ordinary clean up procedure is turned off, aborted, paused, or otherwise disabled for the similar equivalent first object 50X on machine JVM#1. If the answer is no, (that is the similar equivalent first object 50X on each machine is marked as finalizable on all other machines with a similar equivalent object 50X) then the clean up procedure is operated (or resumed or continued, or commenced) and the first object 50X is deleted not only on machine JVM#1 but on all other machines M2...Mn with a similar equivalent object 50X. Preferably, execution of the clean up routine is allocated to one machine, such as the last machine M1 marking the similar equivalent object or class as finalizable. The execution of the finalization routine corresponding to the determination by all machines that the plurality of similar equivalent objects is finalizable, is to execute only once with respect to all machines M1...Mn, and preferably by only one machine, on behalf of all machines M1....Mn. Corresponding to, and preferably following, the execution of the finalization routine, all machines may then delete, reclaim, recycle, free or otherwise clean-up the memory (and other corresponding system resources) utilized by their local similar equivalent object.

Annexures C1, C2, C3, and C4 (also reproduced in part in Tables XVI, XVII, XVIII, and XIX below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of a finalization routine of application program 50 (Annexure C1 and Table XVI), and a post-modification excerpt of the same synchronization routine such as may be used in embodiments of the present invention having multiple machines (Annexures C2 and C3 and Tables XVII and XVIII). Also the modified code that is added to the finalization routine is highlighted in **bold** text.

Annexure C1 is a before-modification excerpt of the disassembled compiled form of the finalize() method of the example.java application of Annexure C4. Annexure C2 is an after-modification form of Annexure C1, modified by FinalLoader.java of Annexure C7 in accordance with the steps of FIG. 22. Annexure C3 is an alternative after-modification form of Annexure C1, modified by FinalLoader.java of Annexure C7 in accordance with the steps of FIG. 22. The modifications are highlighted in **bold.**

Annexure C4 is an excerpt of the source-code of the example.java application used in before/after modification excerpts C1-C3. This example application has a single finalization routine, the finalize() method, which is modified in accordance with this invention by FinalLoader.java of Annexure C7.

Table XVI. Annexure C1- Typical prior art finalization for a single machine

```
Method finalize()
0 getstatic #9 <Field java.io.PrintStream out>
3 ldc #24 <String "Deleted...">
5 invokevirtual #16 <Method void println(java.lang.String)>
8 return
```

Table XVII. Annexure C2 - Finalization For Multiple Machines

```
Method finalize()
0 aload_0
1 invokestatic #3 <Method boolean isLastReference(java.lang.Object)>
4 ifne 8
7 return
8 getstatic #9 <Field java.io.PrintStream out>
11 ldc #24 <String "Deleted...">
13 invokevirtual #16 <Method void println(java.lang.String)>
16 return
```

Table XVIII. Annexure C3 - Finalization For Multiple Machines (Alternative)

```
Method finalize()
0 aload_0
1 invokestatic #3 <Method boolean isLastReference(java.lang.Object)>
4 ifne 8
7 return
```

```
8 getstatic #9 <Field java.io.PrintStream out>
11 ldc #24 <String "Deleted...">
13 invokevirtual #16 <Method void println(java.lang.String)>
16 return
```

Table XIX. Annexure C4 - Source-code of the example.java application used in before/after modification excerpts of Annexures C1-C3

```
import java.lang.*;

public class example{

   /** Finalize method. */
   protected void finalize() throws Throwable{

      // "Deleted..." is printed out when this object is garbaged.
      System.out.println("Deleted...");

   }
}
```

It is noted that the compiled code in the annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure C4. In the procedure of Annexure C1 and Table XVI, the procedure name "Method finalize()" of Step 001 is the name of the displayed disassembled output of the finalize method of the compiled application code "example.java". The method name "finalize()" is the name of an object's finalization method in accordance with the JAVA platform specification, and selected for this example to indicate a typical mode of operation of a JAVA finalization method. Overall the method is responsible for disposing of system resources or to perform other cleanup corresponding to the determination by the garbage collector of a JAVA virtual machine that there are no more references to this object, and the steps the "example.java" code performs are described in turn.

First (Step 002), the JAVA virtual machine instruction "getstatic #9 <Field java.io.PrintStream out>" causes the JAVA virtual machine to retrieve the object reference of the static field indicated by the CONSTANT_Fieldref_info constant_pool item stored in the 2nd index of the classfile structure of the application program containing this example finalize() method and results on a reference to a java.io.PrintStream object in the field to be placed (pushed) on the stack of the current method frame of the currently executing thread.

Next (Step 003), the JAVA virtual machine instruction "ldc #24 <String "Deleted...">" causes the JAVA virtual machine to load the String value "Deleted" onto the stack of the current method frame and results in the String value "Deleted" loaded onto the top of the stack of the current method frame.

Next (Step 004), the JAVA virtual machine instruction "invokevirtual #16 <Method void println(java.lang.String)>" causes the JAVA virtual machine to pop the

topmost item off the stack of the current method frame and invoke the "println" method, passing the popped item to the new method frame as its first argument, and results in the "println" method being invoked.

Finally, the JAVA virtual machine instruction "return" (Step 005) causes the JAVA virtual machine to cease executing this finalize() method by returning control to the previous method frame and results in termination of execution of this finalize method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the JAVA virtual machine can keep track of the object handle count in a consistent, coherent and coordinated manner, and in executing the finalize() method containing the println operation is able to ensure that unwanted behaviour (for example premature or supernumerary finalization operation such as execution of the finalize() method of a single 'example.java' object more than once) such as may be caused by inconsistent and/or incoherent finalization states or handle counts, does not occur. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 with the memory update and propagation replication means of FIGS. 9, 10, 11, 12, and 13, and concurrently executing the application program code 50 on each one of the plurality of machines M1...Mn, the finalization operations of each concurrently executing application program occurrence on each of the one of the machines would be performed without coordination between any other of the occurrences on any other of the machine(s). Given the desirable result of consistent, coordinated and coherent finalization operation across a plurality of a machines, this prior art arrangement would fail to perform such consistent coordinated finalization operation across the plurality of machines, as each machine performs finalization only locally and without any attempt to coordinate their local finalization operation with any other similar finalization operation on any one or more other machines. Such an arrangement would therefore be susceptible to unwanted or other anomalous behaviour due to uncoordinated, inconsistent and/or incoherent finalization states or handle counts, and associated finalization operation. Therefore it is desirable to overcome this limitation of the prior art arrangement.

In the exemplary code in Table XVIII (Annexure C3), the code has been modified so that it solves the problem of consistent, coordinated finalization operation for a plurality of machines M1...Mn, that was not solved in the code example from Table XVI (Annexure C1). In this modified finalize() method code, an "aload_0" instruction is inserted before the "getstatic #9" instruction in order to be the first instruction of the finalize() method. This causes the JAVA virtual machine to load the item in the local variable array at index 0 of the current method frame and store this item on the top of the stack of the current method frame, and results in the object reference of the 'this' object at index 0 being pushed onto the stack.

Furthermore, the JAVA virtual machine instruction "invokestatic #3 <Method boolean isLastReference(java.lang.Object)>" is inserted after the "0 aload_0" instruction so that the JAVA virtual machine pops the topmost item off the stack of the current method frame (which in accordance with the preceding "aload_0" instruction is a reference to the object to which this finalize() method belongs) and invokes the "isLastReference" method, passing the popped item to the new method frame as its first argument, and returning a boolean value onto the stack upon return from this

"invokestatic" instruction. This change is significant because it modifies the finalize() method to execute the "isLastReference" method and associated operations, corresponding to the start of execution of the finalize() method, and returns a boolean argument (indicating whether the object corresponding to this finalize() method is the last remaining reference amongst the similar equivalent object on each of the machines M1...Mn) onto the stack of the executing method frame of the finalize() method.

Next, two JAVA virtual machine instructions "ifne 8" and "return" are inserted into the code stream after the "1 invokestatic #3" instruction and before the "getstatic #9" instruction. The first of these two instructions, the "ifne 8" instruction, causes the JAVA virtual machine to pop the topmost item off the stack and performs a comparison between the popped value and zero. If the performed comparison succeeds (i.e. if and only if the popped value is not equal to zero), then execution continues at the "8 getstatic #9" instruction. If however the performed comparison fails (i.e. if and only if the popped value is equal to zero), then execution continues at the next instruction in the code stream, which is the "7 return" instruction. This change is particularly significant because it modifies the finalize() method to either continue execution of the finalize() method (i.e. instructions 8-16) if the returned value of the "isLastReference" method was positive (i.e. "true"), or discontinue execution of the finalize() method (i.e. the "7 return" instruction causing a return of control to the invoker of this finalize() method) if the returned value of the "isLastReference" method was negative (i.e. "false").

The method void isLastReference(java.lang.Object), part of the FinalClient code of Annexure C5 and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1...Mn to coordinate the execution of the finalize() method amongst the machines M1...Mn. The isLastReference method of this example communicates with the InitServer code of Annexure C6 executing on a machine X of FIG. 15, by means of sending an "clean-up status request" to machine X corresponding to the object being "finalized" (i.e. the object to which this finalize() method belongs). With reference to FIG. 25 and Annexure C6, machine X receives the "clean-up status request" corresponding to the object to which the finalize() method belongs, and consults a table of clean-up counts or finalization states to determine the clean-up count or finalization state for the object to which the request corresponds.

If the plurality of similar equivalent objects one each one of the plurality of machines M1...Mn corresponding to the clean-up status request is marked for clean-up on all other machines than the requesting machine (i.e. n-1 machines), then machine X will send a response indicating that the plurality of similar equivalent objects are marked for clean-up on all other machines, and optionally update a record entry corresponding to the specified similar equivalent objects to indicate the similar equivalent objects as now cleaned up. Alternatively, if the plurality of the similar equivalent objects corresponding to the clean-up status request is not marked for clean-up on all other machines than the requesting machine (i.e. less than n-1 machines), then machine X will send a response indicating that the plurality of similar equivalent objects is not marked for cleanup on all other machines, and increment the "marked for clean-up counter" record (or other similar finalization record means) corresponding to the specified object, to record that the requesting machine has marked its one of the plurality of similar equivalent objects to be cleaned-up. Corresponding to the determination that the plurality of similar equivalent

objects to which this clean-up status request pertains is marked for clean-up on all other machines than the requesting machine, a reply is generated and sent to the requesting machine indicating that the plurality of similar equivalent objects is marked for clean-up on all other machines than the requesting machine. Additionally, and optionally, machine X may update the entry corresponding to the object to which the clean-up status request pertained to indicate the plurality of similar equivalent objects as now "cleaned-up". Following a receipt of such a message from machine X indicating that the plurality of similar equivalent objects is marked for clean-up on all other machines, the isLastReference() method and operations terminate execution and return a 'true' value to the previous method frame, which is the executing method frame of the finalize() method. Alternatively, following a receipt of a message from machine X indicating that the plurality of similar equivalent objects is not marked for clean-up on all other machines, the isLastReference() method and operations terminate execution and return "false" value to the previous method frame, which is the executing method frame of the finalize() method. Following this return operation, the execution of the finalize() method frame then resumes as indicated in the code sequence of Annexure C3.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of finalization routines or other clean-up operations so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as for example erroneous, premature, multiple finalization, or re-finalization operation) does not occur when applying the modified code or procedure.

It may be observed that the code in Annexure C2 and Table XVII is an alternative but lesser preferred form of the code in Annexure C3. It is essentially functionally equivalent to the code and approach in Annexure C3.

As seen in FIG. 15 a modification to the general arrangement of FIG. 8 is provided in that machines M1, M2, ..., Mn are as before and run the same application code 50 (or codes) on all machines M1, M2, ..., Mn simultaneously or concurrently. However, the previous arrangement is modified by the provision of a server machine X which is conveniently able to supply housekeeping functions, for example, and especially the clean up of structures, assets and resources. Such a server machine X can be a low value commodity computer such as a PC since its computational load is low. As indicated by broken lines in FIG. 15, two server machines X and X+1 can be provided for redundancy purposes to increase the overall reliability of the system. Where two such server machines X and X+1 are provided, they are preferably operated as redundant machines in a failover arrangement.

It is not necessary to provide a server machine X as its computational load can be distributed over machines M1, M2, ..., Mn. Alternatively, a database operated by one machine (in a master/slave type operation) can be used for the housekeeping function(s).

FIG. 16 shows a preferred general procedure to be followed. After loading 161 has been commenced, the instructions to be executed are considered in sequence and all clean up routines are detected as indicated in step 162. In the JAVA language these are the finalization routines or finalize method (e.g., "finalize( )"). Other languages use different terms.

Where a clean up routine is detected, it is modified at step 163 in order to perform consistent, coordinated, and coherent clean up or finalization across and between the plurality of machines M1, M2...Mn, typically by inserting further instructions into the clean up routine to, for example, determine if the object (or class or other asset) containing this finalization routine is marked as finalizable across all similar equivalent local objects on all other machines, and if so performing finalization by resuming the execution of the finalization routine, or if not then aborting the execution of the finalization routine, or postponing or pausing the execution of the finalization routine until such a time as all other machines have marked their similar equivalent local objects as finalizable. Alternatively, the modifying instructions could be inserted prior to the routine. Once the modification has been completed the loading procedure continues by loading modified application code in place of the unmodified application code, as indicated in step 164. Altogether, the finalization routine is to be executed only once, and preferably by only one machine, on behalf of all machines M1...Mn corresponding to the determination by all machines M1...Mn that the particular object is finalizable.

FIG. 17 illustrates a particular form of modification. Firstly, the structures, assets or resources (in JAVA termed classes or objects) 50A, 50X...50Y which are possible candidates to be cleaned up, are allocated a name or tag (for example a global name or tag), or have already been allocated a global name or tag, which can be used to identify corresponding similar equivalent local structures, assets, or resources (such as classes and objects in JAVA) globally on each of the machines M1, M2...Mn, as indicated by step 172. This preferably happens when the classes or objects are originally initialized. This is most conveniently done via a table maintained by server machine X. This table also includes the "clean up status" of the class or object (or other asset). It will be understood that this table or other data structure may store only the clean up status, or it may store other status or information as well. In one embodiment, this table also includes a counter which stores a machine asset deletion count value identifying the number of machines (and optionally the identity of the machines although this is not required) which have marked this particular object, class, or other asset for deletion. In one embodiment, the count value is incremented until the count value equals the number of machines. Thus a total machine asset deletion count value of less than (n-1), where n is the total number of machines in Mn indicates a "do not clean up" status for the object, class, or other asset as a network (or machine constellation) whole, because the machine asset deletion count of less than n-1 means that one or more machines have yet to mark their similar equivalent local object (or class or other asset) as finalizable and that object cannot be cleaned up as unwanted or other anomalous behaviour may result. Stated differently, and by way of example but not limitation, if there are six machines and the asset deletion count is less than five then it means that not all the other machines have attempted to finalize this object (i.e., not yet marked this object as finalizable), and therefore the object can't be finalised. If however the asset deletion count is five, then it means that there is only one machine that has yet to attempt to finalize this object (i.e., mark this object as finalizable) and therefore that last machine yet to mark the object as finalizable must be the current machine attempting to finalize the object (i.e., marking the object as finalizable and consequently consulting the finalization table as to finalization status of this object on all other machines). In the configuration of six machines, the count value of n-1=5 means that five machines must have previously marked the object for deletion and the sixth

machine to mark this object for deletion is the machine that actually executes the full finalization routine.

As indicated in FIG. 17, if the global name or identifier is not marked for cleanup or deletion or other finalization on all other machines (i.e., all except on the machine proposing to carry out the clean up or deletion routine) then this means that the proposed clean up or finalization routine of the object or class (or other asset) should be aborted, stopped, suspend, paused, postponed, or cancelled prior to its initiation or if already initiated then to its completion if it has already begun execution, since the object or class is still required by one or more of the machines M1, M2...Mn, as indicated by step 175.

In one embodiment, the clean up or finalization routine is stopped from initiating or beginning execution; however, if some implementations it is difficult or practically impossible to stop the clean up or finalization routine from initiating or beginning execution. Therefore, in an alternative embodiment, the execution of the finalization routine that has already started is aborted such that it does not complete or does not complete in its normal manner. This alternative abortion is understood to include an actual abortion, or a suspend, or postpone, or pause of the execution of a finalization routine that has started to execute (regardless of the stage of execution before completion) and therefore to make sure that the finalization routine does not get the chance to execute to completion to clean up the object (or class or other asset), and therefore the object (or class or other asset) remains "uncleaned" (i.e., "unfinalised", or "not deleted").

However or alternatively, if the global name or other unique number or identifier for a plurality of similar equivalent local objects each on of the plurality of machines M1, M2...Mn is marked for deletion on all other machines, this means that no other machine requires the class or object (or other asset) corresponding to the global name or other unique number or identifier. As a consequence clean up routine and operation, or optionally the regular or conventional ordinary clean up routine and operation, indicated in step 176 can be, and should be, carried out.

FIG. 18 shows the enquiry made by the machine proposing to execute a clean up routine (one of M1, M2...Mn) to the server machine X. The operation of this proposing machine is temporarily interrupted, as shown in step 181 and 182, and corresponding to step 173 of FIG. 17. In step 181 the proposing machine sends an enquiry message to machine X to request the clean-up or finalization status of the object (or class or other asset) to be cleaned-up. Next, the proposing machine awaits a reply from machine X corresponding to the enquiry message sent by the proposing machine at step 181, indicated by step 182.

FIG. 25 shows the activity carried out by machine X in response to such a finalization or clean up status enquiry of step 181 in FIG. 18. The finalization or clean up status is determined as seen in step 192 which determines if the object (or class or other asset) corresponding to the clean-up status request of global name, as received at step 191 (191A), is marked for deletion on all other machines other than the enquiring machine 181 from which the clean-up status request of step 191 originates. The singular term object or class as used in this document (or the equivalent term of asset, or resource used in step 192 (192A) and other Figures) are to be understood to be inclusive of all similar equivalent objects (or classes, or assets, or resources) corresponding to the same global

name on each of the plurality of machines M1, M2, ..., Mn. If the step 193 (193A) determination is made that determines that the global named resource is not marked ("No") for deletion on (n-1) machines (i.e. is utilized elsewhere), then a response to that effect is sent to the enquiring machine 194 (194A) but the "marked for deletion" counter is incremented by one (1), as shown by step 197 (197A). Similarly, if the answer to this determination is marked for deletion ("Yes") indicating that the global named resource is marked for deletion on all other machines other than the waiting enquiring machine 182 then a corresponding reply is sent to the waiting enquiring machine 182 from which the clean-up status request of step 191 originated as indicated by step 195 (195A). The waiting enquiring machine 182 is then able to respond accordingly, such as for example by: (i) aborting (or pausing, or postponing) execution of the finalization routine when the reply from machine X of step 182 indicated that the similar equivalent local objects on the plurality of machines M1, M2, ..., Mn corresponding to the global name of the object proposed to be finalized of step 172 is still utilized elsewhere (i.e., not marked for deletion on all other machines other than the machine proposing to carry out finalization); or (ii) by continuing (or resuming, or starting) execution of the finalization routine when the reply from machine X of step 182 indicated that the similar equivalent local objects on the plurality of machines M1, M2...Mn corresponding to the global name of the object proposed to be finalized of step 172 are not utilized elsewhere (i.e., marked for deletion on all other machines other than the machine proposing to carry out finalization). As indicated by broken lines in FIG. 25, preferably in addition to the "yes" response shown in step 195, the shared table or cleaned-up statuses stored or maintained on machine X is updated so that the status of the globally named asset is changed to "cleaned up" as indicated by step 196.

Reference is made to the accompanying Annexure C in which: Annexure C1 is a typical code fragment from an unmodified finalize routine, Annexure C2 is an equivalent in respect of a modified finalize routine, and Annexure C3 is an alternative equivalent in respect of a modified finalize routine.

Annexures C1 and C2/C3 repeated as Tables XVI and XVII/XVIII are the before (pre-modification or unmodified code) and after (or post-modification or modified code) excerpt of a finalization routine respectively. The modified code that is added to the method is highlighted in bold. In the original code sample of Annexure C1, the finalize method prints "Deleted..." to the computer console on event of finalization (i.e. deletion) of this object. Thus, without management of object finalization in a distributed environment, each machine would re-finalize the same object, thus executing the finalize method more than once for a single globally-named coherent plurality of similar equivalent objects. Clearly this is not what the programmer or user of a single application program code instance expects to happen.

So, taking advantage of the DRT, the application code 50 is modified as it is loaded into the machine by changing the clean-up, deletion, or finalization routine or method. It will be appreciated that the term finalization is typically used in the context of the JAVA language relative to the JAVA virtual machine specification existent at the date of filing of this specification. Therefore, finalization refers to object and/or class cleanup or deletion or reclamation or recycling or any equivalent form of object, class, asset or resource clean-up in the more general sense. The term finalization should therefore be

taken in this broader meaning unless otherwise restricted. The changes made (highlighted in **bold**) are the initial instructions that the finalize method executes. These added instructions check if this particular object is the last remaining object of the plurality of similar equivalent objects on the plurality of machines M1, M2...Mn to be marked as finalizable, by calling a routine or procedure to determine the clean-up status of the object to be finalized, such as the "isLastReference()" procedure or method of a DRT 71 performing the steps of 172-176 of FIG. 17 where the determination as to the clean-up status of the particular object is sought, and which determines either a true result or a false result corresponding to whether or not this particular object on this particular machine that is executing the determination procedure is the last of the plurality of machines M1, M2...Mn, each with one of a similar equivalent peer object, to request finalization. Recall that a peer object refers to a similar equivalent object on a different one of the machines, so that for example, in a configuration having eight machines, there will be eight peer objects (i.e. eight similar equivalent objects each on one of eight machines).

The finalization determination procedure or method "isLastReference()" of the DRT 71 can optionally take an argument which represents a unique identifier for this object (See Annexure C3 and Table XVIII). For example, the name of the object that is being considered for finalization, a reference to the object in question being considered for finalization, or a unique number or identifier representing this object across all machines (or nodes), to be used in the determination of the finalization status of this object or class or other asset. This way, the DRT can support the finalization of multiple objects (or classes or assets) at the same time without becoming confused as to which of the multiple objects are already finalized and which are not, by using the unique identifier of each object to consult the correct record in the finalization table referred to earlier.

The DRT 71 can determine the finalization state of the object in a number of possible ways. Preferably, it (the requesting machine) can ask each other requested machine in turn (such as by using a computer communications network to exchange query and response messages between the requesting machine and the requested machine(s) if their requested machine's similar equivalent object has been marked for finalization, and if any requested machine replies false indicating that their similar equivalent object is not marked for finalization, then return a false result at return from the "isLastReference()" method indicating that the local similar equivalent object should not be finalized, otherwise return a true result at return from the "isLastReference()" method indicating that the local similar equivalent object can be finalized. Of course different logic schemes for true or false result may alternatively be implemented with the same effect. Alternatively, the DRT 71 on the local machine can consult a shared record table (perhaps on a separate machine (e.g., machine X), or a coherent shared record table on each local machine and updated to remain substantially identical, or in a database) to determine if each of the plurality of similar equivalent objects have been marked for finalization by all requested machines except the current requesting machine.

If the "isLastReference()" method of the DRT 71 returns true then this means that this object has been marked for finalization on all other machines in the virtual or distributed computing environment (i.e. the plurality of machines M1...Mn), and hence, the execution of the finalize method is to proceed as this is considered the last remaining

similar equivalent object on the plurality of machines M1, M2...Mn to be marked or declared as finalizable.

On the other hand, if the "isLastReference()" method of the DRT 71 returns false, then this means that the plurality of similar equivalent objects has not been marked for finalization by all other machines in the distributed environment, as recorded in the shared record table on machine X of the finalization states of objects. In such a case, the finalize method is not to be executed (or alternatively resumed, or continued), as it will potentially invalidate the object on those machine(s) that are continuing to use their similar equivalent object and have yet to mark their similar equivalent object for finalization. Thus, when the DRT returns false, the inserted four instructions at the start of the finalize method prevent execution of the remaining code of the finalize method by aborting the execution of the finalize method through the use of a return instruction, and consequently aborting the Java Virtual Machine's finalization operation for this object.

Given the fundamental concept of testing to determine if a finalization, such as a deletion or clean up, is ready to be carried out on a class, object, or other asset; and if ready carrying out the finalization, and if not ready, then not carrying out the finalization, there are several different ways or embodiments in which this finalization concept, method, and procedure may be implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (such as class or object) inclusive of a clean up routine modifies it, and then loads each of the other machines M1, M3, ..., Mn (either sequentially or simultaneously or according to any other order, routine, or procedure) with the modified object (or class or asset) inclusive of the now modified clean up routine or routines. Note that there may be one or a plurality of routines corresponding to only one object in the application code or there can be a plurality of routines corresponding to a plurality of objects in the application code. Note that in one embodiment, the cleanup routine(s) that is (are) loaded is binary executable object code. Alternatively, the cleanup routine(s) that is (are) loaded is executable intermediate code.

In one arrangement, which may be termed "master/slave" (or primary/secondary) each of the slave (or secondary) machines M1, M3, ..., Mn loads the modified object (or class), and inclusive of the now modified clean-up routine(s), that was sent to it over the computer communications network or other communications link or path by the master (or primary) machine, such as machine M2, or some other machine such as a machine X of FIG. 15. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer communications network can be replaced by a shared storage device such as a shared file system, or a shared document/file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that in accordance with the inventive principles described herein, each of the plurality of machines behaves consistently and coherently relative to the other machines to accomplish the operations and objectives described herein. Furthermore, it will be appreciated in light of the description provided herein that there are a myriad of

ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that embodiments of the invention may be implemented within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software, in firmware, or in any combination of these.

In a further variation of this "master/slave" or "primary/secondary" arrangement, machine M2 loads the asset (such as class or object) inclusive of a cleanup routine in unmodified form on machine M2, and then (for example, M2 or each local machine) deletes the unmodified clean up routine that had been present on the machine in whole or part from the asset (such as class or object) and loads by means of a computer communications network the modified code for the asset with the now modified or deleted clean up routine on the other machines. Thus in this instance the modification is not a transformation, instrumentation, translation or compilation of the asset clean up routine but a deletion of the clean up routine on all machines except one. In one embodiment, the actual code-block of the finalization or cleanup routine is deleted on all machines except one, and this last machine therefore is the only machine that can execute the finalization routine because all other machines have deleted the finalization routine. One benefit of this approach is that no conflict arises between multiple machines executing the same finalization routine because only one machine has the routine.

The process of deleting the clean up routine in its entirety can either be performed by the "master" machine (such as machine M2 or some other machine such as machine X of FIG. 15) or alternatively by each other machine M1, M3...Mn upon receipt of the unmodified asset. An additional variation of this "master/slave" or "primary/secondary" arrangement is to use a shared storage device such as a shared file system, or a shared document/file repository such as a shared database as means of exchanging the code for the asset, class or object between machines M1, M2...Mn and optionally a machine X of FIG. 15.

In a still further embodiment, each machine M1, ..., Mn receives the unmodified asset (such as class or object) inclusive of finalization or clean up routine(s), but modifies the routine(s) and then loads the asset (such as class or object) consisting of the now modified routine(s). Although one machine, such as the master or primary machine may customize or perform a different modification to the finalization or clean up routine(s) sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different and to be enhanced, customized or optimized based upon its particular machine architecture, hardware, processor, memory, configuration, operating system or other factors, yet still similar, coherent and consistent with other machines with all other similar modifications and characteristics that may not need to be similar or identical.

In a further arrangement, a particular machine, say M1, loads the unmodified asset (such as class or object) inclusive of a finalization or clean up routine and all other machines M2, M3, ..., Mn perform a modification to delete the clean up routine of the asset (such as class or object) and load the modified version.

In all of the described instances or embodiments, the supply or communication of the asset code (such as class code or object code) to the machines M1, ..., Mn, and optionally inclusive of a machine X of FIG. 15 can be branched, distributed or communicated among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4, etc directly), or by providing or using cascaded or sequential communication (for example, M2 supplies M1 which then supplies M3, which then supplies M4, and so on), or a combination of the direct and cascaded and/or sequential.

In a still further arrangement, the machines M1, ...,Mn, may send some or all load requests to an additional machine X (See for example the embodiment of FIG. 15), which performs the modification to the application program code 50 (such as consisting of assets, and/or classes, and/or objects) and inclusive of finalization or clean up routine(s), via any of the afore mentioned methods, and returns the modified application program code inclusive of the now modified finalization or clean-up routine(s) to each of the machines M1 to Mn, and these machines in turn load the modified application program code inclusive of the modified routine(s) locally. In this arrangement, machines M1 to Mn forward all load requests to machine X, which returns a modified application program code inclusive of modified finalization or clean-up routine(s) to each machine. The modifications performed by machine X can include any of the modifications covered under the scope of the present invention. This arrangement may of course be applied to some of the machines and other arrangements described herein before applied to other of the machines.

Persons skilled in the computing arts will be aware of various possible techniques that may be used in the modification of computer code, including but not limited to instrumentation, program transformation, translation, or compilation means.

One such technique is to make the modification(s) to the application code, without a preceding or consequential change of the language of the application code. Another such technique is to convert the original code (for example, JAVA language source-code) into an intermediate representation (or intermediate-code language, or pseudo code), such as JAVA byte code. Once this conversion takes place the modification is made to the byte code and then the conversion may be reversed. This gives the desired result of modified JAVA code.

A further possible technique is to convert the application program to machine code, either directly from source-code or via the abovementioned intermediate language or through some other intermediate means. Then the machine code is modified before being loaded and executed. A still further such technique is to convert the original code to an intermediate representation, which is thus modified and subsequently converted into machine code.

The present invention encompasses all such modification routes and also a combination of two, three or even more, of such routes.

SYNCHRONIZATION

Turning again to FIG. 14, there is illustrated a schematic representation of a single prior art computer operated as a JAVA virtual machine. In this way, a machine (produced by any one of various manufacturers and having an operating system operating in any one of various different languages) can operate in the particular language of the application program code 50, in this instance the JAVA language. That is, a JAVA virtual machine 72 is able to operate application code 50 in the JAVA language, and utilize the JAVA architecture irrespective of the machine manufacturer and the internal details of the machine.

When implemented in a non-JAVA language or application code environment, the generalized platform, and/or virtual machine and/or machine and/or runtime system is able to operate application code 50 in the language(s) (possibly including for example, but not limited to any one or more of source-code languages, intermediate-code languages, object-code languages, machine-code languages, and any other code languages) of that platform, and/or virtual machine and/or machine and/or runtime system environment, and utilize the platform, and/or virtual machine and/or machine and/or runtime system and/or language architecture irrespective of the machine manufacturer and the internal details of the machine. It will also be appreciated in light of the description provided herein that platform and/or runtime system may include virtual machine and non-virtual machine software and/or firmware architectures, as well as hardware and direct hardware coded applications and implementations.

Furthermore, the single machine (not a plurality of connected or coupled machines) of FIG. 14, or a more general virtual machine or abstract machine environment such as for example but not limited to an object-oriented virtual machine, is able to readily ensure that multiple different and potentially concurrent uses of specific objects 50X-50Z do not conflict or cause unwanted interactions, when specified by the use of mutual exclusion (e.g. "mutex") operators or operations (inclusive for example of locks, semaphores, monitors, barriers, and the like), such as for example by the programmer's use of a synchronizing or synchronization routine in a computer program written in the JAVA language. As each object exists singularly and only locally (that is locally within the machine within which execution is occurring) in this example, the single JAVA virtual machine 72 of FIG. 14 executing within this single machine is able to ensure that an object (or several objects) is (are) properly synchronized as defined by the JAVA Virtual Machine and Language Specifications existent at least as of the date of the filing of this patent application, when specified to do so by the application program (or programmer), and thus the object or objects to be synchronized are only utilized by one executing part of potentially multiple executing parts and potentially concurrently executing parts of the executable application code 50 at once or at the same time, such as for example potentially concurrently executing threads or processes. If another executing part and potentially concurrently executing part (such as for example but not limited to a potentially concurrently executing thread or process) of the executable application code 50 wishes to exclusively use the same object whilst that object is the subject of a mutual exclusion operation by a first executing part (e.g. a first thread or process), such as when a second executing part (e.g. a second thread or process) of a multiple part processing machine of FIG. 14 attempts to synchronize on a same object already synchronized by a first executing part, then the possible conflict is resolved by the JAVA virtual machine 72

such that the second and additional executing parts and potentially concurrently executing part or parts of the application program 50 have to wait until the first executing part has finished the execution of its synchronization routine or other mutual exclusion operation. It may be appreciated that in a conventional situation, a second or multiple executing part(s) (i.e. a second or multiple thread(s)) of the application program or program code may want to use the same object in a multiple-thread processing machine of FIG. 14.

For a more general set of virtual machine or abstract machine environments, and for current and future computers and/or computing machines and/or information appliances or processing systems, and that may not utilize or require utilization of either classes and/or objects, the inventive structure, method, and computer program and computer program product are still applicable. Examples of computers and/or computing machines that do not utilize either classes and/or objects include for example, the x86 computer architecture manufactured by Intel Corporation and others, the SPARC computer architecture manufactured by Sun Microsystems, Inc and others, the PowerPC computer architecture manufactured by International Business Machines Corporation and others, and the personal computer products made by Apple Computer, Inc., and others. For these types of computers, computing machines, information appliances, and the virtual machine or virtual computing environments implemented thereon that do not utilize the idea of classes or objects, the terms 'class' and 'object' may be generalized for example to include primitive data types (such as integer data types, floating point data types, long data types, double data types, string data types, character data types and Boolean data types), structured data types (such as arrays and records) derived types, or other code or data structures of procedural languages or other languages and environments such as functions, pointers, components, modules, structures, references and unions.

A similar procedure applies mutatis mutandis (that is, with suitable or necessary alterations) for classes 50A. In particular, the computer programmer (or if and when applicable, an automated or nonautomated computer program generator or generation means) when writing or generating a program using the JAVA language and architecture in a single machine, need only use a synchronization routine or routines in order to provide for this avoidance of conflict or unwanted interaction. Thus a single JAVA virtual machine can keep track of exclusive utilization of the classes and objects (or other asset) and avoid corresponding problems (such as conflict, race condition, unwanted interaction, or other anomalous behaviour due to unexpected critical dependence on the relative timing of events) as necessary in an unobtrusive fashion. The process whereby only one object or class is exclusively used is termed "synchronization" in the JAVA language. In the JAVA language, synchronization may usually be operationalized or implemented in one of three ways or means. The first way or means is through the use of a synchronization method description that is included in the source-code of an application program written in the JAVA language. The second way or means is by the inclusion of a 'synchronization descriptor' in the method descriptor of a compiled application program of the JAVA virtual machine. And the third way or means for performing synchronization are by the use of the instructions monitor enter (e.g., "monitorenter") and monitor exit (e.g., "monitorexit") of the JAVA virtual machine which signify respectively the beginning and ending of a synchronization routine which results in the acquiring or execution of a "lock" (or other mutual exclusion operator or operation), and the releasing

or termination of a "lock" (or other mutual exclusion operator or operation) respectively which prevents an asset being the subject of conflict (or race condition, or unwanted interaction, or other anomalous behaviour due to unexpected critical dependence on the relative timing of events) between multiple and potentially concurrent uses. An asset may for example include a class or an object, as well as any other software /language/ runtime/ platform/ architecture or machine resource. Such resources may include for example, but are not limited to, software programs (such as for example executable software. modules, subprograms, sub-modules, application program interfaces (API), software libraries, dynamically linkable libraries) and data (such as for example data types, data structures, variables, arrays, lists, structures, unions), and memory locations (such as for example named memory locations, memory ranges, address space(s), registers,) and input/output (I/O) ports and/or interfaces, or other machine, computer ,or information appliance resource or asset.

However, in the arrangement illustrated in FIG. 8, (and also in FIGS. 31-33), a plurality of individual computers or machines M1, M2, ..., Mn are provided, each of which are interconnected via a communications network 53 or other communications link and each of which individual computers or machines is provided with a modifier 51 (See in FIG. 5) and realised by or in for example the distributed run time (DRT) 71 (See FIG. 8) and loaded with a common application code 50. The term common application program is to be understood to mean an application program or application program code written to operate on a single machine, and loaded and/or executed in whole or in part on each one of the plurality of computers or machines M1, M2...Mn, or optionally on each one of some subset of the plurality of computers or machines M1, M2...Mn. Put somewhat differently, there is a common application program represented in application code 50, and this single copy or perhaps a plurality of identical copies are modified to generate a modified copy or version of the application program, each copy or instance prepared for execution on the plurality of machines. At the point after they are modified they are common in the sense that they perform similar operations and operate consistently and coherently with each other. It will be appreciated that a plurality of computers, machines, information appliances, or the like implementing the features of the invention may optionally be connected to or coupled with other computers, machines, information appliances, or the like that do not implement the features of the invention.

In some embodiments, some or all of the plurality of individual computers or machines may be contained within a single housing or chassis (such as so-called "blade servers" manufactured by Hewlett-Packard Development Company, Intel Corporation, IBM Corporation and others) or implemented on a single printed circuit board or even within a single chip or chip set.

Essentially the modifier 51 or DRT 71 ensures that when an executing part (such as a thread or process) of the modified application program 50 running on one or more of the machines exclusively utilizes (e.g., by means of a synchronization routine or similar or equivalent mutual exclusion operator or operation) a particular local asset, such as an objects 50X-50Z or class 50A, no other executing part and potentially concurrently executing part on machines M2...Mn exclusively utilizes the similar equivalent corresponding asset in its local memory at once or at the same time.

It will be appreciated in light of the description provided herein that there are alternative implementations of the modifier 51 and the distributed runtime system 71. For example, the modifier 51 may be implemented as a component of or within the distributed run time 71, and therefore the DRT 71 may implement the functions and operations of the modifier 51. Alternatively, the function and operation of the modifier 51 may be implemented outside of the structure, software, firmware, or other means used to implement the DRT 71. In one embodiment, the modifier 51 and DRT 71 are implemented or written in a single piece of computer program code that provides the functions of the DRT and modifier. The modifier function and structure therefore maybe subsumed into the DRT and considered to be an optional component. Independent of how implemented, the modifier function and structure is responsible for modifying the executable code of the application code program, and the distributed run time function and structure is responsible for implementing communications between and among the computers or machines. The communications functionality in one embodiment is implemented via an intermediary protocol layer within the computer program code of the DRT on each machine. The DRT may for example implement a communications stack in the JAVA language and use the Transmission Control Protocol/Internet Protocol (TCP/IP) to provide for communications or talking between the machines. Exactly how these functions or operations are implemented or divided between structural and/or procedural elements, or between computer program code or data structures within the invention are less important than that they are provided.

It will therefore be understood in light of the description provided here that the invention further includes any means of implementing thread-safety, regardless of whether it is through the use of locks (lock/unlock), synchronizations, monitors, semphafores, mutexes, or other mechanisms.

It will be appreciated that synchronization means or implies "exclusive use" or "mutual exclusion" of an asset or resource. Conventional structures and methods for implementations of single computers or machines have developed some methods for synchronization on such single computer or machine configurations. However, these conventional structures and methods have not provided solutions for synchronization between and among a plurality of computers, machines, or information appliances.

In particular, whilst one particular machine (say, for example machine M3) is exclusively using an object or class (or any other asset or resource), another machine (say, for example machine M5) may also be instructed by the code it is executing to exclusively use the local similar equivalent object or class corresponding to the similar equivalent object or class on machine M3 at the same time or an overlapping time period. Thus if the same corresponding local similar equivalent objects or classes on each machine M3 and M5 were to be exclusively used by both machines, then the behaviour of the object and application as a whole is undefined – that is, in the absence of proper exclusive use of an object (or class) when explicitly specified by the computer program (programmer), conflict, race conditions, unwanted interactions, anomalous behaviour due to unexpected dependence on the relative timing of events, or permanent inconsistency between the similar equivalent objects on machines M5 and M3 is likely to result. Thus the desirable result of achieving or providing consistent, coordinated, and coherent operation of synchronization routines (or other mutual exclusion operations) between and

amongst a plurality of machines, as required for the simultaneous and coordinated operation of the same application program code on each of the plurality of machines M1, M2...Mn, would not be achieved.

In order to ensure consistent synchronization between and amongst machines M1, M2...Mn the application code 50 is analysed or scrutinized by searching through the executable application code 50 in order to detect program steps (such as particular instructions or instruction types) in the application code 50 which define or constitute or otherwise represent a synchronization routine (or other mutual exclusion operation). In the JAVA language, such program steps may for example comprise or consist of an opening monitor enter (e.g. "monitorenter") instruction and one or more closing monitor exit (e.g. "monitorexit") instructions. In one embodiment, a synchronization routine may start with the execution of a "monitorenter" instruction and close with a paired execution of a "monitorexit" instruction.

This analysis or scrutiny of the application code 50 may take place either prior to loading the application program code 50, or during the application program code 50 loading procedure, or even after the application program code 50 loading procedure. It may be likened to an instrumentation, program transformation, translation, or compilation procedure in that the application code may be instrumented with additional instructions, and/or otherwise modified by meaning-preserving program manipulations, and/or optionally translated from an input code language to a different code language (such as for example from source-code language or intermediate-code language to object-code language or machine-code language), and with the understanding that the term compilation normally or conventionally involves a change in code or language, for example, from source code to object code or from one language to another language. However, in the present instance the term "compilation" (and its grammatical equivalents) is not so restricted and can also include or embrace modifications within the same code or language. For example, the compilation and its equivalents are understood to encompass both ordinary compilation (such as for example by way of illustration but not limitation, from source-code to object-code), and compilation from source-code to source-code, as well as compilation from object-code to object-code, and any altered combinations therein. It is also inclusive of so-called "intermediary languages" which are a form of "pseudo object-code".

By way of illustration and not limitation, in one embodiment, the analysis or scrutiny of the application code 50 may take place during the loading of the application program code such as by the operating system reading the application code from the hard disk or other storage device or source and copying it into memory and preparing to begin execution of the application program code. In another embodiment, in a JAVA virtual machine, the analysis or scrutiny may take place during the class loading procedure of the java.lang.ClassLoader loadClass method (e.g., "java.lang.ClassLoader.loadClass()").

Alternatively, the analysis or scrutiny of the application code 50 may take place even after the application program code loading procedure, such as after the operating system has loaded the application code into memory, or optionally even after execution of the application program code has started, such as for example after the JAVA virtual

machine has loaded the application code into the virtual machine via the "java.lang.ClassLoader.loadClass()" method and optionally commenced execution.

Reference is made to the accompanying Annexure D in which: Annexure D1 is a typical code fragment from a synchronization routine prior to modification (e.g., an exemplary unmodified synchronization routine), and Annexure D2 is the same synchronization routine after modification (e.g., an exemplary modified synchronization routine). These code fragments are exemplary only and identify one software code means for performing the modification in an exemplary language. It will be appreciated that other software/firmware or computer program code may be used to accomplish the same or analogous function or operation without departing from the invention.

Annexures D1 and D2 (also reproduced in part in Tables XX and XXI below) are exemplary code listings that set forth the conventional or unmodified computer program software code (such as may be used in a single machine or computer environment) of a synchronization routine of application program 50 and a post-modification excerpt of the same synchronization routine such as may be used in embodiments of the present invention having multiple machines. The modified code that is added to the synchronization method is highlighted in **bold** text. Other embodiments of the invention may provide for code or statements or instructions to be added, amended, removed, moved or reorganized, or otherwise altered.

It is noted that the compiled code in the Annexure and portion repeated in the table is taken from the source-code of the file "example.java" which is included in the Annexure D3. The disassembled compiled code that is listed in the Annexure and Table is taken from compiled source code of the file "EXAMPLE.JAVA". In the procedure of Annexure D1 and Table XX, the procedure name "Method void run()" of Step 001 is the name of the displayed disassembled output of the run method of the compiled application code of "example.java". The name "Method void run()" is arbitrary and selected for this example to indicate a typical JAVA method inclusive of a synchronization operation. Overall the method is responsible for incrementing a memory location ("counter") in a thread-safe manner through the use of a synchronization statement and the steps to accomplish this are described in turn.

First (Step 002), the Java Virtual Machine instruction "getstatic #2 <Field java.lang.Object LOCK>" causes the Java Virtual Machine to retrieve the object reference of the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 2nd index of the classfile structure of the application program containing this example run() method and results in a reference to the object (hereafter referred to as LOCK) in the field to be placed (pushed) on the stack of the current method frame of the currently executing thread.

Next (Step 003), the Java Virtual Machine instruction "dup" causes the Java Virtual Machine to duplicate the topmost item of the stack and push the duplicated item onto the topmost position of the stack of the current method frame and results in the reference to the LOCK object at the top of the stack being duplicated and pushed onto the stack.

Next (Step 004), the Java Virtual Machine instruction "astore_1" causes the Java Virtual Machine to remove the topmost item of the stack of the current method frame and store the item into the local variable array at index 1 of the current method frame and results in the topmost LOCK object reference of the stack being stored in the local variable index 1.

Then (Step 005), the Java Virtual Machine instruction "monitorenter" causes the Java Virtual Machine to pop the topmost object off the stack of the current method frame and acquire an exclusive lock on said popped object and results in a lock being acquired on the LOCK object.

The Java Virtual Machine instruction "getstatic #3 <Field int counter>" (Step 006) causes the Java Virtual Machine to retrieve the integer value of the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example run() method and results in the integer value of said field being placed (pushed) on the stack of the current method frame of the currently executing thread.

The Java Virtual Machine instruction "iconst_1" (Step 007) causes the Java Virtual Machine to load an integer value of "1" onto the stack of the current method frame and results in the integer value of 1 loaded onto the top of the stack of the current method frame.

The Java Virtual Machine instruction "iadd" (Step 008) causes the Java Virtual Machine to perform an integer addition of the two topmost integer values of the stack of the current method frame and results in the resulting integer value of the addition operation being placed on the top of the stack of the current method frame.

The Java Virtual Machine instruction "putstatic #3 <Field int counter>" (Step 009) causes the Java Virtual Machine to pop the topmost value off the stack of the current method frame and store the value in the static field indicated by the CONSTANT_Fieldref_info constant-pool item stored in the 3rd index of the classfile structure of the application program containing this example run() method and results in the topmost integer value of the stack of the current method frame being stored in the integer field named "counter".

The Java Virtual Machine instruction "aload_1" (Step 010) causes the Java Virtual Machine to load the item in the local variable array at index 1 of the current method frame and store this item on the top of the stack of the current method frame and results in the object reference stored in the local variable array at index 1 being pushed onto the stack.

The Java Virtual Machine instruction "monitorexit" (Step 011) causes the Java Virtual Machine to pop the topmost object off the stack of the current method frame and release the exclusive lock on said popped object and results in the LOCK being released on the LOCK object.

Finally, the Java Virtual Machine instruction "return" (Step 012) causes the Java

Virtual Machine to cease executing this run() method by returning control to the previous method frame and results in termination of execution of this run() method.

As a result of these steps operating on a single machine of the conventional configurations in FIG. 1 and FIG. 2, the synchronization statement enclosing the increment operation of the "counter" memory location ensures that no two or more concurrently execution instances of this run() method will conflict, or otherwise result in unwanted interactions such as a race-condition or other anomalous behaviour due to unexpected critical dependence on the relative timing of the incrementing events performed of the one "counter" memory location. Were these steps to be carried out on the plurality of machines of the configurations of FIG. 5 and FIG. 8 with the memory update and propagation replication means of FIGS. 9, 10, 11, 12 and 13, and concurrently executing two or more instances or occurrences of the run() method each on a different one of the plurality of machines M1, M2...Mn, the mutual exclusion operations of each concurrently executing instance of the run() method would be performed on each corresponding one of the machines without coordination between those machines.

Given the desirable result of consistent coordinated synchronization operation across a plurality of machines, this prior art arrangement would fail to perform such consistent coordinated synchronization operation across the plurality of machines, as each machine performs synchronization only locally and without any attempt to coordinate their local synchronization operation with any other similar synchronization operation on any one or more other machines. Such an arrangement would therefore be susceptible to conflict or other unwanted interactions (such as race-conditions or other anomalous behaviour due to unexpected critical dependence on the relative timing of the "counter" increment events on each machine) between the machines M1, M2, ..., Mn. Therefore it is desirable to overcome this limitation of the prior art arrangement.

In the exemplary code in Table XXI (Annexure D2), the code has been modified so that it solves the problem of consistent coordinated synchronization operation for a plurality of machines M1, M2, ..., Mn, that was not solved in the code example from Table XX (Annexure D1). In this modified run() method code, a "dup" instruction is inserted between the "4 astore_1" and "6 monitorenter" instructions. This causes the Java Virtual Machine to duplicate the topmost item of the stack and push said duplicated item onto the topmost position of the stack of the current method frame and results in the reference to the LOCK object at the top of the stack being duplicated and pushed onto the stack.

Furthermore, the Java Virtual Machine instruction "invokestatic #23 <Method void acquireLock(java.lang.Object)>" is inserted after the "6 monitorenter" and before the "10 getstatic #3 <Field int counter>" statements so that the Java Virtual Machine pops the topmost item off the stack of the current method frame and invokes the "acquireLock" method, passing the popped item to the new method frame as its first argument. This change is particularly significant because it modifies the run() method to execute the "acquireLock" method and associated operations, corresponding to the "monitorenter" instruction preceding it.

Annexure D1 is a before-modification excerpt of the disassembled compiled form

of the synchronization operation of example.java of Annexure D3, consisting of an starting "monitorenter" instruction and ending "monitorexit" instruction. Annexure D2 is an after-modification form of Annexure D1, modified by LockLoader.java of Annexure D6 in accordance with the steps of FIG. 26. The modifications are highlighted in bold.

Table XX. Annexure D1

Step	Annexure D1
001	Method void run()
002	0 getstatic #2 <Field java.lang.Object LOCK>
003	3 dup
004	4 astore_1
005	5 monitorenter
006	6 getstatic #3 <Field int counter>
007	9 iconst_1
008	10 iadd
009	11 putstatic #3 <Field int counter>
010	14 aload_1
011	15 monitorexit
012	16 return

Table XXI. Annexure D2

Step	Annexure D2
001	Method void run()
002	0 getstatic #2 <Field java.lang.Object LOCK>
003	3 dup
004	4 astore_1
004A	**5 dup**
005	6 monitorenter
005A	**7 invokestatic #23 <Method void acquireLock(java.lang.Object)>**
006	10 getstatic #3 <Field int counter>
007	13 iconst_1
008	14 iadd
009	15 putstatic #3 <Field int counter>
010	18 aload_1
010A	**19 dup**
010B	**20 invokestatic #24 <Method void releaseLock(java.lang.Object)>**
011	23 monitorexit
	24 return

The method void acquireLock(java.lang.Object), part of the LockClient code of Annexure D4 and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1, ..., Mn to coordinate the execution of the preceding "monitorenter" synchronization operation amongst the machines M1...Mn. The acquireLock method of this example communicates with the LockServer code of Annexure D5 executing on a machine X of FIG. 15, by means of sending an 'acquire lock

request' to machine X corresponding to the object being 'locked' (i.e., the object corresponding to the "monitorenter" instruction), which in the context of Table XXI and Annexure D2 is the 'LOCK' object. With reference to FIG. 29, Machine X receives the 'acquire lock request' corresponding to the LOCK object, and consults a table of locks to determine the lock status corresponding to the plurality of similar equivalent objects on each of the machines, which in the case of Annexure D2 is the plurality of similar equivalent LOCK objects.

If all of the plurality of similar equivalent objects on each of the plurality of machines M1...Mn is presently not locked by any other machine M1...Mn, then Machine X will record the object as now locked and inform the requesting machine of the successful acquisition of the lock. Alternatively, if a similar equivalent object is presently locked by another one of the machines M1...Mn, then Machine X will append this requesting machine to a queue of machines waiting to lock this plurality of similar equivalent objects, until such a time as machine X determines this requesting machine can acquire the lock. Corresponding to the successful acquisition of a lock by a requesting machine, a reply is generated and sent to the successful requesting machine informing that machine of the successful acquisition of the lock. Following a receipt of such a message from Machine X confirming the successful acquisition of a requested lock, the acquireLock method and operations terminate execution and return control to the previous method frame, which is the context of Annexure D2 is the executing method frame of the run() method. Until such a time as the requesting machine receives a reply from machine X confirming the successful acquisition of the requested lock, the operation of the acquireLock method and run() method are suspended until such a confirmatory reply is received. Following this return operation, the execution of the run() method then resumes. Exemplary source-code for an embodiment of the acquireLock method is provided in Annexure D4. Annexure D4 also provides additional detail concerning DRT 71 functionality.

Later, the two statements "dup" and "invokestatic #24 <Method void releaseLock(java.lang.Object)>" are inserted into the code stream after the "18 aload_1" statement and before the "23 monitorexit" statement. These two statements cause the Java Virtual Machine to duplicate the item on the stack and then invoke the releaseLock method with the topmost item of the stack as an argument to the method call and result in the modification of the run() method to execute the "releaseLock" method and associated operations, corresponding to the following "monitorexit" instruction, before the procedure exits and returns.

The method void releaseLock(java.lang.Object), part of the LockClient code of Annexure D4 and part of the distributed runtime system (DRT) 71, performs the communications operations between machines M1...Mn to coordinate the execution of the following "monitorexit" synchronization operation amongst the machines M1...Mn. The releaseLock method of this example communications with LockServer code of Annexure D5 executing on a machine X of FIG. 15, by means of sending a "release lock request" to machine X corresponding to the object being "unlocked" (i.e., the object corresponding to the "monitorexit" instruction), which in the context of Table XXI and Annexure D2 is the 'LOCK' object. Corresponding to FIG. 30, machine X receives the "release lock request" corresponding to the LOCK object, and updates the table of locks to indicate the lock status corresponding to the plurality of similar equivalent 'LOCK'

objects as now "unlocked". Additionally, if there are other machines awaiting acquisition of this lock, then machine X is able to select one of the awaiting machines to be the new owner of the lock by updating the table of locks to indicate this selected one awaiting machine as the new lock owner, and informing the successful one of the awaiting machines of its successful acquisition of the lock by means of a confirmatory reply. The successful one of the awaiting machines then resumes execution of its synchronization routine. Following the notification to machine X of lock release, the releaseLock method terminates execution and returns control to the previous method frame, which in this instance is the method frame of the run() method. Following this return operation, the execution of the run() method resumes.

It will be appreciated that the modified code permits, in a distributed computing environment having a plurality of computers or computing machines, the coordinated operation of synchronization routines or other mutual exclusion operations between and amongst machines M1...Mn so that the problems associated with the operation of the unmodified code or procedure on a plurality of machines M1...Mn (such as conflicts, unwanted interactions, race-conditions, or anomalous behaviour due to unexpected critical dependence on the relative time of events) does not occur when applying the modified code or procedure.

In the unmodified code sample of Annexure D1, the application program code includes instructions or operations that increment a memory location in local memory (used for a counter) within an enclosing synchronization routine. The purpose of the synchronization routine is to ensure thread-safety of the counter memory increment operation in multi-threaded and multi-processing applications and computer systems. The terms thread-safe or thread-safety refer to code that is either re-entrant or protected from multiple simultaneous execution by some form of mutual exclusion. Multi-threaded applications in the context of the invention may, for example, include applications operating two or more threads of execution concurrently each on a different machine. Thus, without the management of coordinated synchronization in environments comprising or consisting of a plurality of machines, each running concurrently executing part of a same application program, and with a memory updating and propagation replication means of FIGS. 9, 10, 11, 12, and 13, each computer or computing machine would perform synchronization in isolation, thus potentially incrementing the shared counter at the same time, leading to potential conflicts or unwanted interactions such as race condition(s) and incoherent memory between the machines M1...Mn. It will be appreciated that although this embodiment is described using a shared counter, the use or provision of such shared counter or memory location is optional and not required for the synchronization aspects of the invention. What is advantageous is that the synchronization routine behaves in a manner as the programming language, runtime system, or machine architecture (or any combination thereof) guarantees – that is, stop two parts (for example, two threads) of the application program from executing the same synchronization routine or same mutual exclusion operation or operator concurrently. Clearly consistent, coherent and coordinated synchronization behaviour is what the programmer or user of the application program code 50 expects to happen.

So, taking advantage of the DRT 71, the application code 50 is modified as it is loaded into the machine by changing the synchronization routine. It will be appreciated

in light of the description provided here that the modifications made on each machine may generally be similar in-so-far as they should advantageously achieve a consistent end result of coordinated synchronization operation amongst all the machines; however, given the broad applicability of the inventive synchronization method and associated procedures, the nature of the modifications may generally vary without altering the effect produced. For example, in a simple variation, one or more additional instructions or statements may be inserted, such as for example a "no-operation" (nop) type instruction into the application will mean the modifications made are technically different, but the modified code still conforms to the invention. Embodiments of the invention may for example, implement the changes by means of program transformation, translation, various forms of compilation, instrumentation, or by other means described herein or known in the art. The changes made (highlighted in bold text) are the starting or initial instructions and the ending instructions that the synchronization routine executes, and which correspond to the entry (start) and exit (finish) of the synchronization routine respectively. These added instructions (or modified instruction stream) act to coordinate the execution of the synchronization routine amongst the multiple concurrently executing instances or occurrences of the modified run method executing on each one of, or some subset of, the plurality of machines M1...Mn, by invoking the acquireLock method corresponding to the start of execution of the synchronization routine, and by invoking the releaseLock method corresponding to the finish of execution of the synchronization routine, thereby providing consistent coordinated operation of the synchronization routine (or other mutual exclusion operation or operator) as required for the simultaneous operation of the modified application program code that is running on or across the plurality of machines M1, M2,..., Mn. This also advantageously provides for operation of the one application program in a coordinated manner across the machines.

The acquire lock (e.g. "acquireLock()") method of the DRT 71 takes an argument "(java.lang.Object)" which represents a reference to (or some other unique identifier for) the particular local object for which the global lock is desired (See Annexure D2 and Table XXI), and is to be used in acquiring a global lock across the plurality of similar equivalent objects on the other machines corresponding to the specified local object. The unique identifier may, for example be the name of the object, a reference to the object in question, or a unique number representing the plurality of similar equivalent objects across all nodes. By using a globally unique identifier across all connected machines to represent the plurality of similar equivalent objects on the plurality of machines, the DRT can support the synchronization of multiple objects at the same time without becoming confused as to which of the multiple objects are already synchronized and which are not as might be the case if object (or class) identifiers were not unique, by using the unique identifier of each object to consult the correct record in the shared synchronization table.

A further advantage of using a global identifier here is as a form of 'meta-name' for all the similar equivalent local objects on each one of the machines. For example, rather than having to keep track of each unique local name of each similar equivalent local object on each machine, one may instead define a global name (e.g., "globalname7787") which each local machine in turn maps to a local object (e.g., "globalname7787" points to object "localobject456" on machine M1, and "globalname7787" points to object "localobject885" on machine M2, and "globalname7787" points to object "localobject111" on machine M3, and so forth). It thereafter is easier to simply say "acquire

lock for globalname7787" which is then translated on machine 1 (M1) to mean "acquire lock for localobject456", and is translated on machine 2 (M2) to mean "acquire lock for localobject885", and so on.

The shared synchronization table that may optionally be used is a table, other storage means, or any other data structure that stores an object (and/or class or other asset) identifier and the synchronization status (or locked or unlocked status) of each object (and/or class or other asset). The table or other storage means operates to relate an object (and/or class or other asset, or a plurality of similar equivalent objects or classes or assets) to a status of either locked or unlocked or some other physical or logical indication of a locked state and an unlocked state. For example: the table (or any other data structure one cares to employ) may advantageously include a named object identifier and a record indicating if a named object (i.e., "globalname7787") is locked or unlocked. In one embodiment, the table or other storage means stores a flag or memory bit, wherein when the flag or memory bit stores a "0" the object is unlocked and when the flag or memory bit stores a "1" the object is locked. Clearly, multiple bit or byte storage may be used and different logic sense or indicators may be used without departing from the invention.

The DRT 71 can determine the synchronization state of the object in any one of a number of ways. Recall, for example that the invention may include any means of implementing thread-safety, regardless of whether it is through the use of locks (lock/unlock), synchronizations, monitors, semphafores, mutexes, or other mechanisms. These means stop or limit concurrently executing parts of a single application program in order to guarantee consistency according to the rules of synchronization, locks, or the like. Preferably, it can ask each machine in turn if their local similar equivalent object (or class or other asset or resource) corresponding to the object being sought to be locked is presently synchronized, and if any machine replies true, then to pause execution of the synchronization routine and wait until that presently synchronized similar equivalent object on the other machine is unsynchronised, otherwise synchronize this object locally and resume execution of the synchronization routine. Each machine may implement synchronization (or mutual exclusion operations or operators) in its own way and this may be different in the different machines. Therefore, although some exemplary implementation details are provided, ultimately how synchronization (or mutual exclusion operations) is (are) implemented, or precisely how synchronization or mutual exclusion status (or locked/unlocked status) is recorded in memory or other storage means, is not critical to the invention. By unsynchronized we generally mean unlocked or otherwise not subject to a mutual exclusion operation, and by synchronized we generally mean locked and subject to a mutual exclusion operation.

Alternatively, the DRT 71 on each local machine can consult a shared record table (perhaps on a separate machine (for example, on machine X which is different from machines M1, M2, ..., Mn)), or can consult a coherent shared record table on each one of the local machines, or a shared database established in a memory or other storage, to determine if this object has been marked or identified as synchronized (or "locked") by any machine and if so, then wait until the status of the object is changed to "unlocked" and then acquire the lock on this machine, otherwise acquire the lock by marking the object as locked (optionally by this machine) in the shared lock table.

In the situation where the shared record table is consulted, this may be considered as a variation of a shared database or data structure, where each machine has a local copy of a shared table (that is a replica of a shared table) with is updated to maintain coherency across the plurality of machines M1, ..., Mn.

In one embodiment, the shared record table refers to a shared table accessible by all machines M1, ..., Mn, that may for example be defined or stored in a commonly accessibly database such that any machine M1, ..., Mn can consult or read this shared database table for the locked or unlocked status of an object. A further alternative arrangement is to implement a shared record table as a table in the memory of an additional machine (which we call "machine X") which stores each object identification name and its lock status, and serves as the central repository which all other machines M1, ..., Mn consult to determine locked status of similar equivalent objects.

In any of these different alternative implementations, the manner in which a one of, or a plurality of, similar equivalent objects is marked or identified as being synchronized (or locked) or unsynchronized (or unlocked) is relatively unimportant, and various stored memory bits or bytes or flags may be utilized as are known in the art to identify either one of the two possible logic states. It will also be appreciated that in the present embodiment, that synchronized is largely synonymous with locked and unsynchronized is largely synonymous with unlocked. These same considerations apply for classes as well as for other assets or resources.

Recall that the DRT 71 is responsible for determining the locked status for an object (or class, or other asset, corresponding to a plurality of similar equivalent objects or classes or assets) seeking to be locked before allowing the synchronization routine corresponding to the acquisition of that lock to proceed. In the exemplary embodiment described here, the DRT consults the shared synchronization record table which in one embodiment resides on an special "machine X", and therefore the DRT needs to communicate via the network or other communications link or path with this machine X to enquire as to and determine the locked (or unlocked) status of the object (or class or other asset corresponding to a plurality of similar equivalent objects or classes or assets).

If the DRT on the local machine that is trying to execute a synchronization routine or other mutual exclusion operation determines that no other machine currently has a lock for this object (i.e., no other machine has synchronized this object) or any other one of a plurality of similar equivalent objects, then to acquire the lock for this object corresponding to the plurality of similar equivalent objects on all other machines, for example by means of modifying the corresponding entry in a shared table of locked states for the object sought to be locked or alternatively, sequentially acquiring the lock on all other similar equivalent objects on all other machines in addition to the current machine. Note that the intent of this procedure is to lock the plurality of similar equivalent objects (or classes or assets) on all the other machines M1, ..., Mn so that simultaneous or concurrent use of any similar equivalent objects by two or more machines is prevented, and any available approach may be utilized to accomplish this coordinated locking. For example, it does not matter if machine M1 instructs M2 to lock its similar equivalent local object, then instructs M3 to lock its similar equivalent local object, and then

instructs M4 and so on; or if M1 instructs M2 to lock its similar equivalent local object, and then M2 instructs M3 to lock its similar equivalent local object, and then M3 instructs M4 to lock its similar equivalent local object, and so forth, what is being sought is the locking of the similar equivalent objects on all other machines so that simultaneous or concurrent use any similar equivalent objects by two or more machines is prevented. Only once this machine has successfully confirmed that no other machine has currently locked a similar equivalent object, and this machine has correspondingly locked its locally similar equivalent object, can the execution of the synchronization routine or code-block begin.

On the other hand, if the DRT 71 within the machine about to execute a synchronization routine (such as machine M1) determines that another machine, such as machine M4 has already synchronized a similar equivalent object, then this machine M1 is to postpone continued execution of the synchronization routine (or code-block) until such a time as the DRT on machine M1 can confirm than no other machine (such as one of machines M2, M3, M4, or M5, ..., Mn) is presently executing a synchronize routine on a corresponding similar equivalent local object, and that this machine M1 has correspondingly synchronized its similar equivalent object locally. Recall that local synchronization refers to prior art conventional synchronization on a single machine, whereas global or coordinated synchronization refers to coordinated synchronization of, across and/or between similar equivalent local objects each on a one of the plurality of machines M1...Mn. In such a case, the synchronization routine (or code-block) is not to continue execution until this machine M1 can guarantee that no other machine M2, M3, M4, ..., Mn is executing a synchronization routine corresponding to the local similar equivalent object being sought to be locked, as it will potentially corrupt the object across the participating machines M1, M2, M3, ..., Mn due to susceptibility to conflicts or other unwanted interactions such as race-conditions, and the like problems resulting from the concurrent execution of synchronization routines. Thus, when the DRT determines that this object, or a similar equivalent object on another machine, is presently "locked", say by machine M4 (relative to all other machines), the DRT on machine M1 pauses execution of the synchronization routine by pausing the execution of the acquire lock (e.g., "acquireLock()") operation until such a time as a corresponding release lock (e.g., "releaseLock()") operation is executed by the present owner of the lock (e.g., machine M4).

Thus, on execution of a release lock (e.g. "releaseLock()") operation, the machine M4 which presently "owns" or holds a lock (i.e., is executing a synchronization routine) indicates the close of its synchronization routine, for example by marking this object as "unlocked" in the shared table of locked states, or alternatively, sequentially releasing locks acquired on all other machines. At this point, a different machine waiting to begin execution of a paused synchronization statement can then claim ownership of this now released lock by resuming execution of its postponed (i.e. delayed) "acquireLock()" operation, for example, by marking itself as executing a lock for this similar equivalent object in the shared table of synchronization states, or alternatively, sequentially acquiring local locks of similar equivalent objects on each of the other machines. It is to be understood that the resumed execution of the acquire lock (e.g., "acquireLock") operation is to be inclusive of the optional resumption of execution of the acquire lock (e.g., "acquireLock") method at the point that execution was paused, as well as the

alternative optional arrangement wherein the execution of the acquire lock (e.g., "acquireLock") operation is repeated so as to re-request the lock. Again, these same considerations also apply for classes and more generally to any asset or resource.

So, according to at least one embodiment and taking advantage of the operation of the DRT 71, the application code 50 is modified as it is loaded into the machine by changing the synchronization routine (consisting of at least a beginning "acquire lock" type instruction (such as a JAVA "monitorenter" instruction) and an ending "release lock" type instruction (such as a JAVA "monitorexit" instruction). "Acquire lock" type instructions commence operation or execution of a mutual exclusion operation, generally corresponding to a particular asset such as a particular memory location or machine resource, and result in the asset corresponding to the mutual exclusion operation being locked with respect to some or all modes of simultaneous or concurrent use, execution or operation. "Release lock" type instructions terminate or otherwise discontinue operation or execution of a mutual exclusion operation, generally corresponding to a particular asset such as a particular memory location or machine resource, and result in the asset corresponding to the mutual exclusion operation being unlocked with respect to some or all modes of simultaneous or concurrent use, execution or operation. The changes made (highlighted in bold) are the modified instructions that the synchronization routine executes. These added instructions for example check if this lock has already been acquired by another machine. If this lock has not been acquired by another machine, then the DRT of this machine notifies all other machines that this machine has acquired the specified lock, and thereby stopping the other machines from executing synchronization routines corresponding to this lock.

The DRT 71 can determine and record the lock status of similar equivalent objects, or other corresponding memory location or machine or software resource on a plurality of machines, in many ways, such as for example, by way of illustration but not limitation:

1. Corresponding to the entry to a synchronization routine by Machine M1, the DRT of machine M1 individually consults or communicates with each machine to ascertain if this global lock is already acquired by any other Machine M2, ..., Mn different from itself. If this global lock corresponding to this asset or object is or has already been acquired by another one of the machines M2, ..., Mn then the DRT of Machine M1 pauses execution of the synchronization routine on machine M1 until all other machines no longer own a global lock on this asset or object (that is to say that none of the other machines any longer own a global lock corresponding to this asset or object), at which point machine M1 can successfully acquire the global lock such that all other machines M2, ..., Mn must now wait for machine M1 to release the global lock before a different machine can in turn acquire it. Otherwise, when it is determined that this global lock corresponding to this asset or object has not already been acquired by another machine M2, ..., Mn the DRT continues execution of the synchronization routine, and such that all other machines M2, ..., Mn must now wait for machine M1 to release the global lock before a different machine can in turn acquire it.

Alternatively, 2. Corresponding to the entry to a synchronization routine, the DRT consults a shared table of records (for example a shared database, or a copy of a shared table on each of the participating machines) which indicate if any machine

currently "owns" this global lock. If so, the DRT then pauses execution of the synchronization routine on this machine until no machine owns a global lock on a similar equivalent object. Otherwise the DRT records this machine in the shared table (or tables, if there are multiple tables of records, e.g., on multiple machines) as the owner of this global lock, and then continues executing the synchronization routine.

Similarly, when a global lock is released, that is to say, when the execution of a synchronization routine is to end, the DRT can "un-record", alter the status indicator, and/or reset the global lock status of machines in many alternative ways, for example by way of illustration but not limitation:

1. Corresponding to the exit to a synchronization routine, the DRT individually notifies each other machine that it no longer owns the global lock.

Alternatively,

2. Corresponding to the exit to a synchronization routine, the DRT updates the record for this globally locked asset or object (such as for example a plurality of similar equivalent objects or assets) in the shared table(s) of records such that this machine is no longer recorded as owning this global lock.

Still further, the DRT can provide an acquire global lock queue to queue machines needing to acquire a global lock in multiple alternative ways, for example by way of illustration but not limitation:

1. Corresponding to the entry to a synchronization routine by Machine M1 say, the DRT of machine M1 notifies the present owning machine (say Machine M4) of the global lock that machine M1 would like to or needs to acquire the corresponding global lock upon release by the current owning machine in order to perform an operation. The specified machine M4, if there are no other waiting machines, then stores a record of the requesting machine's (i.e., machine M1) interest or request in a table or list, such that machine M4 may know subsequent to releasing the corresponding global lock that the machine M1 recorded in the table or list is waiting to acquire the same global lock, which, following the exit of the synchronization routine corresponding to the global lock held by machine M4, then notifies the waiting machine (i.e. machine M1) specified in the record of waiting machines, that the global lock can be acquired, and thus machine M1 can proceed to acquire the global lock and continue executing its own synchronization routine.

2. Corresponding to the entry to a synchronization routine by machine M1 say, the DRT notifies the present owner of the global lock, say machine M4, that a specific machine (say machine M1) would like to acquire the lock upon release by that machine (i.e., machine M4). That machine M4, if it finds after consulting its records of waiting machines for this locked object, finds that there are already one or more other machines (say machines M2 and M7) waiting, then either appends machine M1 to the end of the list of machines M2 and M7 wanting to acquire this locked object, or alternatively, forwards the request from M1 to the first waiting machine (i.e., machine M2), or any other machine waiting (i.e., machine M7), which then, in turn, records machine M1 in their table or records of waiting machines.

In the example above, for example, the records may be kept on Machine M4 and store a queue or other ordered or indexed list of machines waiting to acquire the lock after Machine M4 releases the lock it holds. This list or queue may then be used or referenced by M4 so that M4 can pass the lock on to other machines in accordance with

the order of request or any other prioritization scheme. Alternatively, the list may be unordered, and machine M4 may pass the global lock on to any machine in the list or record.

3. Corresponding to the entry to a synchronization routine, the DRT records itself in a shared table(s) of records (for example, a table stored in a shared database accessible by all machines, or multiple separate tables which are substantially similar).

Still further or in the alternative, the DRT 71 can notify other machines queued to acquire this global lock corresponding to the exit of a synchronization routine by this machine in the following alternative ways, for example:

1. Corresponding to the exit of a synchronization routine, the DRT notifies one of the awaiting machines (for example, this first machine in the queue of waiting machines) that the global lock is released,

2. Corresponding to the exit of a synchronization routine, the DRT notifies one of the awaiting machines (for example, the first machine in the queue of waiting machines) that the global lock is released, and additionally, provides a copy of the entire queue of machines (for example, the second machine and subsequent machines awaiting for this global lock). This way, the second machine inherits the list of waiting machines from the first machine, and thereby ensures the continuity of the queue of waiting machines as each machine in turn down the list acquires and subsequently releases the same global lock.

During the abovementioned scrutiny, "monitorenter" and "monitorexit" instructions (or methods) are initially looked for and, when found, a modifying code is inserted so as to give rise to a modified synchronization routine. This modified routine additionally acquires and releases the global lock. There are several different modes whereby this modification and loading can be carried out.

As seen in FIG. 15 a modification to the general arrangement of FIG. 8 is provided in that machines M1, M2...Mn are as before and run the same application code 50 (or codes) on all machines M1....Mn simultaneously or concurrently. However, the previous arrangement is modified by the provision of a server machine X which is conveniently able to supply housekeeping functions, for example, and especially the synchronization of structures, assets, and resources. Such a server machine X can be a low value commodity computer such as a PC since its computational load is low. As indicated by broken lines in FIG. 15, two server machines X and X+1 can be provided for redundancy purposes to increase the overall reliability of the system. Where two such server machines X and X+1 are provided, they are preferably but optionally operated as redundant machines in a failover arrangement.

It is not necessary to provide a server machine X as its computational load can be distributed over machines M1, M2...Mn. Alternatively, a database operated by one machine (in a master/slave type operation) can be used for the housekeeping function(s).

FIG. 16 shows a preferred general procedure to be followed. After loading 161 has been commenced, the instructions to be executed are considered in sequence and all synchronization routines are detected as indicated in step 162. In the JAVA language

these are the "monitorenter" and "monitorexit" instructions, and methods marked as synchronized in the method descriptor. Other languages use different terms.

Where a synchronization routine is detected 162, it is modified in step 163 in order to perform consistent, coordinated, and coherent synchronization operation (or other mutual exclusion operation) across the plurality of machines M1...Mn, typically by inserting further instructions into the synchronization (or other mutual exclusion) routine to, for example, coordinate the operation of the synchronization routine amongst and between similar equivalent synchronization or other mutual exclusion operations on other one or more of the plurality of machines M1...Mn, so that no two or more machines execute a similar equivalent synchronization or other mutual exclusion operation at once or overlapping. Alternatively, the modifying instructions may be inserted prior to the routine, such as for example prior to the instruction(s) or operation(s) related to a synchronization routine. Once the modification step 163 has been completed the loading procedure continues by loading the modified application code in place of the unmodified application code, as indicated in step 164. The modifications preferably take the form of an "acquire lock on all other machines" operation and a "release lock on all other machines" modification as indicated at step 163.

FIG. 27 illustrates a particular form of modification. Firstly, the structures, assets or resources (in JAVA termed classes or objects eg 50A, 50X-50Y) or more generally "locks" to be synchronized have already been allocated a name or tag (for example a global name or tag) which can be used to identify corresponding similar equivalent local objects, or assets, or resources, or locks on each of the machines M1....Mn, as indicated by step 172. This preferably happens when the classes or objects are originally initialized. This is most conveniently done via a table maintained by server machine X. This table also includes the synchronization status of the class or object or lock. It will be understood that this table or other data structure may store only the synchronization status, or it may store other status or information as well. In the preferred embodiment, this table also includes a queue arrangement which stores the identities of machines which have requested use of this asset or lock.

As indicated in step 173 of FIG. 27, next an "acquire lock" request is sent to machine X, after which, the sending machine awaits for confirmation of lock acquisition as shown in step 174. Thus, if the global name is already locked (i.e. a corresponding similar local asset is in exclusive use by another machine other than the machine proposing to acquire the lock) then this means that the proposed synchronization routine of the corresponding object or class or asset or lock should be paused until the corresponding object or class or asset or lock is unlocked by the current owner.

Alternatively, if the global name is not locked, this means that no other machine is exclusively using a similar equivalent class, object, asset or lock, and confirmation of lock acquisition is received straight away. After receipt of confirmation of lock acquisition, execution of the synchronization routine is allowed to continue, as shown in step 175.

FIG. 28 shows the procedures followed by the application program executing machine which wishes to relinquish a lock. The initial step is indicated at step 181. The

operation of this proposing machine is temporarily interrupted by steps 183, 184 until the reply is received from machine X, corresponding to step 184, and execution then resumes as indicated in step 185. Optionally, and as indicated in step 182, the machine requesting release of a lock is made to lookup the "global name" for this lock preceding a request being made to machine X. This way, multiple locks on multiple machines may be acquired and released without interfering with one another.

FIG. 29 shows the activity carried out by machine X in response to an "acquire lock" enquiry (of FIG. 27). After receiving an "acquire lock" request at step 191, the lock status is determined at steps 192 and 193 and, if no - the named resource is not free or otherwise "locked", the identity of the enquiring machine is added at step 194 to (or forms) the queue of awaiting acquisition requests. Alternatively, if the answer is yes - the named resource is free and "unlocked" - the corresponding reply is sent at step 197. The waiting enquiring machine is then able to execute the synchronization routine accordingly by carrying out step 175 of FIG. 27. In addition to the yes response, the shared table is updated at step 196 so that the status of the globally named asset is changed to "locked".

FIG. 30 shows the activity carried out by machine X in response to a "release lock" request of FIG. 28. After receiving a "release lock" request at step 201, machine X optionally, and preferably, confirms that the machine requesting to release the global lock is indeed the current owner of the lock, as indicated in step 202. Next, the queue status is determined at step 203 and, if no-one is waiting to acquire this lock, machine X marks this lock as "unowned" (or "unlocked") in the shared table, as shown in step 207, and optionally sends a confirmation of release back to the requesting machine, as indicated by step 208. This enables the requesting machine to execute step 185 of FIG. 28.

Alternatively, if yes – that is, other machines are waiting to acquire this lock - machine X marks this lock as now acquired by the next machine in the queue, as shown in step 204, and then sends a confirmation of lock acquisition to the queued machine at step 205, and consequently removes the new lock owner from the queue of waiting machines, as indicated in step 206.

Given the fundamental concept of modifying the synchronization routines (or other mutual exclusion operations or operators) to coordinate operation between and amongst a plurality of machines M1....Mn, there are several different ways or embodiments in which this coordinated, coherent and consistent synchronization (or other mutual exclusion) operation concept, method, and procedure may be carried out or implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (for example a class or object) inclusive of a synchronization routine(s), modifies it, and then loads each of the other machines M1, M3...Mn (either sequentially, or simultaneously or according to any other order, routine, or procedure) with the modified asset (or class or object) inclusive of the new modified synchronization routine(s). Note that there may be one or a plurality of routine(s) corresponding to only one object in the application code, or there may be a plurality of routines corresponding to a plurality of objects in the application code. Note that in one embodiment, the synchronization

routine(s) that is (are) loaded is binary executable object code. Alternatively, the synchronization routine(s) that is (are) loaded is executable intermediate code.

In this arrangement, which may be termed "master/slave" each of the slave (or secondary) machines M1, M3, ..., Mn loads the modified object (or class), and inclusive of the new modified synchronization routine(s), that was sent to it over the computer communications network or other communications link or path by the master (or primary) machine, such as machine M2, or some other machine such as a machine X of FIG. 15. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer communications network can be replaced by a shared storage device such as a shared file system, or a shared document/file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that in accordance with the inventive principles described herein, each of the plurality of machines behaves consistently and coherently relative to the other machines to accomplish the operations and objectives described herein. Furthermore, it will be appreciated in light of the description provided herein that there are a myriad of ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that embodiments of the invention may be implemented within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software, in firmware, or in any combination of these.

In a further variation of this "master/slave" or "primary/secondary" arrangement, machine M2 loads asset (such as class or object) inclusive of an (or even one or more) synchronization routine in unmodified form on machine M2, and then (for example, machine M2 or each local machine) modifies the class (or object or asset) by deleting the synchronization routine in whole or part from the asset (or class or object) and loads by means of a computer communications network or other communications link or path the modified code for the asset with the now modified or deleted synchronization routine on the other machines. Thus in this instance the modification is not a transformation, instrumentation, translation or compilation of the asset synchronization routine but a deletion of the synchronization routine on all machines except one.

The process of deleting the synchronization routine in its entirety can either be performed by the "master" machine (such as machine M2 or some other machine such as machine X of FIG. 15) or alternatively by each other machine M1, M3, ..., Mn upon receipt of the unmodified asset. An additional variation of this "master/slave" or "primary/secondary" arrangement is to use a shared storage device such as a shared file system, or a shared document/file repository such as a shared database as means of exchanging the code (including for example, the modified code) for the asset, class or object between machines M1, M2,..., Mn and optionally a machine X of FIG. 15.

In a still further embodiment, each machine M1, ..., Mn receives the unmodified asset (such as class or object) inclusive of one or more synchronization routines, but modifies the routines and then loads the asset (such as class or object) consisting of the

now modified routines. Although one machine, such as the master or primary machine may customize or perform a different modification to the synchronization routine sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different and to be enhanced, customized, and/or optimized based upon its particular machine architecture, hardware, processor, memory, configuration, operating system, or other factors, yet still similar, coherent and consistent with other machines with all other similar modifications and characteristics that may not need to be similar or identical.

In a further arrangement, a particular machine, say M1, loads the unmodified asset (such as class or object) inclusive of one or more synchronization routines and all other machines M2, M3, ..., Mn perform a modification to delete the synchronization routine(s) of the asset (such as class or object) and load the modified version.

In all of the described instances or embodiments, the supply or the communication of the asset code (such as class code or object code) to the machines M1, ..., Mn, and optionally inclusive of a machine X of FIG. 15, can be branched, distributed or communicated among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4, etc. directly), or by providing or using cascaded or sequential communication (for example, M2 supplies M1 which then supplies M3 which then supplies M4, and so on), or a combination of the direct and cascaded and/or sequential.

In a still further arrangement, the machines M1 to Mn, may send some or all load requests to an additional machine X (see for example the embodiment of FIG. 15), which performs the modification to the application code 50 inclusive of an (and possibly a plurality of) synchronization routine(s) via any of the afore mentioned methods, and returns the modified application code inclusive of the now modified synchronization routine(s) to each of the machines M1 to Mn, and these machines in turn load the modified application code inclusive of the modified routines locally. In this arrangement, machines M1 to Mn forward all load requests to machine X, which returns a modified application program code 50 inclusive of modified synchronization routine(s) to each machine. The modifications performed by machine X can include any of the modifications covered under the scope of the present invention. This arrangement may of course be applied to some of the machines and other arrangements described herein before applied to other of the machines.

Persons skilled in the computing arts will be aware of various possible techniques that may be used in the modification of computer code, including but not limited to instrumentation, program transformation, translation, or compilation means.

One such technique is to make the modification(s) to the application code, without a preceding or consequential change of the language of the application code. Another such technique is to convert the original code (for example, JAVA language source-code) into an intermediate representation (or intermediate-code language, or pseudo code), such as JAVA byte code. Once this conversion takes place the modification is made to the

byte code and then the conversion may be reversed. This gives the desired result of modified JAVA code.

A further possible technique is to convert the application program to machine code, either directly from source-code or via the abovementioned intermediate language or through some other intermediate means. Then the machine code is modified before being loaded and executed. A still further such technique is to convert the original code to an intermediate representation, which is thus modified and subsequently converted into machine code.

The present invention encompasses all such modification routes and also a combination of two, three or even more, of such routes.

EMBODIMENT INCLUDING MEMORY MANAGEMENT AND REPLICATION OBJECT INITIALIZATION, FINALIZATION, AND SYNCHRONIZATION

Having now described structures, procedures, computer program code and tools, and other aspects and features of a multiple computer system and computing method utilizing at least one of memory management and replication object initialization, finalization, and synchronization it may readily be appreciated that these may also optionally but advantageously be applied in any combination.

It may also be appreciated that the memory management, initialization, finalization, and/or synchronization aspects of the invention may be implemented or applied serially or sequentially or in parallel. For example, where the code is being scrutinized or analysed to identify or detect particular code sections relevant to initialization, that same analysis or scrutinization may also attempt to identify or detect code sections relevant to finalization (or synchronization for example). Alternatively, separate sequential (or possibly overlapping) analysis and scrutiny may be utilized to separately detect code relevant to initialization and finalization and synchronization. Any required modification to the code may also be performed in combination or separately, and furthermore, portions may be performed together while other portions are performed separately.

Having now described aspects of the memory management and replication, initialization, finalization, and synchronization, attention is now directed to an exemplary operational scenario illustrating the manner in which application programs on two computers may simultaneously execute the same application program in a consistent, coherent manner.

In this regard, attention is directed to FIGS. 31-33, two laptop computers 101 and 102 are illustrated. The computers 101 and 102 are not necessarily identical and indeed, one can be an IBM or IBM-clone and the other can be an APPLE computer. The computers 101 and 102 have two screens 105, 115 two keyboards 106, 116 but a single mouse 107. The two machines 101, 102 are interconnected by a means of a single coaxial cable or twisted pair cable 314.

Two simple application programs are downloaded onto each of the machines 101, 102, the programs being modified as they are being loaded as described above. In this embodiment the first application is a simple calculator program and results in the image

of a calculator 108 being displayed on the screen 105. The second program is a graphics program which displays four coloured blocks 109 which are of different colours and which move about at random within a rectangular box 310. Again, after loading, the box 310 is displayed on the screen 105. Each application operates independently so that the blocks 109 are in random motion on the screen 105 whilst numerals within the calculator 108 can be selected (with the mouse 107) together with a mathematical operator (such as addition or multiplication) so that the calculator 108 displays the result.

The mouse 107 can be used to "grab" the box 310 and move same to the right across the screen 105 and onto the screen 115 so as to arrive at the situation illustrated in FIG. 32. In this arrangement, the calculator application is being conducted on machine 101 whilst the graphics application resulting in display of box 310 is being conducted on machine 102.

However, as illustrated in FIG. 33, it is possible by means of the mouse 107 to drag the calculator 108 to the right as seen in FIG. 32 so as to have a part of the calculator 108 displayed by each of the screens 105, 115. Similarly, the box 310 can be dragged by means of the mouse 107 to the left as seen in FIG. 32 so that the box 310 is partially displayed by each of the screens 105, 115 as indicated FIG. 33. In this configuration, part of the calculator operation is being performed on machine 101 and part on machine 102 whilst part of the graphics application is being carried out the machine 101 and the remainder is carried out on machine 102.

The foregoing describes only some embodiments of the present invention and modifications, obvious to those skilled in the art, can be made thereto without departing from the scope of the present invention. For example, reference to JAVA includes both the JAVA language and also JAVA platform and architecture.

In all described instances of modification, where the application code 50 is modified before, or during loading, or even after loading but before execution of the unmodified application code has commenced, it is to be understood that the modified application code is loaded in place of, and executed in place of, the unmodified application code subsequently to the modifications being performed.

Alternatively, in the instances where modification takes place after loading and after execution of the unmodified application code has commenced, it is to be understood that the unmodified application code may either be replaced with the modified application code in whole, corresponding to the modifications being performed, or alternatively, the unmodified application code may be replaced in part or incrementally as the modifications are performed incrementally on the executing unmodified application code. Regardless of which such modification routes are used, the modifications subsequent to being performed execute in place of the unmodified application code.

It is advantageous to use a global identifier is as a form of 'meta-name' or 'meta-identity' for all the similar equivalent local objects (or classes, or assets or resources or the like) on each one of the plurality of machines M1, M2...Mn. For example, rather than having to keep track of each unique local name or identity of each similar equivalent local object on each machine of the plurality of similar equivalent objects, one may

instead define or use a global name corresponding to the plurality of similar equivalent objects on each machine (eg "globalname7787"), and with the understanding that each machine relates the global name to a specific local name or object (eg "globalname7787" corresponds to object "localobject456" on machine M1, and "globalname7787" corresponds to object "localobject885" on machine M2, and "globalname7787"corresponds to object "localobject111" on machine M3, and so forth).

It will also be apparent to those skilled in the art in light of the detailed description provided herein that in a table or list or other data structure created by each DRT 71 when initially recording or creating the list of all, or some subset of all objects (eg memory locations or fields), for each such recorded object on each machine M1, M2...Mn there is a name or identity which is common or similar on each of the machines M1, M2...Mn. However, in the individual machines the local object corresponding to a given name or identity will or may vary over time since each machine may, and generally will, store memory values or contents at different memory locations according to its own internal processes. Thus the table, or list, or other data structure in each of the DRTs will have, in general, different local memory locations corresponding to a single memory name or identity, but each global "memory name" or identity will have the same "memory value or content" stored in the different local memory locations. So for each global name there will be a family of corresponding independent local memory location with one family member in each of the computers. Although the local memory name may differ, the asset, object, location etc has essentially the same content or value. So the family is coherent.

It will also be apparent to those skilled in the art in light of the description provided herein that the abovementioned modification of the application program code 50 during loading can be accomplished in many ways or by a variety of means. These ways or means include, but are not limited to at least the following five ways and variations or combinations of these five, including by:

(i) re-compilation at loading,
(ii) a pre-compilation procedure prior to loading,
(iii) compilation prior to loading,
(iv) a "just-in-time" compilations, or
(v) re-compilation after loading (but, for example, before execution of the relevant or corresponding application code in a distributed environment).

Traditionally the term "compilation" implies a change in code or language, for example, from source to object code or one language to another. Clearly the use of the term "compilation" (and its grammatical equivalents) in the present specification is not so restricted and can also include or embrace modifications within the same code or language.

Given the fundamental concept of modifying memory manipulation operations to coordinate operation between and amongst a plurality of machines M1, M2...Mn, there are several different ways or embodiments in which this coordinated, coherent and consistent memory state and manipulation operation concept, method, and procedure may be carried out or implemented.

In the first embodiment, a particular machine, say machine M2, loads the asset (such as class or object) inclusive of memory manipulation operation(s), modifies it, and then loads each of the other machines M1, M3...Mn (either sequentially or simultaneously or according to any other order, routine or procedure) with the modified object (or class or other assert or resource) inclusive of the new modified memory manipulation operation. Note that there may be one or a plurality of memory manipulation operations corresponding to only one object in the application code, or there may be a plurality of memory manipulation operations corresponding to a plurality of objects in the application code. Note that in one embodiment, the memory manipulation operation(s) that is (are) loaded is executable intermediary code.

In this arrangement, which may be termed "master/slave" each of the slave (or secondary) machines M1, M3...Mn loads the modified object (or class), and inclusive of the new modified memory manipulation operation(s), that was sent to it over the computer communications network or other communications link or path be the master (or primary) machine, such as machine M2, or some other machine as a machine X. In a slight variation of this "master/slave" or "primary/secondary" arrangement, the computer communications network can be replaced by a shared storage device such as a shared file system, or a shared document/ file repository such as a shared database.

Note that the modification performed on each machine or computer need not and frequently will not be the same or identical. What is required is that they are modified in a similar enough way that each of the plurality of machines behaves consistently and coherently relative to the other machines. Furthermore, it will be appreciated that there are a myriad of ways to implement the modifications that may for example depend on the particular hardware, architecture, operating system, application program code, or the like or different factors. It will also be appreciated that implementation can be within an operating system, outside of or without the benefit of any operating system, inside the virtual machine, in an EPROM, in software in firmware, or in any combination of these,

In a still further embodiment, each machine M1, M2...Mn receives the unmodified asset (such as class or object) inclusive of one or more memory manipulation operation(s), but modifies the operations and then loads the asset (such as class or object) consisting of the now modified operations. Although one machine, such as the master or primary machine may customize or perform a different modification to the memory manipulation operation(s) sent to each machine, this embodiment more readily enables the modification carried out by each machine to be slightly different. It can thereby be enhanced, customized, and/or optimized based upon its particular machine architecture, hardware processor, memory, configuration, operating system, or other factors yet still be similar, coherent and consistent with the other machines and with all other similar modifications.

In all of the described instances or embodiments, the supply or the communication of the asset code (such as class code or object code) to the machines M1, M2...Mn and optionally inclusive of a machine X, can be branched, distributed or communication among and between the different machines in any combination or permutation; such as by providing direct machine to machine communication (for example, M2 supplies each of M1, M3, M4 etc. directly), or by providing or using cascaded or sequential

communication (for example, M2 supplies M1 which then supplies M3 which then supplies M4, and so on) or a combination of the direct and cascaded and/or sequential.

The abovedescribed arrangement needs to be varied in the situation where the modification relates to a cleanup routine, finalization or similar, which is only to be carried out by one of the plurality of computers In this variation of this "master/slave" or "primary/secondary" arrangement, machine M2 loads the asset (such as class or object) inclusive of a cleanup routine in unmodified form on machine M2, and then (for example, M2 or each local machine) deletes the unmodified cleanup routine that had been present on the machine in whole or part from the asset (such as class or object) and loads by means of a computer communications network the modified code for the asset with the now modified or deleted cleanup routine on the other machines. Thus in this instance the modification is not a transformation, instrumentation, translation or compilation of the asset cleanup routine but a deletion of the cleanup routine on all machines except one. In one embodiment, the actual code-block of the finalization or cleanup routine is deleted on all machines except one, and this last machine therefore is the only machine that can execute the finalization routine because all other machines have deleted the finalization routine. One benefit of this approach is that no conflict arises between multiple machines executing the same finalization routine because only one machine has the routine.

The process of deleting the cleanup routine in its entirety can either be performed by the "master" machine (such as machine M2 or some other machine such as machine X) or alternatively by each other machine M1, M3...Mn upon receipt of the unmodified asset. An additional variation of this "master/slave" or "primary/secondary" arrangement is to use a shared storage device such as a shared file system, or a shared document/file repository such as a shared database as means of exchanging the code for the asset, class or object between machines M1, M2...Mn and optionally the server machine X.

In a further arrangement, a particular machine, say M1, loads the unmodified asset (such as class or object) inclusive of a finalization or cleanup routine and all the other machines M2, M3...Mn perform a modification to delete the cleanup routine of the asset (such as class or object) and load the modified version.

In a still further arrangement, the machines M1, M2...Mn, may send some or all load requests to the additional server machine X , which performs the modification to the application program code 50 (including or consisting of assets, and/or classes, and/or objects) and inclusive of finalization or cleanup routine(s), via any of the afore mentioned methods, and returns in the modified application program code inclusive of the now modified finalization or cleanup routine(s) to each of the machines M1 to Mn, and these machines in turn load the modified application program code inclusive of the modified routine(s) locally. In this arrangement, machines M1 to Mn forward all load requests to machine X, which returns a modified application program code inclusive of modified finalization or cleanup routine(s) to each machine. The modifications performed by machine X can include any of the modifications described. This arrangement may of course be applied to some only of the machines whilst other arrangements described herein are applied to others of the machines.

The abovementioned embodiment in which the code of the JAVA initialisation routine is modified, is based upon the assumption that either the run time system (say, JAVA HOTSPOT VIRTUAL MACHINE written in C and JAVA) or the operating system (LINUX written in C and Assembler, for example) of each machine M1...Mn will call the JAVA initialisation routine. It is possible to leave the JAVA initialisation routine unamended and instead amend the LINUX or HOTSPOT routine which calls the JAVA initialisation routine, so that if the object or class is already loaded, then the JAVA initialisation routine is not called. In order to embrace such an arrangement the term "initialisation routine" is to be understood to include within its scope both the JAVA initialisation routine and the "combination" of the JAVA initialisation routine and the LINUX or HOTSPOT code fragments which call or initiates the JAVA initialisation routine.

The abovementioned embodiment in which the code of the JAVA finalisation or clean up routine is modified, is based upon the assumption that either the run time system (say, JAVA HOTSPOT VIRTUAL MACHINE written in C and JAVA) or the operating system (LINUX written in C and Assembler, for example) of each machine M1...Mn will call the JAVA finalisation routine. It is possible to leave the JAVA finalisation routine unamended and instead amend the LINUX or HOTSPOT routine which calls the JAVA finalisation routine, so that if the object or class is not to be deleted, then the JAVA finalisation routine is not called. In order to embrace such an arrangement the term "finalisation routine" is to be understood to include within its scope both the JAVA finalisation routine and the "combination" of the JAVA finalisation routine and the LINUX or HOTSPOT code fragments which call or initiate the JAVA finalisation routine.

The abovementioned embodiment in which the code of the JAVA synchronization routine is modified, is based upon the assumption that either the run time system (say, JAVA HOTSPOT VIRTUAL MACHINE written in C and JAVA) or the operating system (LINUX written in C and Assembler, for example) of each machine M1...Mn will normally acquire the lock on the local machine (say M2) but not on any other machines (M1, M3 ...Mn). It is possible to leave the JAVA synchronization routine unamended and instead amend the LINUX or HOTSPOT routine which acquires the lock locally, so that it correspondingly acquires the lock on all other machines as well. In order to embrace such an arrangement the term "synchronization routine" is to be understood to include within its scope both the JAVA synchronization routine and the "combination" of the JAVA synchronization routine and the LINUX or HOTSPOT code fragments which perform lock acquisition and release.

The terms object and class used herein are derived from the JAVA environment and are intended to embrace similar terms derived from different environments such as dynamically linked libraries (DLL), or object code packages, or function unit or memory locations.

Those skilled in the programming arts will be aware that when additional code or instructions is/are inserted into an existing code or instruction set to modify same, the existing code or instruction set may well require further modification (such as for

example, by re-numbering of sequential instructions) so that offsets, branching, attributes, mark up and the like are catered for.

Similarly, in the JAVA language memory locations include, for example, both fields and array types. The above description deals with fields and the changes required for array types are essentially the same mutatis mutandis. Also the present invention is equally applicable to similar programming languages (including procedural, declarative and object orientated) to JAVA including Microsoft.NET platform and architecture (Visual Basic, Visual C/C^{++}, and C#) FORTRAN, C/C^{++}, COBOL, BASIC etc.

The terms object and class used herein are derived from the JAVA environment and are intended to embrace similar terms derived from different environments such as dynamically linked libraries (DLL), or object code packages, or function unit or memory locations.

Various means are described relative to embodiments of the invention, including for example but not limited to lock means, distributed run time means, modifier or modifying means, and the like. In at least one embodiment of the invention, any one or each of these various means may be implemented by computer program code statements or instructions (possibly including by a plurality of computer program code statements or instructions) that execute within computer logic circuits, processors, ASICs, microprocessors, microcontrollers or other logic to modify the operation of such logic or circuits to accomplish the recited operation or function. In another embodiment, any one or each of these various means may be implemented in firmware and in other embodiments such may be implemented in hardware. Furthermore, in at least one embodiment of the invention, any one or each of these various means may be implemented by a combination of computer program software, firmware, and/or hardware.

Any and each of the afore described methods, procedures, and/or routines may advantageously be implemented as a computer program and/or computer program product stored on any tangible media or existing in electronic, signal, or digital form. Such computer program or computer program products comprising instructions separately and/or organized as modules, programs, subroutines, or in any other way for execution in processing logic such as in a processor or microprocessor of a computer, computing machine, or information appliance; the computer program or computer program products modifying the operation of the computer in which it executes or on a computer coupled with, connected to, or otherwise in signal communications with the computer on which the computer program or computer program product is present or executing. Such a computer program or computer program product modifies the operation and architectural structure of the computer, computing machine, and/or information appliance to alter the technical operation of the computer and realize the technical effects described herein.

The invention may therefore includes a computer program product comprising a set of program instructions stored in a storage medium or existing electronically in any form and operable to permit a plurality of computers to carry out any of the methods, procedures, routines, or the like as described herein including in any of the claims.

Furthermore, the invention includes a plurality of computers interconnected via a communication network or other communications link or path and each operable to substantially simultaneously or concurrently execute the same or a different portion of an application code written to operate on only a single computer on a corresponding different one of computers. The computers are programmed to carry out any of the methods, procedures. or routines described in the specification or set forth in any of the claims, on being loaded with a computer program product. Similarly, the invention also includes within its scope a single computer arrayed to co-operate with like, or substantially similar, computers to form a multiple computer system.

The term "compromising" (and its grammatical variations) as used herein is used in the inclusive sense of "having" or "including" and not in the exclusive sense of "consisting only of".

Copyright Notice

This patent specification and the Annexures which form a part thereof contains material which is subject to copyright protection. The copyright owner (which is the applicant) has no objection to the reproduction of this patent specification or related materials from publicly available associated Patent Office files for the purposes of review, but otherwise reserves all copyright whatsoever. In particular, the various instructions are not to be entered into a computer without the specific prior written approval of the copyright owner.

Annexure A

The following are program listings in the JAVA language:

A1. This first excerpt is part of the modification code. It searches through the code array, and when it finds a putstatic instruction (opcode 178), it implements the modifications.

```
// START
byte[] code =  Code_attribute.code;      // Bytecode of a given method in a
                                         // given classfile.

int code_length = Code_attribute.code_length;

int DRT = 99;        // Location of the CONSTANT_Methodref_info for the
                     // DRT.alert() method.

for (int i=0; i<code_length; i++){

   if ((code[i] & 0xff) == 179){ // Putstatic instruction.

      System.arraycopy(code, i+3, code, i+6, code_length-(i+3));

      code[i+3] = (byte) 184;     // Invokestatic instruction for the
                                  // DRT.alert() method.

      code[i+4] = (byte) ((DRT >>> 8) & 0xff);

      code[i+5] = (byte) (DRT & 0xff);

   }

}
// END
```

A2. This second excerpt is part of the DRT.alert() method. This is the body of the DRT.alert() method when it is called.

```
// START
public static void alert(){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.notify(); // Alerts a waiting DRT thread in the background.

   }

}
// END
```

A3. This third excerpt is part of the DRT Sending. This code fragment shows the DRT in a separate thread, after being notified, sending the value across the network.

```
// START
MulticastSocket ms = DRT.getMulticastSocket();       // The multicast socket
                                                     // used by the DRT for
                                                     // communication.

byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.

Field field = modifiedClass.getDeclaredField("myField1");   // Stores
                                                            // the field
                                                            // from the
```

```
                                                                    // modified
                                                                    // class.

// In this example, the field is a byte field.
while (DRT.isRunning()){

   synchronized (ALERT_LOCK){

      ALERT_LOCK.wait();   // The DRT thread is waiting for the alert
                           // method to be called.

      byte[] b = new byte[]{nameTag, field.getByte(null)};  // Stores
                                                            // the
                                                            // nameTag
                                                            // and the
                                                            // value
                                                            // of the
                                                            // field from
                                                            // the modified
                                                            // class in a
                                                            // buffer.

      DatagramPacket dp = new DatagramPacket(b, 0, b.length);

      ms.send(dp);  // Send the buffer out across the network.

   }

}
// END
```

A4. The fourth excerpt is part of the DRT receiving. This is a fragment of code to receive a DRT sent alert over the network.

```
// START
MulticastSocket ms = DRT.getMulticastSocket(); // The multicast socket
                                               // used by the DRT for
                                               // communication.

DatagramPacket dp = new DatagramPacket(new byte[2], 0, 2);

byte nameTag = 33;          // This is the "name tag" on the network for this
                            // field.

Field field = modifiedClass.getDeclaredField("myField1");   // Stores the
                                                            // field from
                                                            // the modified
                                                            // class.

// In this example, the field is a byte field.
while (DRT.isRunning){

   ms.receive(dp);  // Receive the previously sent buffer from the network.

   byte[] b = dp.getData();

   if (b[0] == nameTag){   // Check the nametags match.
      field.setByte(null, b[1]); // Write the value from the network packet
                                 // into the field location in memory.

   }

}
// END
```

A5. The fifth excerpt is an example application before modification has occurred.

```
Method void setValues(int, int)
  0 iload_1
  1 putstatic #3 <Field int staticValue>
  4 aload_0
  5 iload_2
  6 putfield #2 <Field int instanceValue>
  9 return
```

A6. The sixth excerpt is the same example application in 5 after modification has been performed. The modifications are highlighted in **bold.**

```
Method void setValues(int, int)
  0 iload_1
  1 putstatic #3 <Field int staticValue>
  4 ldc #4 <String "example">
  6 iconst_0
  7 invokestatic #5 <Method void alert(java.lang.Object, int)>
 10 aload_0
 11 iload_2
 12 putfield #2 <Field int instanceValue>
 15 aload_0
 16 iconst_1
 17 invokestatic #5 <Method void alert(java.lang.Object, int)>
 20 return
```

A7. The seventh excerpt is the source-code of the example application used in excerpt 5 and 6.

```
import java.lang.*;

public class example{

   /** Shared static field. */
   public static int staticValue = 0;

   /** Shared instance field. */
   public int instanceValue = 0;

   /** Example method that writes to memory (instance field). */
   public void setValues(int a, int b){

      staticValue = a;

      instanceValue = b;

   }


}
```

A8. The eighth excerpt is the source-code of FieldAlert, which alerts the "distributed run-time" to propagate a changed value.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;
public class FieldAlert{

   /** Table of alerts. */
   public final static Hashtable alerts = new Hashtable();

   /** Object handle. */
   public Object reference = null;


   /** Table of field alerts for this object. */
   public boolean[] fieldAlerts = null;

   /** Constructor. */
   public FieldAlert(Object o, int initialFieldCount){
      reference = o;
      fieldAlerts = new boolean[initialFieldCount];
   }

   /** Called when an application modifies a value. (Both objects and
       classes) */
   public static void alert(Object o, int fieldID){

      // Lock the alerts table.
      synchronized (alerts){

         FieldAlert alert = (FieldAlert) alerts.get(o);

         if (alert == null){      // This object hasn't been alerted already,
                                  // so add to alerts table.
            alert = new FieldAlert(o, fieldID + 1);
            alerts.put(o, alert);
         }

         if (fieldID >= alert.fieldAlerts.length){
            // Ok, enlarge fieldAlerts array.
            boolean[] b = new boolean[fieldID+1];
            System.arraycopy(alert.fieldAlerts, 0, b, 0,
               alert.fieldAlerts.length);
            alert.fieldAlerts = b;
         }

         // Record the alert.
         alert.fieldAlerts[fieldID] = true;

         // Mark as pending.
         FieldSend.pending = true;       // Signal that there is one or more
                                         // propagations waiting.

         // Finally, notify the waiting FieldSend thread(s)
         if (FieldSend.waiting){
            FieldSend.waiting = false;
            alerts.notify();
         }

      }

   }
```

```
}
```

A9. The ninth excerpt is the source-code of FieldSend, which propagates changes values alerted to it via FieldAlert.

```
import java.lang.*;
import java.lang.reflect.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class FieldSend implements Runnable{

   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int PROPAGATE_OBJECT = 10;
   public final static int PROPAGATE_CLASS = 20;

   /** FieldAlert network values. */
   public final static String group =
      System.getProperty("FieldAlert_network_group");
   public final static int port =
      Integer.parseInt(System.getProperty("FieldAlert_network_port"));

   /** Table of global ID's for local objects. (hashcode-to-globalID
       mappings) */
   public final static Hashtable objectToGlobalID = new Hashtable();

   /** Table of global ID's for local classnames. (classname-to-globalID
       mappings) */
   public final static Hashtable classNameToGlobalID = new Hashtable();

   /** Pending. True if a propagation is pending. */
   public static boolean pending = false;

   /** Waiting. True if the FieldSend thread(s) are waiting. */
   public static boolean waiting = false;

   /** Background send thread. Propagates values as this thread is alerted
       to their alteration. */
   public void run(){

      System.out.println("FieldAlert_network_group=" + group);
      System.out.println("FieldAlert_network_port=" + port);

      try{

         // Create a DatagramSocket to send propagated field values.
         DatagramSocket datagramSocket =
            new DatagramSocket(port, InetAddress.getByName(group));

         // Next, create the buffer and packet for all transmissions.
         byte[] buffer = new byte[512];        // Working limit of 512 bytes
                                               // per packet.
         DatagramPacket datagramPacket =
            new DatagramPacket(buffer, 0, buffer.length);

         while (!Thread.interrupted()){

            Object[] entries = null;
```

```
// Lock the alerts table.
synchronized (FieldAlert.alerts){

   // Await for an alert to propagate something.
   while (!pending){
      waiting = true;
      FieldAlert.alerts.wait();
      waiting = false;
   }

   pending = false;

   entries = FieldAlert.alerts.entrySet().toArray();

   // Clear alerts once we have copied them.
   FieldAlert.alerts.clear();

}

// Process each object alert in turn.
for (int i=0; i<entries.length; i++){

   FieldAlert alert = (FieldAlert) entries[i];

   int index = 0;
   datagramPacket.setLength(buffer.length);

   Object reference = null;
   if (alert.reference instanceof String){
      // PROPAGATE_CLASS field operation.

      buffer[index++] = (byte) ((PROPAGATE_CLASS >> 24) & 0xff);
      buffer[index++] = (byte) ((PROPAGATE_CLASS >> 16) & 0xff);
      buffer[index++] = (byte) ((PROPAGATE_CLASS >> 8) & 0xff);
      buffer[index++] = (byte) ((PROPAGATE_CLASS >> 0) & 0xff);

      String name = (String) alert.reference;
      int length = name.length();
      buffer[index++] = (byte) ((length >> 24) & 0xff);
      buffer[index++] = (byte) ((length >> 16) & 0xff);
      buffer[index++] = (byte) ((length >> 8) & 0xff);
      buffer[index++] = (byte) ((length >> 0) & 0xff);

      byte[] bytes = name.getBytes();
      System.arraycopy(bytes, 0, buffer, index, length);
      index += length;

   }else{                    // PROPAGATE_OBJECT field operation.

      buffer[index++] =
         (byte) ((PROPAGATE_OBJECT >> 24) & 0xff);
      buffer[index++] =
         (byte) ((PROPAGATE_OBJECT >> 16) & 0xff);
      buffer[index++] = (byte) ((PROPAGATE_OBJECT >> 8) & 0xff);
      buffer[index++] = (byte) ((PROPAGATE_OBJECT >> 0) & 0xff);

      int globalID = ((Integer)
         objectToGlobalID.get(alert.reference)).intValue();

      buffer[index++] = (byte) ((globalID >> 24) & 0xff);
      buffer[index++] = (byte) ((globalID >> 16) & 0xff);
      buffer[index++] = (byte) ((globalID >> 8) & 0xff);
      buffer[index++] = (byte) ((globalID >> 0) & 0xff);
```

```
            reference = alert.reference;

        }

        // Use reflection to get a table of fields that correspond to
        // the field indexes used internally.
        Field[] fields = null;
        if (reference == null){
            fields = FieldLoader.loadClass((String)
                alert.reference).getDeclaredFields();
        }else{
            fields = alert.reference.getClass().getDeclaredFields();
        }

        // Now encode in batch mode the fieldID/value pairs.
        for (int j=0; j<alert.fieldAlerts.length; j++){

            if (alert.fieldAlerts[j] == false)
                continue;

            buffer[index++] = (byte) ((j >> 24) & 0xff);
            buffer[index++] = (byte) ((j >> 16) & 0xff);
            buffer[index++] = (byte) ((j >> 8) & 0xff);
            buffer[index++] = (byte) ((j >> 0) & 0xff);

            // Encode value.
            Class type = fields[j].getType();
            if (type == Boolean.TYPE){
                buffer[index++] =(byte)
                    (fields[j].getBoolean(reference)? 1 : 0);
            }else if (type == Byte.TYPE){
                buffer[index++] = fields[j].getByte(reference);
            }else if (type == Short.TYPE){
                short v = fields[j].getShort(reference);
                buffer[index++] = (byte) ((v >> 8) & 0xff);
                buffer[index++] = (byte) ((v >> 0) & 0xff);
            }else if (type == Character.TYPE){
                char v = fields[j].getChar(reference);
                buffer[index++] = (byte) ((v >> 8) & 0xff);
                buffer[index++] = (byte) ((v >> 0) & 0xff);
            }else if (type == Integer.TYPE){
                int v = fields[j].getInt(reference);
                buffer[index++] = (byte) ((v >> 24) & 0xff);
                buffer[index++] = (byte) ((v >> 16) & 0xff);
                buffer[index++] = (byte) ((v >> 8) & 0xff);
                buffer[index++] = (byte) ((v >> 0) & 0xff);
            }else if (type == Float.TYPE){
                int v = Float.floatToIntBits(
                    fields[j].getFloat(reference));
                buffer[index++] = (byte) ((v >> 24) & 0xff);
                buffer[index++] = (byte) ((v >> 16) & 0xff);
                buffer[index++] = (byte) ((v >> 8) & 0xff);
                buffer[index++] = (byte) ((v >> 0) & 0xff);
            }else if (type == Long.TYPE){
                long v = fields[j].getLong(reference);
                buffer[index++] = (byte) ((v >> 56) & 0xff);
                buffer[index++] = (byte) ((v >> 48) & 0xff);
                buffer[index++] = (byte) ((v >> 40) & 0xff);
                buffer[index++] = (byte) ((v >> 32) & 0xff);
                buffer[index++] = (byte) ((v >> 24) & 0xff);
                buffer[index++] = (byte) ((v >> 16) & 0xff);
                buffer[index++] = (byte) ((v >> 8) & 0xff);
                buffer[index++] = (byte) ((v >> 0) & 0xff);
            }else if (type == Double.TYPE){
                long v = Double.doubleToLongBits(
```

```
                        fields[j].getDouble(reference));
                    buffer[index++] = (byte) ((v >> 56) & 0xff);
                    buffer[index++] = (byte) ((v >> 48) & 0xff);
                    buffer[index++] = (byte) ((v >> 40) & 0xff);
                    buffer[index++] = (byte) ((v >> 32) & 0xff);
                    buffer[index++] = (byte) ((v >> 24) & 0xff);
                    buffer[index++] = (byte) ((v >> 16) & 0xff);
                    buffer[index++] = (byte) ((v >> 8) & 0xff);
                    buffer[index++] = (byte) ((v >> 0) & 0xff);
                 }else{
                    throw new AssertionError("Unsupported type.");
                 }


              }

              // Now set the length of the datagrampacket.
              datagramPacket.setLength(index);

              // Now send the packet.
              datagramSocket.send(datagramPacket);

           }

        }

     }catch (Exception e){
        throw new AssertionError("Exception: " + e.toString());
     }

  }

}
```

A10. The tenth excerpt is the source-code of FieldReceive, which receives propagated changed values sent via FieldSend.

```
import java.lang.*;
import java.lang.reflect.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class FieldReceive implements Runnable{

   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int PROPAGATE_OBJECT = 10;
   public final static int PROPAGATE_CLASS = 20;

   /** FieldAlert network values. */
   public final static String group =
      System.getProperty("FieldAlert_network_group");
   public final static int port =
      Integer.parseInt(System.getProperty("FieldAlert_network_port"));

   /** Table of global ID's for local objects. (globalID-to-hashcode
       mappings) */
   public final static Hashtable globalIDToObject = new Hashtable();

   /** Table of global ID's for local classnames. (globalID-to-classname
       mappings) */
```

```
public final static Hashtable globalIDToClassName = new Hashtable();

/** Called when an application is to acquire a lock. */
public void run(){

   System.out.println("FieldAlert_network_group=" + group);
   System.out.println("FieldAlert_network_port=" + port);

   try{

      // Create a DatagramSocket to send propagated field values from
      MulticastSocket multicastSocket = new MulticastSocket(port);
      multicastSocket.joinGroup(InetAddress.getByName(group));

      // Next, create the buffer and packet for all transmissions.
      byte[] buffer = new byte[512];              // Working limit of 512
                                                  // bytes per packet.
      DatagramPacket datagramPacket =
         new DatagramPacket(buffer, 0, buffer.length);

      while (!Thread.interrupted()){

         // Make sure to reset length.
         datagramPacket.setLength(buffer.length);

         // Receive the next available packet.
         multicastSocket.receive(datagramPacket);

         int index = 0, length = datagramPacket.getLength();

         // Decode the command.
         int command = (int) (((buffer[index++] & 0xff) << 24)
            | ((buffer[index++] & 0xff) << 16)
            | ((buffer[index++] & 0xff) << 8)
            | (buffer[index++] & 0xff));

         if (command == PROPAGATE_OBJECT){  // Propagate operation for
                                            // object fields.

            // Decode global id.
            int globalID = (int) (((buffer[index++] & 0xff) << 24)
               | ((buffer[index++] & 0xff) << 16)
               | ((buffer[index++] & 0xff) << 8)
               | (buffer[index++] & 0xff));

            // Now, need to resolve the object in question.
            Object reference = globalIDToObject.get(
               new Integer(globalID));

            // Next, get the array of fields for this object.
            Field[] fields = reference.getClass().getDeclaredFields();

            while (index < length){

               // Decode the field id.
               int fieldID = (int) (((buffer[index++] & 0xff) << 24)
                  | ((buffer[index++] & 0xff) << 16)
                  | ((buffer[index++] & 0xff) << 8)
                  | (buffer[index++] & 0xff));

               // Determine value length based on corresponding field
               // type.
               Field field = fields[fieldID];
               Class type = field.getType();
               if (type == Boolean.TYPE){
```

```
                boolean v = (buffer[index++] == 1 ? true : false);
                field.setBoolean(reference, v);
            }else if (type == Byte.TYPE){
                byte v = buffer[index++];
                field.setByte(reference, v);
            }else if (type == Short.TYPE){
                short v = (short) (((buffer[index++] & 0xff) << 8)
                    | (buffer[index++] & 0xff));
                field.setShort(reference, v);
            }else if (type == Character.TYPE){
                char v = (char) (((buffer[index++] & 0xff) << 8)
                    | (buffer[index++] & 0xff));
                field.setChar(reference, v);
            }else if (type == Integer.TYPE){
                int v = (int) (((buffer[index++] & 0xff) << 24)
                    | ((buffer[index++] & 0xff) << 16)
                    | ((buffer[index++] & 0xff) << 8)
                    | (buffer[index++] & 0xff));
                field.setInt(reference, v);
            }else if (type == Float.TYPE){
                int v = (int) (((buffer[index++] & 0xff) << 24)
                    | ((buffer[index++] & 0xff) << 16)
                    | ((buffer[index++] & 0xff) << 8)
                    | (buffer[index++] & 0xff));
                field.setFloat(reference, Float.intBitsToFloat(v));
            }else if (type == Long.TYPE){
                long v = (long) (((buffer[index++] & 0xff) << 56)
                    | ((buffer[index++] & 0xff) << 48)
                    | ((buffer[index++] & 0xff) << 40)
                    | ((buffer[index++] & 0xff) << 32)
                    | ((buffer[index++] & 0xff) << 24)
                    | ((buffer[index++] & 0xff) << 16)
                    | ((buffer[index++] & 0xff) << 8)
                    | (buffer[index++] & 0xff));
                field.setLong(reference, v);
            }else if (type == Double.TYPE){
                long v = (long) (((buffer[index++] & 0xff) << 56)
                    | ((buffer[index++] & 0xff) << 48)
                    | ((buffer[index++] & 0xff) << 40)
                    | ((buffer[index++] & 0xff) << 32)
                    | ((buffer[index++] & 0xff) << 24)
                    | ((buffer[index++] & 0xff) << 16)
                    | ((buffer[index++] & 0xff) << 8)
                    | (buffer[index++] & 0xff));
                field.setDouble(reference, Double.longBitsToDouble(v));
            }else{
                throw new AssertionError("Unsupported type.");
            }

        }

    }else if (command == PROPAGATE_CLASS){    // Propagate an update
                                              // to class fields.

        // Decode the classname.
        int nameLength = (int) (((buffer[index++] & 0xff) << 24)
            | ((buffer[index++] & 0xff) << 16)
            | ((buffer[index++] & 0xff) << 8)
            | (buffer[index++] & 0xff));
        String name = new String(buffer, index, nameLength);
        index += nameLength;

        // Next, get the array of fields for this class.
        Field[] fields =
            FieldLoader.loadClass(name).getDeclaredFields();
```

```
// Decode all batched fields included in this propagation
// packet.
while (index < length){

   // Decode the field id.
   int fieldID = (int) (((buffer[index++] & 0xff) << 24)
      | ((buffer[index++] & 0xff) << 16)
      | ((buffer[index++] & 0xff) << 8)
      | (buffer[index++] & 0xff));

   // Determine field type to determine value length.
   Field field = fields[fieldID];
   Class type = field.getType();
   if (type == Boolean.TYPE){
      boolean v = (buffer[index++] == 1 ? true : false);
      field.setBoolean(null, v);
   }else if (type == Byte.TYPE){
      byte v = buffer[index++];
      field.setByte(null, v);
   }else if (type == Short.TYPE){
      short v = (short) (((buffer[index++] & 0xff) << 8)
         | (buffer[index++] & 0xff));
      field.setShort(null, v);
   }else if (type == Character.TYPE){
      char v = (char) (((buffer[index++] & 0xff) << 8)
         | (buffer[index++] & 0xff));
      field.setChar(null, v);
   }else if (type == Integer.TYPE){
      int v = (int) (((buffer[index++] & 0xff) << 24)
         | ((buffer[index++] & 0xff) << 16)
         | ((buffer[index++] & 0xff) << 8)
         | (buffer[index++] & 0xff));
      field.setInt(null, v);
   }else if (type == Float.TYPE){
      int v = (int) (((buffer[index++] & 0xff) << 24)
         | ((buffer[index++] & 0xff) << 16)
         | ((buffer[index++] & 0xff) << 8)
         | (buffer[index++] & 0xff));
      field.setFloat(null, Float.intBitsToFloat(v));
   }else if (type == Long.TYPE){
      long v = (long) (((buffer[index++] & 0xff) << 56)
         | ((buffer[index++] & 0xff) << 48)
         | ((buffer[index++] & 0xff) << 40)
         | ((buffer[index++] & 0xff) << 32)
         | ((buffer[index++] & 0xff) << 24)
         | ((buffer[index++] & 0xff) << 16)
         | ((buffer[index++] & 0xff) << 8)
         | (buffer[index++] & 0xff));
      field.setLong(null, v);
   }else if (type == Double.TYPE){
      long v = (long) (((buffer[index++] & 0xff) << 56)
         | ((buffer[index++] & 0xff) << 48)
         | ((buffer[index++] & 0xff) << 40)
         | ((buffer[index++] & 0xff) << 32)
         | ((buffer[index++] & 0xff) << 24)
         | ((buffer[index++] & 0xff) << 16)
         | ((buffer[index++] & 0xff) << 8)
         | (buffer[index++] & 0xff));
      field.setDouble(null, Double.longBitsToDouble(v));
   }else{          // Unsupported field type.
      throw new AssertionError("Unsupported type.");
   }

}
```

```
            }

         }

      }catch (Exception e){
         throw new AssertionError("Exception: " + e.toString());
      }

   }

}
```

A11. **FieldLoader.java**

This excerpt is the source-code of FieldLoader, which modifies an application as it is being loaded.

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class FieldLoader extends URLClassLoader{

   public FieldLoader(URL[] urls){
      super(urls);
   }

   protected Class findClass(String name)
   throws ClassNotFoundException{

      ClassFile cf = null;

      try{

         BufferedInputStream in =
            new BufferedInputStream(findResource(
            name.replace('.', '/').concat(".class")).openStream());

         cf = new ClassFile(in);

      }catch (Exception e){throw new ClassNotFoundException(e.toString());}

      // Class-wide pointers to the ldc and alert index.
      int ldcindex = -1;
      int alertindex = -1;

      for (int i=0; i<cf.methods_count; i++){
         for (int j=0; j<cf.methods[i].attributes_count; j++){
            if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
               continue;

            Code_attribute ca = (Code_attribute) cf.methods[i].attributes[j];

            boolean changed = false;

            for (int z=0; z<ca.code.length; z++){
               if ((ca.code[z][0] & 0xff) == 179){    // Opcode for a PUTSTATIC
                                                      // instruction.
                  changed = true;

                  // The code below only supports fields in this class.
                  // Thus, first off, check that this field is local to this
                  // class.
                  CONSTANT_Fieldref_info fi = (CONSTANT_Fieldref_info)
```

```
   cf.constant_pool[(int) (((ca.code[z][1] & 0xff) << 8) |
   (ca.code[z][2] & 0xff))];
CONSTANT_Class_info ci = (CONSTANT_Class_info)
   cf.constant_pool[fi.class_index];
String className =
   cf.constant_pool[ci.name_index].toString();
if (!name.equals(className)){
   throw new AssertionError("This code only supports fields "
      "local to this class");
}

// Ok, now search for the fields name and index.
int index = 0;
CONSTANT_NameAndType_info ni = (CONSTANT_NameAndType_info)
   cf.constant_pool[fi.name_and_type_index];
String fieldName =
   cf.constant_pool[ni.name_index].toString();
for (int a=0; a<cf.fields_count; a++){
   String fn = cf.constant_pool[
      cf.fields[a].name_index].toString();
   if (fieldName.equals(fn)){
      index = a;
      break;
   }
}

// Next, realign the code array, making room for the
// insertions.
byte[][] code2 = new byte[ca.code.length+3][];
System.arraycopy(ca.code, 0, code2, 0, z+1);
System.arraycopy(ca.code, z+1, code2, z+4,
   ca.code.length-(z+1));
ca.code = code2;

// Next, insert the LDC_W instruction.
if (ldcindex == -1){
   CONSTANT_String_info csi =
      new CONSTANT_String_info(ci.name_index);
   cp_info[] cpi = new cp_info[cf.constant_pool.length+1];
   System.arraycopy(cf.constant_pool, 0, cpi, 0,
      cf.constant_pool.length);
   cpi[cpi.length - 1] = csi;
   ldcindex = cpi.length-1;
   cf.constant_pool = cpi;
   cf.constant_pool_count++;
}
ca.code[z+1] = new byte[3];
ca.code[z+1][0] = (byte) 19;
ca.code[z+1][1] = (byte) ((ldcindex >> 8) & 0xff);
ca.code[z+1][2] = (byte) (ldcindex & 0xff);


// Next, insert the SIPUSH instruction.
ca.code[z+2] = new byte[3];
ca.code[z+2][0] = (byte) 17;
ca.code[z+2][1] = (byte) ((index >> 8) & 0xff);
ca.code[z+2][2] = (byte) (index & 0xff);

// Finally, insert the INVOKESTATIC instruction.
if (alertindex == -1){
   // This is the first time this class is encountering the
   // alert instruction, so have to add it to the constant
   // pool.
   cp_info[] cpi = new cp_info[cf.constant_pool.length+6];
```

```
                    System.arraycopy(cf.constant_pool, 0, cpi, 0,
                       cf.constant_pool.length);
                    cf.constant_pool = cpi;
                    cf.constant_pool_count += 6;

                    CONSTANT_Utf8_info u1 =
                       new CONSTANT_Utf8_info("FieldAlert");
                    cf.constant_pool[cf.constant_pool.length-6] = u1;

                    CONSTANT_Class_info c1 = new CONSTANT_Class_info(
                       cf.constant_pool_count-6);
                    cf.constant_pool[cf.constant_pool.length-5] = c1;

                    u1 = new CONSTANT_Utf8_info("alert");
                    cf.constant_pool[cf.constant_pool.length-4] = u1;

                    u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;I)V");
                    cf.constant_pool[cf.constant_pool.length-3] = u1;

                    CONSTANT_NameAndType_info n1 =
                       new CONSTANT_NameAndType_info(
                       cf.constant_pool.length-4, cf.constant_pool.length-3);
                    cf.constant_pool[cf.constant_pool.length-2] = n1;

                    CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
                       cf.constant_pool.length-5, cf.constant_pool.length-2);
                    cf.constant_pool[cf.constant_pool.length-1] = m1;
                    alertindex = cf.constant_pool.length-1;
                 }
                 ca.code[z+3] = new byte[3];
                 ca.code[z+3][0] = (byte) 184;
                 ca.code[z+3][1] = (byte) ((alertindex >> 8) & 0xff);
                 ca.code[z+3][2] = (byte) (alertindex & 0xff);

                 // And lastly, increase the CODE_LENGTH and ATTRIBUTE_LENGTH
                 // values.
                 ca.code_length += 9;
                 ca.attribute_length += 9;

              }

           }

           // If we changed this method, then increase the stack size by one.
           if (changed){
              ca.max_stack++;          // Just to make sure.
           }

        }
     }


     try{

        ByteArrayOutputStream out = new ByteArrayOutputStream();

        cf.serialize(out);

        byte[] b = out.toByteArray();

        return defineClass(name, b, 0, b.length);

     }catch (Exception e){
        throw new ClassNotFoundException(name);
     }
```

```
    }

}
```

A12. **Attribute_info.java**

Convience class for representing attribute_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** This abstract class represents all types of attribute_info
 *  that are used in the JVM specifications.
 *
 *  All new attribute_info subclasses are to always inherit from this
 *  class.
 */
public abstract class attribute_info{

    public int attribute_name_index;
    public int attribute_length;

    /** This is used by subclasses to register themselves
     *  to their parent classFile.
     */
    attribute_info(ClassFile cf){}

    /** Used during input serialization by ClassFile only. */
    attribute_info(ClassFile cf, DataInputStream in)
        throws IOException{
        attribute_name_index = in.readChar();
        attribute_length = in.readInt();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeChar(attribute_name_index);
        out.writeInt(attribute_length);
    }

    /** This class represents an unknown attribute_info that
     *  this current version of classfile specification does
     *  not understand.
     */
    public final static class Unknown extends attribute_info{

        byte[] info;

        /** Used during input serialization by ClassFile only. */
        Unknown(ClassFile cf, DataInputStream in)
            throws IOException{
            super(cf, in);
            info = new byte[attribute_length];
            in.read(info, 0, attribute_length);
        }

        /** Used during output serialization by ClassFile only. */
        void serialize(DataOutputStream out)
            throws IOException{
            ByteArrayOutputStream baos = new ByteArrayOutputStream();

            super.serialize(out);
            out.write(info, 0, attribute_length);
        }
```

```
    }
}
```

A13. **ClassFile.java**

Convience class for representing ClassFile structures.

```
import java.lang.*;
import java.io.*;
import java.util.*;

/** The ClassFile follows verbatim from the JVM specification.  */
public final class ClassFile {

    public int magic;
    public int minor_version;
    public int major_version;
    public int constant_pool_count;
    public cp_info[] constant_pool;
    public int access_flags;
    public int this_class;
    public int super_class;
    public int interfaces_count;
    public int[] interfaces;
    public int fields_count;
    public field_info[] fields;
    public int methods_count;
    public method_info[] methods;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Constructor. Takes in a byte stream representation and transforms
     *  each of the attributes in the ClassFile into objects to allow for
     *  easier manipulation.
     */
    public ClassFile(InputStream ins)
        throws IOException{
        DataInputStream in = (ins instanceof DataInputStream ?
            (DataInputStream) ins : new DataInputStream(ins));
        magic = in.readInt();
        minor_version = in.readChar();
        major_version = in.readChar();
        constant_pool_count = in.readChar();
        constant_pool = new cp_info[constant_pool_count];
        for (int i=1; i<constant_pool_count; i++){
            in.mark(1);
            int s = in.read();
            in.reset();
            switch (s){
                case 1:
                    constant_pool[i] = new CONSTANT_Utf8_info(this, in);
                    break;
                case 3:
                    constant_pool[i] = new CONSTANT_Integer_info(this, in);
                    break;
                case 4:
                    constant_pool[i] = new CONSTANT_Float_info(this, in);
                    break;
                case 5:
                    constant_pool[i] = new CONSTANT_Long_info(this, in);
                    i++;
                    break;
                case 6:
                    constant_pool[i] = new CONSTANT_Double_info(this, in);
```

```
                    i++;
                    break;
                case 7:
                    constant_pool[i] = new CONSTANT_Class_info(this, in);
                    break;
                case 8:
                    constant_pool[i] = new CONSTANT_String_info(this, in);
                    break;
                case 9:
                    constant_pool[i] = new CONSTANT_Fieldref_info(this, in);
                    break;
                case 10:
                    constant_pool[i] = new CONSTANT_Methodref_info(this, in);
                    break;
                case 11:
                    constant_pool[i] =
                        new CONSTANT_InterfaceMethodref_info(this, in);
                    break;
                case 12:
                    constant_pool[i] = new CONSTANT_NameAndType_info(this, in);
                    break;
                default:
                    throw new ClassFormatError("Invalid ConstantPoolTag");
            }
        }
        access_flags = in.readChar();
        this_class = in.readChar();
        super_class = in.readChar();
        interfaces_count = in.readChar();
        interfaces = new int[interfaces_count];
        for (int i=0; i<interfaces_count; i++)
            interfaces[i] = in.readChar();
        fields_count = in.readChar();
        fields = new field_info[fields_count];
        for (int i=0; i<fields_count; i++) {
            fields[i] = new field_info(this, in);
        }
        methods_count = in.readChar();
        methods = new method_info[methods_count];
        for (int i=0; i<methods_count; i++) {
            methods[i] = new method_info(this, in);
        }
        attributes_count = in.readChar();
        attributes = new attribute_info[attributes_count];
        for (int i=0; i<attributes_count; i++){
            in.mark(2);
            String s = constant_pool[in.readChar()].toString();
            in.reset();
            if (s.equals("SourceFile"))
                attributes[i] = new SourceFile_attribute(this, in);
            else if (s.equals("Deprecated"))
                attributes[i] = new Deprecated_attribute(this, in);
            else if (s.equals("InnerClasses"))
                attributes[i] = new InnerClasses_attribute(this, in);
            else
                attributes[i] = new attribute_info.Unknown(this, in);
        }
    }

    /** Serializes the ClassFile object into a byte stream. */
    public void serialize(OutputStream o)
        throws IOException{
        DataOutputStream out = (o instanceof DataOutputStream ?
            (DataOutputStream) o : new DataOutputStream(o));
        out.writeInt(magic);
```

```
        out.writeChar(minor_version);
        out.writeChar(major_version);
        out.writeChar(constant_pool_count);
        for (int i=1; i<constant_pool_count; i++){
            constant_pool[i].serialize(out);
            if (constant_pool[i] instanceof CONSTANT_Long_info ||
                    constant_pool[i] instanceof CONSTANT_Double_info)
                i++;
        }
        out.writeChar(access_flags);
        out.writeChar(this_class);
        out.writeChar(super_class);
        out.writeChar(interfaces_count);
        for (int i=0; i<interfaces_count; i++)
            out.writeChar(interfaces[i]);
        out.writeChar(fields_count);
        for (int i=0; i<fields_count; i++)
            fields[i].serialize(out);
        out.writeChar(methods_count);
        for (int i=0; i<methods_count; i++)
            methods[i].serialize(out);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
        // Flush the outputstream just to make sure.
        out.flush();
    }

}
```

A14. **Code_attribute.java**

Convience class for representing Code_attribute structures within ClassFiles.

```
import java.util.*;
import java.lang.*;
import java.io.*;

/**
 *  The code[] is stored as a 2D array.  */
public final class Code_attribute extends attribute_info{

    public int max_stack;
    public int max_locals;
    public int code_length;
    public byte[][] code;
    public int exception_table_length;
    public exception_table[] exception_table;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Internal class that handles the exception table. */
    public final static class exception_table{
        public int start_pc;
        public int end_pc;
        public int handler_pc;
        public int catch_type;
    }

    /** Constructor called only by method_info. */
    Code_attribute(ClassFile cf, int ani, int al, int ms, int ml, int cl,
                   byte[][] cd, int etl, exception_table[] et, int ac,
                   attribute_info[] a){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
```

```
        max_stack = ms;
        max_locals = ml;
        code_length = cl;
        code = cd;
        exception_table_length = etl;
        exception_table = et;
        attributes_count = ac;
        attributes = a;
    }

    /** Used during input serialization by ClassFile only. */
    Code_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        max_stack = in.readChar();
        max_locals = in.readChar();
        code_length = in.readInt();
        code = new byte[code_length][];

        int i = 0;
        for (int pos=0; pos<code_length; i++){
            in.mark(1);
            int s = in.read();
            in.reset();
            switch (s){
                case 16:
                case 18:
                case 21:
                case 22:
                case 23:
                case 24:
                case 25:
                case 54:
                case 55:
                case 56:
                case 57:
                case 58:
                case 169:
                case 188:
                case 196:
                    code[i] = new byte[2];
                    break;
                case 17:
                case 19:
                case 20:
                case 132:
                case 153:
                case 154:
                case 155:
                case 156:
                case 157:
                case 158:
                case 159:
                case 160:
                case 161:
                case 162:
                case 163:
                case 164:
                case 165:
                case 166:
                case 167:
                case 168:
                case 178:
                case 179:
                case 180:
```

```
                case 181:
                case 182:
                case 183:
                case 184:
                case 187:
                case 189:
                case 192:
                case 193:
                case 198:
                case 199:
                case 209:
                    code[i] = new byte[3];
                    break;
                case 197:
                    code[i] = new byte[4];
                    break;
                case 185:
                case 200:
                case 201:
                    code[i] = new byte[5];
                    break;
                case 170:{
                    int pad = 3 - (pos % 4);
                    in.mark(pad+13);                                    // highbyte
                    in.skipBytes(pad+5);                                // lowbyte
                    int low = in.readInt();
                    code[i] =
                        new byte[pad + 13 + ((in.readInt() - low + 1) * 4)];
                    in.reset();
                    break;
                }case 171:{
                    int pad = 3 - (pos % 4);
                    in.mark(pad+9);
                    in.skipBytes(pad+5);
                    code[i] = new byte[pad + 9 + (in.readInt() * 8)];
                    in.reset();
                    break;
                }default:
                    code[i] = new byte[1];
            }
            in.read(code[i], 0, code[i].length);
            pos += code[i].length;
        }

        // adjust the array to the new size and store the size
        byte[][] temp = new byte[i][];
        System.arraycopy(code, 0, temp, 0, i);
        code = temp;

        exception_table_length = in.readChar();
        exception_table =
            new Code_attribute.exception_table[exception_table_length];
        for (i=0; i<exception_table_length; i++){
            exception_table[i] = new exception_table();
            exception_table[i].start_pc = in.readChar();
            exception_table[i].end_pc = in.readChar();
            exception_table[i].handler_pc = in.readChar();
            exception_table[i].catch_type = in.readChar();
        }
        attributes_count = in.readChar();
        attributes = new attribute_info[attributes_count];
        for (i=0; i<attributes_count; i++){
            in.mark(2);
            String s = cf.constant_pool[in.readChar()].toString();
            in.reset();
```

```
            if (s.equals("LineNumberTable"))
                attributes[i] = new LineNumberTable_attribute(cf, in);
            else if (s.equals("LocalVariableTable"))
                attributes[i] = new LocalVariableTable_attribute(cf, in);
            else
                attributes[i] = new attribute_info.Unknown(cf, in);
        }
    }

    /** Used during output serialization by ClassFile only.
    */
    void serialize(DataOutputStream out)
        throws IOException{
            attribute_length = 12 + code_length +
                (exception_table_length * 8);
            for (int i=0; i<attributes_count; i++)
                attribute_length += attributes[i].attribute_length + 6;
            super.serialize(out);
            out.writeChar(max_stack);
            out.writeChar(max_locals);
            out.writeInt(code_length);
            for (int i=0, pos=0; pos<code_length; i++){
                out.write(code[i], 0, code[i].length);
                pos += code[i].length;
            }
            out.writeChar(exception_table_length);
            for (int i=0; i<exception_table_length; i++){
                out.writeChar(exception_table[i].start_pc);
                out.writeChar(exception_table[i].end_pc);
                out.writeChar(exception_table[i].handler_pc);
                out.writeChar(exception_table[i].catch_type);
            }
            out.writeChar(attributes_count);
            for (int i=0; i<attributes_count; i++)
                attributes[i].serialize(out);
    }

}
```

A15. **CONSTANT_Class_info.java**

Convience class for representing CONSTANT_Class_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Class subtype of a constant pool entry.  */
public final class CONSTANT_Class_info extends cp_info{

    /** The index to the name of this class. */
    public int name_index = 0;

    /** Convenience constructor.
     */
    public CONSTANT_Class_info(int index) {
        tag = 7;
        name_index = index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Class_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 7)
            throw new ClassFormatError();
        name_index = in.readChar();
```

```
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(name_index);
    }

}
```

A16. **CONSTANT_Double_info.java**

Convience class for representing CONSTANT_Double_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Double subtype of a constant pool entry. */
public final class CONSTANT_Double_info extends cp_info{

    /** The actual value. */
    public double bytes;

    public CONSTANT_Double_info(double d){
        tag = 6;
        bytes = d;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Double_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 6)
            throw new ClassFormatError();
        bytes = in.readDouble();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeDouble(bytes);
        long l = Double.doubleToLongBits(bytes);
    }
}
```

A17. **CONSTANT_Fieldref_info.java**

Convience class for representing CONSTANT_Fieldref_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Fieldref subtype of a constant pool entry.  */
public final class CONSTANT_Fieldref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field if referencing to. */
    public int name_and_type_index;

    /** Convenience constructor. */
    public CONSTANT_Fieldref_info(int class_index, int name_and_type_index) {
        tag = 9;
```

```
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Fieldref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 9)
            throw new ClassFormatError();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
        out.writeChar(name_and_type_index);
    }

}
```

A18. **CONSTANT_Float_info.java**

Convience class for representing CONSTANT_Float_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Float subtype of a constant pool entry. */
public final class CONSTANT_Float_info extends cp_info{

    /** The actual value. */
    public float bytes;

    public CONSTANT_Float_info(float f){
        tag = 4;
        bytes = f;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Float_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 4)
            throw new ClassFormatError();
        bytes = in.readFloat();
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(4);
        out.writeFloat(bytes);
    }

}
```

A19. **CONSTANT_Integer_info.java**

Convience class for representing CONSTANT_Integer_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;
```

```
/** Integer subtype of a constant pool entry.  */
public final class CONSTANT_Integer_info extends cp_info{

    /** The actual value. */
    public int bytes;

    public CONSTANT_Integer_info(int b) {
        tag = 3;
        bytes = b;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Integer_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 3)
            throw new ClassFormatError();
        bytes = in.readInt();
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeInt(bytes);
    }


}
```

A20. **CONSTANT_InterfaceMethodref_info.java**

Convience class for representing CONSTANT_InterfaceMethodref_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** InterfaceMethodref subtype of a constant pool entry.
 */
public final class CONSTANT_InterfaceMethodref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field if referencing to. */
    public int name_and_type_index;

    public CONSTANT_InterfaceMethodref_info(int class_index,
                                            int name_and_type_index) {
        tag = 11;
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }


    /** Used during input serialization by ClassFile only. */
    CONSTANT_InterfaceMethodref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 11)
            throw new ClassFormatError();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }
```

```
    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
        out.writeChar(name_and_type_index);
    }

}
```

A21. **CONSTANT_Long_info.java**

Convience class for representing CONSTANT_Long_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Long subtype of a constant pool entry.  */
public final class CONSTANT_Long_info extends cp_info{

    /** The actual value. */
    public long bytes;

    public CONSTANT_Long_info(long b){
        tag = 5;
        bytes = b;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Long_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 5)
            throw new ClassFormatError();
        bytes = in.readLong();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeLong(bytes);
    }

}
```

A22. **CONSTANT_Methodref_info.java**

Convience class for representing CONSTANT_Methodref_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Methodref subtype of a constant pool entry.
 */
public final class CONSTANT_Methodref_info extends cp_info{

    /** The index to the class that this field is referencing to. */
    public int class_index;

    /** The name and type index this field if referencing to. */
    public int name_and_type_index;
```

```
    public CONSTANT_Methodref_info(int class_index, int name_and_type_index) {
        tag = 10;
        this.class_index = class_index;
        this.name_and_type_index = name_and_type_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_Methodref_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 10)
            throw new ClassFormatError();
        class_index = in.readChar();
        name_and_type_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(class_index);
        out.writeChar(name_and_type_index);
    }

}
```

A23. **CONSTANT_NameAndType_info.java**

Convience class for representing CONSTANT_NameAndType_info structures within ClassFiles.

```
import java.io.*;
import java.lang.*;

/** NameAndType subtype of a constant pool entry.
 */
public final class CONSTANT_NameAndType_info extends cp_info{

    /** The index to the Utf8 that contains the name. */
    public int name_index;

    /** The index fo the Utf8 that constains the signature. */
    public int descriptor_index;

    public CONSTANT_NameAndType_info(int name_index, int descriptor_index) {
        tag = 12;
        this.name_index = name_index;
        this.descriptor_index = descriptor_index;
    }

    /** Used during input serialization by ClassFile only. */
    CONSTANT_NameAndType_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 12)
            throw new ClassFormatError();
        name_index = in.readChar();
        descriptor_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
```

```
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
    }

}
```

A24. **CONSTANT_String_info.java**

Convience class for representing CONSTANT_String_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** String subtype of a constant pool entry.
 */
public final class CONSTANT_String_info extends cp_info{

    /** The index to the actual value of the string. */
    public int string_index;

    public CONSTANT_String_info(int value) {
        tag = 8;
        string_index = value;
    }

    /** ONLY TO BE USED BY CLASSFILE! */
    public CONSTANT_String_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 8)
            throw new ClassFormatError();
        string_index = in.readChar();
    }

    /** Output serialization, ONLY TO BE USED BY CLASSFILE! */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(string_index);
    }

}
```

A25. **CONSTANT_Utf8_info.java**

Convience class for representing CONSTANT_Utf8_info structures within ClassFiles.

```
import java.io.*;
import java.lang.*;

/** Utf8 subtype of a constant pool entry.
 *  We internally represent the Utf8 info byte array
 *  as a String.
 */
public final class CONSTANT_Utf8_info extends cp_info{

    /** Length of the byte array. */
    public int length;

    /** The actual bytes, represented by a String. */
    public String bytes;

    /** This constructor should be used for the purpose
     *  of part creation. It does not set the parent
     *  ClassFile reference.
     */
```

```
    public CONSTANT_Utf8_info(String s) {
        tag = 1;
        length = s.length();
        bytes = s;
    }

    /** Used during input serialization by ClassFile only. */
    public CONSTANT_Utf8_info(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        if (tag != 1)
            throw new ClassFormatError();
        length = in.readChar();
        byte[] b = new byte[length];
        in.read(b, 0, length);

        // WARNING: String constructor is deprecated.
        bytes = new String(b, 0, length);
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeByte(tag);
        out.writeChar(length);

        // WARNING: Handling of String coversion here might be problematic.
        out.writeBytes(bytes);
    }

    public String toString(){
        return bytes;
    }


}
```

A26. ConstantValue_attribute.java

Convience class for representing ConstantValue_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Attribute that allows for initialization of static variables in
 *  classes. This attribute will only reside in a field_info struct.
 */

public final class ConstantValue_attribute extends attribute_info{

    public int constantvalue_index;

    public ConstantValue_attribute(ClassFile cf, int ani, int al, int cvi){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        constantvalue_index = cvi;
    }

    public ConstantValue_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        constantvalue_index = in.readChar();
    }
```

```
    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2;
        super.serialize(out);
        out.writeChar(constantvalue_index);
    }

}
```

A27. **cp_info.java**

Convience class for representing cp_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Represents the common interface of all constant pool parts
 *  that all specific constant pool items must inherit from.
 *
 */
public abstract class cp_info{

    /** The type tag that signifies what kind of constant pool
     *  item it is */
    public int tag;

    /** Used for serialization of the object back into a bytestream. */
    abstract void serialize(DataOutputStream out) throws IOException;

    /** Default constructor. Simply does nothing. */
    public cp_info() {}

    /** Constructor simply takes in the ClassFile as a reference to
     *  it's parent
     */
    public cp_info(ClassFile cf) {}

    /** Used during input serialization by ClassFile only. */
    cp_info(ClassFile cf, DataInputStream in)
        throws IOException{
        tag = in.readUnsignedByte();
    }


}
```

A28. **Deprecated_attribute.java**

Convience class for representing Deprecated_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A fix attributed that can be located either in the ClassFile,
 *  field_info or the method_info attribute. Mark deprecated to
 *  indicate that the method, class or field has been superceded.
 */
public final class Deprecated_attribute extends attribute_info{

    public Deprecated_attribute(ClassFile cf, int ani, int al){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
    }


    /** Used during input serialization by ClassFile only. */
```

```
    Deprecated_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
    }

}
```

A29. **Exceptions_attribute.java**

Convience class for representing Exceptions_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** This is the struct where the exceptions table are located.
 *  <br><br>
 *  This attribute can only appear once in a method_info struct.
 */
public final class Exceptions_attribute extends attribute_info{

    public int number_of_exceptions;
    public int[] exception_index_table;

    public Exceptions_attribute(ClassFile cf, int ani, int al, int noe,
                                int[] eit){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        number_of_exceptions = noe;
        exception_index_table = eit;
    }

    /** Used during input serialization by ClassFile only. */
    Exceptions_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        number_of_exceptions = in.readChar();
        exception_index_table = new int[number_of_exceptions];
        for (int i=0; i<number_of_exceptions; i++)
            exception_index_table[i] = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (number_of_exceptions*2);
        super.serialize(out);
        out.writeChar(number_of_exceptions);
        for (int i=0; i<number_of_exceptions; i++)
            out.writeChar(exception_index_table[i]);
    }

}
```

A30. **field_info.java**

Convience class for representing field_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/**   Represents the field_info structure as specified in the JVM specification.
 */
public final class field_info{

    public int access_flags;
    public int name_index;
```

```
    public int descriptor_index;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Convenience constructor. */
    public field_info(ClassFile cf, int flags, int ni, int di){
        access_flags = flags;
        name_index = ni;
        descriptor_index = di;
        attributes_count = 0;
        attributes = new attribute_info[0];
    }

    /** Constructor called only during the serialization process.
     *  <br><br>
     *  This is intentionally left as package protected as we
     *  should not normally call this constructor directly.
     *  <br><br>
     *  Warning: the handling of len is not correct (after String s =...)
     */
    field_info(ClassFile cf, DataInputStream in)
        throws IOException{
        access_flags = in.readChar();
        name_index = in.readChar();
        descriptor_index = in.readChar();
        attributes_count = in.readChar();
        attributes = new attribute_info[attributes_count];
        for (int i=0; i<attributes_count; i++){
            in.mark(2);
            String s = cf.constant_pool[in.readChar()].toString();
            in.reset();
            if (s.equals("ConstantValue"))
                attributes[i] = new ConstantValue_attribute(cf, in);
            else if (s.equals("Synthetic"))
                attributes[i] = new Synthetic_attribute(cf, in);
            else if (s.equals("Deprecated"))
                attributes[i] = new Deprecated_attribute(cf, in);
            else
                attributes[i] = new attribute_info.Unknown(cf, in);
        }
    }

    /** To serialize the contents into the output format.
     */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeChar(access_flags);
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
    }

}
```

A31. **InnerClasses_attribute.java**

Convience class for representing InnerClasses_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A variable length structure that contains information about an
 *  inner class of this class.
 */
```

```
public final class InnerClasses_attribute extends attribute_info{

    public int number_of_classes;
    public classes[] classes;

    public final static class classes{
        int inner_class_info_index;
        int outer_class_info_index;
        int inner_name_index;
        int inner_class_access_flags;
    }

    public InnerClasses_attribute(ClassFile cf, int ani, int al,
                                  int noc, classes[] c){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        number_of_classes = noc;
        classes = c;
    }

    /** Used during input serialization by ClassFile only. */
    InnerClasses_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        number_of_classes = in.readChar();
        classes = new InnerClasses_attribute.classes[number_of_classes];
        for (int i=0; i<number_of_classes; i++){
            classes[i] = new classes();
            classes[i].inner_class_info_index = in.readChar();
            classes[i].outer_class_info_index = in.readChar();
            classes[i].inner_name_index = in.readChar();
            classes[i].inner_class_access_flags = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    public void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (number_of_classes * 8);
        super.serialize(out);
        out.writeChar(number_of_classes);
        for (int i=0; i<number_of_classes; i++){
            out.writeChar(classes[i].inner_class_info_index);
            out.writeChar(classes[i].outer_class_info_index);
            out.writeChar(classes[i].inner_name_index);
            out.writeChar(classes[i].inner_class_access_flags);
        }
    }

}
```

A32. **LineNumberTable_attribute.java**

Convience class for representing LineNumberTable_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Determines which line of the binary code relates to the
 *  corresponding source code.
 */
public final class LineNumberTable_attribute extends attribute_info{
```

```
    public int line_number_table_length;
    public line_number_table[] line_number_table;

    public final static class line_number_table{
        int start_pc;
        int line_number;
    }

    public LineNumberTable_attribute(ClassFile cf, int ani, int al, int lntl,
                                     line_number_table[] lnt){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        line_number_table_length = lntl;
        line_number_table = lnt;
    }

    /** Used during input serialization by ClassFile only. */
    LineNumberTable_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        line_number_table_length = in.readChar();
        line_number_table = new
LineNumberTable_attribute.line_number_table[line_number_table_length];
        for (int i=0; i<line_number_table_length; i++){
            line_number_table[i] = new line_number_table();
            line_number_table[i].start_pc = in.readChar();
            line_number_table[i].line_number = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (line_number_table_length * 4);
        super.serialize(out);
        out.writeChar(line_number_table_length);
        for (int i=0; i<line_number_table_length; i++){
            out.writeChar(line_number_table[i].start_pc);
            out.writeChar(line_number_table[i].line_number);
        }
    }

}
```

A33. **LocalVariableTable_attribute.java**

Convience class for representing LocalVariableTable_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** Used by debugger to find out how the source file line number is linked
 *  to the binary code. It has many to one correspondence and is found in
 *  the Code_attribute.
 */
public final class LocalVariableTable_attribute extends attribute_info{

    public int local_variable_table_length;
    public local_variable_table[] local_variable_table;

    public final static class local_variable_table{
        int start_pc;
        int length;
```

```
        int name_index;
        int descriptor_index;
        int index;
    }

    public LocalVariableTable_attribute(ClassFile cf, int ani, int al,
                                        int lvtl, local_variable_table[] lvt){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        local_variable_table_length = lvtl;
        local_variable_table = lvt;
    }

    /** Used during input serialization by ClassFile only. */
    LocalVariableTable_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        local_variable_table_length = in.readChar();
        local_variable_table = new
LocalVariableTable_attribute.local_variable_table[local_variable_table_length];
        for (int i=0; i<local_variable_table_length; i++){
            local_variable_table[i] = new local_variable_table();
            local_variable_table[i].start_pc = in.readChar();
            local_variable_table[i].length = in.readChar();
            local_variable_table[i].name_index = in.readChar();
            local_variable_table[i].descriptor_index = in.readChar();
            local_variable_table[i].index = in.readChar();
        }
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2 + (local_variable_table_length * 10);
        super.serialize(out);
        out.writeChar(local_variable_table_length);
        for (int i=0; i<local_variable_table_length; i++){
            out.writeChar(local_variable_table[i].start_pc);
            out.writeChar(local_variable_table[i].length);
            out.writeChar(local_variable_table[i].name_index);
            out.writeChar(local_variable_table[i].descriptor_index);
            out.writeChar(local_variable_table[i].index);
        }
    }

}
```

A34. **method_info.java**

Convience class for representing method_info structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** This follows the method_info in the JVM specification.
 */
public final class method_info {

    public int access_flags;
    public int name_index;
    public int descriptor_index;
    public int attributes_count;
    public attribute_info[] attributes;

    /** Constructor. Creates a method_info, initializes it with
```

```
     *  the flags set, and the name and descriptor indexes given.
     *  A new uninitialized code attribute is also created, and stored
     *  in the <i>code</i> variable.*/
    public method_info(ClassFile cf, int flags, int ni, int di,
                       int ac, attribute_info[] a) {
        access_flags = flags;
        name_index = ni;
        descriptor_index = di;
        attributes_count = ac;
        attributes = a;
    }

    /** This method creates a method_info from the current pointer in the
     *  data stream. Only called by during the serialization of a complete
     *  ClassFile from a bytestream, not normally invoked directly.
     */
    method_info(ClassFile cf, DataInputStream in)
        throws IOException{
        access_flags = in.readChar();
        name_index = in.readChar();
        descriptor_index = in.readChar();
        attributes_count = in.readChar();
        attributes = new attribute_info[attributes_count];
        for (int i=0; i<attributes_count; i++){
            in.mark(2);
            String s = cf.constant_pool[in.readChar()].toString();
            in.reset();
            if (s.equals("Code"))
                attributes[i] = new Code_attribute(cf, in);
            else if (s.equals("Exceptions"))
                attributes[i] = new Exceptions_attribute(cf, in);
            else if (s.equals("Synthetic"))
                attributes[i] = new Synthetic_attribute(cf, in);
            else if (s.equals("Deprecated"))
                attributes[i] = new Deprecated_attribute(cf, in);
            else
                attributes[i] = new attribute_info.Unknown(cf, in);
        }
    }

    /** Output serialization of the method_info to a byte array.
     *  Not normally invoked directly.
    */
    public void serialize(DataOutputStream out)
        throws IOException{
        out.writeChar(access_flags);
        out.writeChar(name_index);
        out.writeChar(descriptor_index);
        out.writeChar(attributes_count);
        for (int i=0; i<attributes_count; i++)
            attributes[i].serialize(out);
    }

}
```

A35. **SourceFile_attribute.java**

Convience class for representing SourceFile_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A SourceFile attribute is an optional fixed_length attribute in
 *  the attributes table. Only located in the ClassFile struct only
```

```
 *  once.
 */
public final class SourceFile_attribute extends attribute_info{

    public int sourcefile_index;

    public SourceFile_attribute(ClassFile cf, int ani, int al, int sfi){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
        sourcefile_index = sfi;
    }

    /** Used during input serialization by ClassFile only. */
    SourceFile_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
        sourcefile_index = in.readChar();
    }

    /** Used during output serialization by ClassFile only. */
    void serialize(DataOutputStream out)
        throws IOException{
        attribute_length = 2;
        super.serialize(out);
        out.writeChar(sourcefile_index);
    }

}
```

A36. **Synthetic_attribute.java**

Convience class for representing Synthetic_attribute structures within ClassFiles.

```
import java.lang.*;
import java.io.*;

/** A synthetic attribute indicates that this class does not have
 *  a generated code source. It is likely to imply that the code
 *  is generated by machine means rather than coded directly. This
 *  attribute can appear in the classfile, method_info or field_info.
 *  It is fixed length.
 */
public final class Synthetic_attribute extends attribute_info{

    public Synthetic_attribute(ClassFile cf, int ani, int al){
        super(cf);
        attribute_name_index = ani;
        attribute_length = al;
    }

    /** Used during output serialization by ClassFile only. */
    Synthetic_attribute(ClassFile cf, DataInputStream in)
        throws IOException{
        super(cf, in);
    }

}
```

ANNEXURE B

B1

Method <clinit>

0 new #2 <Class test>

```
        3 dup
        4 invokespecial #3 <Method test()>
        7 putstatic #4 <Field test thisTest>
       10 return
B2
Method <clinit>
  0 invokestatic #3 <Method boolean isAlreadyLoaded()>
  3 ifeq 7
  6 return
  7 new #5 <Class test>
 10 dup
 11 invokespecial #6 <Method test()>
 14 putstatic #7 <Field test thisTest>
 17 return
B3
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 aload_0
  5 invokestatic #2 <Method long currentTimeMillis()>
  8 putfield #3 <Field long timestamp>
 11 return
B4
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
  4 invokestatic #2 <Method boolean isAlreadyLoaded()>
  7 ifeq 11
 10 return
 11 aload_0
 12 invokestatic #4 <Method long currentTimeMillis()>
 15 putfield #5 <Field long timestamp>
 18 return
B5
Method <clinit>
  0 ldc #2 <String "test">
  2 invokestatic #3 <Method boolean isAlreadyLoaded(java.lang.String)>
  5 ifeq 9
  8 return
  9 new #5 <Class test>
 12 dup
 13 invokespecial #6 <Method test()>
 16 putstatic #7 <Field test thisTest>
 19 return
B6
Method <init>
  0 aload_0
  1 invokespecial #1 <Method java.lang.Object()>
```

4 aload_0
5 invokestatic #2 <Method boolean isAlreadyLoaded(java.lang.Object)>
8 ifeq 12
11 return
12 aload_0
13 invokestatic #4 <Method long currentTimeMillis()>
16 putfield #5 <Field long timestamp>
19 return

ANNEXURE B7

This excerpt is the source-code of InitClient, which queries an "initialisation server" for the initialisation status of the relevant class or object.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class InitClient{

   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int INITIALIZE_CLASS = 10;
   public final static int INITIALIZE_OBJECT = 20;

   /** InitServer network values. */
   public final static String serverAddress =
      System.getProperty("InitServer_network_address");
   public final static int serverPort =
      Integer.parseInt(System.getProperty("InitServer_network_port"));

   /** Table of global ID's for local objects. (hashcode-to-globalID
       mappings) */
   public final static Hashtable hashCodeToGlobalID = new Hashtable();

   /** Called when a object is being initialized. */
   public static boolean isAlreadyLoaded(Object o){

      // First of all, we need to resolve the globalID
      // for object 'o'.  To do this we use the hashCodeToGlobalID
      // table.
      int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

      try{

         // Next, we want to connect to the InitServer, which will inform us
         // of the initialization status of this object.
         Socket socket = new Socket(serverAddress, serverPort);
         DataOutputStream out =
            new DataOutputStream(socket.getOutputStream());
         DataInputStream in =
            new DataInputStream(socket.getInputStream());

         // Ok, now send the serialized request to the InitServer.
         out.writeInt(INITIALIZE_OBJECT);
         out.writeInt(globalID);
         out.flush();

         // Now wait for the reply.
         int status = in.readInt();      // This is a blocking call.  So we
                                         // will wait until the remote side
```

```
                                        // sends something.

            if (status == NACK){
               throw new AssertionError(
                  "Negative acknowledgement. Request failed.");
            }else if (status != ACK){
               throw new AssertionError("Unknown acknowledgement: "
                  + status + ". Request failed.");
            }

            // Next, read in a 32bit argument which is the count of previous
            // initializations.
            int count = in.readInt();

            // If the count is equal to 0, then this is the first
            // initialization, and hence isAlreadyLoaded should be false.
            // If however, the count is greater than 0, then this is already
            // initialized, and thus isAlreadyLoaded should be true.
            boolean isAlreadyLoaded = (count == 0 ? false : true);

            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // Return the value of the isAlreadyLoaded variable.
            return isAlreadyLoaded;

         }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
         }
      }

      /** Called when a class is being initialized. */
      public static boolean isAlreadyLoaded(String name){

         try{

            // First of all, we want to connect to the InitServer, which will
            // inform us of the initialization status of this class.
            Socket socket = new Socket(serverAddress, serverPort);
            DataOutputStream out =
               new DataOutputStream(socket.getOutputStream());
            DataInputStream in =
               new DataInputStream(socket.getInputStream());

            // Ok, now send the serialized request to the InitServer.
            out.writeInt(INITIALIZE_CLASS);
            out.writeInt(name.length());          // A 32bit length argument of
                                                  // the String name.
            out.write(name.getBytes(), 0, name.length());      // The byte-
                                                               // encoded
                                                               // String name.

            out.flush();

            // Now wait for the reply.
            int status = in.readInt();      // This is a blocking call. So we
                                            // will wait until the remote side
                                            // sends something.

            if (status == NACK){
               throw new AssertionError(
```

```
                "Negative acknowledgement. Request failed.");
            }else if (status != ACK){
               throw new AssertionError("Unknown acknowledgement: "
                  + status + ". Request failed.");
            }

            // Next, read in a 32bit argument which is the count of the
            // previous intializations.
            int count = in.readInt();

            // If the count is equal to 0, then this is the first
            // initialization, and hence isAlreadyLoaded should be false.
            // If however, the count is greater than 0, then this is already
            // loaded, and thus isAlreadyLoaded should be true.
            boolean isAlreadyLoaded = (count == 0 ? false : true);

            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // Return the value of the isAlreadyLoaded variable.
            return isAlreadyLoaded;

         }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
         }
      }

   }
```

ANNEXURE B8

This excerpt is the source-code of InitServer, which receives an initialisation status query by InitClient and in response returns the corresponding status.

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class InitServer implements Runnable{

   /** Protocol specific values */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int INITIALIZE_CLASS = 10;
   public final static int INITIALIZE_OBJECT= 20;
   /** InitServer network values. */
   public final static int serverPort = 20001;
   /** Table of initialization records. */
   public final static Hashtable initializations = new Hashtable();

   /** Private input/output objects. */
   private Socket socket = null;
   private DataOutputStream outputStream;
   private DataInputStream inputStream;
   private String address;

   public static void main(String[] s)
   throws Exception{
```

```
        System.out.println("InitServer_network_address="
           + InetAddress.getLocalHost().getHostAddress());
        System.out.println("InitServer_network_port=" + serverPort);

        // Create a serversocket to accept incoming initialization operation
        // connections.
        ServerSocket serverSocket = new ServerSocket(serverPort);

        while (!Thread.interrupted()){

           // Block until an incoming initialization operation connection.
           Socket socket = serverSocket.accept();

           // Create a new instance of InitServer to manage this
           // initialization operation connection.
           new Thread(new InitServer(socket)).start();

        }

     }

     /** Constructor. Initialize this new InitServer instance with necessary
         resources for operation. */
     public InitServer(Socket s){
        socket = s;
        try{
           outputStream = new DataOutputStream(s.getOutputStream());
           inputStream = new DataInputStream(s.getInputStream());
           address = s.getInetAddress().getHostAddress();
        }catch (IOException e){
           throw new AssertionError("Exception: " + e.toString());
        }
     }
     /** Main code body. Decode incoming initialization operation requests and
         execute accordingly. */
     public void run(){

        try{

           // All commands are implemented as 32bit integers.
           // Legal commands are listed in the "protocol specific values"
           // fields above.
           int command = inputStream.readInt();

           // Continue processing commands until a CLOSE operation.
           while (command != CLOSE){

              if (command == INITIALIZE_CLASS){          // This is an
                                                         // INITIALIZE_CLASS
                                                         // operation.

                 // Read in a 32bit length field 'l', and a String name for
                 // this class of length 'l'.
                 int length = inputStream.readInt();
                 byte[] b = new byte[length];
                 inputStream.read(b, 0, b.length);
                 String className = new String(b, 0, length);

                 // Synchronize on the initializations table in order to
                 // ensure thread-safety.
                 synchronized (initializations){

                    // Locate the previous initializations entry for this
                    // class, if any.
                    Integer entry = (Integer) initializations.get(className);
```

```
            if (entry == null){    // This is an unknown class so
                                   // update the table with a
                                   // corresponding entry.

               initializations.put(className, new Integer(1));

               // Send a positive acknowledgement to InitClient,
               // together with the count of previous initializations
               // of this class - which in this case of an unknown
               // class must be 0.
               outputStream.writeInt(ACK);
               outputStream.writeInt(0);
               outputStream.flush();

            }else{                 // This is a known class, so update
                                   // the count of initializations.

               initializations.put(className,
                  new Integer(entry.intValue() + 1));

               // Send a positive acknowledgement to InitClient,
               // together with the count of previous initializtions
               // of this class - which in this case of a known class
               // must be the value of "entry.intValue()".
               outputStream.writeInt(ACK);
               outputStream.writeInt(entry.intValue());
               outputStream.flush();

            }

         }

      }else if (command == INITIALIZE_OBJECT){ // This is an
                                               // INITIALIZE_OBJECT
                                               // operation.

         // Read in the globalID of the object to be initialized.
         int globalID = inputStream.readInt();

         // Synchronize on the initializations table in order to
         // ensure thread-safety.
         synchronized (initializations){

            // Locate the previous initializations entry for this
            // object, if any.
            Integer entry = (Integer) initializations.get(
               new Integer(globalID));

            if (entry == null){    // This is an unknown object so
                                   // update the table with a
                                   // corresponding entry.

               initializations.put(new Integer(globalID),
                  new Integer(1));

               // Send a positive acknowledgement to InitClient,
               // together with the count of previous initializations
               // of this object - which in this case of an unknown
               // object must be 0.
               outputStream.writeInt(ACK);
               outputStream.writeInt(0);
               outputStream.flush();

            }else{          // This is a known object so update the
```

```
                                    // count of initializations.

                        initializations.put(new Integer(globalID),
                           new Integer(entry.intValue() + 1));

                        // Send a positive acknowledgement to InitClient,
                        // together with the count of previous initializations
                        // of this object - which in this case of a known
                        // object must be value "entry.intValue()".
                        outputStream.writeInt(ACK);
                        outputStream.writeInt(entry.intValue());
                        outputStream.flush();

                     }

                  }

               }else{          // Unknown command.
                  throw new AssertionError(
                     "Unknown command. Operation failed.");
               }

               // Read in the next command.
               command = inputStream.readInt();

            }

         }catch (Exception e){
            throw new AssertionError("Exception: " + e.toString());
         }finally{
            try{
               // Closing down. Cleanup this connection.
               outputStream.flush();
               outputStream.close();
               inputStream.close();
               socket.close();
            }catch (Throwable t){
               t.printStackTrace();
            }
            // Garbage these references.
            outputStream = null;
            inputStream = null;
            socket = null;
         }
      }

   }
```

ANNEXURE B9

This excerpt is the source-code of the example application used in the before/after examples of Annexure B.

```
import java.lang.*;

public class example{

   /** Shared static field. */
   public static example currentExample;

   /** Shared instance field. */
   public long timestamp;

   /** Static intializer. (clinit) */
   static{
```

```
        currentExample = new example();

    }

    /** Instance intializer (init) */
    public example(){

        timestamp = System.currentTimeMillis();

    }

}
```

ANNEXURE B10

InitLoader.java

This excerpt is the source-code of InitLoader, which modifies an application as it is being loaded.

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class InitLoader extends URLClassLoader{

    public InitLoader(URL[] urls){
        super(urls);
    }

    protected Class findClass(String name)
    throws ClassNotFoundException{

        ClassFile cf = null;

        try{
            BufferedInputStream in = new
                BufferedInputStream(findResource(name.replace('.',
                '/').concat(".class")).openStream());

            cf = new ClassFile(in);

        }catch (Exception e){throw new ClassNotFoundException(e.toString());}

        for (int i=0; i<cf.methods_count; i++){

            // Find the <clinit> method_info struct.
            String methodName = cf.constant_pool[
                cf.methods[i].name_index].toString();
            if (!methodName.equals("<clinit>")){
                continue;
            }

            // Now find the Code_attribute for the <clinit> method.
            for (int j=0; j<cf.methods[i].attributes_count; j++){
                if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
                    continue;

                Code_attribute ca = (Code_attribute) cf.methods[i].attributes[j];

                // First, shift the code[] down by 4 instructions.
                byte[][] code2 = new byte[ca.code.length+4][];
                System.arraycopy(ca.code, 0, code2, 4, ca.code.length);
                ca.code = code2;
```

```
            // Then enlarge the constant_pool by 7 items.
            cp_info[] cpi = new cp_info[cf.constant_pool.length+7];
            System.arraycopy(cf.constant_pool, 0, cpi, 0,
               cf.constant_pool.length);
            cf.constant_pool = cpi;
            cf.constant_pool_count += 7;

            // Now add the constant pool items for these instructions, starting
            // with String.
            CONSTANT_String_info csi = new CONSTANT_String_info(

        ((CONSTANT_Class_info)cf.constant_pool[cf.this_class]).name_index);
            cf.constant_pool[cf.constant_pool.length-7] = csi;

            // Now add the UTF for class.
            CONSTANT_Utf8_info u1 = new CONSTANT_Utf8_info("InitClient");
            cf.constant_pool[cf.constant_pool.length-6] = u1;

            // Now add the CLASS for the previous UTF.
            CONSTANT_Class_info c1 =
               new CONSTANT_Class_info(cf.constant_pool.length-6);
            cf.constant_pool[cf.constant_pool.length-5] = c1;

            // Next add the first UTF for NameAndType.
            u1 = new CONSTANT_Utf8_info("isAlreadyLoaded");
            cf.constant_pool[cf.constant_pool.length-4] = u1;

            // Next add the second UTF for NameAndType.
            u1 = new CONSTANT_Utf8_info("(Ljava/lang/String;)Z");
            cf.constant_pool[cf.constant_pool.length-3] = u1;

            // Next add the NameAndType for the previous two UTFs.
            CONSTANT_NameAndType_info n1 = new CONSTANT_NameAndType_info(
               cf.constant_pool.length-4, cf.constant_pool.length-3);
            cf.constant_pool[cf.constant_pool.length-2] = n1;

            // Next add the Methodref for the previous CLASS and NameAndType.
            CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
               cf.constant_pool.length-5, cf.constant_pool.length-2);
            cf.constant_pool[cf.constant_pool.length-1] = m1;

            // Now with that done, add the instructions into the code, starting
            // with LDC.
            ca.code[0] = new byte[3];
            ca.code[0][0] = (byte) 19;
            ca.code[0][1] = (byte) (((cf.constant_pool.length-7) >> 8) & 0xff);
            ca.code[0][2] = (byte) ((cf.constant_pool.length-7) & 0xff);

            // Now Add the INVOKESTATIC instruction.
            ca.code[1] = new byte[3];
            ca.code[1][0] = (byte) 184;
            ca.code[1][1] = (byte) (((cf.constant_pool.length-1) >> 8) & 0xff);
            ca.code[1][2] = (byte) ((cf.constant_pool.length-1) & 0xff);

            // Next add the IFEQ instruction.
            ca.code[2] = new byte[3];
            ca.code[2][0] = (byte) 153;
            ca.code[2][1] = (byte) ((4 >> 8) & 0xff);
            ca.code[2][2] = (byte) (4 & 0xff);

            // Finally, add the RETURN instruction.
            ca.code[3] = new byte[1];
            ca.code[3][0] = (byte) 177;

            // Lastly, increment the CODE_LENGTH and ATTRIBUTE_LENGTH values.
```

```
                ca.code_length += 10;
                ca.attribute_length += 10;

            }
        }

        try{
            ByteArrayOutputStream out = new ByteArrayOutputStream();
            cf.serialize(out);

            byte[] b = out.toByteArray();

            return defineClass(name, b, 0, b.length);

        }catch (Exception e){
         e.printStackTrace();
            throw new ClassNotFoundException(name);
        }

    }
}
```

Annexure C

C1. TYPICAL PRIOR ART FINALIZATION FOR A SINGLE MACHINE:

Method finalize()
0 getstatic #9 <Field java.io.PrintStream out>
3 ldc #24 <String "Deleted...">
5 invokevirtual #16 <Method void println(java.lang.String)>
8 return

C2. PREFERRED FINALIZATION FOR MULTIPLE MACHINES

Method finalize()
0 invokestatic #3 <Method boolean isLastReference()>
3 ifne 7
6 return
7 getstatic #9 <Field java.io.PrintStream out>
10 ldc #24 <String "Deleted...">
12 invokevirtual #16 <Method void println(java.lang.String)>
15 return

C3. PREFERRED FINALIZATION FOR MULTIPLE MACHINES (Alternative)

Method finalize()
0 aload_0
1 invokestatic #3 <Method boolean isLastReference(java.lang.Object)>
4 ifne 8
7 return
8 getstatic #9 <Field java.io.PrintStream out>
11 ldc #24 <String "Deleted...">
13 invokevirtual #16 <Method void println(java.lang.String)>
16 return

Annexure C4

```
import java.lang.*;

public class example{
```

```
   /** Finalize method. */
   protected void finalize() throws Throwable{

      // "Deleted..." is printed out when this object is garbaged.
      System.out.println("Deleted...");

   }
}
```

Annexure C5

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class FinalClient{

   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int FINALIZE_OBJECT = 10;

   /** FinalServer network values. */
   public final static String serverAddress =
      System.getProperty("FinalServer_network_address");
   public final static int serverPort =
      Integer.parseInt(System.getProperty("FinalServer_network_port"));

   /** Table of global ID's for local objects. (hashcode-to-globalID
       mappings) */
   public final static Hashtable hashCodeToGlobalID = new Hashtable();

   /** Called when a object is being finalized. */
   public static boolean isLastReference(Object o){

      // First of all, we need to resolve the globalID for object 'o'.
      // To do this we use the hashCodeToGlobalID table.
      int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

      try{

         // Next, we want to connect to the FinalServer, which will inform
         // us of the finalization status of this object.
         Socket socket = new Socket(serverAddress, serverPort);
         DataOutputStream out =
            new DataOutputStream(socket.getOutputStream());
         DataInputStream in = new DataInputStream(socket.getInputStream());

         // Ok, now send the serialized request to the FinalServer.
         out.writeInt(FINALIZE_OBJECT);
         out.writeInt(globalID);
         out.flush();

         // Now wait for the reply.
         int status = in.readInt();      // This is a blocking call.  So we
                                         // will wait until the remote side
                                         // sends something.

         if (status == NACK){
            throw new AssertionError(
               "Negative acknowledgement. Request failed.");
         }else if (status != ACK){
            throw new AssertionError("Unknown acknowledgement: "
```

```
                + status + ". Request failed.");
            }

            // Next, read in a 32bit argument which is the count of the
            // remaining finalizations
            int count = in.readInt();

            // If the count is equal to 1, then this is the last finalization,
            // and hence isLastReference should be true.
            // If however, the count is greater than 1, then this is not the
            // last finalization, and thus isLastReference should be false.
            boolean isLastReference = (count == 1 ? true : false);

            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // Return the value of the isLastReference variable.
            return isLastReference;

        }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
        }
    }
}
```

Annexure C6

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class FinalServer implements Runnable{

    /** Protocol specific values */
    public final static int CLOSE = -1;
    public final static int NACK = 0;
    public final static int ACK = 1;
    public final static int FINALIZE_OBJECT = 10;

    /** FinalServer network values. */
    public final static int serverPort = 20001;

    /** Table of finalization records. */
    public final static Hashtable finalizations = new Hashtable();

    /** Private input/output objects. */
    private Socket socket = null;
    private DataOutputStream outputStream;
    private DataInputStream inputStream;
    private String address;

    public static void main(String[] s)
    throws Exception{

        System.out.println("FinalServer network address="
            + InetAddress.getLocalHost().getHostAddress());
        System.out.println("FinalServer network port=" + serverPort);

        // Create a serversocket to accept incoming initialization operation
        // connections.
```

```
      ServerSocket serverSocket = new ServerSocket(serverPort);

      while (!Thread.interrupted()){

         // Block until an incoming initialization operation connection.
         Socket socket = serverSocket.accept();

         // Create a new instance of InitServer to manage this
         // initialization operation connection.
         new Thread(new FinalServer(socket)).start();

      }

   }

   /** Constructor. Initialize this new FinalServer instance with necessary
       resources for operation. */
   public FinalServer(Socket s){
      socket = s;
      try{
         outputStream = new DataOutputStream(s.getOutputStream());
         inputStream = new DataInputStream(s.getInputStream());
         address = s.getInetAddress().getHostAddress();
      }catch (IOException e){
         throw new AssertionError("Exception: " + e.toString());
      }
   }

   /** Main code body. Decode incoming finalization operation requests and
       execute accordingly. */
   public void run(){

      try{

         // All commands are implemented as 32bit integers.
         // Legal commands are listed in the "protocol specific values"
         // fields above.
         int command = inputStream.readInt();

         // Continue processing commands until a CLOSE operation.
         while (command != CLOSE){

            if (command == FINALIZE_OBJECT){          // This is a
                                                      // FINALIZE_OBJECT
                                                      // operation.

               // Read in the globalID of the object to be finalized.
               int globalID = inputStream.readInt();

               // Synchronize on the finalizations table in order to ensure
               // thread-safety.
               synchronized (finalizations){

                  // Locate the previous finalizations entry for this
                  // object, if any.
                  Integer entry = (Integer) finalizations.get(
                     new Integer(globalID));

                  if (entry == null){
                     throw new AssertionError("Unknown object.");
                  }else if (entry.intValue() < 1){
                     throw new AssertionError("Invalid count.");
                  }else if (entry.intValue() == 1){   // Count of 1 means
                                                      // this is the last
                                                      // reference, hence
```

```
                                                    // remove from table.
                        finalizations.remove(new Integer(globalID));

                        // Send a positive acknowledgement to FinalClient,
                        // together with the count of remaining references -
                        // which in this case is 1.
                        outputStream.writeInt(ACK);
                        outputStream.writeInt(1);
                        outputStream.flush();

                     }else{          // This is not the last remaining
                                     // reference, as count is greater than 1.
                                     // Decrement count by 1.

                        finalizations.put(new Integer(globalID),
                           new Integer(entry.intValue() - 1));

                        // Send a positive acknowledgement to FinalClient,
                        // together with the count of remaining references to
                        // this object - which in this case of must be value
                        // "entry.intValue()".
                        outputStream.writeInt(ACK);
                        outputStream.writeInt(entry.intValue());
                        outputStream.flush();

                     }

                  }

               }else{          // Unknown command.
                  throw new AssertionError(
                     "Unknown command. Operation failed.");
               }

               // Read in the next command.
               command = inputStream.readInt();

            }

         }catch (Exception e){
            throw new AssertionError("Exception: " + e.toString());
         }finally{
            try{
               // Closing down. Cleanup this connection.
               outputStream.flush();
               outputStream.close();
               inputStream.close();
               socket.close();
            }catch (Throwable t){
               t.printStackTrace();
            }
            // Garbage these references.
            outputStream = null;
            inputStream = null;
            socket = null;
         }
      }

}
```

ANNEXURE C7

FinalLoader.java

This excerpt is the source-code of FinalLoader, which modifies an application as it is being loaded.

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class FinalLoader extends URLClassLoader{

   public FinalLoader(URL[] urls){
      super(urls);
   }

   protected Class findClass(String name)
   throws ClassNotFoundException{

      ClassFile cf = null;

      try{
         BufferedInputStream in =
            new BufferedInputStream(findResource(name.replace('.',
            '/').concat(".class")).openStream());

         cf = new ClassFile(in);

      }catch (Exception e){throw new ClassNotFoundException(e.toString());}

      for (int i=0; i<cf.methods_count; i++){

         // Find the finalize method_info struct.
         String methodName = cf.constant_pool[
            cf.methods[i].name_index].toString();
         if (!methodName.equals("finalize")){
            continue;
         }

         // Now find the Code_attribute for the finalize method.
         for (int j=0; j<cf.methods[i].attributes_count; j++){
            if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
               continue;

            Code_attribute ca = (Code_attribute) cf.methods[i].attributes[j];

            // First, shift the code[] down by 4 instructions.
            byte[][] code2 = new byte[ca.code.length+4][];
            System.arraycopy(ca.code, 0, code2, 4, ca.code.length);
            ca.code = code2;

            // Then enlarge the constant_pool by 6 items.
            cp_info[] cpi = new cp_info[cf.constant_pool.length+6];
            System.arraycopy(cf.constant_pool, 0, cpi, 0,
               cf.constant_pool.length);
            cf.constant_pool = cpi;
            cf.constant_pool_count += 6;

            // Now add the UTF for class.
            CONSTANT_Utf8_info u1 = new CONSTANT_Utf8_info("FinalClient");
            cf.constant_pool[cf.constant_pool.length-6] = u1;

            // Now add the CLASS for the previous UTF.
            CONSTANT_Class_info c1 =
               new CONSTANT_Class_info(cf.constant_pool.length-6);
            cf.constant_pool[cf.constant_pool.length-5] = c1;

            // Next add the first UTF for NameAndType.
```

```
                u1 = new CONSTANT_Utf8_info("isLastReference");
                cf.constant_pool[cf.constant_pool.length-4] = u1;

                // Next add the second UTF for NameAndType.
                u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;)Z");
                cf.constant_pool[cf.constant_pool.length-3] = u1;

                // Next add the NameAndType for the previous two UTFs.
                CONSTANT_NameAndType_info n1 = new CONSTANT_NameAndType_info(
                   cf.constant_pool.length-4, cf.constant_pool.length-3);
                cf.constant_pool[cf.constant_pool.length-2] = n1;

                // Next add the Methodref for the previous CLASS and NameAndType.
                CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
                   cf.constant_pool.length-5, cf.constant_pool.length-2);
                cf.constant_pool[cf.constant_pool.length-1] = m1;

                // Now with that done, add the instructions into the code, starting
                // with LDC.
                ca.code[0] = new byte[1];
                ca.code[0][0] = (byte) 42;

                // Now Add the INVOKESTATIC instruction.
                ca.code[1] = new byte[3];
                ca.code[1][0] = (byte) 184;
                ca.code[1][1] = (byte) (((cf.constant_pool.length-1) >> 8) & 0xff);
                ca.code[1][2] = (byte) ((cf.constant_pool.length-1) & 0xff);

                // Next add the IFNE instruction.
                ca.code[2] = new byte[3];
                ca.code[2][0] = (byte) 154;
                ca.code[2][1] = (byte) ((4 >> 8) & 0xff);
                ca.code[2][2] = (byte) (4 & 0xff);

                // Finally, add the RETURN instruction.
                ca.code[3] = new byte[1];
                ca.code[3][0] = (byte) 177;

                // Lastly, increment the CODE_LENGTH and ATTRIBUTE_LENGTH values.
                ca.code_length += 8;
                ca.attribute_length += 8;

             }
          }

          try{

             ByteArrayOutputStream out = new ByteArrayOutputStream();
             cf.serialize(out);

             byte[] b = out.toByteArray();

             return defineClass(name, b, 0, b.length);

          }catch (Exception e){
           e.printStackTrace();
             throw new ClassNotFoundException(name);
          }
       }

    }
```

Annexure D1

```
Method void run()
  0 getstatic #2 <Field java.lang.Object LOCK>
  3 dup
  4 astore_1
  5 monitorenter
  6 getstatic #3 <Field int counter>
  9 iconst_1
 10 iadd
 11 putstatic #3 <Field int counter>
 14 aload_1
 15 monitorexit
 16 return
```

Annexure D2

```
Method void run()
  0 getstatic #2 <Field java.lang.Object LOCK>
  3 dup
  4 astore_1
  5 dup
  6 monitorenter
  7 invokestatic #23 <Method void acquireLock(java.lang.Object)>
 10 getstatic #3 <Field int counter>
 13 iconst_1
 14 iadd
 15 putstatic #3 <Field int counter>
 18 aload_1
 19 dup
 20 invokestatic #24 <Method void releaseLock(java.lang.Object)>
 23 monitorexit
 24 return
```

Annexure D3

```
import java.lang.*;

public class example{

   /** Shared static field. */
   public final static Object LOCK = new Object();

   /** Shared static field. */
   public static int counter = 0;

   /** Example method using synchronization. This method serves to
       illustrate the use of synchronization to implement thread-safe
       modification of a shared memory location by potentially multiple
       threads. */
   public void run(){

      // First acquire the lock, otherwise any memory writes we do will be
      // prone to race-conditions.
      synchronized (LOCK){

         // Now that we have acquired the lock, we can safely modify memory
         // in a thread-safe manner.
         counter++;
```

```
        }

    }

}
```

Annexure D4

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class LockClient{

   /** Protocol specific values. */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int ACQUIRE_LOCK = 10;
   public final static int RELEASE_LOCK = 20;
   /** LockServer network values. */
   public final static String serverAddress =
      System.getProperty("LockServer_network_address");
   public final static int serverPort =
      Integer.parseInt(System.getProperty("LockServer_network_port"));

   /** Table of global ID's for local objects. (hashcode-to-globalID
       mappings) */
   public final static Hashtable hashCodeToGlobalID = new Hashtable();

   /** Called when an application is to acquire a lock. */
   public static void acquireLock(Object o){

      // First of all, we need to resolve the globalID for object 'o'.
      // To do this we use the hashCodeToGlobalID table.
      int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

      try{

         // Next, we want to connect to the LockServer, which will grant us
         // the global lock.
         Socket socket = new Socket(serverAddress, serverPort);
         DataOutputStream out =
            new DataOutputStream(socket.getOutputStream());
         DataInputStream in = new DataInputStream(socket.getInputStream());

         // Ok, now send the serialized request to the lock server.
         out.writeInt(ACQUIRE_LOCK);
         out.writeInt(globalID);
         out.flush();

         // Now wait for the reply.
         int status = in.readInt();      // This is a blocking call.  So we
                                         // will wait until the remote side
                                         // sends something.

         if (status == NACK){
            throw new AssertionError(
               "Negative acknowledgement. Request failed.");
         }else if (status != ACK){
            throw new AssertionError("Unknown acknowledgement: "
               + status + ". Request failed.");
         }
```

```
            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // This is a good acknowledgement, thus we can return now because
            // global lock is now acquired.
            return;

         }catch (IOException e){
            throw new AssertionError("Exception: " + e.toString());
         }

      }

      /** Called when an application is to release a lock. */
      public static void releaseLock(Object o){

         // First of all, we need to resolve the globalID for object 'o'.
         // To do this we use the hashCodeToGlobalID table.
         int globalID = ((Integer) hashCodeToGlobalID.get(o)).intValue();

         try{

            // Next, we want to connect to the LockServer, which records us as
            // the owner of the global lock for object 'o'.
            Socket socket = new Socket(serverAddress, serverPort);
            DataOutputStream out =
               new DataOutputStream(socket.getOutputStream());
            DataInputStream in = new DataInputStream(socket.getInputStream());

            // Ok, now send the serialized request to the lock server.
            out.writeInt(RELEASE_LOCK);
            out.writeInt(globalID);
            out.flush();

            // Now wait for the reply.
            int status = in.readInt();      // This is a blocking call.  So we
                                            // will wait until the remote side
                                            // sends something.

            if (status == NACK){
               throw new AssertionError(
                  "Negative acknowledgement. Request failed.");
            }else if (status != ACK){
               throw new AssertionError("Unknown acknowledgement: "
                  + status + ". Request failed.");
            }

            // Close down the connection.
            out.writeInt(CLOSE);
            out.flush();
            out.close();
            in.close();

            socket.close();          // Make sure to close the socket.

            // This is a good acknowledgement, return because global lock is
            // now released.
            return;
```

```
        }catch (IOException e){
           throw new AssertionError("Exception: " + e.toString());
        }

    }

}
```

Annexure D5

```
import java.lang.*;
import java.util.*;
import java.net.*;
import java.io.*;

public class LockServer implements Runnable{

   /** Protocol specific values */
   public final static int CLOSE = -1;
   public final static int NACK = 0;
   public final static int ACK = 1;
   public final static int ACQUIRE_LOCK = 10;
   public final static int RELEASE_LOCK = 20;

   /** LockServer network values. */
   public final static int serverPort = 20001;

   /** Table of lock records. */
   public final static Hashtable locks = new Hashtable();

   /** Linked list of waiting LockManager objects. */
   public LockServer next = null;

   /** Address of remote LockClient. */
   public final String address;

   /** Private input/output objects. */
   private Socket socket = null;
   private DataOutputStream outputStream;
   private DataInputStream inputStream;

   public static void main(String[] s)
   throws Exception{

      System.out.println("LockServer network address="
         + InetAddress.getLocalHost().getHostAddress());
      System.out.println("LockServer network port=" + serverPort);

      // Create a serversocket to accept incoming lock operation
      // connections.
      ServerSocket serverSocket = new ServerSocket(serverPort);

      while (!Thread.interrupted()){

         // Block until an incoming lock operation connection.
         Socket socket = serverSocket.accept();

         // Create a new instance of LockServer to manage this lock
         // operation connection.
         new Thread(new LockServer(socket)).start();

      }

   }
```

```
/** Constructor. Initialise this new LockServer instance with necessary
    resources for operation. */
public LockServer(Socket s){
   socket = s;
   try{
      outputStream = new DataOutputStream(s.getOutputStream());
      inputStream = new DataInputStream(s.getInputStream());
      address = s.getInetAddress().getHostAddress();
   }catch (IOException e){
      throw new AssertionError("Exception: " + e.toString());
   }
}

/** Main code body. Decode incoming lock operation requests and execute
    accordingly. */
public void run(){

   try{

      // All commands are implemented as 32bit integers.
      // Legal commands are listed in the "protocol specific values"
      // fields above.
      int command = inputStream.readInt();

      // Continue processing commands until a CLOSE operation.
      while (command != CLOSE){

         if (command == ACQUIRE_LOCK){              // This is an
                                                    // ACQUIRE_LOCK
                                                    // operation.

            // Read in the globalID of the object to be locked.
            int globalID = inputStream.readInt();

            // Synchronize on the locks table in order to ensure thread-
            // safety.
            synchronized (locks){

               // Check for an existing owner of this lock.
               LockServer lock = (LockServer) locks.get(
                  new Integer(globalID));

               if (lock == null){     // No-one presently owns this lock,
                                      // so acquire it.

                  locks.put(new Integer(globalID), this);

                  acquireLock();      // Signal to the client the
                                      // successful acquisition of this
                                      // lock.

               }else{                 // Already owned. Append ourselves
                                      // to end of queue.

                  // Search for the end of the queue. (Implemented as
                  // linked-list)
                  while (lock.next != null){
                     lock = lock.next;
                  }

                  lock.next = this;  // Append this lock request at end.
               }

            }
```

```
            }else if (command == RELEASE_LOCK){       // This is a
                                                      // RELEASE_LOCK
                                                      // operation.

               // Read in the globalID of the object to be locked.
               int globalID = inputStream.readInt();

               // Synchronize on the locks table in order to ensure thread-
               // safety.
               synchronized (locks){

                  // Check to make sure we are the owner of this lock.
                  LockServer lock = (LockServer) locks.get(
                     new Integer(globalID));

                  if (lock == null){
                     throw new AssertionError("Unlocked. Release failed.");
                  }else if (lock.address != this.address){
                     throw new AssertionError("Trying to release a lock "
                        + "which this client doesn't own. Release "
                        + "failed.");
                  }

                  lock = lock.next;
                  lock.acquireLock();   // Signal to the client the
                                        // successful acquisition of this
                                        // lock.

                  // Shift the linked list of pending acquisitions forward
                  // by one.
                  locks.put(new Integer(globalID), lock);

                  // Clear stale reference.
                  next = null;
               }

               releaseLock();     // Signal to the client the successful
                                  // release of this lock.

            }else{         // Unknown command.
               throw new AssertionError(
                  "Unknown command. Operation failed.");
            }

            // Read in the next command.
            command = inputStream.readInt();

         }

      }catch (Exception e){
         throw new AssertionError("Exception: " + e.toString());
      }finally{
         try{
            // Closing down. Cleanup this connection.
            outputStream.flush();
            outputStream.close();
            inputStream.close();
            socket.close();
         }catch (Throwable t){
            t.printStackTrace();
         }
         // Garbage these references.
         outputStream = null;
         inputStream = null;
         socket = null;
```

```
      }

   }

   /** Send a positive acknowledgement of an ACQUIRE_LOCK operation. */
   public void acquireLock() throws IOException{
      outputStream.writeInt(ACK);
      outputStream.flush();
   }

   /** Send a positive acknowledgement of a RELEASE_LOCK operation. */
   public void releaseLock() throws IOException{
      outputStream.writeInt(ACK);
      outputStream.flush();
   }


}
```

ANNEXURE D6

LockLoader.java

This excerpt is the source-code of LockLoader, which modifies an application as it is being loaded.

```
import java.lang.*;
import java.io.*;
import java.net.*;

public class LockLoader extends URLClassLoader{

   public LockLoader(URL[] urls){
      super(urls);
   }

   protected Class findClass(String name)
   throws ClassNotFoundException{

      ClassFile cf = null;

      try{
         BufferedInputStream in =
            new BufferedInputStream(findResource(name.replace('.',
            '/').concat(".class")).openStream());

         cf = new ClassFile(in);

      }catch (Exception e){throw new ClassNotFoundException(e.toString());}

      // Class-wide pointers to the enterindex and exitindex.
      int enterindex = -1;
      int exitindex = -1;

      for (int i=0; i<cf.methods_count; i++){
         for (int j=0; j<cf.methods[i].attributes_count; j++){
            if (!(cf.methods[i].attributes[j] instanceof Code_attribute))
               continue;

            Code_attribute ca = (Code_attribute) cf.methods[i].attributes[j];

            boolean changed = false;

            for (int z=0; z<ca.code.length; z++){
               if ((ca.code[z][0] & 0xff) == 194){    // Opcode for a
                                                      // MONITORENTER
                                                      // instruction.
```

```
                changed = true;

                // Next, realign the code array, making room for the
                // insertions.
                byte[][] code2 = new byte[ca.code.length+2][];
                System.arraycopy(ca.code, 0, code2, 0, z);
                code2[z+1] = ca.code[z];
                System.arraycopy(ca.code, z+1, code2, z+3,
                   ca.code.length-(z+1));
                ca.code = code2;


                // Next, insert the DUP instruction.
                ca.code[z] = new byte[1];
                ca.code[z][0] = (byte) 89;


                // Finally, insert the INVOKESTATIC instruction.
                if (enterindex == -1){
                   // This is the first time this class is encountering the
                   // acquirelock instruction, so have to add it to the
                   // constant pool.
                   cp_info[] cpi = new cp_info[cf.constant_pool.length+6];
                   System.arraycopy(cf.constant_pool, 0, cpi, 0,
                      cf.constant_pool.length);
                   cf.constant_pool = cpi;
                   cf.constant_pool_count += 6;

                   CONSTANT_Utf8_info u1 =
                      new CONSTANT_Utf8_info("LockClient");
                   cf.constant_pool[cf.constant_pool.length-6] = u1;

                   CONSTANT_Class_info c1 = new CONSTANT_Class_info(
                      cf.constant_pool_count-6);
                   cf.constant_pool[cf.constant_pool.length-5] = c1;

                   u1 = new CONSTANT_Utf8_info("acquireLock");
                   cf.constant_pool[cf.constant_pool.length-4] = u1;

                   u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;)V");
                   cf.constant_pool[cf.constant_pool.length-3] = u1;

                   CONSTANT_NameAndType_info n1 =
                      new CONSTANT_NameAndType_info(
                      cf.constant_pool.length-4, cf.constant_pool.length-3);
                   cf.constant_pool[cf.constant_pool.length-2] = n1;

                   CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
                      cf.constant_pool.length-5, cf.constant_pool.length-2);
                   cf.constant_pool[cf.constant_pool.length-1] = m1;
                   enterindex = cf.constant_pool.length-1;
                }
                ca.code[z+2] = new byte[3];
                ca.code[z+2][0] = (byte) 184;
                ca.code[z+2][1] = (byte) ((enterindex >> 8) & 0xff);
                ca.code[z+2][2] = (byte) (enterindex & 0xff);

                // And lastly, increase the CODE_LENGTH and ATTRIBUTE_LENGTH
                // values.
                ca.code_length += 4;
                ca.attribute_length += 4;

                z += 1;

             }else if ((ca.code[z][0] & 0xff) == 195){    // Opcode for a
```

```
                                                    // MONITOREXIT
                                                    // instruction.
                changed = true;

                // Next, realign the code array, making room for the
                // insertions.
                byte[][] code2 = new byte[ca.code.length+2][];
                System.arraycopy(ca.code, 0, code2, 0, z);
                code2[z+1] = ca.code[z];
                System.arraycopy(ca.code, z+1, code2, z+3,
                   ca.code.length-(z+1));
                ca.code = code2;

                // Next, insert the DUP instruction.
                ca.code[z] = new byte[1];
                ca.code[z][0] = (byte) 89;

                // Finally, insert the INVOKESTATIC instruction.
                if (exitindex == -1){
                   // This is the first time this class is encountering the
                   // acquirelock instruction, so have to add it to the
                   // constant pool.
                   cp_info[] cpi = new cp_info[cf.constant_pool.length+6];
                   System.arraycopy(cf.constant_pool, 0, cpi, 0,
                      cf.constant_pool.length);
                   cf.constant_pool = cpi;
                   cf.constant_pool_count += 6;

                   CONSTANT_Utf8_info u1 =
                      new CONSTANT_Utf8_info("LockClient");
                   cf.constant_pool[cf.constant_pool.length-6] = u1;

                   CONSTANT_Class_info c1 = new CONSTANT_Class_info(
                      cf.constant_pool_count-6);
                   cf.constant_pool[cf.constant_pool.length-5] = c1;

                   u1 = new CONSTANT_Utf8_info("releaseLock");
                   cf.constant_pool[cf.constant_pool.length-4] = u1;

                   u1 = new CONSTANT_Utf8_info("(Ljava/lang/Object;)V");
                   cf.constant_pool[cf.constant_pool.length-3] = u1;

                   CONSTANT_NameAndType_info n1 =
                      new CONSTANT_NameAndType_info(
                      cf.constant_pool.length-4, cf.constant_pool.length-3);
                   cf.constant_pool[cf.constant_pool.length-2] = n1;

                   CONSTANT_Methodref_info m1 = new CONSTANT_Methodref_info(
                      cf.constant_pool.length-5, cf.constant_pool.length-2);
                   cf.constant_pool[cf.constant_pool.length-1] = m1;
                   exitindex = cf.constant_pool.length-1;
                }
                ca.code[z+2] = new byte[3];
                ca.code[z+2][0] = (byte) 184;
                ca.code[z+2][1] = (byte) ((exitindex >> 8) & 0xff);
                ca.code[z+2][2] = (byte) (exitindex & 0xff);


                // And lastly, increase the CODE_LENGTH and ATTRIBUTE_LENGTH
                // values.
                ca.code_length += 4;
                ca.attribute_length += 4;

                z += 1;
```

```
                }

            }

            // If we changed this method, then increase the stack size by one.
            if (changed){
               ca.max_stack++;          // Just to make sure.
            }
         }
      }


      try{

         ByteArrayOutputStream out = new ByteArrayOutputStream();
         cf.serialize(out);

         byte[] b = out.toByteArray();

         return defineClass(name, b, 0, b.length);

      }catch (Exception e){
         throw new ClassNotFoundException(name);
      }

   }

}
```

CLAIMS

1. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network, and
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s) whereby the corresponding replicated memory location of each said computer of said multiple system can be updated via said communicating network and all said replicated memory locations can remain substantially identical.

2. The computer as claimed in claim 1 wherein each said replicated local memory location is part of an independent local memory accessible only by the corresponding portion of said application program executing on said computer.

3. The computer as claimed in claim 1 or 2 wherein said memory location includes at least one of an asset, object or resource and has a value or content.

4. The computer as claimed in any one of claims 1 - 3 wherein a substantially identical global name is allocated to each corresponding said replicated memory location.

5. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
initialization means which determine the initial content or value of said replicated memory location and which can be disabled.

6. The computer as claimed in claim 5 wherein said initialization means is connected to said communications port to receive an initial content or value of said replicated memory location via said communications network.

7. The computer as claimed in claim 5 or 6 wherein said initialization means is connected to said communications port to send an initial content or value of said replicated memory location via said communications network.

8. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least

one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
finalization means which deletes said replicated memory location when all said computers no longer need to refer thereto, said finalization means being connected to said communications port to receive therefrom data transmitted over said network relating to continued reference of other computers of said multiple computer system to said replicated memory location.

9. The computer as claimed in claim 8 wherein said finalization means when said computer no longer needs to refer to said replicated memory location transfers to said communications port data indicative of the terminated reference for transmission via said communications network.

10. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,
updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s), and
lock acquisition and relinquishing means to respectively permit said replicated local memory location to be written to, and prevent said replicated local memory being written to, on command.

11. The computer as claimed in claim 10 wherein said command is received from said communications port.

12. The computer as claimed in claim 10 or 11 wherein said command is generated by said computer and sent to said communications port for transmission via said communications network.

13. A single computer intended to operate in a multiple computer system which comprises a plurality of computers each having a local memory and each being interconnected via a communications network, wherein a different portion of at least one application program each written to execute on only a single computer executes substantially simultaneously on a corresponding one of said plurality of computers, and at least one memory location is replicated in the local memory of each said computer, said single computer comprising:
a local memory having at least one memory location intended to be updated via said communications network,
a communications port for connection to said communications network,

updating means to transfer to said communications port any updated content(s) of said replicated local memory location(s) whereby the corresponding replicated memory location of each said computer of said multiple system can be updated via said communicating network and all said replicated memory locations can remain substantially identical,

initialization means which determine the initial content or value of said replicated memory location and which can be disabled,

finalization means which deletes said replicated memory location when all said computers no longer need to refer thereto, said finalization means being connected to said communications port to receive therefrom data transmitted over said network relating to continued reference of other computers of said multiple computer system to said replicated memory location, and

lock acquisition and relinquishing means to respectively permit said replicated local memory location to be written to, and prevent said replicated local memory being written to, on command.

14. The computer as claimed in claim 13 wherein each said replicated local memory location is part of an independent local memory accessible only by the corresponding portion of said application program executing on said computer.

15. The computer as claimed in claim 13 or 14 wherein said memory location includes at least one of an asset, object or resource and has a value or content.

16. The computer as claimed in any one of claims 13-15 wherein a substantially identical global name is allocated to each corresponding said replicated memory location.

17. The computer as claimed in any one of claims 13-16 wherein said initialization means is connected to said communications port to receive an initial content or value of said replicated memory location via said communications network.

18. The computer as claimed in any one of claims 13-17 wherein said initialization means is connected to said communications port to send an initial content or value of said replicated memory location via said communications network.

19. The computer as claimed in any one of claims 13-18 wherein said finalization means when said computer no longer needs to refer to said replicated memory location transfers to said communications port data indicative of the terminated reference for transmission via said communications network.

20. The computer as claimed in any one of claims 13-19 wherein said command is received from said communications port.

21. The computer as claimed in any one of claims 13-20 wherein said command is generated by said computer and sent to said communications port for transmission via said communications network.

22. A multiple computer system having at least one application program each written to operate on only a single computer but running simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers, wherein each computer has an independent local memory accessible only by the corresponding portion of said application program(s) and wherein for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer.

23. The system as claimed in claim 22 wherein each computer has an independent local memory accessible only by the corresponding portion of said application program(s)

24. The system as claimed in claim 22 or 23 wherein each of said plurality of substantially identical objects has a substantially identical name.

25. The system as claimed in claim 24 wherein each said computer includes a distributed run time means with the distributed run time means of each said computer able to communicate with all other computers whereby if a portion of said application program(s) running on one of said computers changes the contents or value of an object in that computer then the change in content or value for said object is propagated by the distributed run time means of said one computer to all other computers to change the content or value of the corresponding object in each of said other computers.

26. The system as claimed in claim 25 wherein each said application program is modified before, during, or after loading by inserting an updating propagation routine to modify each instance at which said application program writes to memory, said updating propagation routine propagating every memory write by one computer to said other computers.

27. The system as claimed in claim 26 wherein the application program is modified in accordance with a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

28. The system as claimed in claim 26 or 27 wherein said modified application program is transferred to all said computers in accordance with a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

29. A plurality of computers interconnected via a communications link and each having an independent local memory and substantially simultaneously operating a different portion at least one application program each written to operate on only a single computer, each local memory being accessible only by the corresponding portion of said application program.

30. The plurality of computers as claimed in claim 29 wherein each said computer in operating said at least one application program reads and writes only to local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has distribution update means to distribute to all other said computers the content of any memory location updated by said one computer.

31. The plurality of computers as claimed in claim 29 or 30 wherein the local memory capacity allocated to the or each said application program is substantially identical and the total memory capacity available to the or each said application program is said allocated memory capacity.

32. The plurality of computers as claimed in claim 30 or 31 wherein all said distribution update means communicate via said communications link at a data transfer rate which is substantially less than the local memory read rate.

33. The plurality of computers as claimed in any one of claims 29-32 wherein at least some of said computers are manufactured by different manufacturers and/or have different operating systems.

34. A multiple computer system having at least one application program each written to operate on only a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and wherein the initial contents of each of said identically named objects is substantially the same.

35. The system as claimed in claim 34 wherein each said computer has a local memory accessible only by the corresponding portion of said application program.

36. The system as claimed in claim 34 or 35 wherein each said computer includes a distributed run time means with the distributed run time means of each said computer able to communicate with all other computers whereby if a portion of said application program(s) running on one of said computers creates an object in that computer then the created object is propagated by the distributed run time means of said one computer to all the other computers.

37. The system as claimed in any one of claims 34-36 wherein each said application program is modified before, during, or after loading by inserting an initialization routine to modify each instance at which said application program creates an object, said initialization routine propagating every object newly created by one computer to all said other computers.

38. The system as claimed in claim 37 wherein said inserted initialization routine modifies a pre-existing initialization routine to enable the pre-existing initialization routine to execute on creation of the first of said like plurality of objects, and to disable the pre-existing initialization routine on creation of all subsequent ones of said like plurality of objects.

39. The system as claimed in claim 37 or 38 wherein the application program is modified in accordance with a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

40. The system as claimed in any one of claims 37-39 wherein said modified application program is transferred to all said computers in accordance with a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

41. A plurality of computers interconnected via a communications link and substantially simultaneously operating at least one application program each written to operation on only a single computer wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion creates objects only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer are fundamentally similar but not, at each instant, identical, and every one of said computers has distribution update means to distribute to all other said computers objects created by said one computer.

42. The plurality of computers as claimed in claim 41 wherein the local memory capacity allocated to the or each said application program is substantially identical and the total memory capacity available to the or each said application program is said allocated memory capacity.

43. The plurality of computers as claimed in claim 41 or 42 wherein all said distribution update means communicate via said communications link at a data transfer rate which is substantially less than the local memory read rate.

44. The plurality of computers as claimed in any one of claims 41-43 wherein at least some of said computers are manufactured by different manufacturers and/or have different operating systems.

45. A multiple computer system having at least one application program each written to operate only on a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each said portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and wherein all said identical objects are collectively deleted when each one of said plurality of computers no longer needs to refer to their corresponding object.

46. The system as claimed in claim 45 wherein each said computer includes a distributed run time means with the distributed run time means of each said computer able to communicate with all other computers whereby if a portion of said application program(s) running on one of said computers no longer needs to refer to an object in that computer then the identity of the unreferenced object is transmitted by the distributed run time means of said one computer to a shared table or record accessible by all the other computers.

47. The system as claimed in claim 46 or 47 wherein each said application program is modified before, during, or after loading by inserting a finalization routine to modify each instance at which said application program no longer needs to refer to an object.

48. The system as claimed in claim 47 wherein said inserted finalization routine modifies a pre-existing finalization routine to enable the pre-existing finalization routine to execute if all computers no longer need to refer to their corresponding object, and to disable the pre-existing finalization routine if at least one computer does need to refer to a corresponding object.

49. The system as claimed in claim 47 or 48 wherein the application program is modified in accordance with a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

50. The system as claimed in any one of claims 47-49 wherein said modified application program is transferred to all said computers in accordance with a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

51. A plurality of computers interconnected via a communications link and operating substantially simultaneously at least one application program each written to operate only on a single computer, wherein each said computer substantially simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion needs, or no longer needs to refer to an object only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has a finalization

routine which deletes a non-referenced object only if each one of said plurality of computers no longer needs to refer to their corresponding object.

52. The plurality of computers as claimed in claim 51 wherein the local memory capacity allocated to the or each said application program is substantially identical and the total memory capacity available to the or each said application program is said allocated memory capacity.

53. The plurality of computers as claimed in claim 51 or 52 wherein all said distribution update means communicate via said communications link at a data transfer rate which is substantially less than the local memory read rate.

54. The plurality of computers as claimed in any one of claims 51-53 wherein at least some of said computers are manufactured by different manufacturers and/or have different operating systems.

55. A multiple computer system having at least one application program each written to operate on only a single computer but running substantially simultaneously on a plurality of computers interconnected by a communications network, wherein different portions of said application program(s) execute substantially simultaneously on different ones of said computers and for each portion a like plurality of substantially identical objects are created, each in the corresponding computer and each having a substantially identical name, and said system including a lock means applicable to all said computers wherein any computer wishing to utilize a named object therein acquires an authorizing lock from said lock means which permits said utilization and which prevents all the other computers from utilizing their corresponding named object until said authorizing lock is relinquished.

56. The system as claimed in claim 55 wherein said lock means includes an acquire lock routine and a release lock routine, and both said routines are included in modifications made to said application program running on all said computers.

57. The system as claimed in claim 55 or 56 wherein said lock means further includes a shared table or record listing said named objects in use by any said computer, a lock status for each said object, and a queue of any pending lock acquisitions.

58. The system as claimed in any one of claims 55-57 wherein said lock means is located within an additional computer not running said application program and connected to said communications network.

59. The system as claimed in any one of claims 56-58 wherein each said application program is modified before, during, or after loading by inserting said acquire lock routine and said release lock routine to modify each instance at which said application program acquires and releases respectively a lock on an object.

60. The system as claimed in claim 59 wherein the application program is modified in accordance with a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

61. The system as claimed in claim 59 or 60 wherein said modified application program is transferred to all said computers in accordance with a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

62. A plurality of computers interconnected via a communications link and operating substantially simultaneously at least one application program each written to operate on only a single computer, wherein each said computer substantially

simultaneously executes a different portion of said application program(s), each said computer in operating its application program portion utilizes an object only in local memory physically located in each said computer, the contents of the local memory utilized by each said computer is fundamentally similar but not, at each instant, identical, and every one of said computers has an acquire lock routine and a release lock routine which permit utilization of the local object only by one computer and each of the remainder of said plurality of computers is locked out of utilization of their corresponding object.

63. The plurality of computers as claimed in claim 62 wherein the local memory capacity allocated to the or each said application program is substantially identical and the total memory capacity available to the or each said application program is substantially said allocated memory capacity.

64. The plurality of computers as claimed in claim 62 or 63 wherein each said computer has a distribution update means communicate via said communications link at a data transfer rate which is substantially less than the local memory read rate.

65. The plurality of computers as claimed in any one of claims 62-64 wherein at least some of said computers are manufactured by different manufacturers and/or have different operating systems.

66. A method of running simultaneously on a plurality of computers at least one application program each written to operate on only a single computer, said computers being interconnected by means of a communications network, said method comprising the step of,

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each accessible only by the corresponding portion of said application program.

67. The method as claimed in claim 66 wherein each computer has an independent local memory which includes the corresponding identical object.

68. The method as claimed in claim 66 or 67 comprising the further step of,

(ii) naming each of said plurality of substantially identical objects with a substantially identical global name.

69. The method as claimed in any one of claims 66-68 comprising the further step of,

(iii) if a portion of said application program running on one of said computers changes the contents or value of an object in that computer, then the change in content or value of said object is propagated to all of the other computers via said communications network to change the content or value of the corresponding object in each of said other computers.

70. The method as claimed in any one of claims 66-69 including the further step of:

(iv) modifying said application program before, during or after loading by inserting an updating propagation routine to modify each instance at which said application program writes to memory, said updating propagation routine propagating every memory write by one computer to all said other computers.

71. The method as claimed in claim 70 including the further step of:

(v) modifying said application program utilizing a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to

loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

72. The method as claimed in claim 70 or 71 including the further step of:

(vi) transferring the modified application program to all said computers utilizing a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

73. A method of loading an application program written to operate only on a single computer onto each of a plurality of computers, the computers being interconnected via a communications link, and different portions of said application program(s) being substantially simultaneously executable on different computers with each computer having an independent local memory accessible only by the corresponding portion of said application program(s), the method comprising the step of modifying the application before, during, or after loading and before execution of the relevant portion of the application program.

74. The method as claimed in claim 73 wherein the modification of the application is different for different computers.

75. The method as claimed in claim 73 or 74 wherein said modifying step comprises:-

(i) detecting instructions which share memory records utilizing one of said computers,

(ii) listing all such shared memory records and providing a naming tag for each listed memory record,

(iii) detecting those instructions which write to, or manipulate the contents of, any of said listed memory records, and

(iv) generating an updating propagation routine corresponding to each said detected write or manipulate instruction, said updating propagation routine forwarding the re-written or manipulated contents and name tag of each said re-written or manipulated listed memory record to all of the others of said computers.

76. A method of operating simultaneously on a plurality of computers all interconnected via a communications link at least one application program each written to operate on only a single computer, each of said computers having at least a minimum predetermined local memory capacity, different portions of said application program(s) being substantially simultaneously executed on different ones of said computers with the local memory of each computer being only accessible by the corresponding portion of said application program(s), said method comprising the steps of:

(i) initially providing each local memory in substantially identical condition,

(ii) satisfying all memory reads and writes generated by each said application program portion from said corresponding local memory, and

(iii) communicating via said communications link all said memory writes at each said computer which take place locally to all the remainder of said plurality of computers whereby the contents of the local memory utilised by each said computer, subject to an updating data transmission delay, remains substantially identical.

77. The method as claimed in claim 76 including the further step of:

(iv) communicating said local memory writes constituting an updating data transmission at a data transfer rate which is substantially less than the local memory read rate.

78. A method of compiling or modifying an application program written to operate on only a single computer but to run simultaneously on a plurality of computers

interconnected via a communications link, with different portions of said application program(s) executing substantially simultaneously on different ones of said computers each of which has an independent local memory accessible only by the corresponding portion of said application program, said method comprising the steps of:

(i) detecting instructions which share memory records utilizing one of said computers,

(ii) listing all such shared memory records and providing a naming tag for each listed memory record,

(iii) detecting those instructions which write to, or manipulate the contents of, any of said listed memory records, and

(iv) activating an updating propagation routine following each said detected write or manipulate instruction, said updating propagation routine forwarding the re-written or manipulated contents and name tag of each said re-written or manipulated listed memory record to the remainder of said computers.

79. The method as claimed in claim 78 and carried out prior to loading the application program onto each said computer, or during loading of the application program onto each said computer, or after loading of the application program onto each said computer and before execution of the relevant portion of the application program.

80. In a multiple thread processing computer operation in which individual threads of a single application program written to operate on only a single computer are simultaneously being processed each on a different corresponding one of a plurality of computers each having an independent local memory accessible only by the corresponding thread and each being interconnected via a communications link, the improvement comprising communicating changes in the contents of local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

81. The improvement as claimed in claim 80 wherein changes to the memory associated with one said thread are communicated by the computer of said one thread to all other said computers.

82. The improvement as claimed in claim 80 wherein changes to the memory associated with one said thread are transmitted to the computer associated with another said thread and are transmitted thereby to all said other computers.

83. A method of running substantially simultaneously on a plurality of computers at least one application program each written to operate on only a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) creating the initial contents of each of said identically named objects substantially the same.

84. The method as claimed in claim 83 comprising the further step of,

(iii) if a portion of said application program running on one of said computers creates an object in that computer, then the created object is propagated to all of the other computers via said communications network.

85. The method as claimed in claim 83 or 84 including the further step of:

(iv) modifying said application program before, during or after loading by

inserting an initialization routine to modify each instance at which said application program creates an object, said initialization routine propagating every object created by one computer to all said other computers.

86. The method as claimed in claim 85 including the further step of:

(v) modifying said application program utilizing a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

87. The method as claimed in claim 85 or 86 including the further step of:

(vi) transferring the modified application program to all said computers utilizing a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

88. A method of compiling or modifying an application program written to operate on only a single computer to have different portions thereof to execute substantially simultaneously on different ones of a plurality of computers interconnected via a communications link, said method comprising the steps of:

(i) detecting instructions which create objects utilizing one of said computers,

(ii) activating an initialization routine following each said detected object creation instruction, said initialization routine forwarding each created object to the remainder of said computers.

89. The method as claimed in claim 88 and carried out prior to loading the application program onto each said computer, or during loading of the application program onto each said computer, or after loading of the application program onto each said computer and before execution of the relevant portion of the application program.

90. In a multiple thread processing computer operation in which individual threads of a single application program written to operate on only a single computer are substantially simultaneously being processed each on a different corresponding one of a plurality of computers interconnected via a communications link, the improvement comprising communicating objects created in local memory physically associated with the computer processing each thread to the local memory of each other said computer via said communications link.

91. The improvement as claimed in claim 90 wherein objects created in the memory associated with one said thread are communicated by the computer of said one thread to all other said computers.

92. The improvement as claimed in claim 91 wherein objects created the memory associated with one said thread are transmitted to the computer associated with another said thread and are transmitted thereby to all said other computers.

93. A method of ensuring consistent initialization of an application program written to operate on only a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:

(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an initialization routine, and

(ii) modifying said initialization routine to ensure consistent operation of all said computers.

94. The method as claimed in claim 93 wherein said initialization routine is modified to execute once only on the creation of a first object by any one of said

computers and is modified to be disabled on the creation of each subsequent peer copy of said object by the remainder of said computers.

95. The method claimed in claim 93 or 94 wherein step (ii) comprises the steps of:

(iii) loading and executing said initialization routine on one of said computers,

(iv) modifying said initialization routine by said one computer, and

(v) transferring said modified initialization routine to each of the remaining computers.

96. The method as claimed in claim 95 wherein said modified initialization routine is supplied by said one computer direct to each of said remaining computers.

97. The method as claimed in claim 96 wherein said modified initialization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

98. The method claimed in claim 93 or 94 wherein step (ii) comprises the steps of:

(vi) loading and modifying said initialization routine on one of said computers,

(vii) said one computer sending said unmodified initialization routine to each of the remaining computers, and

(viii) each of said remaining computers modifying said initialization routine after receipt of same.

99. The method claimed in claim 98 wherein said unmodified initialization routine is supplied by said one computer directly to each of said remaining computers.

100. The method claimed in claim 98 wherein said unmodified initialization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

101. A method of running substantially simultaneously on a plurality of computers at least one application program each written to operate only on a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) deleting all said identical objects collectively when all of said plurality of computers no longer need to refer to their corresponding object.

102. A method as claimed in claim 101 including the further step of:

(iii) providing each said computer with a distributed run time means to communicate between said computers via said communications network.

103. A method as claimed in claim 102 including the further step of:

(iv) providing a shared table or record accessible by each said distributed run time means and in which is stored the identity of any computer which no longer requires to access an object, together with the identity of the object.

104. A method as claimed in claim 103 including the further step of:

(v) associating a counter means with said shared table, said counter means storing a count of the number of said computers which no longer require to access said object.

105. A method as claimed in claim 104 including the further step of:

(vi) providing an additional computer on which said shared program does not run and which hosts said shared table and counter, said additional computer being connected to said communications network.

106. A method of ensuring consistent finalization of an application program written to operate only on a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:

(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an finalization routine, and

(ii) modifying said finalization routine to ensure collective deletion of corresponding objects in all said computers only when each one of said computers no longer needs to refer to their corresponding object.

107. The method as claimed in claim 106 wherein said finalization routine is modified to execute to clean-up an object once only and on only one of said computers and when all of said computers no longer need to refer to said object.

108. The method claimed in claim 106 or 107 wherein step (ii) comprises the steps of:

(iii) loading and executing said finalization routine on one of said computers,

(iv) modifying said finalization routine by said one computer, and

(v) transferring said modified finalization routine to each of the remaining computers.

109. The method as claimed in claim 108 wherein said modified finalization routine is supplied by said one computer direct to each of said remaining computers.

110. The method as claimed in claim 108 wherein said modified finalization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

111. The method claimed in claim 106 or 107 wherein step (ii) comprises the steps of:

(vi) loading and modifying said finalization routine on one of said computers,

(vii) said one computer sending said unmodified finalization routine to each of the remaining computers, and

(viii) each of said remaining computers modifying said finalization routine after receipt of same.

112. The method claimed in claim 111 wherein said unmodified finalization routine is supplied by said one computer directly to each of said remaining computers.

113. The method claimed in claim 111 wherein said unmodified finalization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

114. The method as claimed in any one of claims 106-113 including the further step of:

(ix) modifying said application program utilizing a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

115. The method as claimed in claim 114 including the further step of:

(x) transferring the modified application program to all said computers utilizing a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

116. In a multiple thread processing computer operation in which individual threads of a single application program written to operate only on a single computer are substantially simultaneously being processed each on a corresponding different one of a

plurality of computers interconnected via a communications link, and in which objects in local memory physically associated with the computer processing each thread have corresponding objects in the local memory of each other said computer, the improvement comprising collectively deleting all said corresponding objects when each one of said plurality of computers no longer needs to refer to their corresponding object.

117. The improvement as claimed in claim 116 wherein an object residing in the memory associated with one said thread and to be deleted has its identity communicated by the computer of said one thread to a shared table or record accessible by all other said computers.

118. The improvement as claimed in claim 116 wherein an object residing in the memory associated with one said thread and to be deleted has its identity transmitted to the computer associated with another said thread and is transmitted thereby to a shared table or record accessible by all said other computers.

119. A method of running substantially simultaneously on a plurality of computers at least one application program each written to operate only on a single computer, said computers being interconnected by means of a communications network, said method comprising the steps of:

(i) executing different portions of said application program(s) on different ones of said computers and for each said portion creating a like plurality of substantially identical objects each in the corresponding computer and each having a substantially identical name, and

(ii) requiring any of said computers wishing to utilize a named object therein to acquire an authorizing lock which permits said utilization and which prevents all the other computers from utilizing their corresponding named object until said authorizing lock is relinquished.

120. A method as claimed in claim 119 including the further step of:

(iii) providing each said computer with a distributed run time means to communicate between said computers via said communications network.

121. A method as claimed in claim 120 including the further step of:

(iv) providing a shared table or record accessible by each said distributed run time means and in which is stored the identity of any computer which currently has to access an object, together with the identity of the object.

122. A method as claimed in claim 121 including the further step of:

(v) associating a counter means with said shared table, said counter means storing a count of the number of said computers which seek access to said object.

123. A method as claimed in claim 122 including the further step of:

(vi) providing an additional computer on which said shared program does not run and which hosts said shared table and counter, said additional computer being connected to said communications network.

124. A method of ensuring consistent synchronization of an application program written to operate only on a single computer but different portions of which are to be executed substantially simultaneously each on a different one of a plurality of computers interconnected via a communications network, said method comprising the steps of:

(i) scrutinizing said application program at, or prior to, or after loading to detect each program step defining an synchronization routine, and

(ii) modifying said synchronization routine to ensure utilization of an object by

only one computer and preventing all the remaining computers from simultaneously utilizing their corresponding objects.

125. The method claimed in claim 124 wherein step (ii) comprises the steps of:

(iii) loading and executing said synchronization routine on one of said computers,

(iv) modifying said synchronization routine by said one computer, and

(v) transferring said modified synchronization routine to each of the remaining computers.

126. The method as claimed in claim 125 wherein said modified synchronization routine is supplied by said one computer direct to each of said remaining computers.

127. The method as claimed in claim 125 wherein said modified synchronization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

128. The method claimed in any one of claims 124-127 wherein step (ii) comprises the steps of:

(vi) loading and modifying said synchronization routine on one of said computers,

(vii) said one computer sending said unmodified synchronization routine to each of the remaining computers, and

(viii) each of said remaining computers modifying said synchronization routine after receipt of same.

129. The method claimed in claim 128 wherein said unmodified synchronization routine is supplied by said one computer directly to each of said remaining computers.

130. The method claimed in claim 128 wherein said unmodified synchronization routine is supplied in cascade fashion from said one computer sequentially to each of said remaining computers.

131. The method as claimed in any one of claims 124-130 including the further step of:

(ix) modifying said application program utilizing a procedure selected from the group of procedures consisting of re-compilation at loading, pre-compilation prior to loading, compilation prior to loading, just-in-time compilation, and re-compilation after loading and before execution of the relevant portion of application program.

132. The method as claimed in claim 131 including the further step of:

(x) transferring the modified application program to all said computers utilizing a procedure selected from the group consisting of master/slave transfer, branched transfer and cascaded transfer.

133. In a multiple thread processing computer operation in which individual threads of a single application program written to operate only on a single computer are substantially simultaneously being processed each on a corresponding different one of a plurality of computers interconnected via a communications link, and in which objects in local memory physically associated with the computer processing each thread have corresponding objects in the local memory of each other said computer, the improvement comprising permitting only one of said computers to utilize an object and preventing all the remaining computers from simultaneously utilizing their corresponding object.

134. The improvement as claimed in claim 133 wherein an object residing in the memory associated with one said thread and to be utilized has its identity communicated by the computer of said one thread to a shared table or record accessible by all other said computers.

135. The improvement as claimed in claim 133 wherein an object residing in the memory associated with one said thread and to be utilized has its identity transmitted to the computer associated with another said thread and is transmitted thereby to a shared table accessible by all said other computers.

136. A computer program product comprising a set of program instructions stored in a storage medium and operable to permit at least one computer to carry out the method as claimed in any one of claims 66-135.

137. A plurality of computers interconnected via a communication network and operable to ensure consistent operation of an application program written to operate only on a single computer but running substantially simultaneously of said computers, said computers being programmed to carry out the method as claimed in any one of claims 66-135 or being loaded with the computer program product as claimed in claim 136.

138. A distributed run time adapted to enable communications between a plurality of computers, computing machines, or information appliances.

139. A modifier for modifying an application program written to execute on a single computer whereby the modified application program executes substantially simultaneously on a plurality of network computers or computing machines.