US 20040225923A1

(19) United States

(12) Patent Application Publication (10) Pub. No.: US 2004/0225923 A1

Teegan et al. (43) Pub. Date: Nov. 11, 2004

(54) OBJECT-BASED SOFTWARE MANAGEMENT

(75) Inventors: **Hugh A. Teegan**, Bellevue, WA (US); **Scott Matsumoto**, Andover, MA (US)

Correspondence Address:
KLARQUIST SPARKMAN LLP
121 S.W. SALMON STREET
SUITE 1600
PORTLAND, OR 97204 (US)

(73) Assignee: **Microsoft Corporation**, Redmond, WA

(21) Appl. No.: **10/864,855**

(22) Filed: **Jun. 8, 2004**

Related U.S. Application Data

(63) Continuation of application No. 09/393,011, filed on Sep. 9, 1999, now Pat. No. 6,748,555.

Publication Classification

(51) **Int. Cl.**[7] **G06F 7/00**

(52) **U.S. Cl.** **714/38**

(57) **ABSTRACT**

An execution environment accommodating object-based software transparently monitors interactions with software objects to generate operational management information for managing programs executing at plural computers. Notifications are directed to a software manager in the form of events, which can additionally be provided to applications or user programs. The software manager can group the events into sets and derive various operational management metrics from them to provide an overall picture of a program's performance, including availability. A hierarchical arrangement feature facilitates gathering information for programs scattered over plural computers. An alert feature provides warnings if metrics fall outside a specified threshold. In addition, the alert feature can automatically subscribe to additional sets of events to dynamically select the information collected by the software manager. Since the operational management information is collected transparently by logic outside the objects, manual instrumentation of the program is unnecessary, and software management technology is made available to organizations without software management expertise.

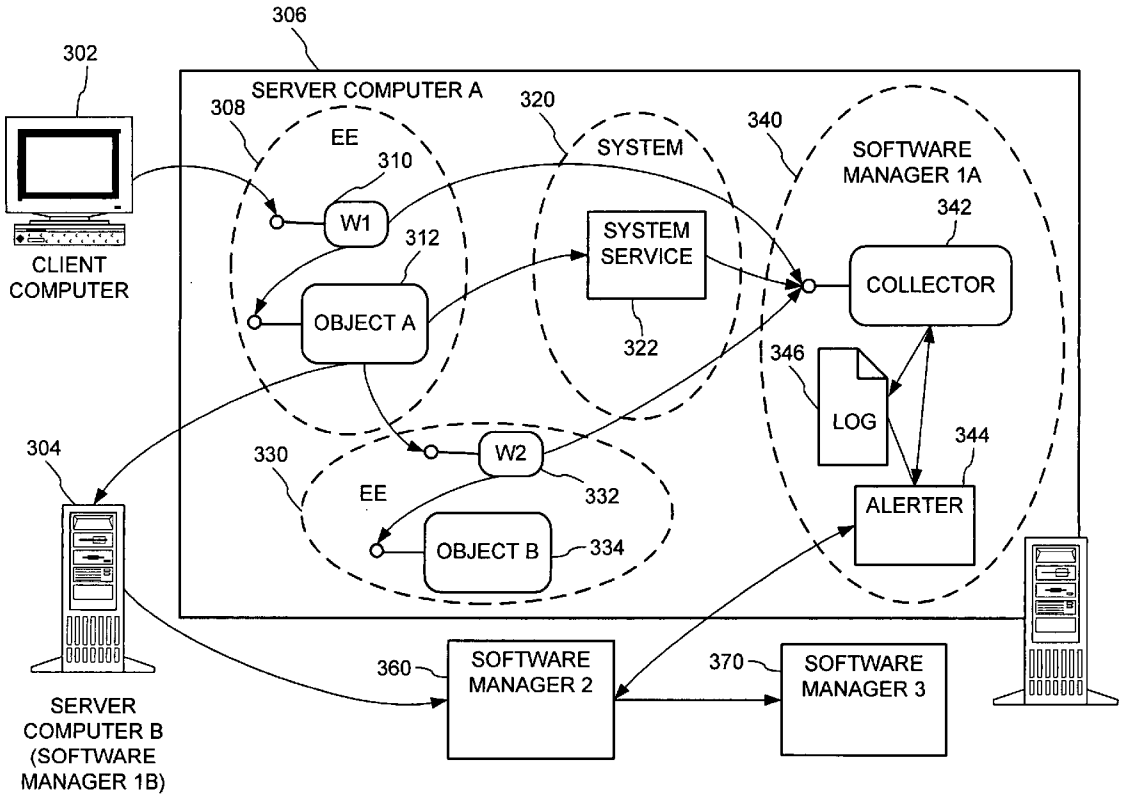


FIG. 1

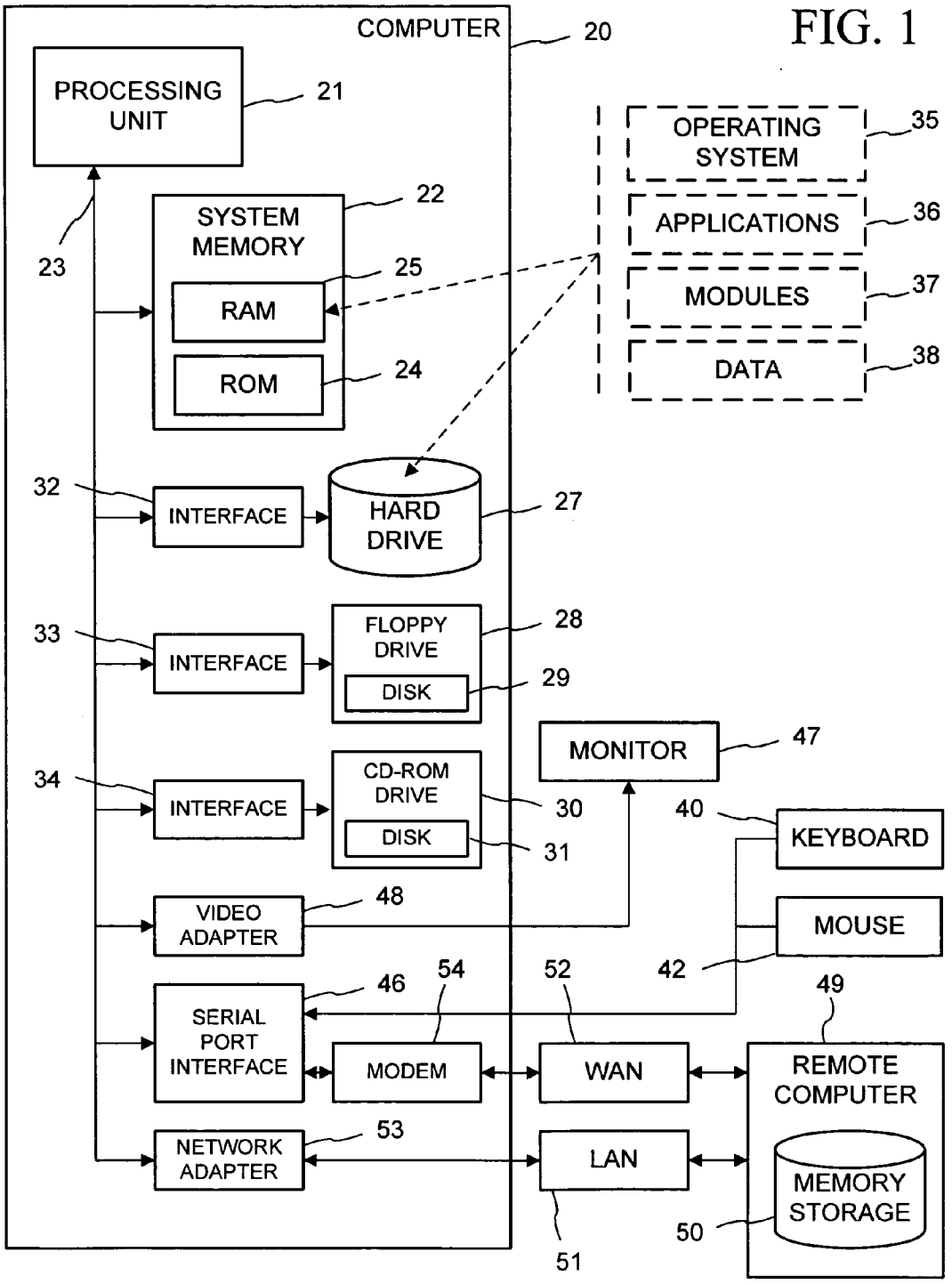

COMPUTER
20
PROCESSING UNIT
21
SYSTEM MEMORY
22
23
RAM
25
ROM
24
OPERATING SYSTEM
35
APPLICATIONS
36
MODULES
37
DATA
38
32
INTERFACE
HARD DRIVE
27
33
INTERFACE
FLOPPY DRIVE
28
DISK
29
34
INTERFACE
CD-ROM DRIVE
30
DISK
31
MONITOR
47
40
KEYBOARD
MOUSE
42
VIDEO ADAPTER
48
46
54
52
49
SERIAL PORT INTERFACE
MODEM
WAN
REMOTE COMPUTER
NETWORK ADAPTER
53
LAN
MEMORY STORAGE
50
51

FIG. 2

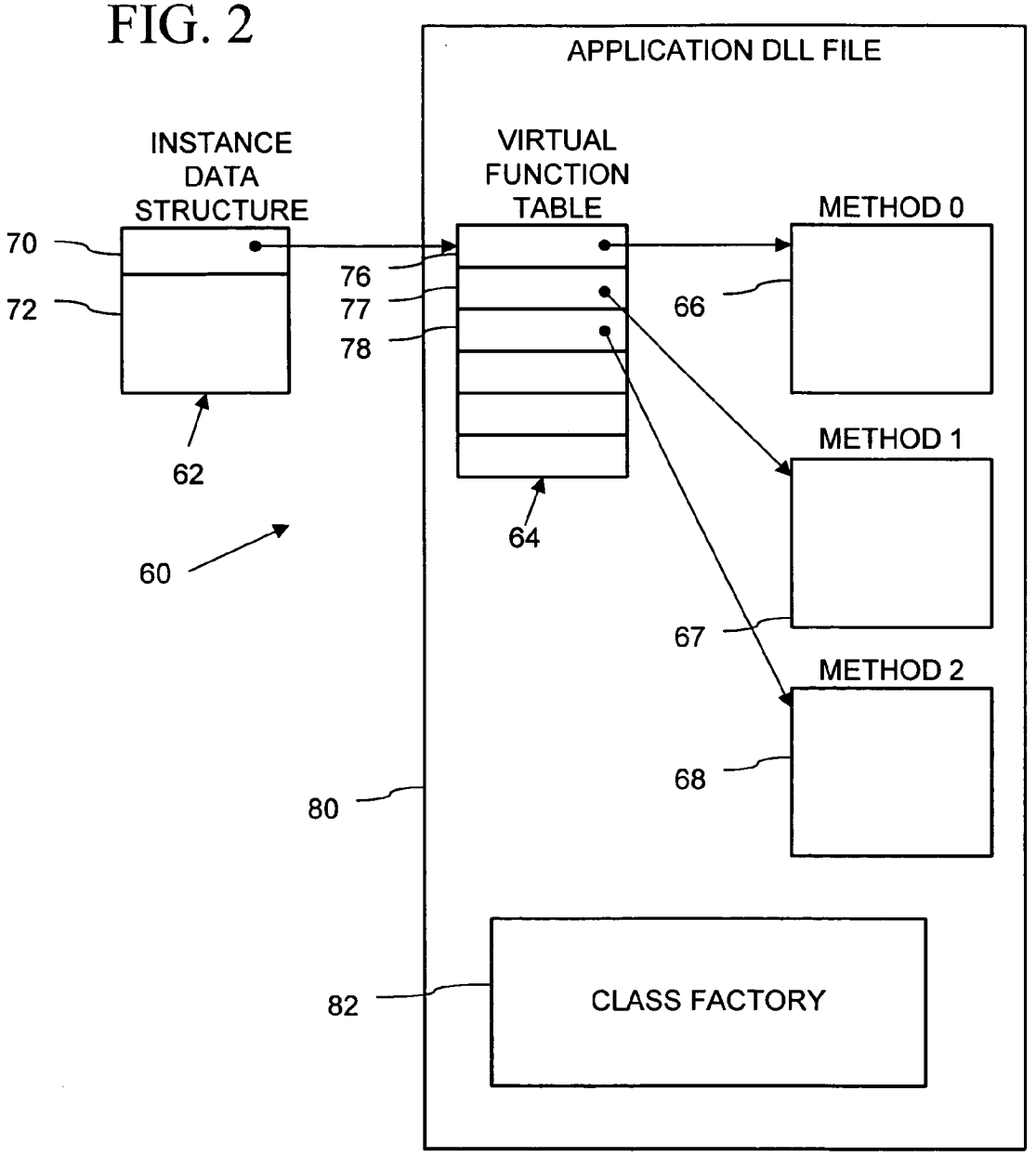

APPLICATION DLL FILE
INSTANCE DATA STRUCTURE
VIRTUAL FUNCTION TABLE
METHOD 0
METHOD 1
METHOD 2
CLASS FACTORY
70
72
62
60
76
77
78
64
66
67
68
80
82

FIG. 3

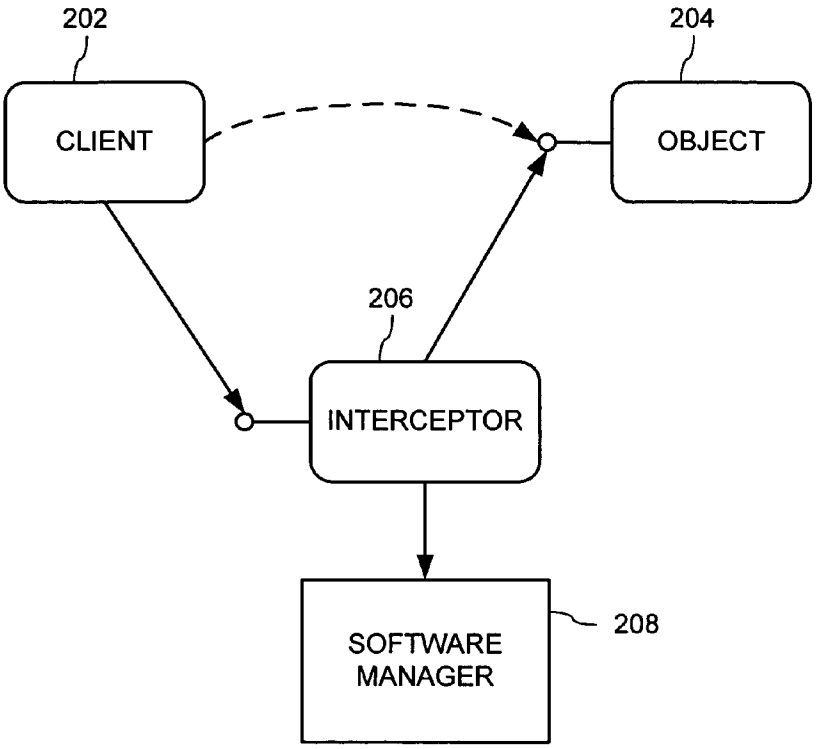

202
204
CLIENT
OBJECT
206
INTERCEPTOR
SOFTWARE MANAGER
208

FIG. 4

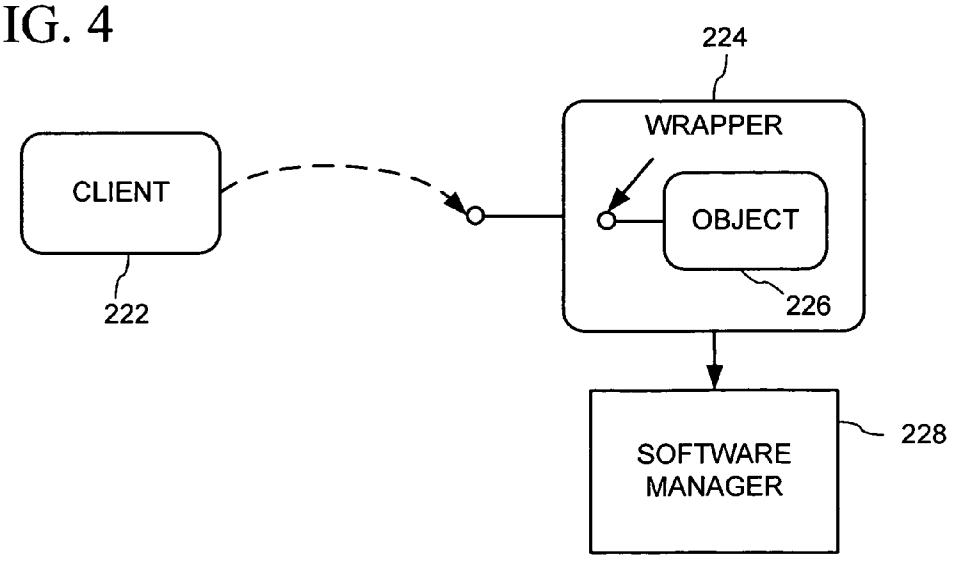

224
WRAPPER
CLIENT
OBJECT
222
226
SOFTWARE MANAGER
228

FIG. 5

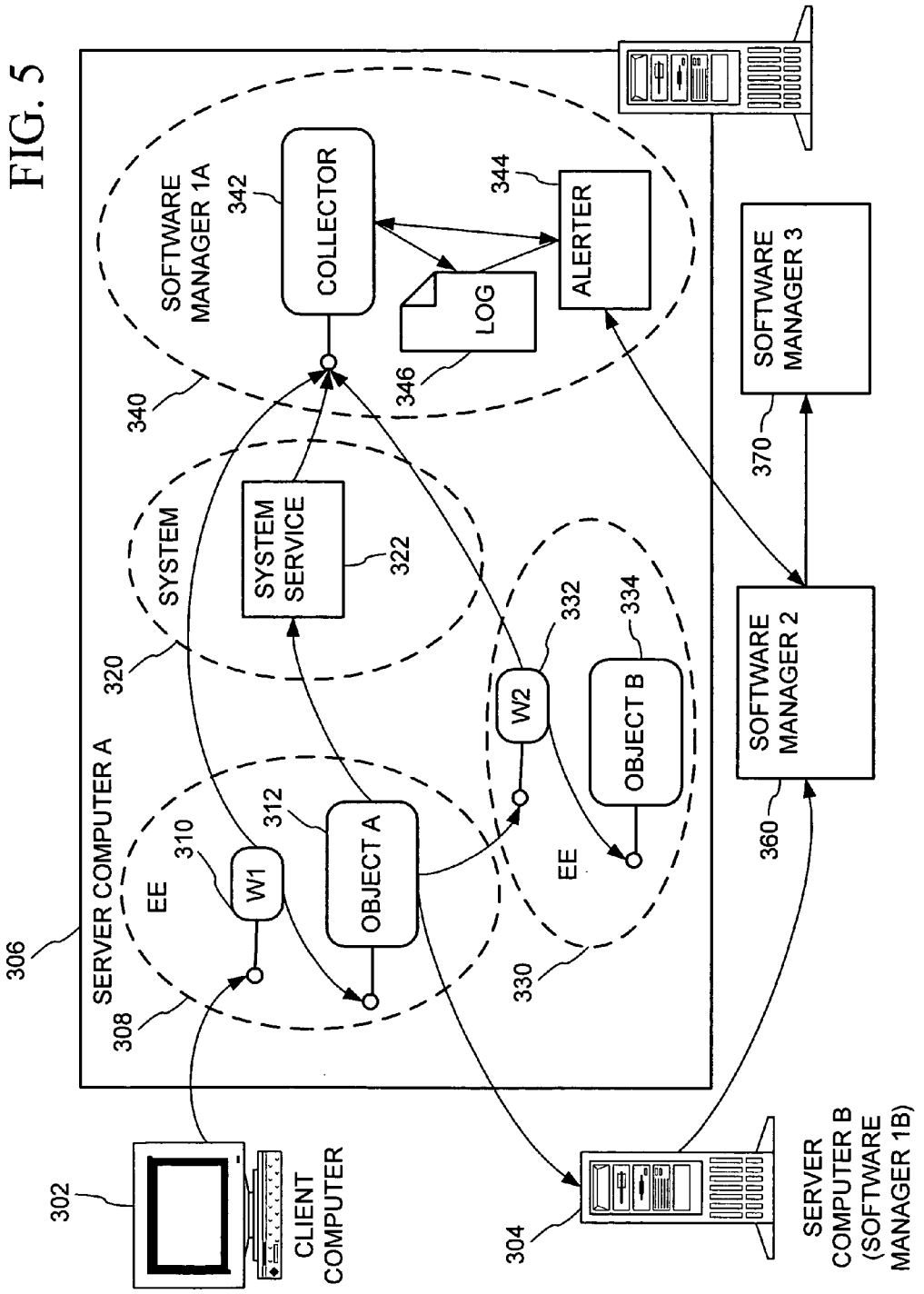

SOFTWARE MANAGER 1A
342
COLLECTOR
344
ALERTER
LOG
346
340
SOFTWARE MANAGER 3
370
SYSTEM
SYSTEM SERVICE
322
320
SOFTWARE MANAGER 2
360
W2
332
334
OBJECT B
EE
330
SERVER COMPUTER A
306
EE
310
W1
312
OBJECT A
308
302
CLIENT COMPUTER
304
SERVER COMPUTER B (SOFTWARE MANAGER 1B)

FIG. 6

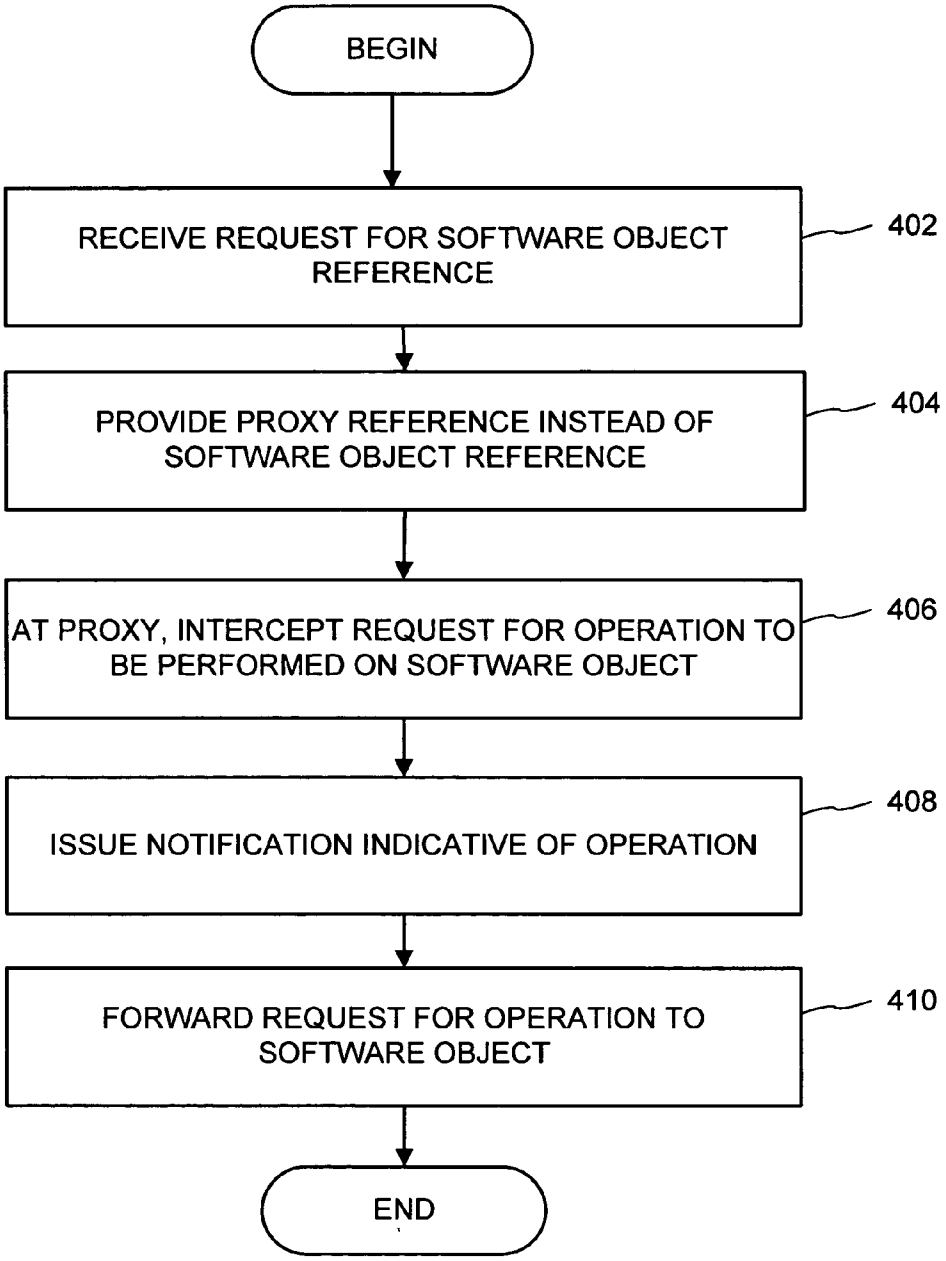

BEGIN
RECEIVE REQUEST FOR SOFTWARE OBJECT REFERENCE
402
PROVIDE PROXY REFERENCE INSTEAD OF SOFTWARE OBJECT REFERENCE
404
AT PROXY, INTERCEPT REQUEST FOR OPERATION TO BE PERFORMED ON SOFTWARE OBJECT
406
ISSUE NOTIFICATION INDICATIVE OF OPERATION
408
FORWARD REQUEST FOR OPERATION TO SOFTWARE OBJECT
410
END

FIG. 7

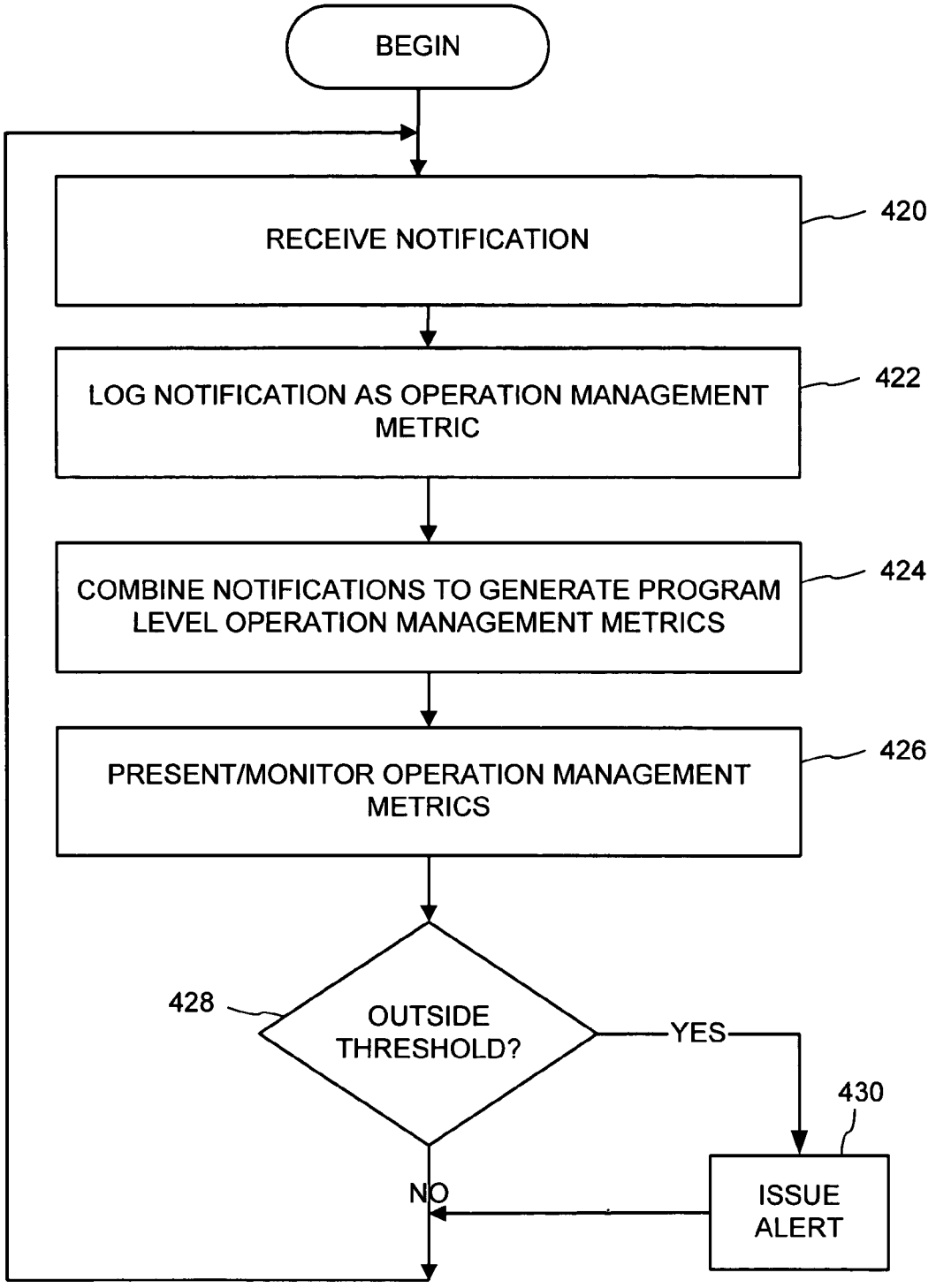
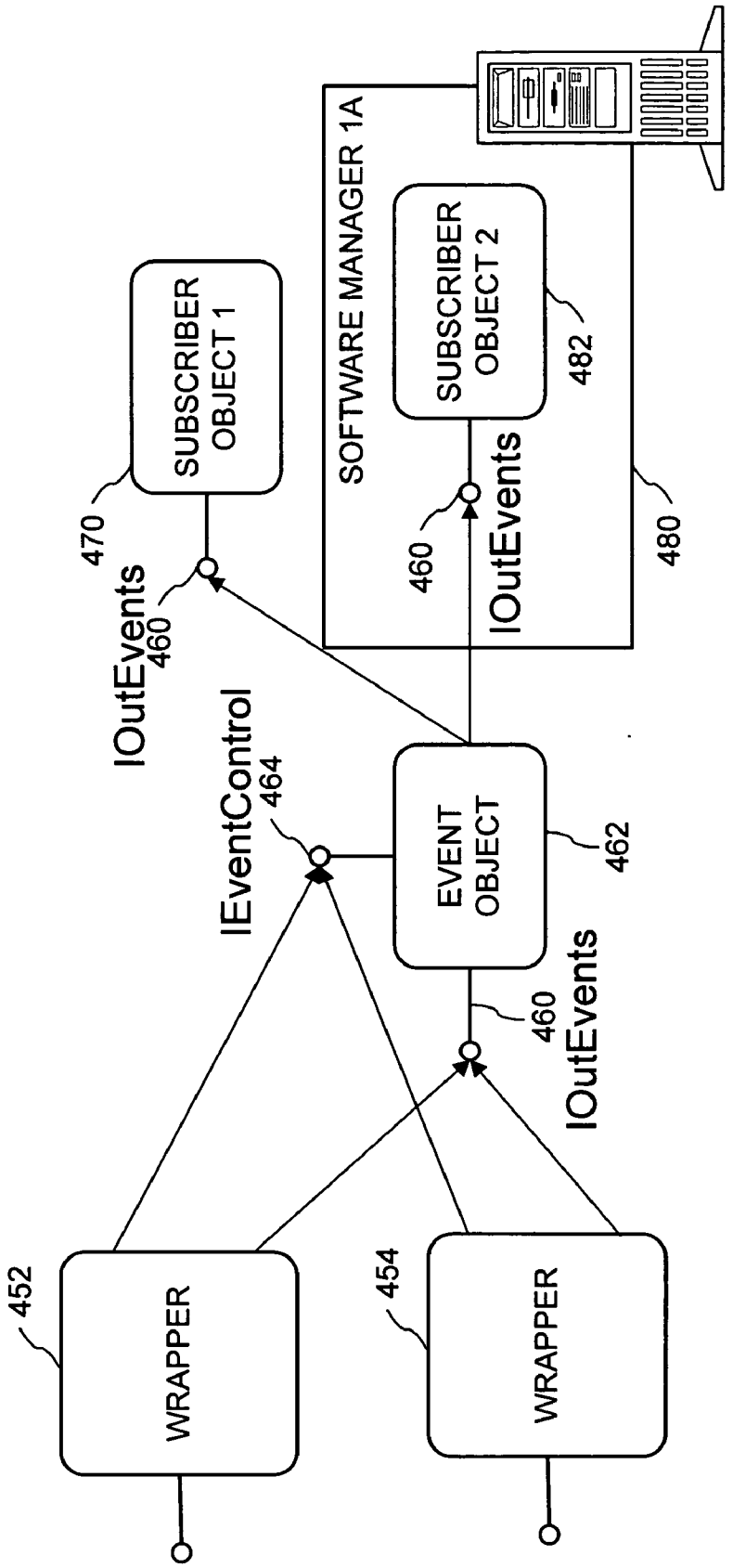

BEGIN
RECEIVE NOTIFICATION
420
LOG NOTIFICATION AS OPERATION MANAGEMENT METRIC
422
COMBINE NOTIFICATIONS TO GENERATE PROGRAM LEVEL OPERATION MANAGEMENT METRICS
424
PRESENT/MONITOR OPERATION MANAGEMENT METRICS
426
428
OUTSIDE THRESHOLD?
YES
430
NO
ISSUE ALERT

FIG. 8


SOFTWARE MANAGER 1A
SUBSCRIBER OBJECT 1
SUBSCRIBER OBJECT 2
EVENT OBJECT
WRAPPER
WRAPPER
IOutEvents
IEventControl
470
460
464
462
482
480
452
454

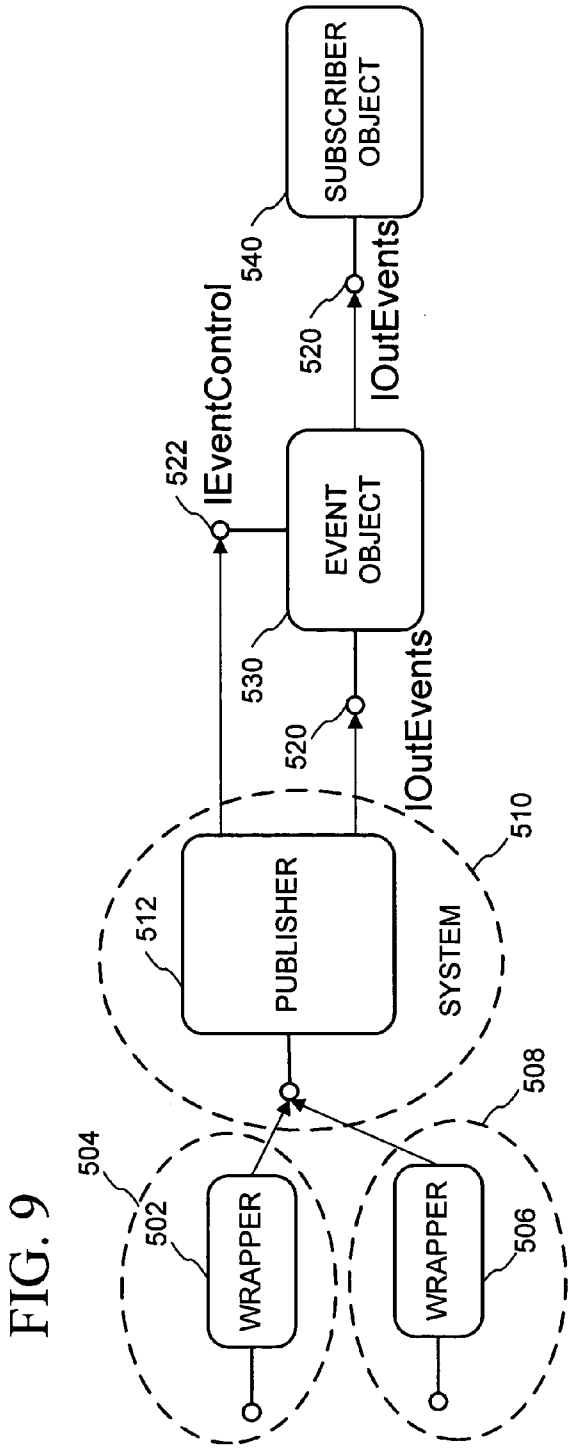

SUBSCRIBER OBJECT
540
520
IOutEvents
522
IEventControl
EVENT OBJECT
530
520
IOutEvents
510
512
PUBLISHER
SYSTEM
508
504
502
WRAPPER
WRAPPER
506

FIG. 9

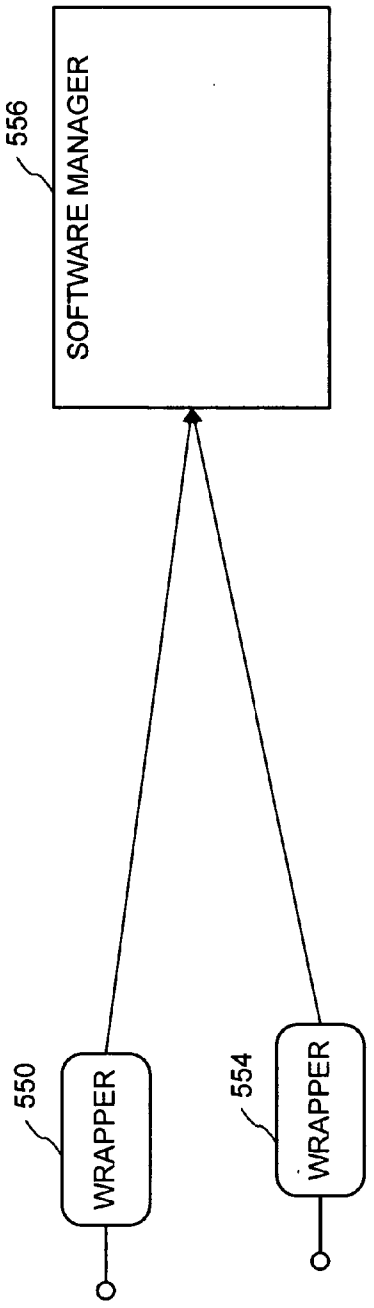

556
SOFTWARE MANAGER
550
WRAPPER
554
WRAPPER

FIG. 10

FIG. 11

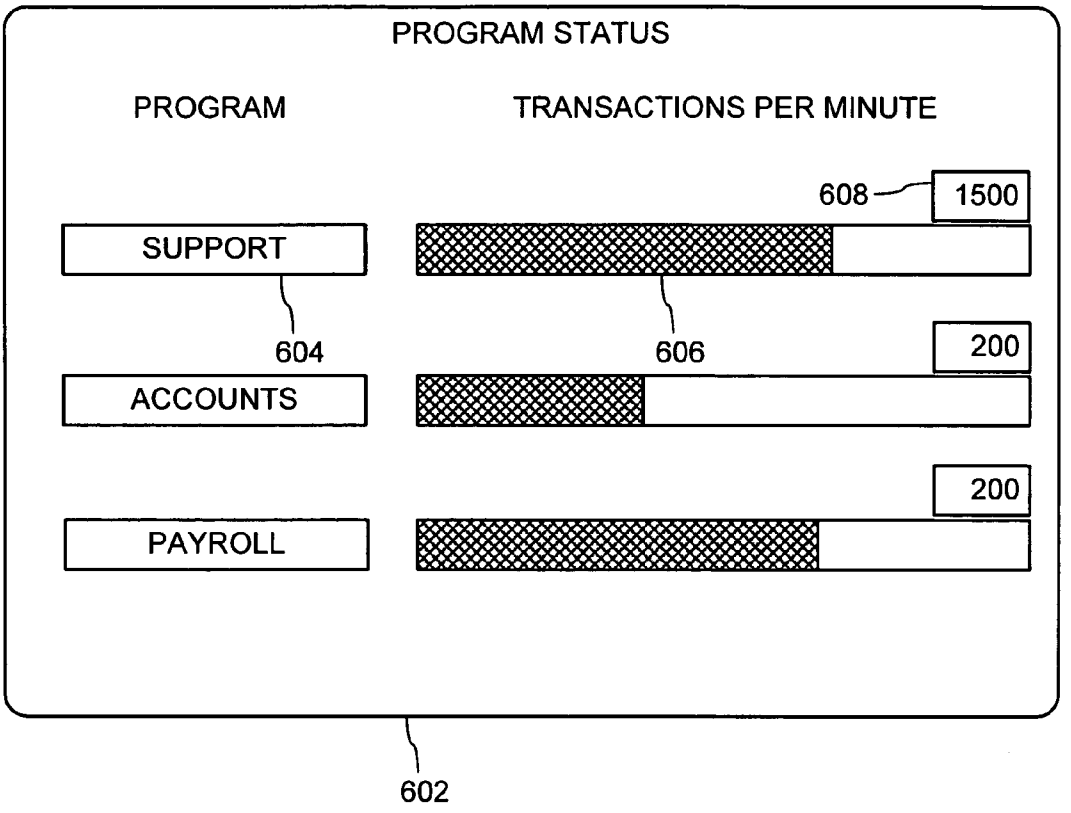
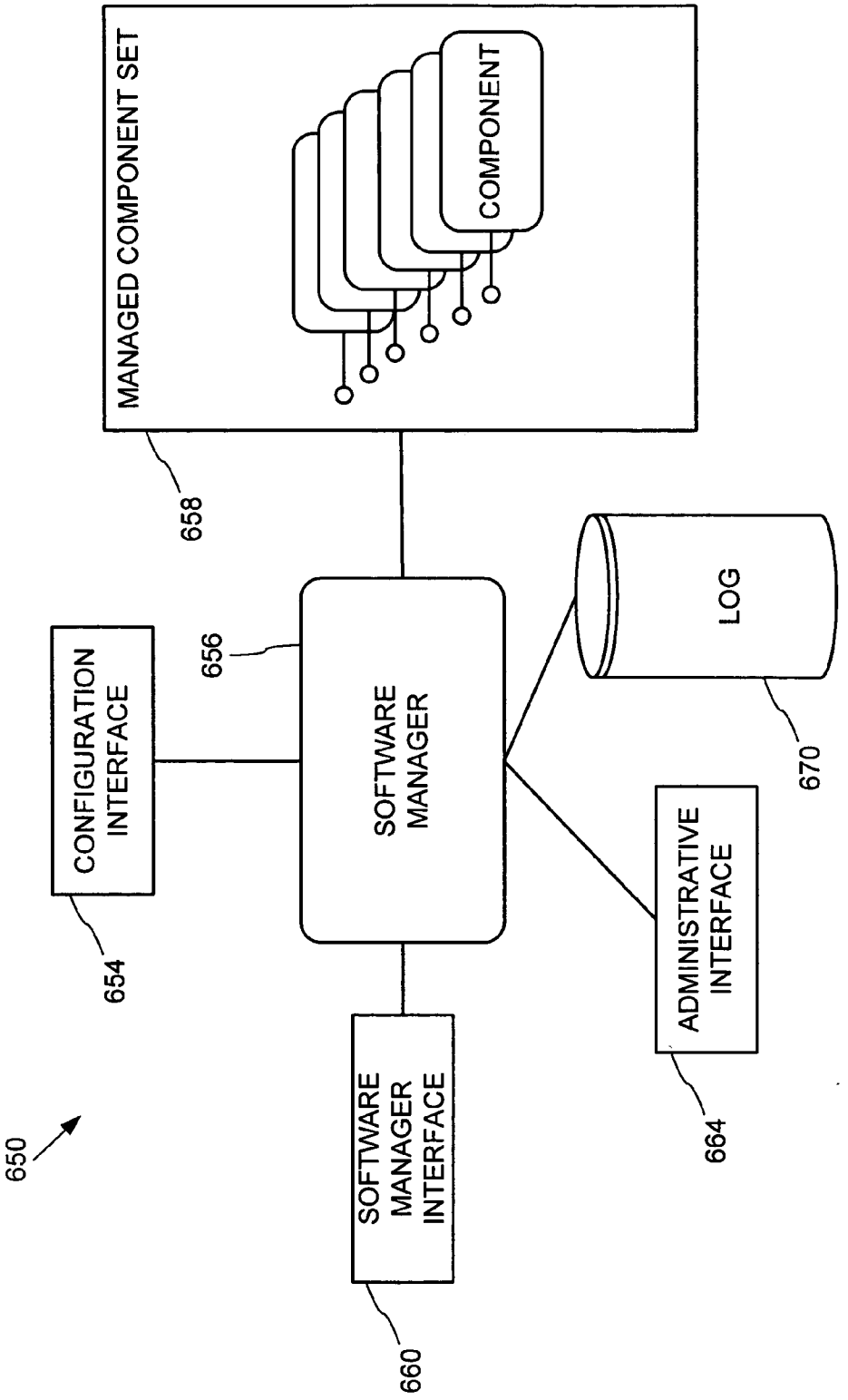

PROGRAM STATUS
PROGRAM
TRANSACTIONS PER MINUTE
608
1500
SUPPORT
604
606
200
ACCOUNTS
200
PAYROLL
602

FIG. 12


MANAGED COMPONENT SET
COMPONENT
658
656
SOFTWARE MANAGER
CONFIGURATION INTERFACE
654
LOG
670
ADMINISTRATIVE INTERFACE
664
SOFTWARE MANAGER INTERFACE
660
650

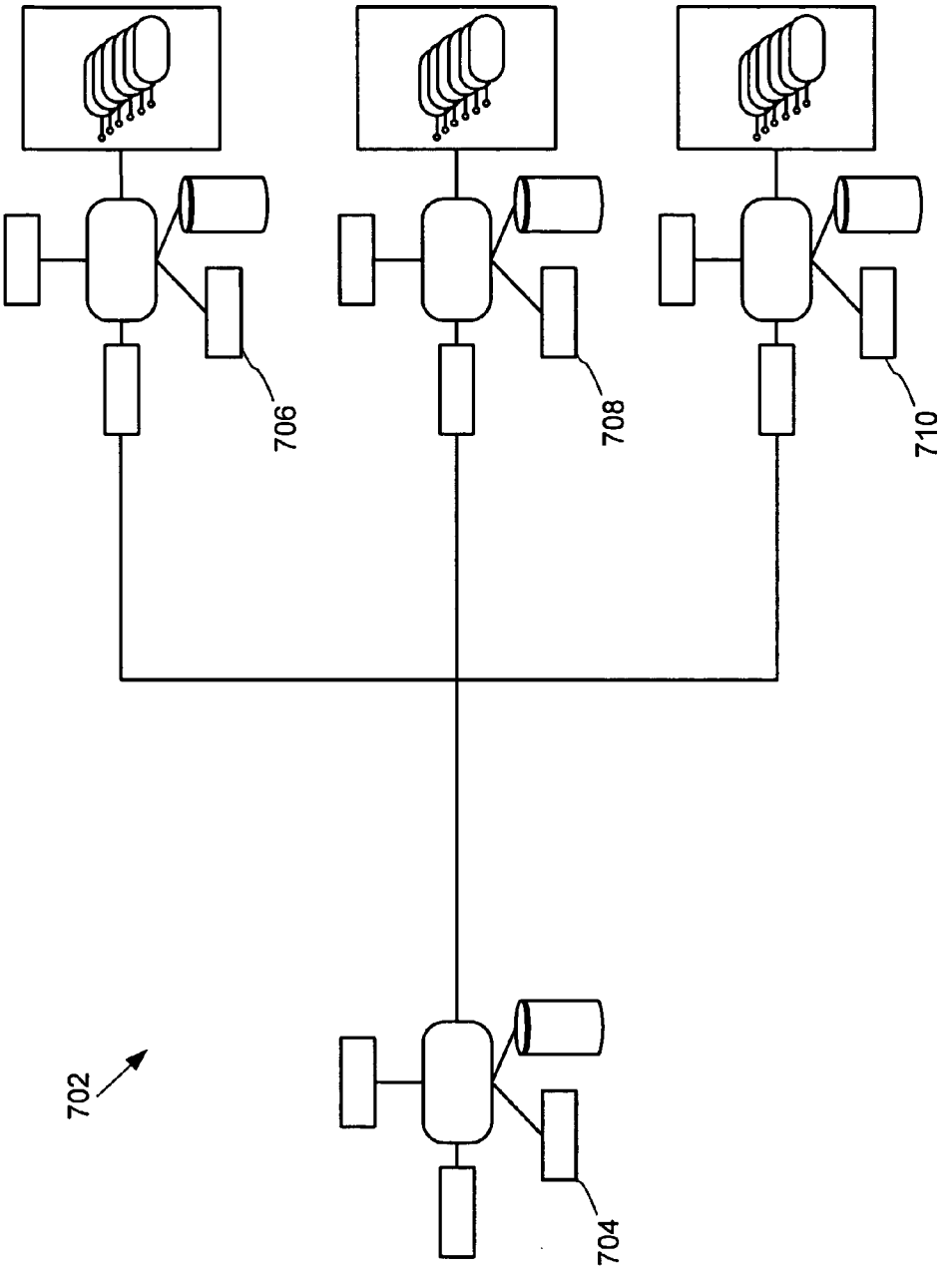


FIG. 13

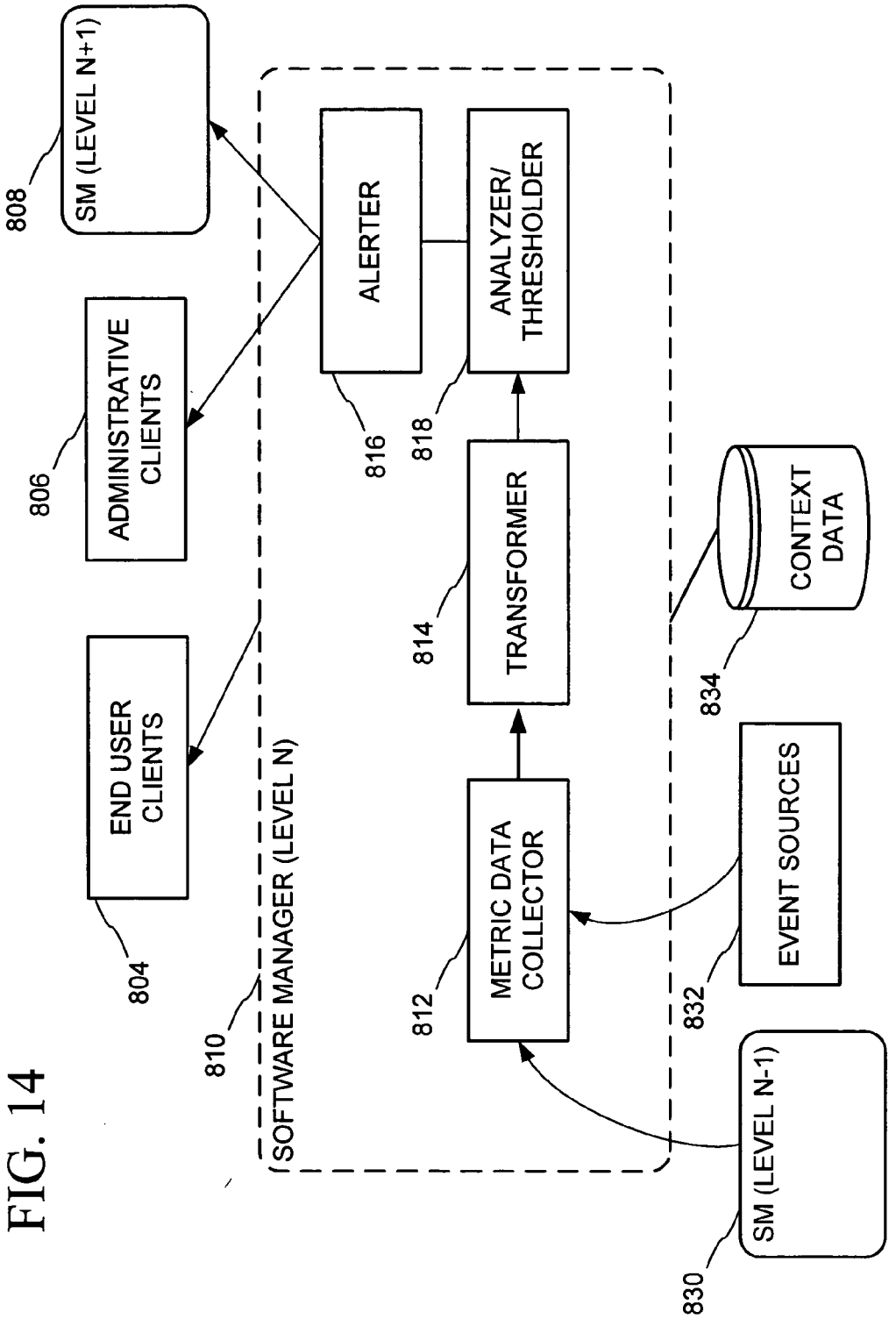

SM (LEVEL N+1)
808
ADMINISTRATIVE CLIENTS
806
END USER CLIENTS
804
SOFTWARE MANAGER (LEVEL N)
810
ALERTER
816
ANALYZER/ THRESHOLDER
818
TRANSFORMER
814
METRIC DATA COLLECTOR
812
CONTEXT DATA
834
EVENT SOURCES
832
SM (LEVEL N-1)
830

FIG. 14

FIG. 15

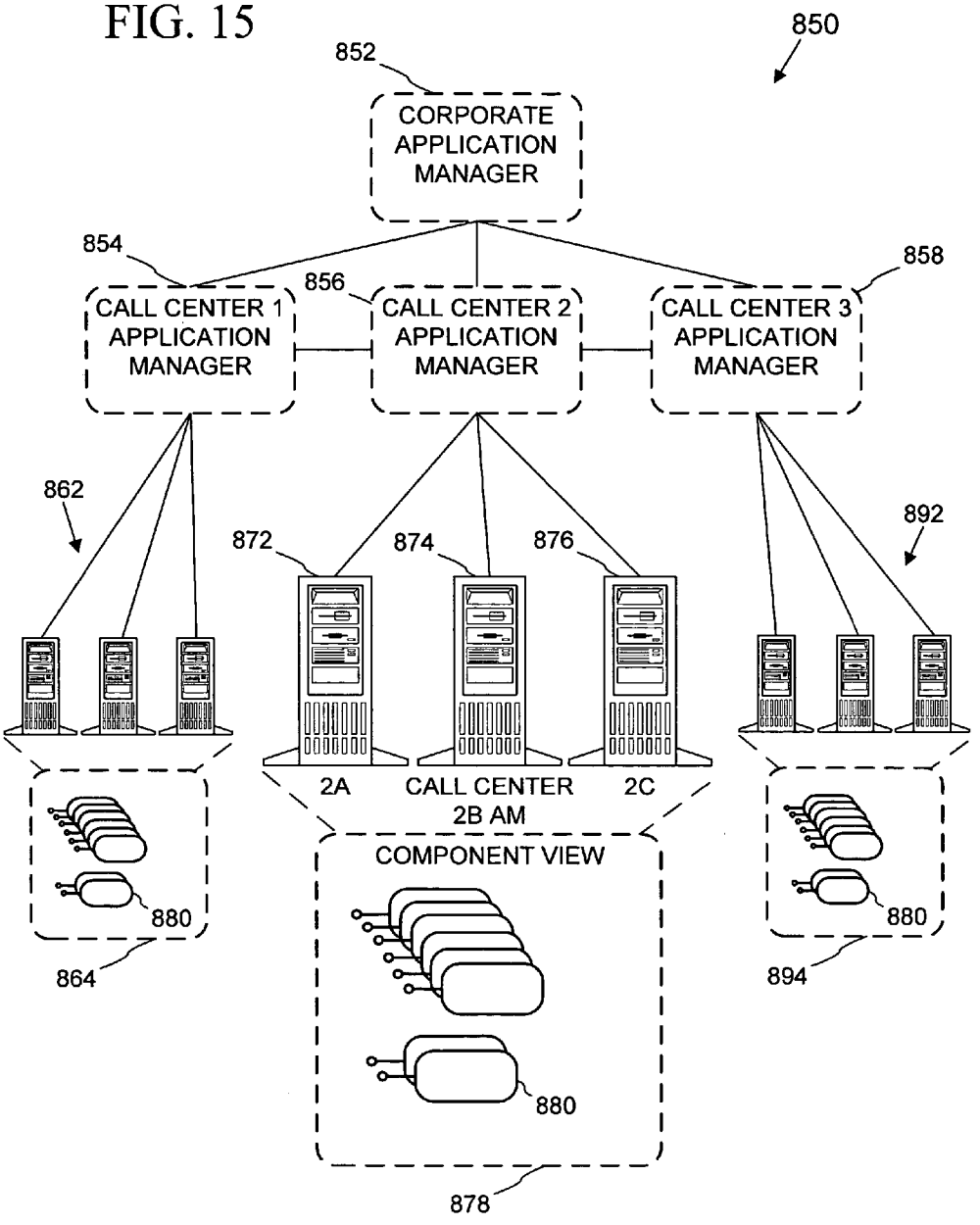

850
852
CORPORATE APPLICATION MANAGER
854
856
858
CALL CENTER 1 APPLICATION MANAGER
CALL CENTER 2 APPLICATION MANAGER
CALL CENTER 3 APPLICATION MANAGER
862
872
874
876
892
2A
CALL CENTER 2B AM
2C
COMPONENT VIEW
880
880
880
864
878
894

FIG. 16

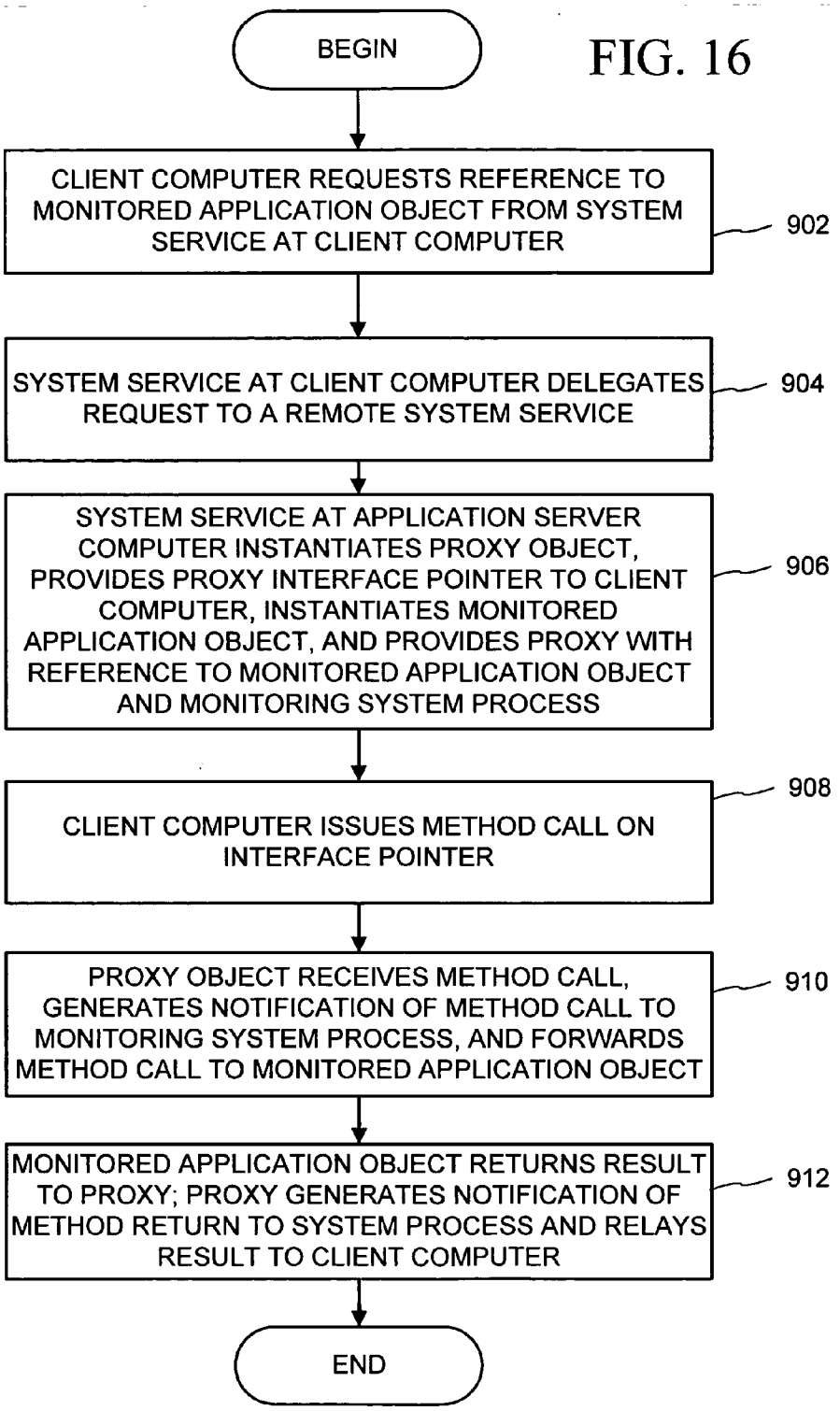

BEGIN
CLIENT COMPUTER REQUESTS REFERENCE TO MONITORED APPLICATION OBJECT FROM SYSTEM SERVICE AT CLIENT COMPUTER
902
SYSTEM SERVICE AT CLIENT COMPUTER DELEGATES REQUEST TO A REMOTE SYSTEM SERVICE
904
SYSTEM SERVICE AT APPLICATION SERVER COMPUTER INSTANTIATES PROXY OBJECT, PROVIDES PROXY INTERFACE POINTER TO CLIENT COMPUTER, INSTANTIATES MONITORED APPLICATION OBJECT, AND PROVIDES PROXY WITH REFERENCE TO MONITORED APPLICATION OBJECT AND MONITORING SYSTEM PROCESS
906
CLIENT COMPUTER ISSUES METHOD CALL ON INTERFACE POINTER
908
PROXY OBJECT RECEIVES METHOD CALL, GENERATES NOTIFICATION OF METHOD CALL TO MONITORING SYSTEM PROCESS, AND FORWARDS METHOD CALL TO MONITORED APPLICATION OBJECT
910
MONITORED APPLICATION OBJECT RETURNS RESULT TO PROXY; PROXY GENERATES NOTIFICATION OF METHOD RETURN TO SYSTEM PROCESS AND RELAYS RESULT TO CLIENT COMPUTER
912
END

FIG. 17

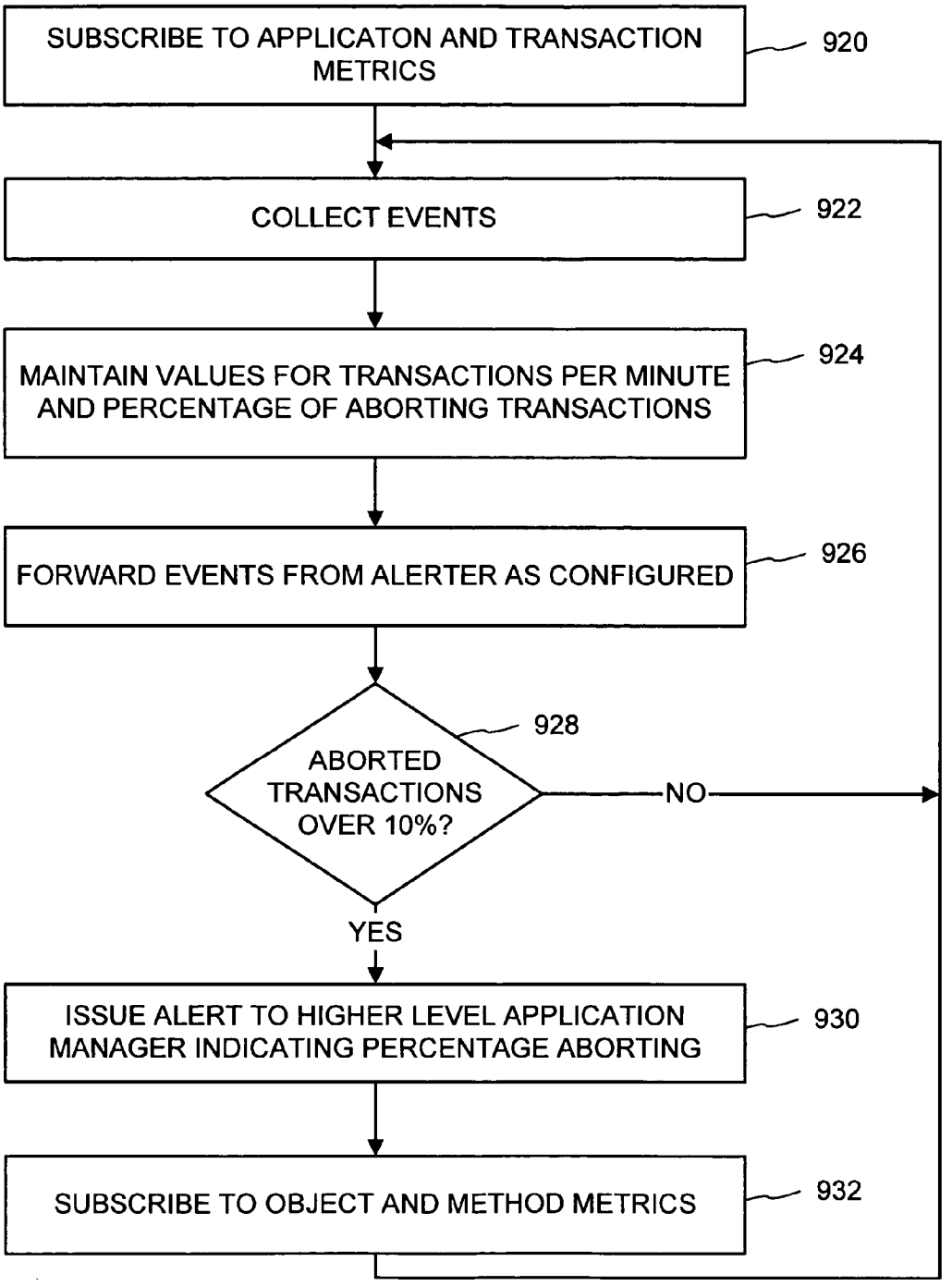
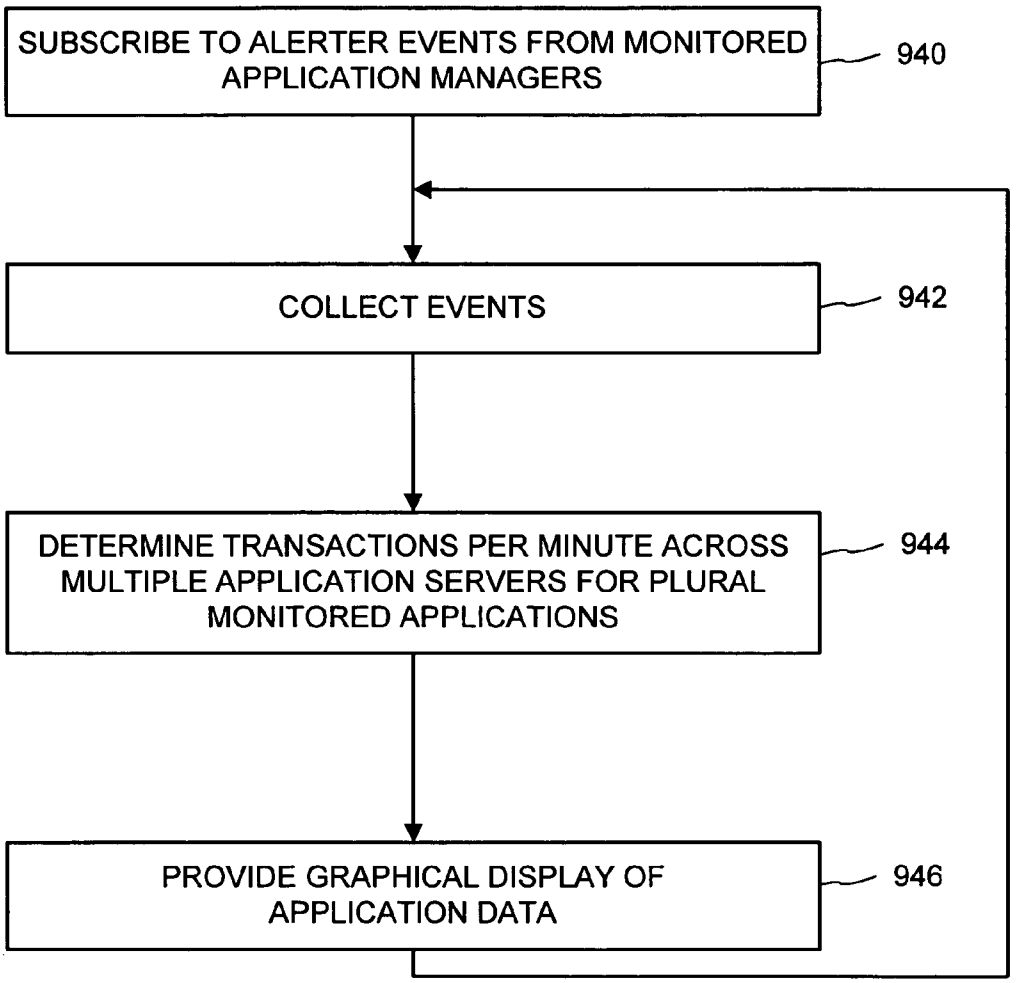

SUBSCRIBE TO APPLICATON AND TRANSACTION METRICS
920
COLLECT EVENTS
922
MAINTAIN VALUES FOR TRANSACTIONS PER MINUTE AND PERCENTAGE OF ABORTING TRANSACTIONS
924
FORWARD EVENTS FROM ALERTER AS CONFIGURED
926
928
ABORTED TRANSACTIONS OVER 10%?
NO
YES
ISSUE ALERT TO HIGHER LEVEL APPLICATION MANAGER INDICATING PERCENTAGE ABORTING
930
SUBSCRIBE TO OBJECT AND METHOD METRICS
932

FIG. 18


SUBSCRIBE TO ALERTER EVENTS FROM MONITORED APPLICATION MANAGERS
940
COLLECT EVENTS
942
DETERMINE TRANSACTIONS PER MINUTE ACROSS MULTIPLE APPLICATION SERVERS FOR PLURAL MONITORED APPLICATIONS
944
PROVIDE GRAPHICAL DISPLAY OF APPLICATION DATA
946

OBJECT-BASED SOFTWARE MANAGEMENT

CROSS-REFERENCE TO RELATED APPLICATION

[0001] This is a continuation of U.S. patent application Ser. No. 09/393,011, filed Sep. 9, 1999, now U.S. Pat. No. 6,748,555, the disclosure of which is hereby incorporated herein by reference.

TECHNICAL FIELD

[0002] The invention relates to managing object-based software, and more particularly relates to generating operational management information transparently to monitored software objects.

BACKGROUND OF THE INVENTION

[0003] A variety of systems have evolved for accommodating software objects in a variety of information processing scenarios. For example, a server application running software objects on a host or server computer in a distributed network can provide services or functions for client applications running on terminal or workstation computers of the network which are operated by a multitude of users. Common examples of such server applications include software for processing class registrations at a university, travel reservations, money transfers at a bank, and sales at a retail business. In these examples, the processing services provided by the server application may update databases of class schedules, hotel reservations, account balances, product shipments, payments, or inventory for actions initiated by the individual users at their respective stations. A common way to implement these applications is by exchanging data through a web site hosted on the server.

[0004] As organizations become more dependent on their information systems, successful business operation is increasingly tied to application software availability. Thus, certain applications need to be available at all times; any interruption in service results in lost customers or missed deadlines. Applications playing an integral part in business operations are sometimes called "mission critical" or "24×7" applications. For example, if an order center is open twenty-four hours a day to accept customer information requests and orders, inferior performance or failure at any time impairs business operation. To avoid service interruption, an organization assigns the task of monitoring application performance and availability to a team of information technology professionals known as system administrators.

[0005] The system administrators strive to ensure the server applications provide consistent, quality service. However, maintaining service is an ongoing battle against a variety of factors. Inevitably, an application becomes overloaded with requests for service, or software anomalies crash the application altogether, leading to inferior or interrupted performance and loss of mission critical functions. If the system administrators wait for customer complaints before taking action, some users have already experienced poor service. Also, if the system administrators wait until a server fails completely (or "crashes"), they must expend considerable time and effort to restore service. And, as the number of applications and servers grows into an enterprise-wide system, inferior performance may go unnoticed. Finally, the system administrators typically find themselves chasing down urgent failures rather than focusing on improving application performance. Ideally, then, system administrators should monitor application performance to avoid problems instead of reacting to user complaints.

[0006] To achieve this end, system administrators turn to management software, to provide an indication of how each system is performing and whether the system has failed. In this way, the system administrators avoid service outages and can see a particular system needs attention because performance is degrading.

[0007] Two techniques for gathering information about a system's operation (sometimes called "operational management information") have developed for management software: non-intrusive and intrusive. Non-intrusive techniques require little or no modification to existing applications but provide limited information. For example, non-instrusive management software may monitor free disk space or sniff network packets. Additional features include an alert system; the system administrator can specify criteria (e.g., disk free space falls to under 1 percent) that will trigger an alert (e.g., page the administrator). However, non-intrusive techniques are of limited use because they typically monitor the underlying system rather than a particular application. Thus, a non-intrusive technique typically cannot pinpoint what application functionality is causing trouble. For example, in the above example, the alert does not explain why the disk usage has increased or which application is responsible for the increase.

[0008] Intrusive techniques offer additional information not provided by non-intrusive techniques. In one intrusive technique, a process called instrumentation is applied to each application. To instrument an application, programming instructions are added throughout the application to send information to management software. The instructions may relay information indicating a location within the application, allowing the management software to determine what portions of the application are responsible for generating error conditions or triggering alarms.

[0009] For example, code could be placed in a customer order application to send a notification to the management software when a customer order is received and another notification when processing for the order is completed. In this way, the management software can provide information about the number of orders received and the number of orders completed per minute. If the number of orders completed per minute drops to zero while the number of orders received per minute remains constant, it is likely that some portion of the system has failed; further it appears the problem is with processing orders, not receiving them. Thus, an alarm set to inform the administrator when the orders completed rate drops below 20% of the orders received rate indicates both that there is a problem and that the administrator should investigate why orders are not being completed.

[0010] However, intrusive management techniques suffer from various problems. First, the instrumentation process requires an application developer to undergo the process of including extra code at development time or retrofitting a current application with instrumentation code. And, during the instrumentation process, the developer must determine how much instrumentation is sufficient. There are numerous degrees of instrumentation, and it is not always clear at

application development time how much instrumentation is desired. Excess instrumentation can degrade performance, but too little might not provide sufficient information to adequately manage the application. If the wrong decisions are made, the application must be modified yet again.

[0011] Thus, instrumentation requires exercise of seasoned judgment and care on the part of the application developer, who may consult with the system administrators to incorporate their experience into the instrumentation process. As a result, instrumentation requires expertise in high demand, and the process drains resources from the primary tasks of developing, improving, and maintaining the application. In addition, since instrumentation itself can introduce new problems, the instrumented version of the software must be tested to detect newly introduced software bugs.

[0012] Second, instrumentation can be implemented according to one of a variety of instrumentation standards, and an application instrumented according to one standard may not work with management software expecting a different instrumentation standard. Thus, if two departments using different standards are combined, two different application management systems must be run in parallel unless the software is re-instrumented.

[0013] Thus, system administrators are forced to choose between a non-intrusive monitoring technique which provides no information at the application level and instrumentation, which requires an experienced software developer who modifies an application to accommodate specific management software.

SUMMARY OF THE INVENTION

[0014] The invention includes a method and system for managing a set of objects, such as those forming an application or other program. In an architecture accommodating software objects, operations on software objects are monitored to generate information for a software management software system. The result is automatic instrumentation performed at run time.

[0015] Thus, a software developer can write software without knowledge of software management technology. Subsequently, when objects are created at run time, wrappers associated with the objects generate notifications representing operational management information. Further, plural notifications can be transformed into further operational management information for incorporation into an enterprise software management system. Thus, the logic for sending notifications is contained outside the software objects, and software developers can avoid instrumenting their software. As a result, a developer can focus energy on other tasks related to software development, such as business logic. Thus, software management technology is brought to organizations without access to software management expertise and experience.

[0016] The operational management information can include information for grouping the information, such as for grouping information related to software residing on plural computers. Thus, a system administrator or an automated software manager can monitor a program's performance, even if the program is composed of objects scattered over several computers. The system supports a comfort screen to provide assurance that software is operating normally and can generate an alert when operational management information meets certain criteria. The system can automatically act on alerts by, for example, paging an administrator when a particular value falls outside an acceptable threshold.

[0017] In one aspect of the invention, the management software provides object-related notifications based on external interactions with the objects. For example, a notification is sent when a client program invokes a software object's method. In this way, execution of the software can be traced and stored in a log. Such a log is useful, for example, in determining the source of software failure.

[0018] In another aspect of the invention, notifications are published as events in a loosely coupled publish and subscribe arrangement. In this way, management software can subscribe to the events, log them, and transform them into additional operational management metrics such as operational management metrics indicating a particular program's performance. The events are grouped into categories to facilitate selection of events of interest, and the architecture supports user-defined events. To improve performance and avoid creating numerous publishers, plural notifications can be collected by a single publisher in a system process. A subscriber can specify which notifications it wishes to receive, allowing an automated software manager to automatically (e.g., via an alert) subscribe to additional events related to a problem when the problem is detected.

[0019] Since the publisher and subscriber are loosely coupled, the architecture easily accommodates a custom software manager. Also, a program can monitor itself by subscribing to the events and monitoring them.

[0020] In yet another aspect of the invention, notifications are generated by system services for incorporation into the object-related notifications. In this way, a more complete picture of a program's performance is assembled, including information on transactions and resource allocation.

[0021] In still another aspect of the invention, events can be collected from a software manager of a lower hierarchical position and relayed to an enterprise software manager of a higher hierarchical position. In this way, software performance can be evaluated at various levels throughout the enterprise. If a problem is detected at a high level, lower levels can be examined to zoom to problem details.

[0022] Yet another aspect of the invention involves tracing software operation by activity. In this way, the system provides a trace of the string of actions performed for a particular user interaction with the software and aids in tuning system performance based on activity volume and resources consumed by the activities.

[0023] Additional features and advantages of the invention will be made apparent from the following detailed description of illustrated embodiments, which proceeds with reference to the accompanying drawings.

BRIEF DESCRIPTION OF THE DRAWINGS

[0024] **FIG. 1** is a block diagram of a computer system that may be used to implement the described software management framework for object-based software.

[0025] **FIG. 2** is a block diagram of an object conforming to the Component Object Model specification of Microsoft Corporation, which may be used to construct objects for developing an object-based program managed by the described software management framework.

[0026] **FIG. 3** is a block diagram showing an interceptor interposed between a client program and an object monitored by a software manager.

[0027] **FIG. 4** is a block diagram showing a wrapper interposed between a client program and a software object monitored by a software manager.

[0028] **FIG. 5** is a block diagram showing an architecture for implementing an enterprise software management system transparently to monitored software objects.

[0029] **FIG. 6** is a flowchart showing a method for collecting notifications transparently to the software objects being monitored.

[0030] **FIG. 7** is a flowchart showing a method for collecting and monitoring notifications generated by a method such as that shown in **FIG. 6**.

[0031] **FIG. 8** is a block diagram showing a model for collecting notifications in a loosely coupled publish and subscribe event arrangement.

[0032] **FIG. 9** is a block diagram showing a model for collecting notifications in a loosely coupled publish and subscribe event arrangement using an intermediary system publisher.

[0033] **FIG. 10** is a block diagram showing a model for collecting notifications in a tightly coupled notification arrangement.

[0034] **FIG. 11** is a view of a user interface providing visual confirmation of programs' normal operation.

[0035] **FIG. 12** is a block diagram showing a software manager and accompanying interfaces.

[0036] **FIG. 13** is a block diagram showing a hierarchical arrangement of software managers.

[0037] **FIG. 14** is a block diagram showing a software manager and information flow into and out of the software manager.

[0038] **FIG. 15** is a block diagram showing an exemplary enterprise application management arrangement.

[0039] **FIG. 16** is a flowchart showing a method for generating notifications for monitoring an object transparently to the monitored object.

[0040] **FIG. 17** is a flowchart showing a method for collecting events generated by a method such as that shown in **FIG. 16** and dynamically selecting monitored operation management metrics.

[0041] **FIG. 18** is a flowchart showing a method for collecting events generated by a method such as that shown in **FIG. 16** and providing program information for a program scattered across multiple application servers.

DETAILED DESCRIPTION OF THE INVENTION

[0042] The invention is directed toward a method and system for providing an object execution environment with a software management framework providing automatic collection of operational management information for programs. In one embodiment illustrated herein, the invention is incorporated into an object services component entitled "COM+" of an operating system entitled "MICROSOFT WINDOWS 2000," both marketed by Microsoft Corporation of Redmond, Washington. Briefly described, this software is a scaleable, high-performance network and computer operating system providing an object execution environment for object programs conforming to COM. COM+ also supports distributed client/server computing. The COM+ component incorporates new technology as well as object services from prior object systems, including the MICROSOFT Component Object Model (COM), the MICROSOFT Distributed Component Object Model (DCOM), and the MICROSOFT Transaction Server (MTS).

Exemplar Operating Environment

[0043] **FIG. 1** and the following discussion are intended to provide a brief, general description of a suitable computing environment in which the invention may be implemented. While the invention will be described in the general context of computer-executable instructions of a computer program that runs on a computer, those skilled in the art will recognize the invention also may be implemented in combination with other programs. Generally, programs include routines, software objects (also called components), data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including single- or multiprocessor computer systems, minicomputers, mainframe computers, as well as personal computers, hand-held computing devices, microprocessor-based or programmable consumer electronics, and the like. The illustrated embodiment of the invention also is practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a communications network. But, some embodiments of the invention can be practiced on stand-alone computers. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

[0044] With reference to **FIG. 1**, an exemplary system for implementing the invention includes a conventional computer **20**, including a processing unit **21**, a system memory **22**, and a system bus **23** that couples various system components including the system memory to the processing unit **21**. The processing unit may be any of various commercially available processors, including Intel ×86, Pentium and compatible microprocessors from Intel and others, including Cyrix, AMD and Nexgen; Alpha from Digital; MIPS from MIPS Technology, NEC, IDT, Siemens, and others; and the PowerPC from IBM and Motorola. Dual microprocessors and other multi-processor architectures also can be used as the processing unit **21**.

[0045] The system bus may be any of several types of bus structure including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of conventional bus architectures such as PCI, VESA, Microchannel, ISA and EISA, to name a few. The system memory includes read only memory (ROM) **24** and random access memory (RAM) **25**. A basic input/output system (BIOS), containing the basic routines that help to transfer informa-

tion between elements within the computer **20**, such as during start-up, is stored in ROM **24**.

[**0046**] The computer **20** further includes a hard disk drive **27**, a magnetic disk drive **28**, e.g., to read from or write to a removable disk **29**, and an optical disk drive **30**, e.g., for reading a CD-ROM disk **31** or to read from or write to other optical media. The hard disk drive **27**, magnetic disk drive **28**, and optical disk drive **30** are connected to the system bus **23** by a hard disk drive interface **32**, a magnetic disk drive interface **33**, and an optical drive interface **34**, respectively. The drives and their associated computer-readable media provide nonvolatile storage of data, data structures, computer-executable instructions, etc. for the computer **20**. Although the description of computer-readable media above refers to a hard disk, a removable magnetic disk and a CD, it should be appreciated by those skilled in the art that other types of media which are readable by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, and the like, may also be used in the exemplary operating environment.

[**0047**] A number of programs may be stored in the drives and RAM **25**, including an operating system **35**, one or more application programs **36**, other programs **37**, and program data **38**. The operating system **35** in the illustrated computer may be the MICROSOFT WINDOWS NT Server operating system, together with the before mentioned MICROSOFT Transaction Server.

[**0048**] A user may enter commands and information into the computer **20** through a keyboard **40** and pointing device, such as a mouse **42**. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit **21** through a serial port interface **46** that is coupled to the system bus, but may be connected by other interfaces, such as a parallel port, game port or a universal serial bus (USB). A monitor **47** or other type of display device is also connected to the system bus **23** via an interface, such as a video adapter **48**. In addition to the monitor, computers typically include other peripheral output devices (not shown), such as speakers and printers.

[**0049**] The computer **20** may operate in a networked environment using logical connections to one or more remote computers, such as a remote client computer **49**. The remote computer **49** may be a workstation, a terminal computer, another server computer, a router, a peer device or other common network node, and typically includes many or all of the elements described relative to the computer **20**, although only a memory storage device **50** has been illustrated in **FIG. 1**. The logical connections depicted in **FIG. 1** include a local area network (LAN) **51** and a wide area network (WAN) **52**. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets, extranets, and the Internet.

[**0050**] When used in a LAN networking environment, the computer **20** is connected to the local network **51** through a network interface or adapter **53**. When used in a WAN networking environment, the computer **20** typically includes a modem **54**, or is connected to a communications server on the LAN, or has other means for establishing communications over the wide area network **52**, such as the Internet. The modem **54**, which may be internal or external, is connected to the system bus **23** via the serial port interface **46**. In a networked environment, program modules depicted relative to the computer **20**, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

[**0051**] In accordance with the practices of persons skilled in the art of computer programming, the present invention is described below with reference to acts and symbolic representations of operations that are performed by the computer **20**, unless indicated otherwise. Such acts and operations are sometimes referred to as being computer-executed. It will be appreciated that the acts and symbolically represented operations include the manipulation by the processing unit **21** of electrical signals representing data bits which causes a resulting transformation or reduction of the electrical signal representation, and the maintenance of data bits at memory locations in the memory system (including the system memory **22**, hard drive **27**, floppy disks **29**, and CD-ROM **31**) to thereby reconfigure or otherwise alter the computer system's operation, as well as other processing of signals. The memory locations where data bits are maintained are physical locations that have particular electrical, magnetic, or optical properties corresponding to the data bits.

Object Overview

[**0052**] **FIG. 2** and the following discussion are intended to provide an overview of software objects, using the MICROSOFT Component Object Model (COM) as an exemplary object model. In the illustrated embodiments, a software management framework is implemented in an extension to the MICROSOFT COM Environment termed "COM+." COM is a model for accommodating software objects and can be implemented on a variety of platforms, such as the MICROSOFT WINDOWS NT operating system. In the illustrated embodiments of the invention, the software objects conform to the MICROSOFT Component Object Model ("COM") specification (i.e., are implemented as a "COM Object"**76**) and are executed using the COM+ services of the MICROSOFT WINDOWS 2000 operating system, but alternatively may be implemented according to other object standards (including the CORBA (Common Object Request Broker Architecture) specification of the Object Management Group and JavaBeans by Sun Microsystems) and executed under object services of another operating system. The COM specification defines binary standards for objects and their interfaces which facilitate the integration of software objects into programs. (For a detailed discussion of COM and OLE, see Kraig Brockschmidt, *Inside OLE, Second Edition*, Microsoft Press, Redmond, Wash. (1995)).

[**0053**] In accordance with COM, the COM object **60** is represented in the computer system **20** (**FIG. 1**) by an instance data structure **62**, a virtual function table **64**, and member methods (also called member functions) **66-68**. The instance data structure **62** contains a pointer **70** to the virtual function table **64** and data **72** (also referred to as data members, or properties of the object). A pointer is a data value that holds the address of an item. The virtual function table **64** contains entries **76-78** for the member methods **66-68**. Each of the entries **76-78** contains a reference to the code **66-68** that implements the corresponding member methods.

[0054] The pointer **70**, the virtual function table **64**, and the member methods **66-68** implement an interface of the COM object **60**. By convention, the interfaces of a COM object are illustrated graphically as a plug-in jack as shown for the software objects **312** and **334** in **FIG. 5**. Also, interfaces conventionally are given names beginning with a capital "I." In accordance with COM, the COM object **60** can include multiple interfaces, which are implemented with one or more virtual function tables. The member function of an interface is denoted as "IInterfaceName::MethodName."

[0055] The virtual function table **64** and member methods **66-68** of the COM object **60** are provided by an object server program **80** (hereafter "object server DLL") which is stored in the computer **20** (**FIG. 1**) as a dynamic link library file (denoted with a ".dll" file name extension). In accordance with COM, the object server DLL **80** includes code for the virtual function table **64** and member methods **66-68** of the classes that it supports, and also includes a class factory **82** that generates the instance data structure **62** for an object of the class.

[0056] Other objects and programs (referred to as a "client" of the COM object **60**) access the functionality of the COM object by invoking the member methods through the COM object's interfaces. Typically however, the COM object is first instantiated (i.e., by causing the class factory to create the instance data structure **62** of the object); and the client obtains an interface pointer to the COM object.

[0057] Before the COM object **60** can be instantiated, the object is first installed on the computer **20**. Typically, installation involves installing a group of related objects called a package. The COM object **60** is installed by storing the object server DLL file(s) **80** that provides the object in data storage accessible by the computer **20** (typically the hard drive **27**, shown in **FIG. 1**), and registering COM attributes (e.g., class identifier, path and name of the object server DLL file **80**, etc.) of the COM object in one or more data stores storing configuration information. Configuration data stores for the object include the registry and the catalog.

[0058] A client requests instantiation of the COM object using system-provided services and a set of standard, system-defined component interfaces based on class and interface identifiers assigned to the COM Object's class and interfaces. More specifically, the services are available to client programs as application programming interface (API) functions provided in the COM+ library, which is a component of the MICROSOFT WINDOWS 2000 operating system in a file named "OLE32.DLL." Other versions of COM+ or other object services may use another file or another mechanism. Also in COM+, classes of COM objects are uniquely associated with class identifiers ("CLSIDs"), and registered by their CLSID in the registry (or the catalog, or both). The registry entry for a COM object class associates the CLSID of the class with information identifying an executable file that provides the class (e.g., a DLL file having a class factory to produce an instance of the class). Class identifiers are 128-bit globally unique identifiers ("GUIDs") that the programmer creates with a COM+ service named "CoCreateGUID" (or any of several other APIs and utilities that are used to create universally unique identifiers) and assigns to the respective classes. The interfaces of a component additionally are associated with interface identifiers ("IIDs").

[0059] In particular, the COM+ library provides an API function, "CoCreateInstance( )," that the client program can call to request creation of a component using its assigned CLSID and an IID of a desired interface. In response, the "CoCreateInstance( )" API looks up the registry entry of the requested CLSID in the registry to identify the executable file for the class. The "CoCreateInstance( )" API function then loads the class' executable file, and uses the class factory in the executable file to create an instance of the COM object **60**. Finally, the "CoCreateInstance( )" API function returns a pointer of the requested interface to the client program. The "CoCreateInstance( )" API function can load the executable file either in the client program's process, or into a server process which can be either local or remote (i.e., on the same computer or a remote computer in a distributed computer network) depending on the attributes registered for the COM object **60** in the system registry.

[0060] Once the client of the COM object **60** has obtained this first interface pointer of the COM object, the client can obtain pointers of other desired interfaces of the component using the interface identifier associated with the desired interface. COM+ defines several standard interfaces generally supported by COM objects including the "IUnknown" interface. This interface includes a member function named "QueryInterface( )." The "QueryInterface" function can be called with an interface identifier as an argument, and returns a pointer to the interface associated with that interface identifier. The "IUnknown" interface of each COM object also includes member functions, "AddRef( )" and "Release( )", for maintaining a count of client programs holding a reference (e.g., an interface pointer) to the COM object. By convention, the "IUnknown" interface's member functions are included as part of each interface on a COM object. Thus, any interface pointer that the client obtains to an interface of the COM object **60** can be used to call the QueryInterface function.

ILLUSTRATED EMBODIMENTS

[0061] In the following illustrated embodiments, a framework for accommodating objects collects operational management information transparently to programs being monitored. The framework thus transparently provides various operational management metrics to facilitate software management. The arrangement frees software developers from including logic for generating operational management information, allowing software developers to focus on developing other aspects (e.g., the business logic) of their programs.

Overview of an Implementation of Enterprise Software Management for Object-based Software

[0062] In the illustrated implementations, an architecture for facilitating enterprise software management is used to collect operational management metrics by transparently monitoring interactions with software objects. Monitoring can be accomplished using an object operation monitor such as an interceptor or a wrapper.

[0063] **FIG. 3** depicts an exemplary monitoring arrangement. When a client program **202** (e.g., a calling object) performs an operation (e.g., a method invocation on an interface with particular parameters) on a monitored object **204**, the operation is intercepted by an interceptor **206**,

which sends operational management information to a software manager **208** and forwards the operation to the monitored object **204**. The interceptor **206** can also intercept information directed by the object **204** back to the client program **202** (e.g., a value returned from a method call) and generate appropriate operational management information (e.g., indicating the method call has returned and the value returned). A variation on this arrangement (not shown) divides the logic of the interceptor into a proxy and a stub component, both of which can generate operational management information; the proxy and stub can reside in two different processes or on two different computers.

[**0064**] **FIG. 4** shows an alternative depiction of a monitoring arrangement. The wrapper **224** wraps the monitored object **226**; external interactions with the monitored object **226** pass through the wrapper **224**, which sends operational management information to a software manager **228** based on interaction between the wrapped object **226** and the client **222**. The wrapper **224** can also direct operational management information to the software manager **228** when the monitored object **226** directs operations back to the client **222** (e.g., a return from a call on a method). Although the wrapper **224** is graphically depicted as wrapping the entire object **226**, it may instead wrap one or more interfaces of the object. Thus, two wrappers may wrap the same object **226**, each covering the same or different interfaces of the object **226**.

[**0065**] The operational management information generated by the interceptor **206** or the wrapper **224** generally indicates the particular interaction between a client and a monitored software object. For example, the information could indicate that a particular method of a particular interface was called. Optionally, the information may provide more detail, such as the parameters used during the method call.

[**0066**] Under either scenario, the monitored software object **204** or **226** need not contain logic for generating or sending operational management information. Thus, the developer of the monitored software object can write code for the monitored software object without knowledge of the enterprise software management arrangement or architecture.

[**0067**] Another way to describe the monitoring arrangement is by calling it "run time instrumentation of an object-based program." In other words, logic for generating and sending operational management information is inserted at program run time. Again, the developer avoids incorporating logic for generating or sending operational management information to a software manager because the illustrated architecture automatically does so at run time.

[**0068**] The monitoring arrangement could also be described as instrumentation of an object execution environment external of the objects. In other words, the execution environment is instrumented to generate notifications upon interaction with the objects. Again, the software developer avoids incorporating operational management logic in software objects because such logic resides outside the objects, in the execution environment.

[**0069**] The operational management information represents operational management metrics (measurements of the operation of the objects). These metrics are collected by a software manager, which can keep a log of activity on the monitored objects. As explained in a later section, two pieces of information are particularly helpful in the field of enterprise software management: the program originating the metrics the activity originating the metric. The collected (or "primary") operational management metrics can be transformed into other (or "derived") metrics (e.g., average response time for customer orders). A management console can present the metrics graphically to provide a comfort (or "heartbeat") screen (e.g., displaying number of transactions per minute) to provide visual confirmation of normal operation. In addition, various alerts can be configured to warn an administrator when a metric falls outside of a particular threshold. Finally, the log of events can prove useful for determining what caused a machine to crash (sometimes called "postmortem analysis"). If, for example, the last entry for a crashing server computer indicates a customer placed an order for 7.2 million widgets, it is likely that the program crashed because some portion of the software was unable to handle such a large quantity.

[**0070**] Overview of Software Management Architecture

[**0071**] An overview of an architecture used to collect operational management information is shown at **FIG. 5**. A monitored software object **312** runs in an execution environment **308** on a server computer **306**. When a client program at the client computer **302** wishes to access the functionality of the monitored software object **312**, it typically requests a reference (e.g., an interface pointer) to the monitored software object **312**. In the illustrated architecture, a reference to the proxy object **310** is instead provided to the client program at the client computer **302**.

[**0072**] Consequently, when the client program at the client computer **302** wishes to perform an operation on the monitored software object **312** (e.g., a method call), the client program at the client computer **302** does so via the reference to proxy **310** (e.g., using the normal method call semantics of COM described above). The proxy **310** then both performs the action on the monitored software object **312** and directs a notification to the collector **342** in the software manager **340**. As the software object **312** performs work, it may access the functionality of another software object **334** executing in another (or the same) execution environment **330** through a proxy **332**, which may also direct a notification to the collector **342**. Additionally, if the monitored software object **312** accesses a system service **322** running in a system process **320**, the system service **322** may direct additional notifications to the collector **342**. Finally, the monitored software object **312** may invoke the functionality of a monitored software object on a remote server **304**; the monitored software object on the remote server **304** directs a notification to a software manager running on the remote server **304**. The notification could be sent back to the server computer **306** and collected by collector **342**.

[**0073**] Notifications received by the collector **342** are typically recorded in a log **346**. The software manager **340** further comprises an alerter **344**, which monitors notifications and generates an alert (e.g., to a system administrator or to an automated management agent) when particular conditions are met. If, for example, the notifications include an activity identifier, an alert can be generated when notifications indicate certain conditions (e.g., no notifications or more than x notifications per minute) for a certain activity.

[0074] In addition, the alerter **344** of the software manager **340** can provide notifications to other software managers at positions higher in a hierarchy, such as the software manager **360**. These notifications can be either forwarded notifications (e.g., from the wrapper **310**) or generated anew based on specified conditions. Similarly, the server computer **304** can also direct notifications to the software manager **360**. In this way, a single software manager **360** can monitor the operation of a program, even if its components are software objects spread out over plural machines **306** and **304**. In the illustrated example, each software manager **304**, **340**, **360**, and **370** is on a different computer; however, plural software managers can reside on the same computer. Thus, one computer could serve as a software manager both for a particular computer and a set of computers including the particular computer. Finally, an alternative architecture might accommodate monitoring a computer not having a software manager. For example, notifications could be sent from a system process **320** to a software manager on a remote server **304**.

[0075] As the software manager **360** receives notifications, it can in turn provide notifications to other software managers still higher in the hierarchy, such as the software manager **370**. Thus, for example, a low-level software manager might provide a notification to an intermediate-level software manager when a program begins to exhibit poor performance and when the program fails. The intermediate-level software manager might only forward notifications to a high-level software manager when the program fails. Thus, the high-level software manager can monitor program availability across an enterprise without receiving all notifications.

[0076] Overview of Software Management Operation

[0077] An overview of a method used to collect operational management information for software management is shown at **FIGS. 6 and 7**. Generally, **FIG. 6** depicts a method for sending notifications to a software manager; **FIG. 7** depicts a method for handling the notifications.

[0078] With reference now to **FIG. 6**, a request is received by an object request service for a software object reference (box **402**). Instead of providing a reference to a software object, the object request service provides a reference to a proxy (box **404**). A request for an operation to be performed on the object is intercepted at the proxy (box **406**). The proxy then issues a notification indicative of the operation (box **408**) and forwards the request for the operation to the software object (box **410**). In such an arrangement, the proxy serves as an interceptor (such as the interceptor **206** in **FIG. 3**) or wrapper (such as the wrapper **224** in **FIG. 4**) or both; however, some other interceptor or wrapper could be used as an alternative to the proxy. Similarly, when the software object performs an operation (e.g., a method return), the proxy intercepts the operation, issues a notification, and forwards the operation. Thus, the proxy can monitor various interactions with the object, including both operations performed on and by the software object.

[0079] With reference now to **FIG. 7**, the notification sent in box **408** (**FIG. 6**) is received at box **420**. The notification is then logged (box **422**) and combined with other notifications to generate program-level operational management information (box **424**). The program-level operational management information is presented, monitored, or both (box **426**); if a particular metric falls outside a threshold (box **428**), an alert is provided (box **430**). The alert may be in the form of an action taken to notify a system administrator (e.g., an email) or an event sent to a software manager at a higher position in a software manager hierarchy.

[0080] Generating Notifications in the Software Management Architecture

[0081] On a general level, a program is monitored by observing the interactions with monitored software objects comprising the program. When an interaction with a monitored software object takes place, a notification is directed to a software manager. Further, when objects request certain system functions, a notification is directed to the software manager. In the illustrated exemplary embodiments, these notifications are generically called operational management metrics because they provide measurements useful for managing the operation of monitored software objects. The software manager in turn transforms (or "munges") the notifications into program-level operational management metrics by, for instance, resolving timestamps and references between the metrics. The operational management metrics can be monitored by an administrator or an automated monitor.

[0082] With reference now to the overview of a software management architecture shown at **FIG. 5**, a particular implementation of the architecture places a monitored software object (e.g., object **312**) in an object context. Object contexts are an environment extensibility mechanism described at length in Thatte et al., "Environment Extensibility and Automatic Services For Component Applications Using Contexts, Policies And Activators," U.S. patent application Ser. No. 09/135,397, filed Aug. 17, 1998, the disclosure of which is incorporated herein by reference. In such an embodiment, the wrapper **310** is a proxy object, which generates a notification and forwards the method invocation to the monitored software object. Thus, the object contexts are used to automatically generate notifications transparently to the monitored object when a client program from another context (e.g., on another computer or in another process) directs a method invocation to the monitored object.

[0083] The proxy is automatically put into place by an object request service for cross-context calls if the software object is designated as a monitored software object. The object can be so designated by configuring a catalog of object information residing on the computer on which the object executes. The catalog can be configured to monitor particular objects or all software objects for a particular application or other program (i.e., a monitored program). In one implementation, if the catalog indicates the object is monitored, a policy is included in the policy set of the object context; the policy generates notifications. In another implementation, the proxy contains code to determine whether the object has been designated as monitored, and the proxy generates notifications.

[0084] A variety of arrangements can be used to direct operational management metrics to a software manager. In one implementation, notifications are events fired by an event publisher in a loosely coupled publish and subscribe arrangement. Loosely coupled publish and subscribe events are described in Hinson et al., "Object Connectivity Through Loosely Coupled Publish and Subscribe Events," U.S.

patent application Ser. No. 09/247,363, filed Feb. 23, 1999, the disclosure of which is incorporated herein by reference. Such an arrangement is shown generally in **FIG. 8**. The wrappers **452** and **454** become publishers via the interface IEventControl **464**, then fire events to the event object **462** through the interface IOutEvents **460**. The event object **464** distributes the events to various subscribers, such as a software manager **480**. The subscribers **470** and **482** subscribe to a set of events by adding a subscription to a subscription set; the event object **462** then sends the events to the interface IOutEvents **460** of the objects **470** and **482**. Since the identity of the subscriber is stored in the subscription, the publisher need have no a priori knowledge of a subscriber's identity; also the lifetime of the publisher need not overlap that of the subscriber.

[**0085**] Such an arrangement has several advantages in enterprise software management. One advantage is that a custom software manager (e.g., a third party plug in or a user program) can monitor the notifications in addition to or instead of the provided software manager **480**. Other software managers are easily incorporated into the architecture because they need only register a subscription to appropriate events in order to receive notifications for the monitored program's objects. Thus, a program can monitor its own performance using the software management architecture and avoid incorporating logic for generating notifications into the program. As a result, the software management architecture provides program monitoring with logic residing outside and transparent to the program.

[**0086**] Alternatively, an intermediate collector (e.g., in a system process) can be provided to collect notifications before publishing them. For example, as shown in **FIG. 9**, a publisher **512** in a system process **510** collects notifications from the wrapper **502** in the process **504** and the wrapper **506** in the process **508**, and publishes the notifications as events. The publisher **512** accesses the interfaces **522** and **520** of the event object **530**, which sends the events to a subscriber object **540** through the interface **520**. The arrangement may or may not expose the intermediate interfaces (e.g., the interface to the publisher **512**). Such an arrangement avoids creating numerous publishers; there is but one publisher per computer (or, alternatively, per program). However, the arrangement still benefits from the advantages of a loosely coupled publish and subscribe event model.

[**0087**] Yet another alternative is to send a notification directly to a software manager without employing the loosely coupled arrangement. Such an arrangement is shown in **FIG. 10**, in which wrappers **550** and **554** send notifications directly to a software manager **556** in a tightly coupled arrangement. Thus, the term "notification" includes direct notification, an event published according to a loosely coupled publish and subscribe arrangement, or a notification to a system process, which publishes an event in response to the notification.

[**0088**] Another useful feature of the illustrated arrangements involving subscriptions is that semantically-related operational management metrics can be placed into various groups, for which individual subscriptions can be registered. For example, a subscription to metrics related to transactions (e.g., transaction started, transaction completed, and transaction aborted) can be registered separately from metrics related to methods (e.g., method call, method return, and method exception). This feature accommodates dynamic operational management metric selection. In other words, the metrics selected for monitoring by a software manager can be changed at run time. For example, if transaction information indicates that too many transactions are aborting, method-level monitoring can be started to study object behavior and determine why so many transactions are aborting. Thus, the architecture avoids sending excessive notifications to the software manager when the program is functioning normally.

[**0089**] Notification Contents

[**0090**] The notifications provided to a software manager contain information useful for monitoring program performance. For example, a timestamp in the notification allows tracking of various time-based metrics. Also, including information identifying the program responsible for generating the notification (e.g., which program caused a method call) enables the information to be grouped by program.

[**0091**] Another useful way of grouping the information is by activity. An activity is a unit of work composed of the tasks to be completed to form one user level interaction. For example, if a user clicks an "order" button, the activity of processing the order begins; when the order processing is completed and the user has been notified of the result, the activity ends. Each activity is identified by an activity identifier (a GUID). Notifications generated while performing the work to complete the activity (e.g. calling a method, instantiating a software object, allocating threads and memory, accessing a database, and completing a transaction) are associated with the activity via the activity identifier. The software manager can then generate information based on notifications collected for the activity. The activity information can help determine which activities are most popular and what resources are consumed by what activities. Such information is particularly useful for tuning program performance. Finally, since the activity information correlates with a user interaction, tracking activity performance roughly corresponds to the user's perception of program performance.

[**0092**] The architecture also supports user-defined notifications to accommodate functionality not covered by the other events. Thus it would be possible, for example, to generate a plug in software manager which monitors a program in a very specialized way.

[**0093**] Finally, the notifications can contain information to facilitate combining them. For example, a "transaction started" metric might contain a key referenced by a "transaction completed" metric. Thus, the notifications can be recognized as related.

[**0094**] Handling Notifications in the Software Management Architecture

[**0095**] As notifications from various sources arrive at a software manager, they are logged as primary operational management metrics and transformed into derived operational management metrics. For example, two notifications indicating when a transaction started and completed can be transformed into a metric indicating average transaction completion time and transactions completed per minute. Program-level operational management metrics indicate

performance of a particular program, and may be derived from primary operational management metrics originating from plural computers.

[0096] The log of notifications is useful in assisting in a variety of software management tasks. Since notifications are associated with an activity and a program, it is possible to examine log entries to determine the source of various problems. For example, the last entry for a particular program can be examined in a post mortem analysis to determine why the program crashed. The log can also be used as a source of information to generate custom reports.

[0097] The software manager can additionally present operational management information graphically. For example, a user interface **602** commonly called a "comfort screen" (because it assures an administrator that operations are proceeding normally) is shown in **FIG. 11**. This interface provides an indication of the operation (or "heartbeat") of the programs being monitored. For each monitored program, a selectable button **604**, a thermometer **606**, and a ceiling value **608** are displayed. The thermometer **606** shows the percentage of the ceiling value **608** exhibited by the program-level operational management metric. The program-level operational management metric can be one generated by monitoring a program scattered across plural computers (e.g., plural instances of order processing software objects for an order processing application or instances of various software objects for a banking application). By selecting the selectable button **604**, a system administrator can navigate to an analysis screen for the associated program. In this way, the software manager provides a running indication of program availability and provides an easy way to navigate to a more detailed view showing an analysis screen associated with the program. Thus, the user interface **602** is a useful interface for presentation to a system administrator during day-to-day operations.

[0098] Finally, the software manager can be configured to generate a variety of alerts when program-level operational management metrics go outside specified thresholds or if a particular event is received. Alerts can take various forms, such as changing a screen condition (e.g., highlighting an icon representing a program or server), sending an email, or paging an administrator. Alerts can also be used to communicate from one software manager to another, as described in more detail below.

[0099] Software Manager Hierarchy

[0100] Software Managers can be connected together in a hierarchical fashion to facilitate enterprise software management. With reference now to **FIG. 12**, an exemplary basic building block in an enterprise software management system is shown as a managed unit **650**. The managed unit **650** comprises the managed component set **658** and a software manager **656** responsible for managing the managed component set **658**. The managed unit **650** further comprises interfaces **654**, **660**, and **664** to the software manager **656**.

[0101] A configuration interface **654** facilitates configuration of various software manager **656** features (e.g., alerts). The software manager interface **660** serves as a sender and a recipient of events to and from other software managers, as shown in more detail below. The administrative interface **664** allows a user (e.g., a system administrator) to examine the log **670** and other information collected and generated for the managed component set **658**.

[0102] As shown in **FIG. 13**, software managers can be arranged in a hierarchical fashion to provide an enterprise software management system **702**. The managed units **706**, **708**, and **710** are monitored by a software manager in the managed unit **704**. From one of the lower level managed units (**706**, **708**, or **710**), alerts can be sent through the software manager interface **660** (**FIG. 12**) of a higher level managed unit **704**. For example, if too many transactions for a particular program are aborting, the managed unit **706** may alert the managed unit **704** so a system administrator monitoring several programs at various locations will be provided an alarm. The alerts can also be used for communication between the managed units. For example, a managed unit **708** may provide the higher-level managed unit **704** with an update on the number of transactions completed per minute for a particular program. The frequency of this communication can be varied by the system administrator.

[0103] Software Manager Implementation

[0104] **FIG. 14** shows an exemplary implementation of a software manager **810**. The metric data collector **812** is a subscriber to events from software managers at lower levels (e.g., the software manager **830**) and other event sources **832** (e.g., events generated by a transaction server, the system, object wrappers, and programs local to the software manager **810**).

[0105] The transformer (or "munger") **814** transforms the metrics into program-level operational management metrics for analysis by the analyzer/thresholder **818**. For example, timestamp information for plural metrics is converted into elapsed time to determine an activity completion time, and various metrics are grouped by program. Context data **834** derived from a monitored object's object context object can be utilized during the transformation process.

[0106] The analyzer/thresholder **818** further filters the metrics to determine whether they should be fed to the alerter **816**, which can publish events to administrative clients **806** or subscribing software managers at higher levels (e.g., the software manager **808**). End user clients **804** can access the software manager **810** for examination and configuration. For example, an Internet connection could be used to remotely access and configure the software manager **810**.

[0107] Other Features

[0108] The architecture can accommodate a wide variety of features not described above. For example, the architecture can monitor program security. If a notification is provided to the software manager when user authentication fails, an alarm can alert a remote administrator, who can take steps to stop a potential intruder or help a user who has forgotten a password.

[0109] Additionally, although various examples make reference to managing an application or other program, the architecture can monitor any software comprising a designated set of software objects. Thus, for example, instead of monitoring software objects for a particular application, the architecture can monitor software objects originating from a particular author or vendor.

[0110] Operational Management Metrics

[0111] Operational management metrics measure a managed operation. Each operation performed to do work for a managed program is a potential operational management metric. Thus, operational management metrics might measure a wide variety of actions related to program initialization, transactions, objects, object methods, object pooling, contexts, resource allocation, and security. For example, when an object's method is called to do work for a program, one potential operational management metric indicates the particular method called and a timestamp indicating when the method was called. The metric can also include information identifying the responsible program and activity to facilitate grouping and tracking a set of related metrics.

[0112] Operational management metrics representing direct measurements of interactions (e.g., a method call) with a software object are called "primary" operational management metrics. These primary operational management metrics can be combined using various mathematical formulas to derive other "derived" operational management metrics, which can in turn be combined to derive still others. Derivation is generally accomplished by combining plural operational management metrics into a set and performing a calculation on the set. The calculation may comprise a set of mathematical operations (e.g., averaging, adding, subtracting, and counting). For example, a first metric indicating a timestamp of a method call and a second metric indicating a timestamp of the method return could be combined (by subtracting timestamps) into a third metric indicating the time required to complete the method call. Further, a set of such metrics could be combined and averaged to indicate an average time required to complete a method call. Or, the number of transactions completed per minute can be derived by counting the number of "transaction completed" metrics showing appropriate timestamps.

[0113] Program-level operational management metrics are generated by combining a set of metrics for a single program into a set and performing a calculation on the set. For example, metrics showing that transactions have been completed for a particular program could be grouped together to determine how many transactions per minute have been completed for the program, even if the program is scattered across plural computers.

[0114] The potential number of operational metrics is limitless, and the illustrated architecture accommodates user-defined events to facilitate user-created metrics not derivable using the other metrics provided by the architecture. The metrics, including user defined ones, can be monitored graphically or used to trigger alerts.

Exemplary Setup of the Software Management Architecture

[0115] In the illustrated exemplary setup, operational management metrics monitor a wide variety of object-related operations associated with software objects to determine application performance. The object-related operations include object creation, object method calling, method exception generation, object activation, and queuing a method call.

[0116] Sample Hierarchical Arrangement

[0117] The scalability of the architecture is demonstrated by the arrangement shown in **FIG. 15**. An enterprise application management system **850** includes software managers at three levels: the corporate level (the application manager **852**), the call center level (the application managers **854**, **856**, and **858**), and the machine level (e.g., the application managers at each of the computers in the groups **862** and **892** and the computers **872**, **874**, and **876**).

[0118] The management system **850** monitors the operations of the component sets **864**, **878**, and **894**, which include payroll application components. In the illustrated arrangement, the application manager **854** administers the applications running on the computers in group **862** (the component set **864**) and the payroll components **880** throughout the system **850** (i.e., at each of the component sets **864**, **878**, and **894**). Events pertaining to the payroll components **880** received by the application managers **856** and **858** are sent to the application manager **854**.

[0119] The application managers **856** and **858** administer their respective local applications (i.e., applications running at the computers **872**, **874** and **876** for the application manager **856**, and applications running at the computer group **892** for the application manager **858**). Filtered management information is sent to the corporate application manager **852**, which administers applications throughout the system **850**.

[0120] Sample Log

[0121] Table 1 shows a sample log of entries from an application manager. A variety of filters can be applied to the log (e.g., show only those entries for a particular application), and the log can be used to generate custom reports.

TABLE 1

Log Entries
Application WebOrders Activated
Process: ps_x; Time t_x; Application: a_1; Machine: m_x
Activity CustomerOrder Started
Process: ps_x; Time t_x; Application: a_1; Machine: m_x
Transaction tx Started
Process: ps_x; Time t_x; Application: a_1; Machine: m_x
Object of object class CLSID created in Context ctxt for Activity CustomerOrder in Transaction tx (ID: ObjectID)
Process: ps_x; Time t_x; Application: a_1; Machine: m_x
Application TechSupport Activated
Process: ps_x; Time t_x; Application: a_2; Machine m_x
Activity ProblemReport Started
Process: ps_x; Time t_x; Application: a_2; Machine m_x
Method IID of object class CLSID called (ID: ObjectID)
Process: ps_x; Time t_x; Application: a_1; Machine: m_x
Resource "Database Connection" created and allocated in a transaction (ID: ObjectID)
Process: ps_x; Time t_x; Application: a_1; Machine: m_x
Method IID of object class CLSID called (ID: ObjectID)
Process: ps_x; Time t_x; Application: a_2; Machine: m_x
Method IID of object class CLSID called (ID: ObjectID)
Process: ps_x; Time t_x; Application: a_1; Machine: m_x
Method IID of object class CLSID returned (ID: ObjectID)
Process: ps_x; Time t_x; Application: a_1; Machine: m_x
Method IID of object class CLSID returned (ID: ObjectID)
Process: ps_x; Time t_x; Application: a_2; Machine: m_x
...
Transaction tx committed
Process: ps_x; Time t; Application: a_1; Machine: m_x
Activity CustomerOrder Finished
Process: ps_x; Time t; Application: a_1; Machine: m_x
...
Application WebOrders shut down
Process: ps_x; Time t; Application: a_1; Machine: m_x

[0122] Sample Events and Event Formats

[0123] An exemplary set of events is described in this section. Each of the events represents an operational management metric.

[0124] Each metric provides a standard structure COMSVCSEVENTINFO as its first value. The structure is shown in Table 2. The structure contains contextual information about the metric, such as the time it was generated, from which process and the software application responsible for its generation. Optionally, the COMSVCSEVENTINFO structure may contain a version of the application, facilitating, for example, separately tracking test and production versions of the same application. A field other than or in addition to the application responsible for generating the metric could be included in the structure. The architecture thus accommodates monitoring any set of software objects placed into a logical group.

[0125] A metric key field can be referenced by other metrics for correlation. Specifically, value stored in the key field of a first metric is stored in the reference field of other related metrics to relate the first metric with the others.

TABLE 2

COMSVCSEVENTINFO structure

DWORD dwPid;	process id from which the event originated
LONGLONG iTime;	Coordinated Universal Time of event as seconds elapsed since midnight (00:00:00), Jan. 1, 1970
LONG lMicroTime;	microseconds added to tTime for time to microsecond resolution
LONGLONG perfCount;	
GUID guidApp;	the application GUID for the first component instantiated in dwPid
LPOLESTR sMachineName;	fully qualified name of the machine where the event originated

[0126] The individual metrics are divided into groups as shown in the tables below. Each group can be individually subscribed. Instead of grouping by type of operation performed, the metrics could be grouped, for example, by level of detail revealed by the metric. Thus, metrics providing a general measurement of application performance could be placed in a group different from those providing detailed measurements. Such an arrangement would permit an application manager to automatically subscribe to the more detailed metrics upon detection of a problem as revealed by the general metrics. The system illustrated below supports such a scheme in that, for instance, method metrics can be individually subscribed after detecting a problem with transaction metrics.

[0127] Metrics in relation to operations performed at objects are generated at various times of the objects' lifetimes. For example, when an object is created, a metric is generated having a key field, a data field denoting the time the object creation was observed, and a data field identifying the application for which the object is performing work. Subsequently, when a method call is performed on the object, the system generates a second metric having a reference to the key and a data field denoting the time the method call was observed. The metrics can thus be correlated using the key and reference fields.

TABLE 3

Application Metrics

OnAppActivation	Generated when an application server is loaded
COMSVCSEVENTINFO * pInfo	
GUID guidApp	GUID for the Application
OnAppForceShutdown	Generated when an application server is shut down through the operator's console
COMSVCSEVENTINFO * pInfo	
GUID guidApp	
OnAppShutdown	Generated when an application server shuts down
COMSVCSEVENTINFO * pInfo	
GUID guidApp	

[0128]

TABLE 4

Activity Metrics (Activities are logical synchronization units)

OnActivityCreate	Generated when an activity starts
COMSVCSEVENTINFO * pInfo	
REFGUID guidActivity	KEY - identifies the activity
OnActivityDestroy	Generated when an activity is finished
COMSVCSEVENTINFO * pInfo	
REFGUID guidActivity	REFERENCES OnActivityCreate
OnActivityTimeout	Generated when a call into an Activity times out
COMSVCSEVENTINFO * pInfo	
REFGUID guidCurrent	REFERENCES OnActivityCreate::GuidApp for caller
REFGUID guidEntered	REFERENCES OnActivityCreate::GuidApp for the activity being entered (attempted entry)
DWORD dwThread	WINDOWS 2000 thread ID executing the call
DWORD dwTimeout	Timeout period

[0129]

TABLE 5

Transaction Metrics

OnTransactionStart	Generated when a DTC (Distributed Transaction Coordinator) transaction starts
COMSVCSEVENTINFO * pInfo	
REFGUID guidTx	KEY - unique identifier for the transaction
REFGUID tsid	KEY - unique identifier for correlation to objects
BOOL fRoot	TRUE if this is a root transaction
OnTransactionPrepare	Generated on Prepare phase of a DTC transaction
COMSVCSEVENTINFO * pInfo	
REFGUID guidTx	REFERENCES OnTransactionStart
BOOL fVoteYes	How the Resource Manager generating the prepare voted
OnTransactionAbort	Generated when a transaction aborts
COMSVCSEVENTINFO * pInfo	
REFGUID guidTx	REFERENCES OnTransactionStart
OnTransactionCommit	Generated when a transaction commits
COMSVCSEVENTINFO * pInfo	
REFGUID guidTx	REFERENCES OnTransactionStart

[0130]

TABLE 6

Object Metrics

OnObjectCreate	Generated when an object is created by a client
COMSVCSEVENTINFO * pInfo	
REFGUID guidActivity	REFERENCES - OnActivityCreate
REFCLSID clsid	CLSID for the object being created
REFGUID tsid	REFERENCES OnTransactionStart
ULONG64 CtxtID	KEY - Context for this object
ULONG64 ObjectID	KEY - Initial JIT activated object
OnObjectDestroy	Generated when an object is released by a client
COMSVCSEVENTINFO * pInfo	
ULONG64 CtxtID	REFERENCES - OnObjectCreate
OnObjectActivate	Generated when an object gets a new JITed object instance
COMSVCSEVENTINFO * pInfo	
ULONG64 CtxtID	REFERENCES OnObjectCreate
ULONG64 ObjectID	KEY - JIT activated object
OnObjectDeactivate	Generated when the JITed object is freed (by SetComplete or SetAbort)
COMSVCSEVENTINFO * pInfo	
ULONG64 CtxtID	REFERENCES OnObjectCreate
ULONG64 ObjectID	REFERENCES OnObjectActivate

[0131]

TABLE 7

Context Metrics

OnDisableCommit	Generated when the client calls DisableCommit on a context
COMSVCSEVENTINFO * pInfo	
ULONG64 CtxtID	REFERENCES OnObjectCreate
OnEnableCommit	Generated when the client calls EnableCommit on a context
COMSVCSEVENTINFO * pInfo	
ULONG64 CtxtID	REFERENCES OnObjectCreate
OnSetComplete	Generated when the client calls SetComplete on a context
COMSVCSEVENTINFO * pInfo	
ULONG64 CtxtID	REFERENCES OnObjectCreate
OnSetAbort	Generated when the client calls SetAbort on a context
COMSVCSEVENTINFO * pInfo	
ULONG64 CtxtID	REFERENCES OnObjectCreate

[0132]

TABLE 8

Method Metrics

OnMethodCall	Generated when an object's method is called
COMSVCSEVENTINFO * pInfo	
ULONG64 oid	REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
REFCLSID guidCid	CLSID for the object being called
REFIID guidRid	IID of the method being called
ULONG iMeth	v-table index of said method
OnMethodReturn	Generated when an object's method returns
COMSVCSEVENTINFO * pInfo	
ULONG64 oid	REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
REFCLSID guidCid	CLSID for the object being called

TABLE 8-continued

Method Metrics

REFIID guidRid	IID of the method returning
ULONG iMeth	v-table index of said method
OnMethodException	Generated when an object's method generates an exception
COMSVCSEVENTINFO * pInfo	
ULONG64 oid	REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
REFCLSID guidCid	CLSID for the object being called
REFIID guidRid	IID of the method generating the exception
ULONG iMeth	v-table index of said method

[0133]

TABLE 9

Resource Dispenser Management Metrics

OnResourceCreate	Generated when a NEW resource is created and allocated
COMSVCSEVENTINFO * pInfo	
ULONG64 ObjectID	REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
LPCOLESTR pszType	String describing resource being created
ULONG64 resId	KEY - unique identifier for resource
BOOL enlisted	TRUE if enlisted in a transaction
OnResourceAllocate	Generated when an existing resource is allocated
COMSVCSEVENTINFO * pInfo	
ULONG64 ObjectID	REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
LPCOLESTR pszType	String describing resource
ULONG64 resId	REFERENCES OnResourceCreate
BOOL enlisted	TRUE if enlisted in a transaction
DWORD NumRated	Number of possible resources evaluated for match
DWORD Rating	The rating of the resource actually selected
OnResourceRecycle	Generated when an object is finished with a resource
COMSVCSEVENTINFO * pInfo	
ULONG64 ObjectID	REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
LPCOLESTR pszType	String describing resource
ULONG64 resId	REFERENCES OnResourceCreate
OnResourceDestroy	Generated when a resource is permanently removed from the resource pool
COMSVCSEVENTINFO * pInfo	
ULONG64 ObjectID	REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
HRESULT hr	Result from Resource Dispenser's Destroy
LPCOLESTR pszType	String describing resource
ULONG64 resId	REFERENCES OnResourceCreate

[0134]

TABLE 10

Security and Authentication Metrics

OnAuthenticate Generated when a method call level authentication succeeds
COMSVCSEVENTINFO * pInfo
REFGUID guidActivity REFERENCES - OnActivityCreate
ULONG64 ObjectID REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
REFGUID guidIID IID of the method
ULONG iMeth v-table index of said method
BYTE * pSidOriginalUser SID of Original Caller
BYTE * pSidCurrentUser SID of Current Caller
BOOL bCurrentUserInpersonatingInProc TRUE if Current User is impersonating
OnAuthenticateFail Generated when a method call level authentication fails
COMSVCSEVENTINFO * pInfo
REFGUID guidActivity REFERENCES - OnActivityCreate
ULONG64 ObjectID REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
REFGUID guidIID IID of the method
ULONG iMeth v-table index of said method
BYTE * pSidOriginalUser SID of Original Caller
BYTE * pSidCurrentUser SID of Current Caller
BOOL bCurrentUserInpersonatingInProc TRUE if Current User is impersonating
OnIISRequestInfo Generated when an activity is part of an IIS ASP
COMSVCSEVENTINFO * pInfo
ULONG64 ObjId REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject
LPCOLESTR pszClientIP IP Address of IIS client
LPCOLESTR pszServerIP IP Address of IIS server
LPCOLESTR pszURL URL on IIS server generating object reference

[0135]

TABLE 11

COM+ Object Pooling Metrics

OnObjPoolPutObject Generated when a non-transactional object is returned to the pool
COMSVCSEVENTINFO * pInfo
REFGUID guidObjectCLSID for the objects in the pool
int nReason Reserved - always 0
DWORD dwAvailable Number of objects in the pool
ULONG64 oid REFERENCES OnObjPoolGetObject
OnObjPoolGetObject Generated when a non-transactional object is obtained from the pool
COMSVCSEVENTINFO * pInfo
REFGUID guidActivity REFERENCES - OnActivityCreate
REFGUID guidObjectCLSID for the objects in the pool
DWORD dwAvailable Number of objects in the pool
ULONG64 oid KEY - the unique identifier for this object
OnObjPoolRecycleToTx Generated when a transactional object is returned to the pool
COMSVCSEVENTINFO * pInfo
REFGUID guidActivity REFERENCES - OnActivityCreate
REFGUID guidObjectCLSID for the objects in the pool
REFGUID guidTx REFERENCES OnTransactionStart

TABLE 11-continued

COM+ Object Pooling Metrics

ULONG64 objid REFERENCES OnObjectCreate::ObjectID or OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid
OnObjPoolGetFromTx Generated when a transactional object is obtained from the pool
COMSVCSEVENTINFO * pInfo
REFGUID guidActivity REFERENCES - OnActivityCreate
REFGUID guidObjectCLSID for the objects in the pool
REFGUID guidTx REFERENCES OnTransactionStart
ULONG64 objid KEY - the unique identifier for this object
OnObjPoolCreateObject Generated when an object is created for the pool
COMSVCSEVENTINFO * pInfo
REFGUID guidObject CLSID for the objects in the pool
DWORD dwObjsCreated Number of objects in the pool
ULONG64 oid KEY - unique pooled object ID
OnObjPoolDestroyObject Generated when an object is permanently removed from the pool
COMSVCSEVENTINFO * pInfo
REFGUID guidObject CLSID for the objects in the pool
DWORD dwAvailable Number of objects in the Pool
ULONG64 oid REFERENCES OnObjPoolCreateObject
OnObjPoolCreateDecision Generated when the pool must decide to give out an existing object or create a new one
COMSVCSEVENTINFO * pInfo
DWORD dwThreadsWaiting Number of threads waiting for an object
DWORD dwAvail Number of free objects in the pool
DWORD dwCreated Number of total objects in the pool
DWORD dwMin Pool's Min object value
DWORD dwMax Pool's Max object value
OnObjPoolTimeout Generated when the request for a pool object times out
COMSVCSEVENTINFO * pInfo
REFGUID guidObject CLSID for the objects in the pool
REFGUID guidActivity REFERENCES - OnActivityCreate
DWORD dwTimeout Pool's timeout value
OnObjPoolCreatePool Generated when a new pool is created
COMSVCSEVENTINFO * pInfo
REFGUID guidObject CLSID for the objects in the pool
DWORD dwMin Pool's Min object value
DWORD dwMax Pool's Max object value
DWORD dwTimeout Pool's timeout value
OnObjectConstruct Generated for when a Constructed object is created
COMSVCSEVENTINFO * pInfo
REFGUID guidObject CLSID for the objects in the pool
LPCOLESTR sConstructString Object construction string
ULONG64 objid KEY - unique constructed Object ID

[0136]

TABLE 12

Queued Components Metrics	
OnQCRecord	Generated when the QC recorder creates the queued message
COMSVCSEVENTINFO * pInfo	
ULONG64 objid	REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject - object whose method calls are being queued
WCHAR szQueue	MSMQ Queue name
REFGUID guidMsgId	KEY - Unique message ID for this queued message
REFGUID guidWorkFlowId	Reserved
HRESULT msmqhr	MSMQ return status for queue message
OnQCQueueOpen	Generated when the queue for a QC queue is opened (used to generated the QueueID)
COMSVCSEVENTINFO * pInfo	
WCHAR szQueue	MSMQ Queue name
ULONG64 QueueID	KEY - unique identifier for queue
HRESULT hr	Status from MSMQ queue open
OnQCReceive	Generated for a successful de-queuing of a message (although QC might find something wrong with the contents)
COMSVCSEVENTINFO * pInfo	
ULONG64 QueueID	REFERENCE OnQCQueueOpen
REFGUID guidMsgId	REFERENCE - OnQCRecord
REFGUID guidWorkFlowId	Reserved
HRESULT hr	Status from QC processing of received message
OnQCReceiveFail	Generated when the receive message fails
COMSVCSEVENTINFO * pInfo	
ULONG64 QueueID	REFERENCE OnQCQueueOpen
HRESULT msmqhr	Status from MSMQ receive message
OnQCMoveToReTryQueue	Generated when a message is moved to a QC retry queue
COMSVCSEVENTINFO * pInfo	
REFGUID guidMsgId	REFERENCE - OnQCRecord
REFGUID guidWorkFlowId	Reserved
ULONG RetryIndex	Which retry queue to move to
OnQCMoveToDeadQueue	Generated when a message cannot be delivered
COMSVCSEVENTINFO * pInfo	
REFGUID guidMsgId	REFERENCE - OnQCRecord
REFGUID guidWorkFlowId	Reserved
OnQCPlayback	Generated when a message's contents are replayed
COMSVCSEVENTINFO * pInfo	
ULONG64 objid	REFERENCES OnObjectCreate::ObjectID, OnObjectActivate::ObjectID, OnObjPoolGetFromTx::objid, OnObjectConstruct::objid or OnObjPoolGetObject - object playing back the QC messages
REFGUID guidMsgId	REFERENCE - OnQCRecord
REFGUID guidWorkFlowId	Reserved
HRESULT hr	Status from MSMQ receive message

[0137]

TABLE 13

Component Exception Metrics	
OnExceptionUser	Generated for transactional components when a user exception is encountered
COMSVCSEVENTINFO * pInfo	
ULONG code	Exception code
ULONG64 address	Address of Exception
LPCOLESTR pszStackTrace	Stack trace

[0138]

TABLE 14

User Defined Event Metrics	
OnUserEvent	Provided for User components to generate user specific metrics
COMSVCSEVENTINFO * pInfo	
VARIANT * pvarEvent	User defined content

[0139]

TABLE 15

STA Thread Pool Metrics

OnThreadStart	Generated when a new STA (Single Threaded Apartment) thread is created
COMSVCSEVENTINFO * pInfo	
ULONG64 ThreadID	KEY - unique thread identifier
DWORD dwThread	WINDOWS 2000 thread ID
DWORD dwTheadCnt	Number of threads in STA thread pool
OnThreadTerminate	Generated when an STA thread is terminated
COMSVCSEVENTINFO * pInfo	
ULONG64 ThreadID	REFERENCES OnThreadStart
DWORD dwThread	WINDOWS 2000 thread ID
DWORD dwTheadCnt	Number of threads in the STA thread pool
OnThreadBindToApartment	Generated when an STA thread needs an apartment (thread) to run in. Either allocates one from the pool or creates one
COMSVCSEVENTINFO * pInfo	
ULONG64 ThreadID	REFERENCES OnThreadStart
ULONG64 AptID	Apartment ID
DWORD dwActCnt	Number of activities bound to this apartment
DWORD dwLowCnt	Reserved - currently 0
OnThreadUnBind	Generated when the apartment (thread) is no longer needed
COMSVCSEVENTINFO * pInfo	
ULONG64 ThreadID	REFERENCES OnThreadStart
ULONG64 AptID	REFERENCES OnThreadAssignApartment
DWORD dwActCnt	Number of activities active on the Apartment (thread)
OnThreadAssignApartment	Generated when an activity is assigned to an apartment (thread)
COMSVCSEVENTINFO * pInfo	
REFGUID guidActivity	REFERENCES OnActivityCreate
ULONG64 AptID	KEY - unique apartment ID
OnThreadUnassignApartment	Generated when the activity is no longer associated with that apartment (thread)
COMSVCSEVENTINFO * pInfo	
ULONG64 AptID	REFERENCES OnThreadAssignApartment

[0140] Alerts

[0141] The illustrated exemplary architecture accommodates a set of rules considered when issuing alerts. The alerts can take the form of scripts. For example, a payroll administrator observes that 20 percent of transactions are aborting. After further investigation, she learns that the transactions are aborting due to an expired password. A similar situation occurred six months ago when passwords expired at the end of a six-month password rotation cycle.

[0142] Consequently, the administrator configures a rule to run a script whenever payroll transactions abort over the 15 percent level. The script acquires information on authentication failures and sends an advisory email to the password administrator if authentication failures are the cause of the failed transactions.

[0143] The scripts can be further used to start an application at a computer. For example, an organization has ten servers; five servers normally run a web application to process customer credit card orders from a web page, and five normally run an in-house application to process telephone orders as entered by staff. After the telephone order center closes, the application manager detects that the load on the in-house application has dropped dramatically; meanwhile, orders from the web page are beginning to overwhelm the five servers running the web application. The application manager can execute a script to start the web application on one or more of the five servers normally running the in-house application and shut down the in-house application on those servers if desired.

[0144] Generating Notifications in the Exemplary Architecture

[0145] **FIG. 16** shows a method for generating notifications in the exemplary architecture. A client computer **302** (**FIG. 5**) requests a reference to a monitored application object from a system service at the client computer (box **902**). Responsive to the request, the system service at the client computer delegates the request to a remote system service (box **904**). The remote system service could be a service at a server computer (e.g., **306**) or at a load balancing computer (e.g., a router), which forwards the request to an appropriate server. Responsive to the delegation, the system service at the server **306** instantiates a proxy object **310**, provides a proxy interface pointer to the client computer **302**, instantiates the monitored application object **312**, and provides the proxy **310** with a reference to the monitored application object **312** and the publisher (e.g., publisher **510** in **FIG. 9**) in the system process **320** (box **906**).

[0146] The client computer **302** then issues a method call on the interface pointer provided to it by the object creation service (box **908**). The proxy object **310** intercepts the method call, generates a notification of the method call to the monitoring system process **320**, and forwards the method call to the monitored application object **312** (box **910**).

[0147] The monitored application object **312** returns the result of the method call to the proxy **310**, which generates a notification indicating the method call has returned and relays the result to the client computer **302** (box **912**).

[0148] Fielding Notifications in the Exemplary Architecture

[0149] **FIG. 17** illustrates a method for acquiring and fielding the notifications generated by the method of **FIG. 16**. An application manager **340** (**FIG. 5**) subscribes to application and transaction metrics (box **920**). As the system process **320** publishes events, they are collected (box **922**). The application manager **340** maintains values for transactions per minute and percentage of aborting transactions (box **924**). The application manager's alerter **344** can forward various events or generate new ones as configured (box **926**). If the percentage of aborted transactions exceeds 10 percent (box **928**), the alerter **344** issues an alert to a higher level application manager indicating the percentage of transactions aborting (box **930**) and subscribes to additional metrics (i.e., object and method metrics) (box **932**).

[0150] The example of **FIG. 17** could be varied in many ways to monitor a variety of other rules instead of rules associated with aborting transactions. For example, the application manager could monitor the number of transactions per minute across a set of application servers. For example, **FIG. 18** shows an arrangement in which a higher level application manager **360** subscribes to events from monitored application managers (e.g., application managers at computers **304** and **306**) (box **940**). The events are collected and logged (box **942**). Plural events are combined to determine the number of transactions per minute across

plural computers for the monitored applications (box **944**). A graphical display depicts application data (e.g., application availability) (box **946**).

[**0151**] Having described and illustrated the principles of our invention with reference to illustrated embodiments, it will be recognized that the illustrated embodiments can be modified in arrangement and detail without departing from such principles. It should be understood that the programs, processes, or methods described herein are not related or limited to any particular type of computer apparatus, unless indicated otherwise. Various types of general purpose or specialized computer apparatus may be used with or perform operations in accordance with the teachings described herein. Elements of the illustrated embodiment shown in software may be implemented in hardware and vice versa. In view of the many possible embodiments to which the principles of our invention may be applied, it should be recognized that the detailed embodiments are illustrative only and should not be taken as limiting the scope of our invention. Rather, we claim as our invention all such embodiments as may come within the scope and spirit of the following claims and equivalents thereto.

We claim:

1. In a computer system, a method of monitoring the execution of software comprising a software object, the method comprising:

at run time of the software system, observing a plurality of object-related operations associated with the software object;

responsive to the observing the plurality of object-related operations associated with the software object, generating a first notification indicative of a first object-related operation associated with the software object and a second notification indicative of a second object-related operation associated with the software object; and

performing a calculation on the first notification and the second notification to determine an operational management metric indicative of performance of the software system.

2. The method of claim 1 wherein:

the software object has a plurality of methods;

one of the plurality of object-related operations is a call to one of the plurality of methods; and

the notification indicates the one of the plurality of methods.

3. The method of claim 1 wherein:

the software object has one or more methods;

one of the plurality of object-related operations is a call to one of the one or more methods, the call having at least one parameter having a value; and

the operational management metric indicates the value of the at least one parameter.

4. The method of claim 1 wherein:

the software object has a plurality of methods;

one of the plurality of object-related operations is a return by the software object from a call to one of the plurality of methods by a client of the software object; and

the notification indicates occurrence of the return.

5. The method of claim 1 wherein the notification is published in a loosely coupled publish and subscribe model.

6. The method of claim 1 wherein the observing is performed by an object operation monitor.

7. The method of claim 6 wherein the object operation monitor is interposed between the software object and a client of the software object when the client of the software object requests a reference to the software object.

8. The method of claim 6 wherein the software object is of an object class and the object operation monitor is interposed between the software object and a client of the software object responsive to a request by the client for a newly-created instance of a software object of the object class.

9. The method of claim 6 wherein the object operation monitor is a proxy object interposed between the software object and a client of the software object.

10. The method of claim 6 wherein the object operation monitor is an interceptor interposed between the software object and a client of the software object.

11. The method of claim 6 wherein the object operation monitor is a wrapper interposed between an interface of the software object and a client of the software object.

12. A computer-readable medium having computer-executable instructions stored thereon for causing a computer to perform the method of claim 1.

13. In a computer system, a method of monitoring a software system in an object execution environment, the software system comprising a software object, the method comprising:

at run time of the software system, observing at least one object-related operation associated with the software object;

responsive to the observing the at least one object-related operation associated with the software object, generating a notification indicative of the at least one object-related operation associated with the software object, wherein the generating is performed by object monitoring instrumentation in the execution environment; and

determining an operational management metric to measure performance of the software system based at least in part on the notification.

14. A computer-readable medium having computer-executable instructions stored thereon for causing a computer to perform the method of claim 13.

15. In an execution environment accommodating software objects, a method of monitoring the execution of a software object having one or more methods, wherein the software is operable as a component part of a program, the method comprising:

at run time of the software system, placing an object operation monitor at the software object to monitor at least one operation performed at the monitored software object;

responsive to observation of the at least one object-related operation by the object operation monitor, generating a notification indicative of the at least one object-related operation performed at the monitored software object; and

monitoring the notification to determine an operational management metric associated with the monitored software object.

16. The method of claim 15 wherein the object operation monitor is interposed between the monitored software object and a client of the monitored software object when the client of the monitored software object requests a reference to the monitored software object.

17. The method of claim 15 wherein the monitored software object is of an object class and the object operation monitor is interposed between the monitored software object and a client of the software object responsive to a request by the client for a newly-created instance of a software object of the object class.

18. The method of claim 15 wherein the object operation monitor is a proxy object interposed between the monitored software object and a client of the monitored software object.

19. The method of claim 15 wherein the object operation monitor is an interceptor interposed between the monitored software object and a client of the monitored software object.

20. The method of claim 15 wherein the object operation monitor is a wrapper interposed between an interface of the monitored software object and a client of the monitored software object.

21. A computer-readable medium having computer-executable instructions stored thereon for causing a computer to perform the method of claim 15.

22. A software management system for managing the execution of an object-based program comprising a set of software objects, the management system comprising:

an object execution environment having instrumentation operative to receive a request to perform an interaction with one of the software objects and, responsive to the request, direct a notification indicative of the interaction to a program execution monitor; and

the program execution monitor, operative to receive the notification indicative of the interaction from the instrumented object execution environment and further operative to receive other notifications indicative of other interactions and combine the notification and the other notifications to generate an operational management metric.

* * * * *