(19) **United States**
(12) **Patent Application Publication**
Walsh
(10) **Pub. No.: US 2014/0214745 A1**
(43) **Pub. Date: Jul. 31, 2014**

(54) **METHODS AND SYSTEMS OF PREDICTIVE MONITORING OF OBJECTS IN A DISTRIBUTED NETWORK SYSTEM**

(71) Applicant: **RACKSPACE US, INC.**, San Antonio, TX (US)

(72) Inventor: **Alexander Leonard Walsh**, Waverly (CA)

(21) Appl. No.: **13/841,552**

(22) Filed: **Mar. 15, 2013**

Related U.S. Application Data

(63) Continuation-in-part of application No. 13/752,147, filed on Jan. 28, 2013, Continuation-in-part of application No. 13/752,255, filed on Jan. 28, 2013, Continuation-in-part of application No. 13/752,234, filed on Jan. 28, 2013.

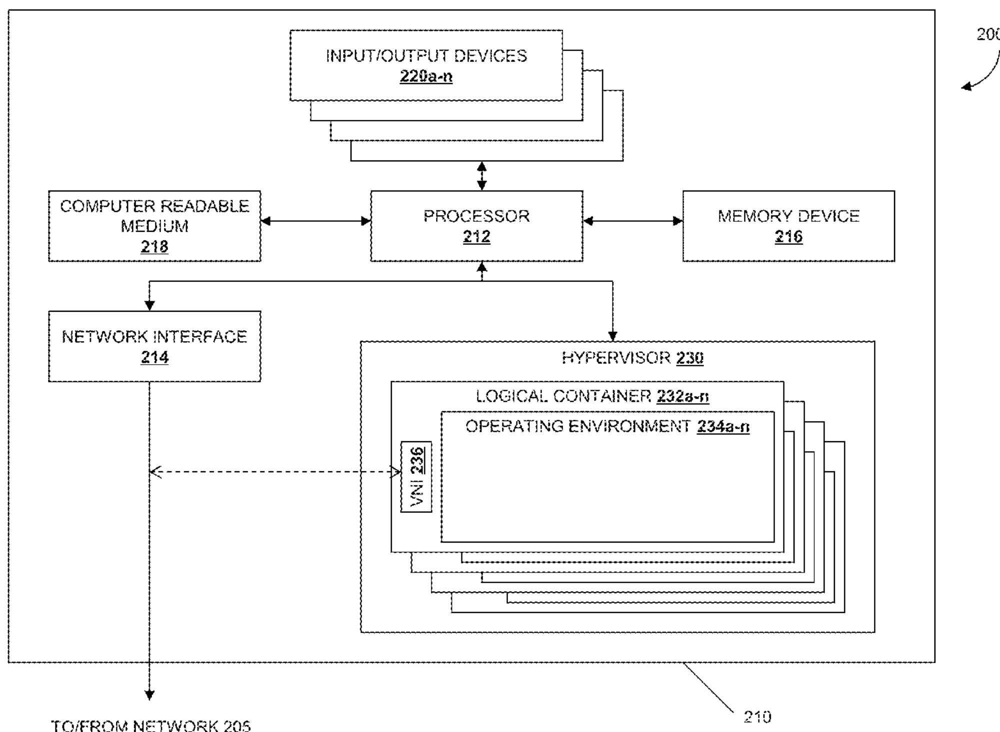
Publication Classification

(51) **Int. Cl.**
G06N 5/02 (2006.01)
H04L 12/26 (2006.01)

(52) **U.S. Cl.**
CPC . ***G06N 5/02*** (2013.01); ***H04L 43/08*** (2013.01)
USPC **706/46**

(57) **ABSTRACT**

Predictive monitoring of objects in a distributed network system providing cloud services is disclosed. In one embodiment, the methods and systems observing one or more update messages sent and received among components of the distributed network system, the update messages comprising information associated with a state of an object on the distributed network system, determine the state of the object in response to the one or more update messages, and reference a predictive object state model to predict occurrence of an artifact in response to the state of the object. Advantageously, the present embodiments provide advanced warning of potential failures in a distributed network systems, which may enable a system administrator or dynamic process to resolve the failure before it ever occurs. Additional benefits and advantages of the present embodiments will become evident in the following description.



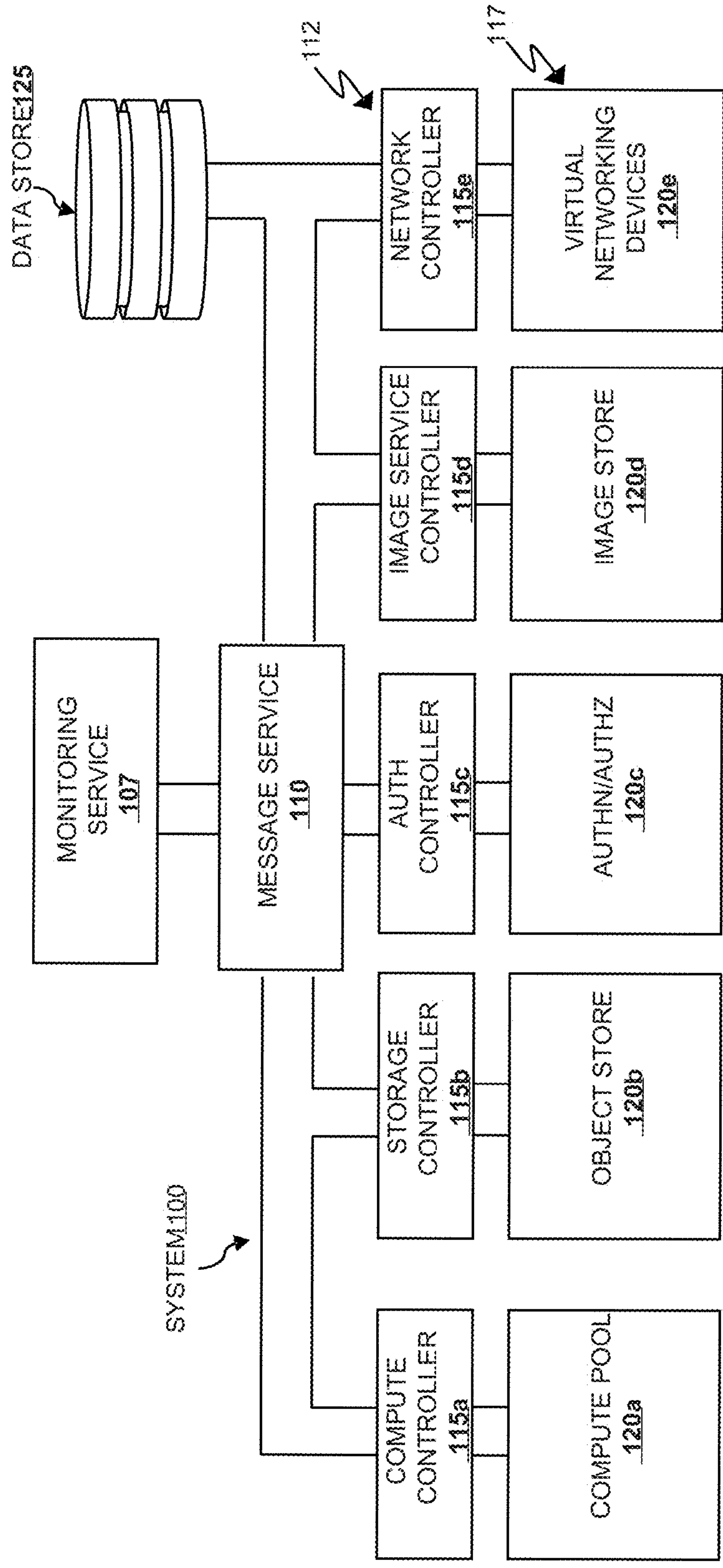


FIGURE1A

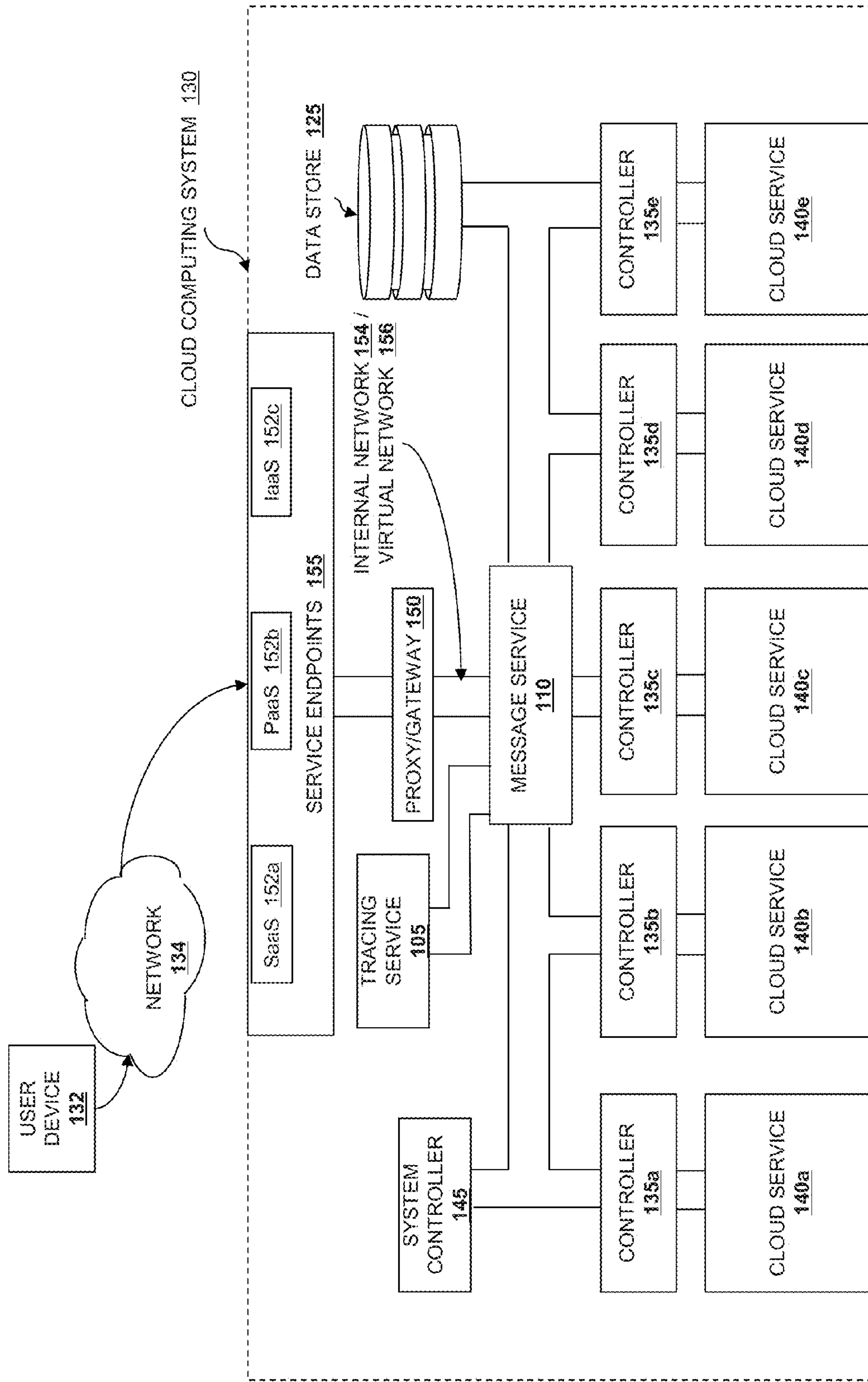


FIGURE 1B

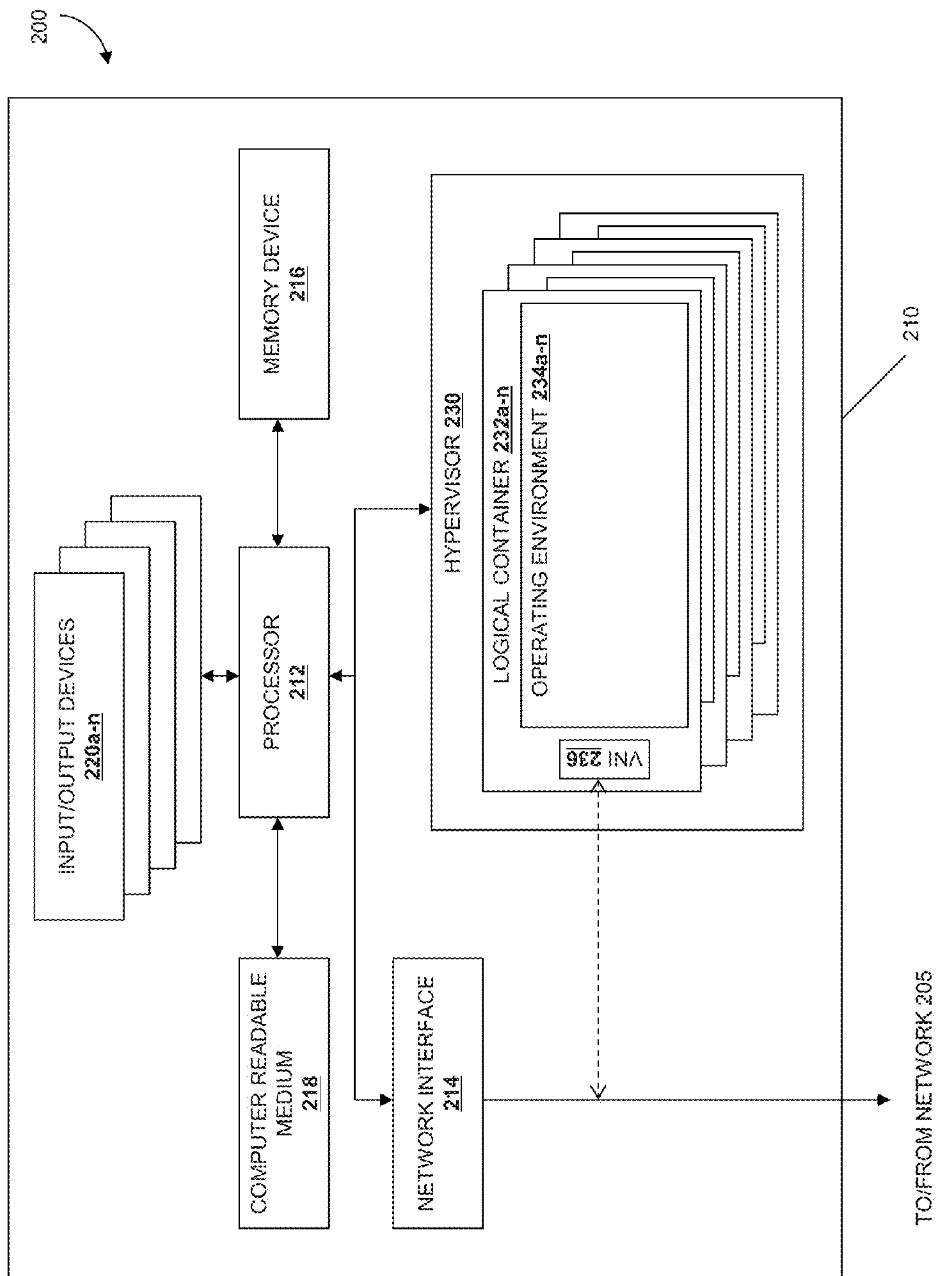


FIGURE 2

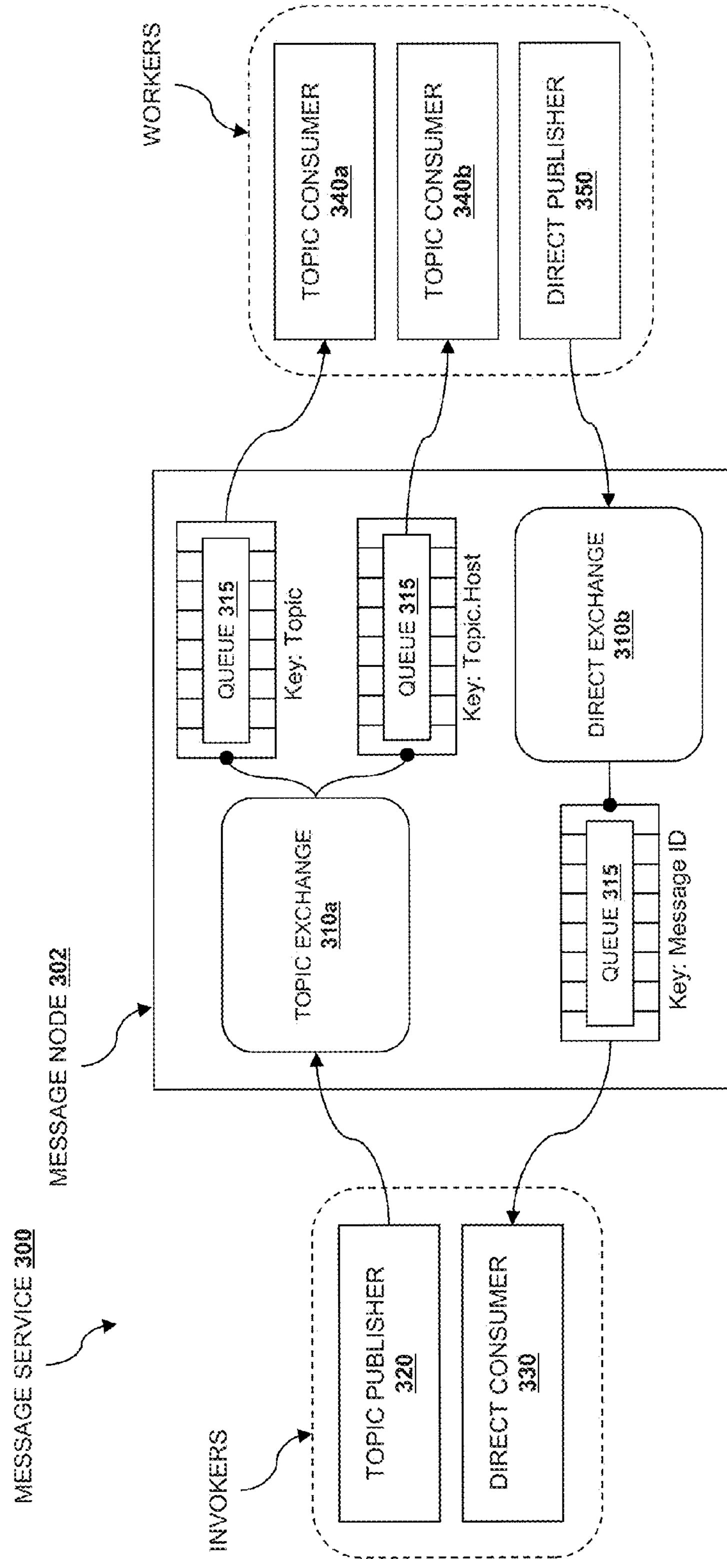


FIGURE 3a

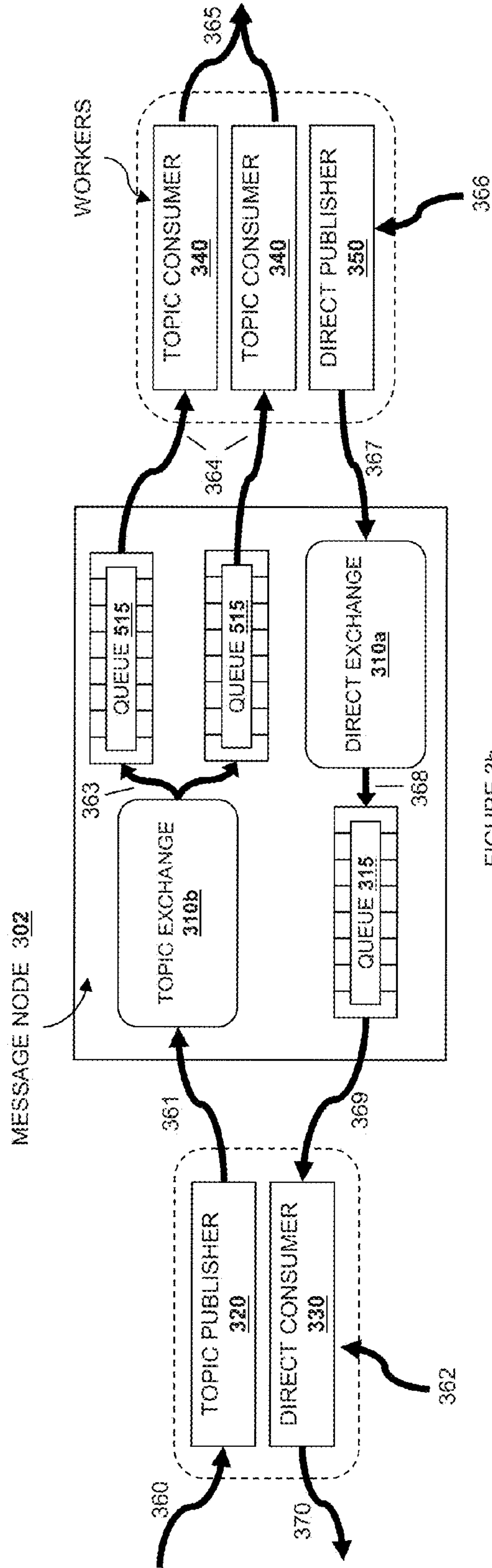


FIGURE 3b

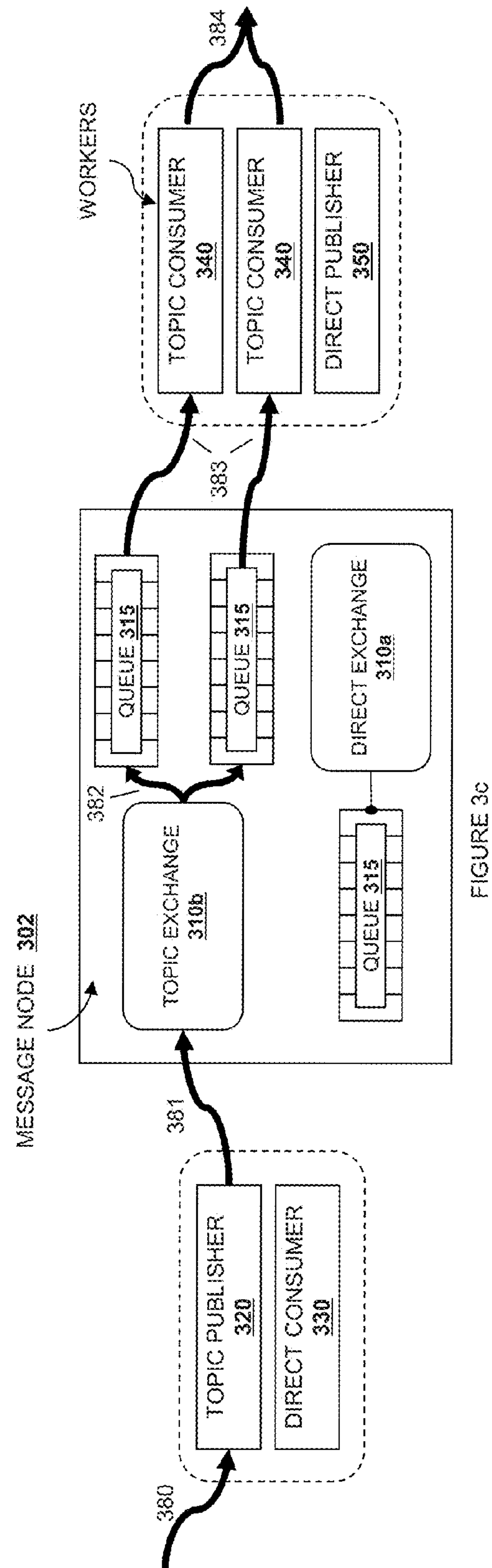
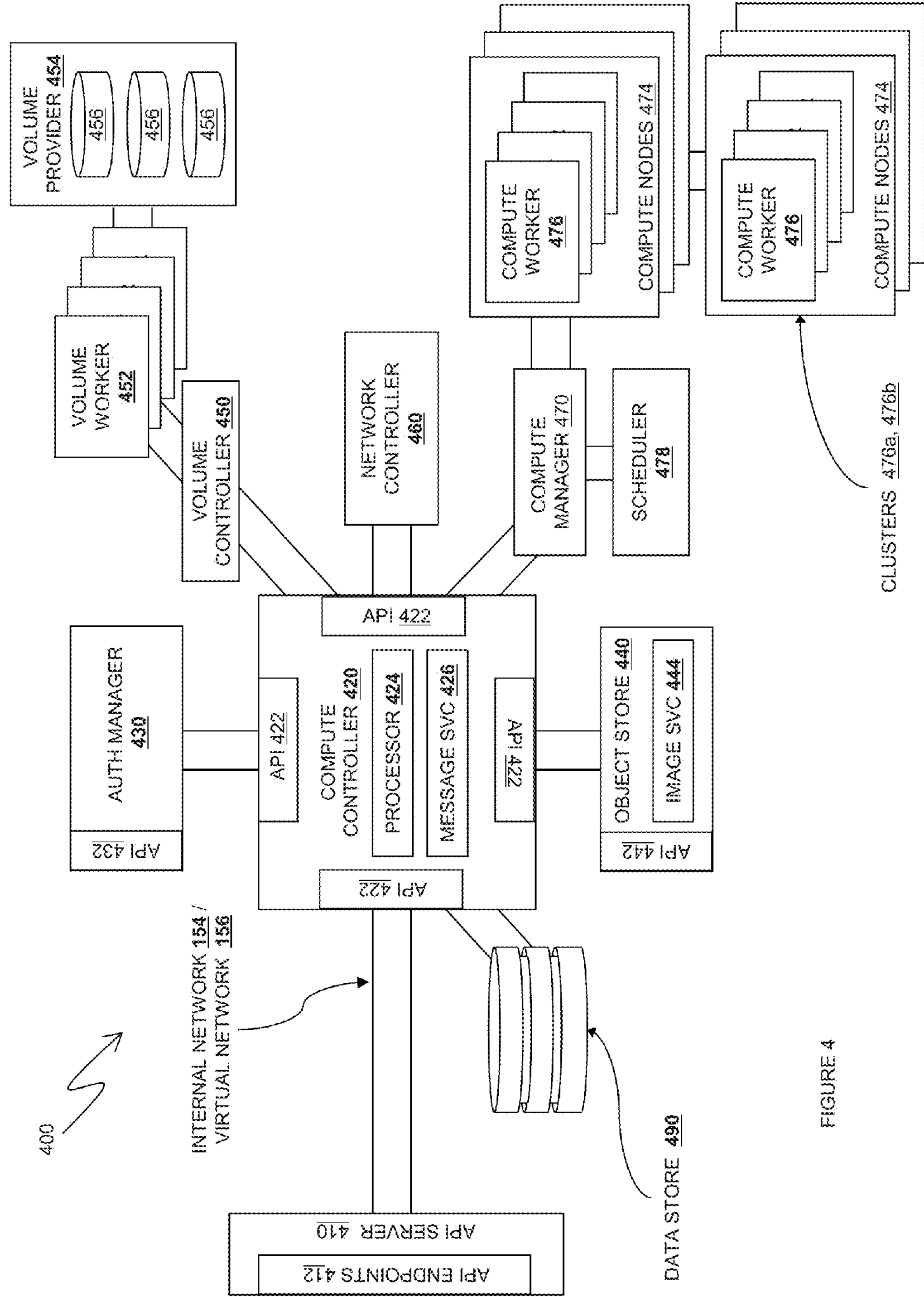


FIGURE 3c


400
INTERNAL NETWORK 154 /
VIRTUAL NETWORK 156
API SERVER 410
API ENDPOINTS 412
DATA STORE 490
AUTH MANAGER 430
API 432
API 422
COMPUTE CONTROLLER 420
PROCESSOR 424
MESSAGE SVC 426
OBJECT STORE 440
IMAGE SVC 444
API 442
VOLUME CONTROLLER 450
VOLUME WORKER 452
VOLUME PROVIDER 454
456
NETWORK CONTROLLER 460
COMPUTE MANAGER 470
SCHEDULER 478
COMPUTE WORKER 476
COMPUTE NODES 474
CLUSTERS 476a, 476b
FIGURE 4

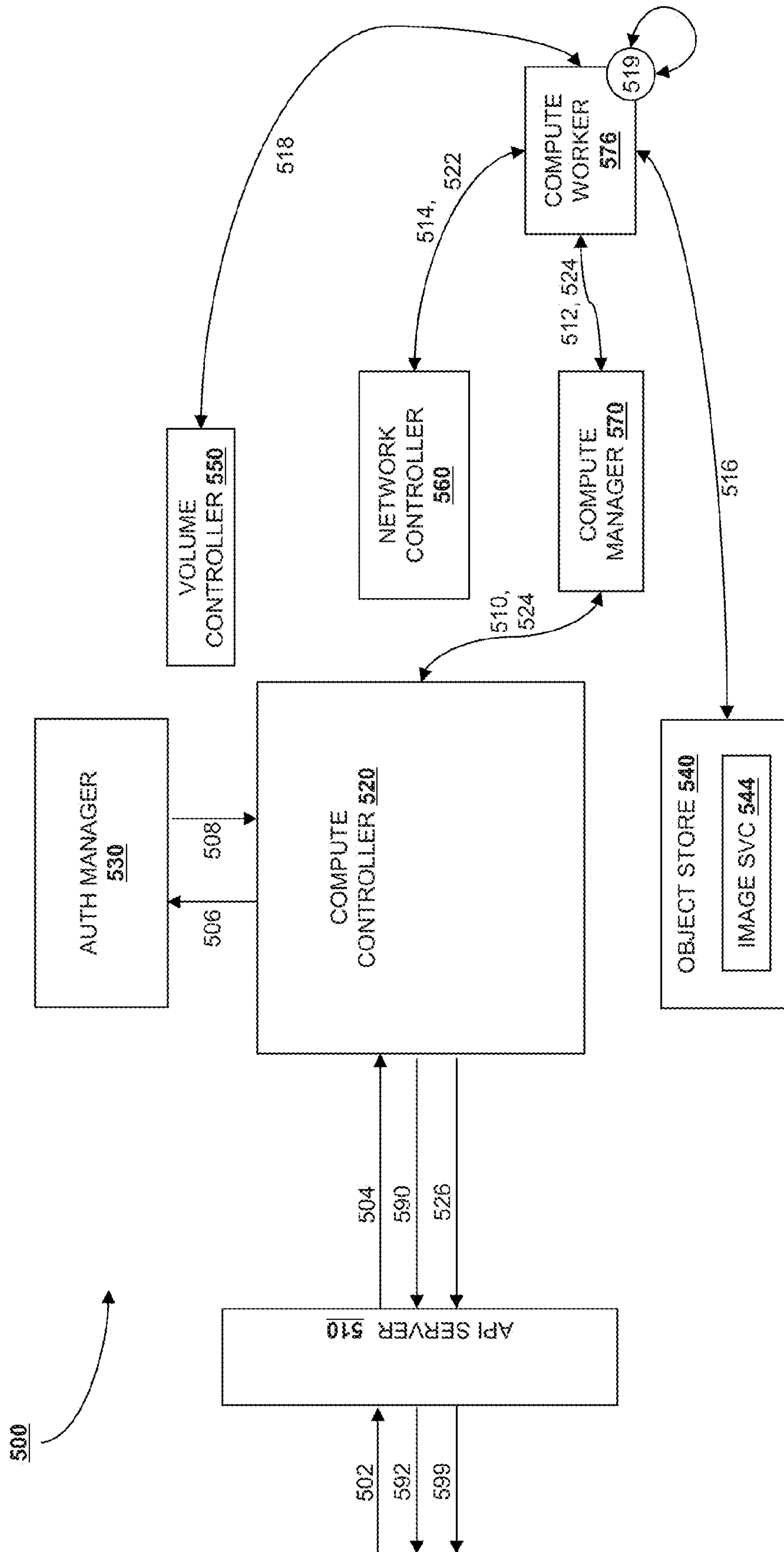


FIGURE 5

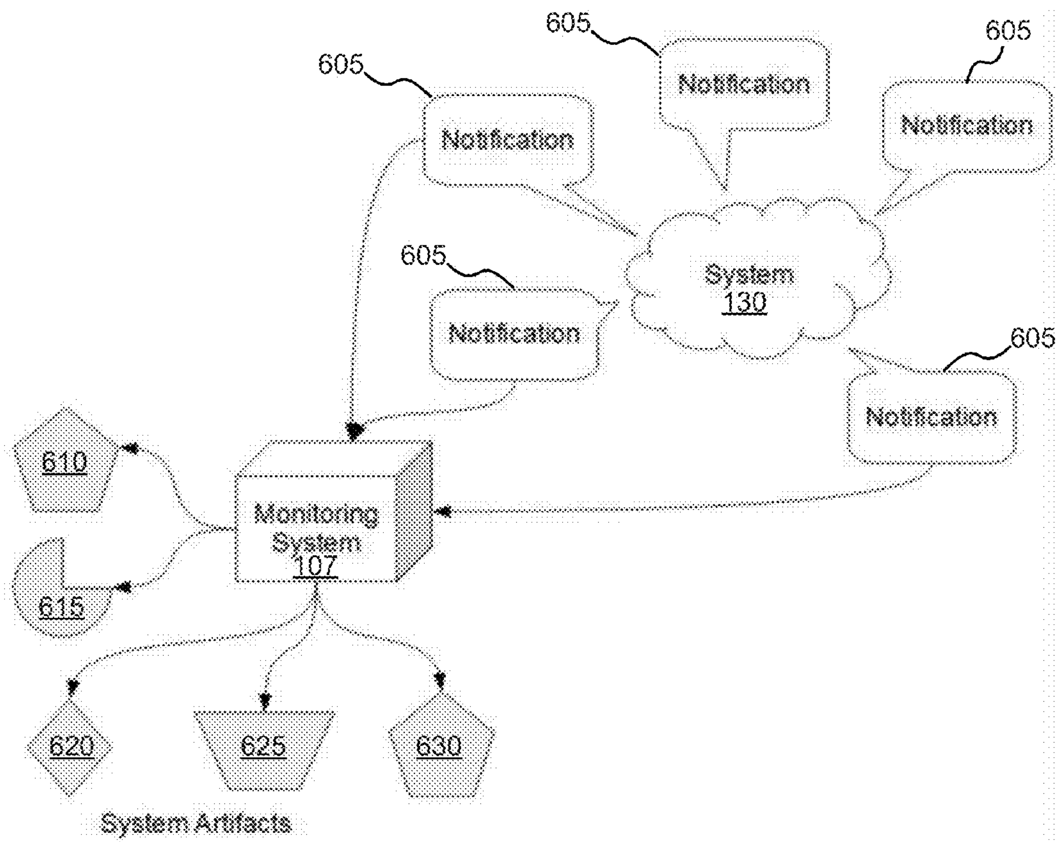


FIG. 6

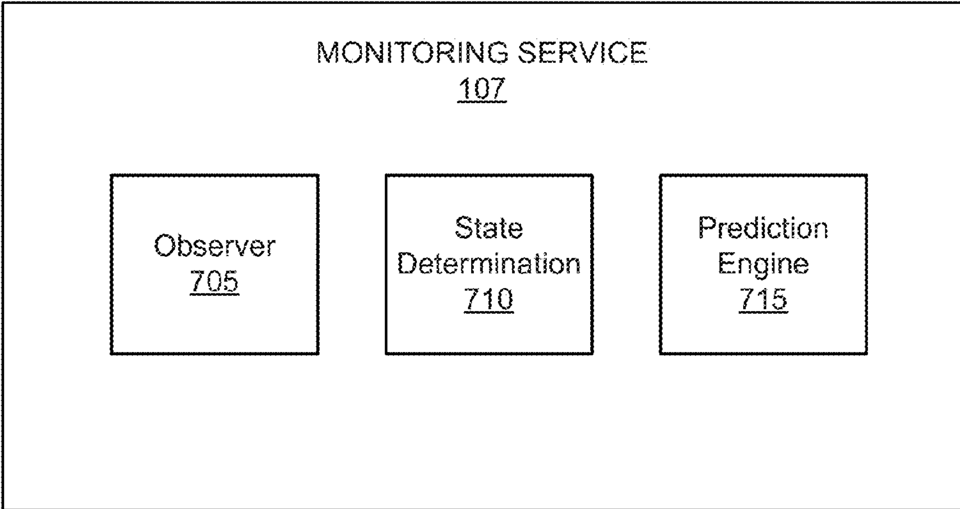


FIG. 7

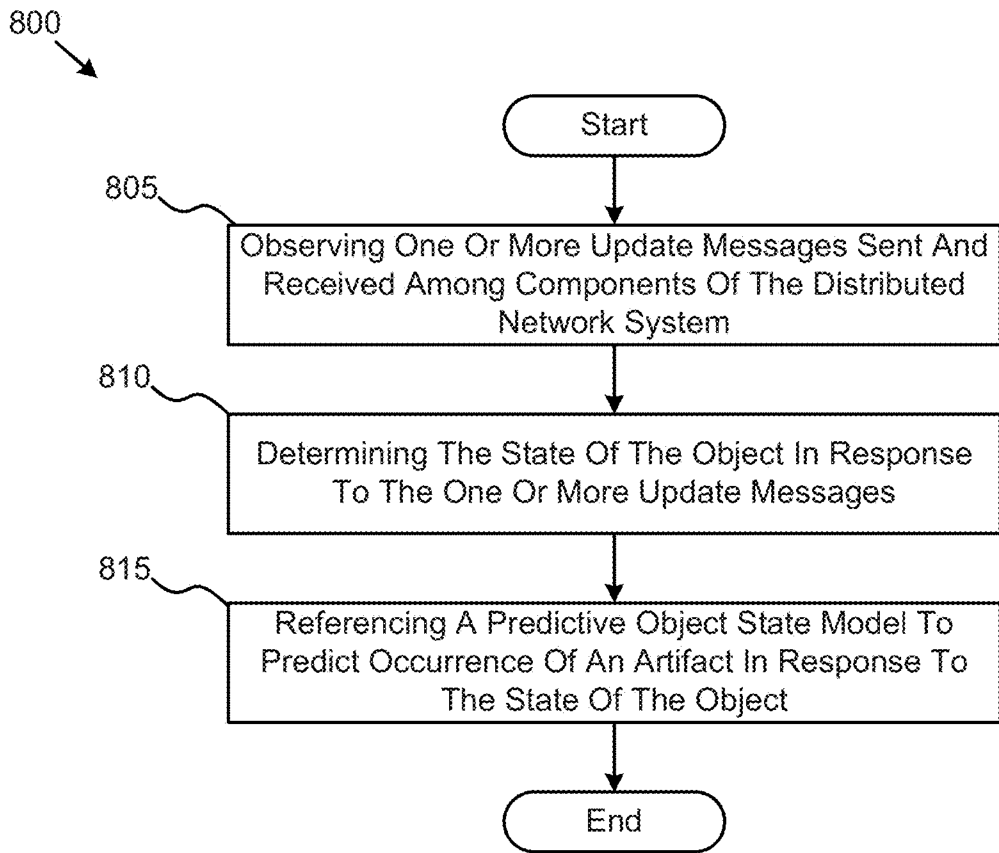


FIG. 8

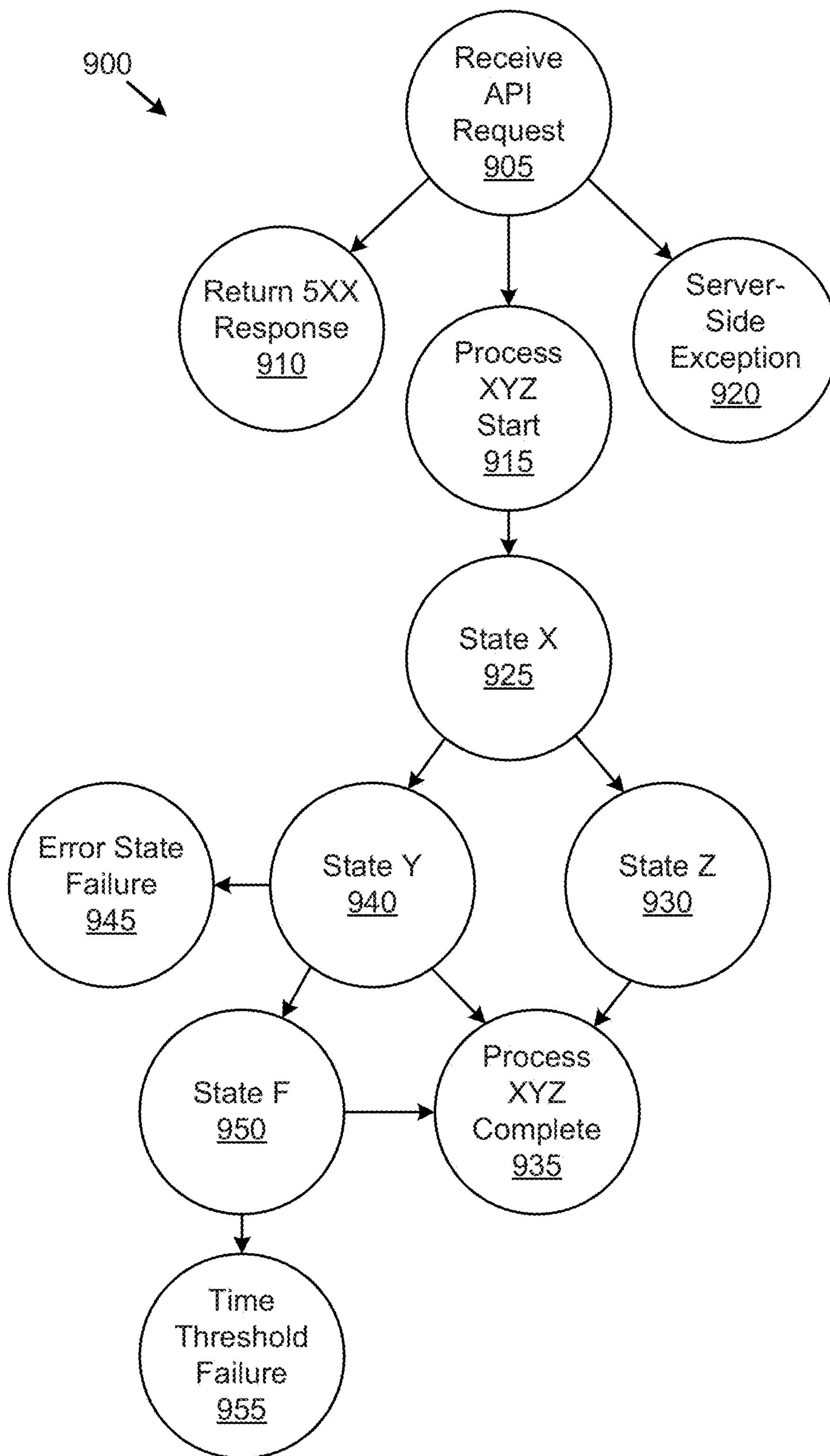
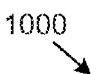


FIG. 9



	Receive API Request	Return 5XX Resposne	Server-Side Exception	Process XYZ Start	State X	State Y	State Z	State F	Process XYZ Complete	Error State Failure	Time Threshold Failure
Receive API Request	N/A	1	2	97	0	0	0	0	0	0	0
Return 5XX Resposne	0	N/A	0	0	0	0	0	0	0	0	0
Server-Side Exception	0	0	N/A	0	0	0	0	0	0	0	0
Process XYZ Start	0	0	0	N/A	97	0	0	0	0	0	0
State X	0	0	0	0	N/A	30	67	0	0	0	0
State Y	0	0	0	0	0	N/A	0	10	17	3	0
State Z	0	0	0	0	0	0	N/A	0	67	0	0
State F	0	0	0	0	0	0	0	N/A	6	0	4
Process XYZ Complete	0	0	0	0	0	0	0	0	N/A	0	0
Error State Failure	0	0	0	0	0	0	0	0	0	N/A	0
Time Threshold Failure	0	0	0	0	0	0	0	0	0	0	N/A

FIG. 10

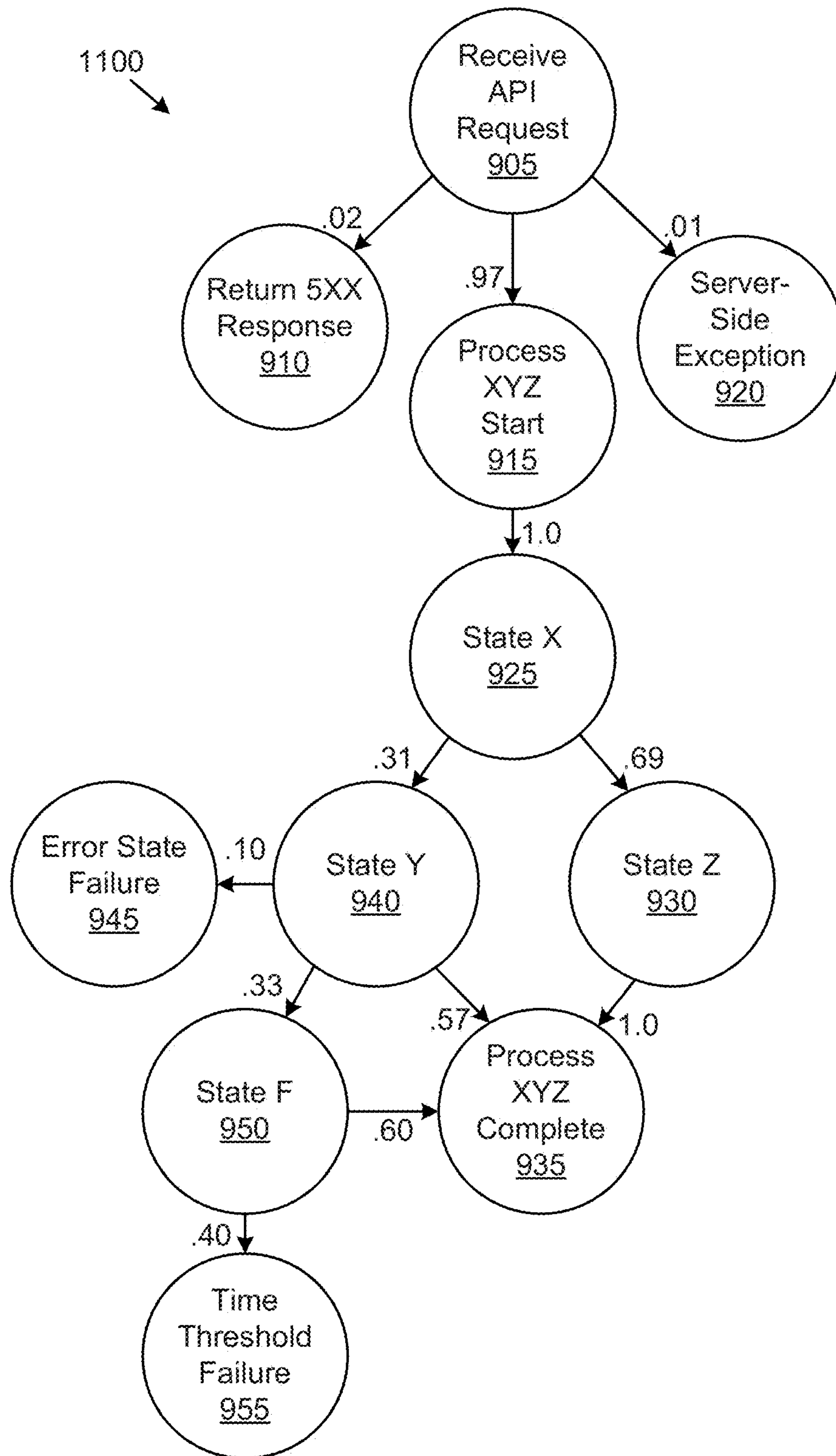


FIG. 11

METHODS AND SYSTEMS OF PREDICTIVE MONITORING OF OBJECTS IN A DISTRIBUTED NETWORK SYSTEM

[0001] This application is a continuation-in-part of, and claims priority to, co-pending non-provisional U.S. patent application Ser. No. 13/752,147 entitled "Methods and Systems of Distributed Tracing," filed Jan. 28, 2013, Ser. No. 13/752,255 entitled "Methods and Systems of Generating a billing feed of a distributed network," filed Jan. 28, 2013, and Ser. No. 13/752,234 entitled "Methods and Systems of Function-Specific Tracing," filed Jan. 28, 2013, each of which are incorporated, in their entirety, herein by reference. This application is related to U.S. patent application Ser. No. 13/______, entitled "Methods and Systems of Monitoring Failures in a Distributed Network System," filed Mar. 15, 2013, and Ser. No. 13/______, entitled "Methods and Systems of Tracking and Verifying Records of System Change Events in a Distributed Network System," filed Mar. 15, 2013, each of which are incorporated, in their entirety, herein by reference.

BACKGROUND

[0002] Distributed network systems are generally large and complex systems with resources distributed on diverse host devices and even on different networks. Monitoring of objects in distributed network systems can often be a challenging, particularly when aspects of the objects are distributed across multiple portions of the distributed network. Nonetheless, it is desirable to monitor objects in a distributed network in order to identify faults associated with the system, its processes and associated objects. It is also commonly desirable to gather performance metrics, usage metrics, and other metrics. Administrators of distributed network systems are often vigilant in efforts to minimize object failures and associated impacts of object failures because system reliability and availability standards are typically part of Service Level Agreements (SLAs), which are agreements that are commonly included in service purchase agreements. Thus, in certain instances, there may be financial consequences associated with object faults.

BRIEF DESCRIPTION OF THE DRAWINGS

[0003] FIG. **1***a* is a schematic view of a distributed system.

[0004] FIG. **1***b* is a schematic view illustrating an external view of a cloud computing system.

[0005] FIG. **2** is a schematic view illustrating an information processing system as used in various embodiments.

[0006] FIG. **3***a* shows a message service system according to various embodiments.

[0007] FIG. **3***b* is a diagram showing how a directed message is sent using the message service according to various embodiments.

[0008] FIG. **3***c* is a diagram showing how a broadcast message is sent using the message service according to various embodiments.

[0009] FIG. **4** shows IaaS-style computational cloud service according to various embodiments.

[0010] FIG. **5** shows an instantiating and launching process for virtual resources according to various embodiments.

[0011] FIG. **6** illustrates a schematic block diagram illustrating an environment in which the present embodiments are configured to operate.

[0012] FIG. **7** is a schematic block diagram illustrating one embodiment of an apparatus for tracking and verifying records of system change events in a distributed network system.

[0013] FIG. **8** is a flowchart diagram illustrating one embodiment of an apparatus configured for predictive monitoring of processes in a distributed network system.

[0014] FIG. **9** illustrates one embodiment of a process state flow.

[0015] FIG. **10** shows an embodiment of a predictive monitoring model.

[0016] FIG. **11** shows another embodiment of a predictive monitoring model.

DETAILED DESCRIPTION

[0017] The following disclosure has reference to predictive monitoring of objects in a distributed network system providing cloud services.

[0018] In one embodiment, the methods and systems observing one or more update messages sent and received among components of the distributed network system, the update messages comprising information associated with a state of an object on the distributed network system, determine the state of the object in response to the one or more update messages, and reference a predictive object state model to predict occurrence of an artifact in response to the state of the object. Advantageously, the present embodiments provide advanced warning of potential failures in a distributed network systems, which may enable a system administrator or dynamic process to resolve the failure before it ever occurs. Additional benefits and advantages of the present embodiments will become evident in the following description.

[0019] Monitored objects may include VM instances, system processes, VM images, data files, IP addresses, and other components of system **100** described below, which may exist or operate in a plurality of states and which may be subject to failures or failure-related errors, degradation, corruption, loss, orphaning, unavailability, or the like.

[0020] FIG. **1**A illustrates a simplified diagram of a distributed application **100** for which various embodiments of verification of records of system change events in a distributed network system may be implemented. It should be appreciated that application **100** is provided merely as an example and that other suitable distributed applications, middleware, or computing systems can benefit from distributed system status verification capabilities described herein. According to one embodiment, application **100** may be a cloud service.

[0021] According to one embodiment, application **100** includes monitoring service **107** configured to provide predictive monitoring of objects in system **100**. In certain embodiments, monitoring service **107** may be implemented in association with event manager **106** (not shown in FIG. **1***a*, but illustrated in related U.S. patent application Ser. No. 13/______, entitled "Methods and Systems of Monitoring Failures in a Distributed Network System," filed Mar. 15, 2013, and 13/______, entitled "Methods and Systems of Tracking and Verifying Records of System Change Events in a Distributed Network System," filed Mar. 15, 2013 which are both incorporated herein by reference in their entirety). As will be described in more detail below, predictive monitoring can provide advanced warning of object-related failures and increase reliability of the distributed application. By way of example, monitoring service **107** can observe messages

within the distributed application across queues and from particular components of the application. As depicted in FIG. 1A, monitoring service **107** interfaces with message service **110** of application **100**. Message service **110** connects various subsystems of the application **100**, and message service **110** may be configured to pass messages relative to one or more elements of system **100**.

[0022] System **100** may include one or more subsystems, such as controllers **112** and services **117**. System **100** may include one or more controllers **112** for the application to be employed in a distributed architecture, such as cloud computing services. As depicted in FIG. **1**A, controllers **112** include a compute controller **115***a*, a storage controller **115***b*, auth controller **115***c*, image service controller **115***d* and network controller **115***e*. Controllers **115** are described with reference to a cloud computing architecture in FIG. **1**. By way of example, network controller **115***a* deals with host machine network configurations and can perform operations for allocating IP addresses, configuring VLANs, implementing security groups and configuring networks. Each of controllers **112** may interface with one or more services. As depicted in FIG. 1A, compute controller **115***a* interfaces with compute pool **120***a*, storage controller **115***b* may interface with object store **120***b*, auth controller **115***c* may interface with authentication/authorization controller **120***c*, image service controller **115***d* may interface with image store **120***d* and network controller **115***e* may interface with virtual networking devices **120***e*. Although controllers **115** and services **120** are with reference to an open architecture, it should be appreciated that the methods and systems for predictive modeling may be equally applied to other distributed applications.

[0023] Referring now to FIG. **1***b*, an external view of a cloud computing system **130** is illustrated. Cloud computing system **130** includes monitoring service **107** and message service **110**. According to one embodiment, monitoring service **107** can observe messages of cloud computing system **130** and constructs a predictive state model of objects on the cloud system **130**. According to another embodiment, controllers and services of the cloud computing system **130** may include monitoring services **107** to provide state predictions related to local objects.

[0024] The cloud computing system **130** includes a user device **132** connected to a network **134** such as, for example, a Transport Control Protocol/Internet Protocol (TCP/IP) network (e.g., the Internet.) The user device **132** is coupled to the cloud computing system **130** via one or more service endpoints **155**. Depending on the type of cloud service provided, these endpoints give varying amounts of control relative to the provisioning of resources within the cloud computing system **130**. For example, SaaS endpoint **152***a* typically only gives information and access relative to the application running on the cloud storage system, and the scaling and processing aspects of the cloud computing system is obscured from the user. PaaS endpoint **152***b* typically gives an abstract Application Programming Interface (API) that allows developers to declaratively request or command the backend storage, computation, and scaling resources provided by the cloud, without giving exact control to the user. IaaS endpoint **152***c* typically provides the ability to directly request the provisioning of resources, such as computation units (typically virtual machines), software-defined or software-controlled network elements like routers, switches, domain name servers, etc., file or object storage facilities, authorization services, database services, queue services and endpoints, etc. In addition, users interacting with an IaaS cloud are typically able to provide virtual machine images that have been customized for user-specific functions. This allows the cloud computing system **130** to be used for new, user-defined services without requiring specific support.

[0025] It is important to recognize that the control allowed via an IaaS endpoint is not complete. Within the cloud computing system **130** are one or more cloud controllers **135** (running what is sometimes called a "cloud operating system") that work on an even lower level, interacting with physical machines, managing the contradictory demands of the multi-tenant cloud computing system **130**. In one embodiment, these correspond to the controllers and services discussed relative to FIG. **1***a*. The workings of the cloud controllers **135** are typically not exposed outside of the cloud computing system **130**, even in an IaaS context. In one embodiment, the commands received through one of the service endpoints **155** are then routed via one or more internal networks **154**. The internal network **154** couples the different services to each other. The internal network **154** may encompass various protocols or services, including but not limited to electrical, optical, or wireless connections at the physical layer; Ethernet, Fiber channel, ATM, and SONET at the MAC layer; TCP, UDP, ZeroMQ or other services at the connection layer; and XMPP, HTTP, AMPQ, STOMP, SMS, SMTP, SNMP, or other standards at the protocol layer. The internal network **154** is typically not exposed outside the cloud computing system, except to the extent that one or more virtual networks **156** may be exposed that control the internal routing according to various rules. The virtual networks **156** typically do not expose as much complexity as may exist in the actual internal network **154**; but varying levels of granularity can be exposed to the control of the user, particularly in IaaS services.

[0026] In one or more embodiments, it may be useful to include various processing or routing nodes in the network layers **154** and **156**, such as proxy/gateway **150**. Other types of processing or routing nodes may include switches, routers, switch fabrics, caches, format modifiers, or correlators. These processing and routing nodes may or may not be visible to the outside. It is typical that one level of processing or routing nodes may be internal only, coupled to the internal network **154**, whereas other types of network services may be defined by or accessible to users, and show up in one or more virtual networks **156**. Either of the internal network **154** or the virtual networks **156** may be encrypted or authenticated according to the protocols and services described below.

[0027] In various embodiments, one or more parts of the cloud computing system **130** may be disposed on a single host. Accordingly, some of the "network" layers **154** and **156** may be composed of an internal call graph, inter-process communication (IPC), or a shared memory communication system.

[0028] Once a communication passes from the endpoints via a network layer **154** or **156**, as well as possibly via one or more switches or processing devices **150**, it is received by one or more applicable cloud controllers **135**. The cloud controllers **135** are responsible for interpreting the message and coordinating the performance of the necessary corresponding services, returning a response if necessary. Although the cloud controllers **135** may provide services directly, more typically the cloud controllers **135** are in operative contact with the service resources **140** necessary to provide the corresponding services. For example, it is possible for different

services to be provided at different levels of abstraction. For example, a service **140***a* may be a "compute" service that will work at an IaaS level, allowing the creation and control of user-defined virtual computing resources. In addition to the services discussed relative to FIG. **1***a*, a cloud computing system **130** may provide a declarative storage API, a SaaS-level Queue service **140***c*, a DNS service **140***d*, or a Database service **140***e*, or other application services without exposing any of the underlying scaling or computational resources. Other services are contemplated as discussed in detail below.

[0029] In various embodiments, various cloud computing services or the cloud computing system itself may require a message passing system. The message routing service **110** is available to address this need, but it is not a required part of the system architecture in at least one embodiment. In one embodiment, the message routing service is used to transfer messages from one component to another without explicitly linking the state of the two components. Note that this message routing service **110** may or may not be available for user-addressable systems; in one preferred embodiment, there is a separation between storage for cloud service state and for user data, including user service state.

[0030] In various embodiments, various cloud computing services or the cloud computing system itself may require a persistent storage for system state. The data store **125** is available to address this need, but it is not a required part of the system architecture in at least one embodiment. In one embodiment, various aspects of system state are saved in redundant databases on various hosts or as special files in an object storage service. In a second embodiment, a relational database service is used to store system state. In a third embodiment, a column, graph, or document-oriented database is used. Note that this persistent storage may or may not be available for user-addressable systems; in one preferred embodiment, there is a separation between storage for cloud service state and for user data, including user service state.

[0031] In various embodiments, it may be useful for the cloud computing system **130** to have a system controller **145**. In one embodiment, the system controller **145** is similar to the cloud computing controllers **135**, except that it is used to control or direct operations at the level of the cloud computing system **130** rather than at the level of an individual service.

[0032] For clarity of discussion above, only one user device **132** has been illustrated as connected to the cloud computing system **130**, and the discussion generally referred to receiving a communication from outside the cloud computing system, routing it to a cloud controller **135**, and coordinating processing of the message via a service **130**, the infrastructure described is also equally available for sending out messages. These messages may be sent out as replies to previous communications, or they may be internally sourced. Routing messages from a particular service **130** to a user device **132** is accomplished in the same manner as receiving a message from user device **132** to a service **130**, just in reverse. The precise manner of receiving, processing, responding, and sending messages is described below with reference to the various discussed service embodiments. One of skill in the art will recognize, however, that a plurality of user devices **132** may, and typically will, be connected to the cloud computing system **130** and that each element or set of elements within the cloud computing system is replicable as necessary. Further, the cloud computing system **130**, whether or not it has one endpoint or multiple endpoints, is expected to encompass embodiments including public clouds, private clouds, hybrid clouds, and multi-vendor clouds.

[0033] Each of the user device **132**, the cloud computing system **130**, the endpoints **152**, the cloud controllers **135** and the cloud services **140** typically include a respective information processing system, a subsystem, or a part of a subsystem for executing processes and performing operations (e.g., processing or communicating information). An information processing system is an electronic device capable of processing, executing or otherwise handling information, such as a computer. FIG. **2** shows an information processing system **210** that is representative of one of, or a portion of, the information processing systems described above.

[0034] Referring now to FIG. **2**, diagram **200** shows an information processing system **210** configured to host one or more virtual machines, coupled to a network **205**. The network **205** could be one or both of the networks **154** and **156** described above. An information processing system is an electronic device capable of processing, executing or otherwise handling information. Examples of information processing systems include a server computer, a personal computer (e.g., a desktop computer or a portable computer such as, for example, a laptop computer), a handheld computer, and/or a variety of other information handling systems known in the art. The information processing system **210** shown is representative of, one of, or a portion of, the information processing systems described above.

[0035] The information processing system **210** may include any or all of the following: (a) a processor **212** for executing and otherwise processing instructions, (b) one or more network interfaces **214** (e.g., circuitry) for communicating between the processor **212** and other devices, those other devices possibly located across the network **205**; (c) a memory device **216** (e.g., FLASH memory, a random access memory (RAM) device or a read-only memory (ROM) device for storing information (e.g., instructions executed by processor **212** and data operated upon by processor **212** in response to such instructions)). In some embodiments, the information processing system **210** may also include a separate computer-readable medium **218** operably coupled to the processor **212** for storing information and instructions as described further below.

[0036] In one embodiment, there is more than one network interface **214**, so that the multiple network interfaces can be used to separately route management, production, and other traffic. In one exemplary embodiment, an information processing system has a "management" interface at **1** GB/s, a "production" interface at **10** GB/s, and may have additional interfaces for channel bonding, high availability, or performance. An information processing device configured as a processing or routing node may also have an additional interface dedicated to public Internet traffic, and specific circuitry or resources necessary to act as a VLAN trunk.

[0037] In some embodiments, the information processing system **210** may include a plurality of input/output devices **220***a-n* which are operably coupled to the processor **212**, for inputting or outputting information, such as a display device **220***a*, a print device **220***b*, or other electronic circuitry **220***c-n* for performing other operations of the information processing system **210** known in the art.

[0038] With reference to the computer-readable media, including both memory device **216** and secondary computer-readable medium **218**, the computer-readable media and the processor **212** are structurally and functionally interrelated

with one another as described below in further detail, and information processing system of the illustrative embodiment is structurally and functionally interrelated with a respective computer-readable medium similar to the manner in which the processor **212** is structurally and functionally interrelated with the computer-readable media **216** and **218**. As discussed above, the computer-readable media may be implemented using a hard disk drive, a memory device, and/or a variety of other computer-readable media known in the art, and when including functional descriptive material, data structures are created that define structural and functional interrelationships between such data structures and the computer-readable media (and other aspects of the system **200**). Such interrelationships permit the data structures' functionality to be realized. For example, in one embodiment the processor **212** reads (e.g., accesses or copies) such functional descriptive material from the network interface **214**, the computer-readable media **218** onto the memory device **216** of the information processing system **210**, and the information processing system **210** (more particularly, the processor **212**) performs its operations, as described elsewhere herein, in response to such material stored in the memory device of the information processing system **210**. In addition to reading such functional descriptive material from the computer-readable medium **218**, the processor **212** is capable of reading such functional descriptive material from (or through) the network **105**. In one embodiment, the information processing system **210** includes at least one type of computer-readable media that is non-transitory. For explanatory purposes below, singular forms such as "computer-readable medium," "memory," and "disk" are used, but it is intended that these may refer to all or any portion of the computer-readable media available in or to a particular information processing system **210**, without limiting them to a specific location or implementation.

[0039] The information processing system **210** includes a hypervisor **230**. The hypervisor **230** may be implemented in software, as a subsidiary information processing system, or in a tailored electrical circuit or as software instructions to be used in conjunction with a processor to create a hardware-software combination that implements the specific functionality described herein. To the extent that software is used to implement the hypervisor, it may include software that is stored on a computer-readable medium, including the computer-readable medium **218**. The hypervisor may be included logically "below" a host operating system, as a host itself, as part of a larger host operating system, or as a program or process running "above" or "on top of" a host operating system. Examples of hypervisors include Xenserver, KVM, VMware, Microsoft's Hyper-V, and emulation programs such as QEMU.

[0040] The hypervisor **230** includes the functionality to add, remove, and modify a number of logical containers **232***a-n* associated with the hypervisor. Zero, one, or many of the logical containers **232***a-n* contain associated operating environments **234***a-n*. The logical containers **232***a-n* can implement various interfaces depending upon the desired characteristics of the operating environment. In one embodiment, a logical container **232** implements a hardware-like interface, such that the associated operating environment **234** appears to be running on or within an information processing system such as the information processing system **210**. For example, one embodiment of a logical container **234** could implement an interface resembling an x86, x86-64, ARM, or other computer instruction set with appropriate RAM, busses, disks, and network devices. A corresponding operating environment **234** for this embodiment could be an operating system such as Microsoft Windows, Linux, Linux-Android, or Mac OS X. In another embodiment, a logical container **232** implements an operating system-like interface, such that the associated operating environment **234** appears to be running on or within an operating system. For example one embodiment of this type of logical container **232** could appear to be a Microsoft Windows, Linux, or Mac OS X operating system. Another possible operating system includes an Android operating system, which includes significant runtime functionality on top of a lower-level kernel. A corresponding operating environment **234** could enforce separation between users and processes such that each process or group of processes appeared to have sole access to the resources of the operating system. In a third environment, a logical container **232** implements a software-defined interface, such a language runtime or logical process that the associated operating environment **234** can use to run and interact with its environment. For example one embodiment of this type of logical container **232** could appear to be a Java, Dalvik, Lua, Python, or other language virtual machine. A corresponding operating environment **234** would use the built-in threading, processing, and code loading capabilities to load and run code. Adding, removing, or modifying a logical container **232** may or may not also involve adding, removing, or modifying an associated operating environment **234**. For ease of explanation below, these operating environments will be described in terms of an embodiment as "Virtual Machines," or "VMs," but this is simply one implementation among the options listed above.

[0041] In one or more embodiments, a VM has one or more virtual network interfaces **236**. How the virtual network interface is exposed to the operating environment depends upon the implementation of the operating environment. In an operating environment that mimics a hardware computer, the virtual network interface **236** appears as one or more virtual network interface cards. In an operating environment that appears as an operating system, the virtual network interface **236** appears as a virtual character device or socket. In an operating environment that appears as a language runtime, the virtual network interface appears as a socket, queue, message service, or other appropriate construct. The virtual network interfaces (VNIs) **236** may be associated with a virtual switch (Vswitch) at either the hypervisor or container level. The VNI **236** logically couples the operating environment **234** to the network, and allows the VMs to send and receive network traffic. In one embodiment, the physical network interface card **214** is also coupled to one or more VMs through a Vswitch.

[0042] In one or more embodiments, each VM includes identification data for use naming, interacting, or referring to the VM. This can include the Media Access Control (MAC) address, the Internet Protocol (IP) address, and one or more unambiguous names or identifiers.

[0043] In one or more embodiments, a "volume" is a detachable block storage device. In some embodiments, a particular volume can only be attached to one instance at a time, whereas in other embodiments a volume works like a Storage Area Network (SAN) so that it can be concurrently accessed by multiple devices. Volumes can be attached to either a particular information processing device or a particular virtual machine, so they are or appear to be local to that machine. Further, a volume attached to one information pro-

cessing device or VM can be exported over the network to share access with other instances using common file sharing protocols. In other embodiments, there are areas of storage declared to be "local storage." Typically a local storage volume will be storage from the information processing device shared with or exposed to one or more operating environments on the information processing device. Local storage is guaranteed to exist only for the duration of the operating environment; recreating the operating environment may or may not remove or erase any local storage associated with that operating environment.

[0044] Message Service

[0045] Between the various virtual machines and virtual devices, it may be necessary to have a reliable messaging infrastructure. In various embodiments, a message queuing service is used for both local and remote communication so that there is no requirement that any of the services exist on the same physical machine. Various existing messaging infrastructures are contemplated, including AMQP, ZeroMQ, STOMP and XMPP. Note that this messaging system may or may not be available for user-addressable systems; in one preferred embodiment, there is a separation between internal messaging services and any messaging services associated with user data.

[0046] In one embodiment, the message service sits between various components and allows them to communicate in a loosely coupled fashion. This can be accomplished using Remote Procedure Calls (RPC hereinafter) to communicate between components, built atop either direct messages and/or an underlying publish/subscribe infrastructure. In a typical embodiment, it is expected that both direct and topic-based exchanges are used. This allows for decoupling of the components, full asynchronous communications, and transparent balancing between equivalent components. In some embodiments, calls between different APIs can be supported over the distributed system by providing an adapter class which takes care of marshalling and unmarshalling of messages into function calls.

[0047] In one embodiment, a cloud controller **135** (or the applicable cloud service **140**) creates two queues at initialization time, one that accepts node-specific messages and another that accepts generic messages addressed to any node of a particular type. This allows both specific node control as well as orchestration of the cloud service without limiting the particular implementation of a node. In an embodiment in which these message queues are bridged to an API, the API can act as a consumer, server, or publisher.

[0048] Turning now to FIG. **3***a*, one implementation of a message service **110** is shown. For simplicity of description, FIG. **3***a* shows the message service **300** when a single instance is deployed and shared in the cloud computing system **130**, but the message service can be either centralized or fully distributed.

[0049] In one embodiment, the message service **300** keeps traffic associated with different queues or routing keys separate, so that disparate services can use the message service without interfering with each other. Accordingly, the message queue service may be used to communicate messages between network elements, between cloud services **140**, between cloud controllers **135**, between network elements, or between any group of sub-elements within the above. More than one message service may be used, and a cloud service **140** may use its own message service as required.

[0050] For clarity of exposition, access to the message service will be described in terms of "Invokers" and "Workers," but these labels are purely expository and are not intended to convey a limitation on purpose; in some embodiments, a single component (such as a VM) may act first as an Invoker, then as a Worker, the other way around, or simultaneously in each role. An Invoker is a component that sends messages in the system via two operations: 1) an RPC (Remote Procedure Call) directed message and ii) an RPC broadcast. A Worker is a component that receives messages from the message system and replies accordingly.

[0051] In one embodiment, there is a message node **302** including one or more exchanges **310**. In a second embodiment, the message system is "brokerless," and one or more exchanges are located at each client. The exchanges **310** act as internal message routing elements so that components interacting with the message service can send and receive messages. In one embodiment, these exchanges are subdivided further into a topic exchange **310***a* and a direct exchange **310***b*. An exchange **310** is a routing structure or system that exists in a particular context. In a one embodiment, multiple contexts can be included within a single message service with each one acting independently of the others. In one embodiment, the type of exchange, such as a topic exchange **310***a* vs. direct exchange **310***b* determines the routing policy. In a second embodiment, the routing policy is determined via a series of routing rules evaluated by the exchange **310**.

[0052] The direct exchange **310***a* is a routing element created during or for RPC directed message operations. In one embodiment, there are many instances of a direct exchange **310***a* that are created as needed for the message service. In a further embodiment, there is one direct exchange **310***a* created for each RPC directed message received by the system.

[0053] The topic exchange **310***a* is a routing element created during or for RPC directed broadcast operations. In one simple embodiment, every message received by the topic exchange is received by every other connected component. In a second embodiment, the routing rule within a topic exchange is described as publish-subscribe, wherein different components can specify a discriminating function and only topics matching the discriminator are passed along. In one embodiment, there are many instances of a topic exchange **310***b* that are created as needed for the message service. In one embodiment, there is one topic-based exchange for every topic created in the cloud computing system. In a second embodiment, there are a set number of topics that have pre-created and persistent topic exchanges **310***b*.

[0054] Within one or more of the exchanges **310**, it may be useful to have a queue element **315**. A queue **315** is a message stream; messages sent into the stream are kept in the queue **315** until a consuming component connects to the queue and fetches the message. A queue **315** can be shared or can be exclusive. In one embodiment, queues with the same topic are shared amongst Workers subscribed to that topic.

[0055] In a typical embodiment, a queue **315** will implement a FIFO policy for messages and ensure that they are delivered in the same order that they are received. In other embodiments, however, a queue **315** may implement other policies, such as LIFO, a priority queue (highest-priority messages are delivered first), or age (oldest objects in the queue are delivered first), or other configurable delivery policies. In other embodiments, a queue **315** may or may not make any guarantees related to message delivery or message persistence.

[0056] In one embodiment, element **320** is a topic publisher. A topic publisher **320** is created, instantiated, or awakened when an RPC directed message or an RPC broadcast operation is executed; this object is instantiated and used to push a message to the message system. Every publisher connects always to the same topic-based exchange; its life-cycle is limited to the message delivery.

[0057] In one embodiment, element **330** is a direct consumer. A direct consumer **330** is created, instantiated, or awakened if an RPC directed message operation is executed; this component is instantiated and used to receive a response message from the queuing system. Every direct consumer **330** connects to a unique direct-based exchange via a unique exclusive queue, identified by a UUID or other unique name. The life-cycle of the direct consumer **330** is limited to the message delivery. In one embodiment, the exchange and queue identifiers are included the message sent by the topic publisher **320** for RPC directed message operations.

[0058] In one embodiment, elements **340** (elements **340***a* and **340***b*) are topic consumers. In one embodiment, a topic consumer **340** is created, instantiated, or awakened at system start. In a second embodiment, a topic consumer **340** is created, instantiated, or awakened when a topic is registered with the message system **300**. In a third embodiment, a topic consumer **340** is created, instantiated, or awakened at the same time that a Worker or Workers are instantiated and persists as long as the associated Worker or Workers have not been destroyed. In this embodiment, the topic consumer **340** is used to receive messages from the queue and it invokes the appropriate action as defined by the Worker role. A topic consumer **340** connects to the topic-based exchange either via a shared queue or via a unique exclusive queue. In one embodiment, every Worker has two associated topic consumers **340**, one that is addressed only during an RPC broadcast operations (and it connects to a shared queue whose exchange key is defined by the topic) and the other that is addressed only during an RPC directed message operations, connected to a unique queue whose with the exchange key is defined by the topic and the host.

[0059] In one embodiment, element **350** is a direct publisher. In one embodiment, a direct publisher **350** is created, instantiated, or awakened for RPC directed message operations and it is instantiated to return the message required by the request/response operation. The object connects to a direct-based exchange whose identity is dictated by the incoming message.

[0060] Turning now to FIG. **3***b*, one embodiment of the process of sending an RPC directed message is shown relative to the elements of the message system **300** as described relative to FIG. **3***a*. All elements are as described above relative to FIG. **3***a* unless described otherwise. At step **360**, a topic publisher **320** is instantiated. At step **361**, the topic publisher **320** sends a message to an exchange **310***b*. At step **362**, a direct consumer **330** is instantiated to wait for the response message. At step **363**, the message is dispatched by the exchange **310***b*. At step **364**, the message is fetched by the topic consumer **340** dictated by the routing key (either by topic or by topic and host). At step **365**, the message is passed to a Worker associated with the topic consumer **340**. If needed, at step **366**, a direct publisher **350** is instantiated to send a response message via the message system **300**. At step **367**, the direct publisher **340** sends a message to an exchange **310***a*. At step **368**, the response message is dispatched by the exchange **310***a*. At step **369**, the response message is fetched by the direct consumer **330** instantiated to receive the response and dictated by the routing key. At step **370**, the message response is passed to the Invoker.

[0061] Turning now to FIG. **3***c*, one embodiment of the process of sending an RPC broadcast message is shown relative to the elements of the message system **300** as described relative to FIG. **3***a*. All elements are as described above relative to FIG. **3***a* unless described otherwise. At step **580**, a topic publisher **520** is instantiated. At step **381**, the topic publisher **320** sends a message to an exchange **310***a*. At step **382**, the message is dispatched by the exchange **310***b*. At step **383**, the message is fetched by a topic consumer **340** dictated by the routing key (either by topic or by topic and host). At step **384**, the message is passed to a Worker associated with the topic consumer **340**.

[0062] In some embodiments, a response to an RPC broadcast message can be requested. In that case, the process follows the steps outlined relative to FIG. **3***b* to return a response to the Invoker. As the process of instantiating and launching a VM instance in FIG. **5** shows, requests to a distributed service or application may move through various software components, which may be running on one physical machine or may span across multiple machines and network boundaries.

[0063] Turning now to FIG. **4**, an IaaS-style computational cloud service (a "compute" service) is shown at **400** according to one embodiment. This is one embodiment of a cloud controller **135** with associated cloud service **140** as described relative to FIG. **1***b*. Except as described relative to specific embodiments, the existence of a compute service does not require or prohibit the existence of other portions of the cloud computing system **130** nor does it require or prohibit the existence of other cloud controllers **135** with other respective services **140**.

[0064] To the extent that some components described relative to the compute service **400** are similar to components of the larger cloud computing system **130**, those components may be shared between the cloud computing system **130** and a compute service **400**, or they may be completely separate. Further, to the extent that "controllers," "nodes," "servers," "managers," "VMs," or similar terms are described relative to the compute service **400**, those can be understood to comprise any of a single information processing device **210** as described relative to FIG. **2**, multiple information processing devices **210**, a single VM as described relative to FIG. **2**, a group or cluster of VMs or information processing devices as described relative to FIG. **3**. These may run on a single machine or a group of machines, but logically work together to provide the described function within the system.

[0065] In one embodiment, compute service **400** includes an API Server **410**, a Compute Controller **420**, an Auth Manager **430**, an Object Store **440**, a Volume Controller **450**, a Network Controller **460**, and a Compute Manager **470**. These components are coupled by a communications network of the type previously described. In one embodiment, communications between various components are message-oriented, using HTTP or a messaging protocol such as AMQP, ZeroMQ, or STOMP.

[0066] Although various components are described as "calling" each other or "sending" data or messages, one embodiment makes the communications or calls between components asynchronous with callbacks that get triggered when responses are received. This allows the system to be architected in a "shared-nothing" fashion. To achieve the shared-nothing property with multiple copies of the same

component, compute service **400** further includes distributed data store **490**. Global state for compute service **400** is written into this store using atomic transactions when required. Requests for system state are read out of this store. In some embodiments, results are cached within controllers for short periods of time to improve performance. In various embodiments, the distributed data store **490** can be the same as, or share the same implementation as Object Store **440**.

[0067] In one embodiment, the API server **410** includes external API endpoints **412**. In one embodiment, the external API endpoints **412** are provided over an RPC-style system, such as CORBA, DCE/COM, SOAP, or XML-RPC. These follow the calling structure and conventions defined in their respective standards. In another embodiment, the external API endpoints **412** are basic HTTP web services following a REST pattern and identifiable via URL. Requests to read a value from a resource are mapped to HTTP GETs, requests to create resources are mapped to HTTP PUTs, requests to update values associated with a resource are mapped to HTTP POSTs, and requests to delete resources are mapped to HTTP DELETEs. In some embodiments, other REST-style verbs are also available, such as the ones associated with WebDay. In a third embodiment, the API endpoints **412** are provided via internal function calls, IPC, or a shared memory mechanism. Regardless of how the API is presented, the external API endpoints **412** are used to handle authentication, authorization, and basic command and control functions using various API interfaces. In one embodiment, the same functionality is available via multiple APIs, including APIs associated with other cloud computing systems. This enables API compatibility with multiple existing tool sets created for interaction with offerings from other vendors.

[0068] The Compute Controller **420** coordinates the interaction of the various parts of the compute service **400**. In one embodiment, the various internal services that work together to provide the compute service **400**, are internally decoupled by adopting a service-oriented architecture (SOA). The Compute Controller **420** serves as an internal API server, allowing the various internal controllers, managers, and other components to request and consume services from the other components. In one embodiment, all messages pass through the Compute Controller **420**. In a second embodiment, the Compute Controller **420** brings up services and advertises service availability, but requests and responses go directly between the components making and serving the request. In a third embodiment, there is a hybrid model in which some services are requested through the Compute Controller **420**, but the responses are provided directly from one component to another.

[0069] In one embodiment, communication to and from the Compute Controller **420** is mediated via one or more internal API endpoints **422**, provided in a similar fashion to those discussed above. The internal API endpoints **422** differ from the external API endpoints **412** in that the internal API endpoints **422** advertise services only available within the overall compute service **400**, whereas the external API endpoints **412** advertise services available outside the compute service **400**. There may be one or more internal APIs **422** that correspond to external APIs **412**, but it is expected that there will be a greater number and variety of internal API calls available from the Compute Controller **420**.

[0070] In one embodiment, the Compute Controller **420** includes an instruction processor **424** for receiving and processing instructions associated with directing the compute service **400**. For example, in one embodiment, responding to an API call involves making a series of coordinated internal API calls to the various services available within the compute service **400**, and conditioning later API calls on the outcome or results of earlier API calls. The instruction processor **424** is the component within the Compute Controller **420** responsible for marshaling arguments, calling services, and making conditional decisions to respond appropriately to API calls.

[0071] In one embodiment, the instruction processor **424** is implemented as a tailored electrical circuit or as software instructions to be used in conjunction with a hardware processor to create a hardware-software combination that implements the specific functionality described herein. To the extent that one embodiment includes computer-executable instructions, those instructions may include software that is stored on a computer-readable medium. Further, one or more embodiments have associated with them a buffer. The buffer can take the form of data structures, a memory, a computer-readable medium, or an off-script-processor facility. For example, one embodiment uses a language runtime as an instruction processor **424**, running as a discrete operating environment, as a process in an active operating environment, or can be run from a low-power embedded processor. In a second embodiment, the instruction processor **424** takes the form of a series of interoperating but discrete components, some or all of which may be implemented as software programs. In another embodiment, the instruction processor **424** is a discrete component, using a small amount of flash and a low power processor, such as a low-power ARM processor. In a further embodiment, the instruction processor includes a rule engine as a submodule as described herein.

[0072] In one embodiment, the Compute Controller **420** includes a message queue as provided by message service **426**. In accordance with the service-oriented architecture described above, the various functions within the compute service **400** are isolated into discrete internal services that communicate with each other by passing data in a well-defined, shared format, or by coordinating an activity between two or more services. In one embodiment, this is done using a message queue as provided by message service **426**. The message service **426** brokers the interactions between the various services inside and outside the Compute Service **400**.

[0073] In one embodiment, the message service **426** is implemented similarly to the message service described relative to FIGS. **3*a*-3*c***. The message service **426** may use the message service **110** directly, with a set of unique exchanges, or may use a similarly configured but separate service.

[0074] The Auth Manager **430** provides services for authenticating and managing user, account, role, project, group, quota, and security group information for the compute service **400**. In a first embodiment, every call is necessarily associated with an authenticated and authorized entity within the system, and so is or can be checked before any action is taken. In another embodiment, internal messages are assumed to be authorized, but all messages originating from outside the service are suspect. In this embodiment, the Auth Manager checks the keys provided associated with each call received over external API endpoints **412** and terminates and/or logs any call that appears to come from an unauthenticated or unauthorized source. In a third embodiment, the Auth Manager **430** is also used for providing resource-specific information such as security groups, but the internal API calls for that information are assumed to be authorized. External

calls are still checked for proper authentication and authorization. Other schemes for authentication and authorization can be implemented by flagging certain API calls as needing verification by the Auth Manager **430**, and others as needing no verification.

[0075] In one embodiment, external communication to and from the Auth Manager **430** is mediated via one or more authentication and authorization API endpoints **632**, provided in a similar fashion to those discussed above. The authentication and authorization API endpoints **432** differ from the external API endpoints **612** in that the authentication and authorization API endpoints **432** are only used for managing users, resources, projects, groups, and rules associated with those entities, such as security groups, RBAC roles, etc. In another embodiment, the authentication and authorization API endpoints **432** are provided as a subset of external API endpoints **412**.

[0076] In one embodiment, the Auth Manager **430** includes rules processor **434** for processing the rules associated with the different portions of the compute service **400**. In one embodiment, this is implemented in a similar fashion to the instruction processor **424** described above.

[0077] The Object Store **440** provides redundant, scalable object storage capacity for arbitrary data used by other portions of the compute service **400**. At its simplest, the Object Store **440** can be implemented one or more block devices exported over the network. In a second embodiment, the Object Store **440** is implemented as a structured, and possibly distributed data organization system. Examples include relational database systems—both standalone and clustered—as well as non-relational structured data storage systems like MongoDB, Apache Cassandra, or Redis. In a third embodiment, the Object Store **440** is implemented as a redundant, eventually consistent, fully distributed data storage service.

[0078] In one embodiment, external communication to and from the Object Store **440** is mediated via one or more object storage API endpoints **442**, provided in a similar fashion to those discussed above. In one embodiment, the object storage API endpoints **442** are internal APIs only. In a second embodiment, the Object Store **440** is provided by a separate cloud service **130**, so the "internal" API used for compute service **400** is the same as the external API provided by the object storage service itself.

[0079] In one embodiment, the Object Store **440** includes an Image Service **444**. The Image Service **444** is a lookup and retrieval system for virtual machine images. In one embodiment, various virtual machine images can be associated with a unique project, group, user, or name and stored in the Object Store **440** under an appropriate key. In this fashion multiple different virtual machine image files can be provided and programmatically loaded by the compute service **400**.

[0080] The Volume Controller **450** coordinates the provision of block devices for use and attachment to virtual machines. In one embodiment, the Volume Controller **450** includes Volume Workers **452**. The Volume Workers **452** are implemented as unique virtual machines, processes, or threads of control that interact with one or more backend volume providers **454** to create, update, delete, manage, and attach one or more volumes **456** to a requesting VM.

[0081] In a first embodiment, the Volume Controller **450** is implemented using a SAN that provides a sharable, network-exported block device that is available to one or more VMs, using a network block protocol such as iSCSI. In this embodiment, the Volume Workers **452** interact with the SAN to manage and iSCSI storage to manage LVM-based instance volumes, stored on one or more smart disks or independent processing devices that act as volume providers **454** using their embedded storage **456**. In a second embodiment, disk volumes **456** are stored in the Object Store **440** as image files under appropriate keys. The Volume Controller **450** interacts with the Object Store **440** to retrieve a disk volume **456** and place it within an appropriate logical container on the same information processing system **440** that contains the requesting VM. An instruction processing module acting in concert with the instruction processor and hypervisor on the information processing system **240** acts as the volume provider **454**, managing, mounting, and unmounting the volume **456** on the requesting VM. In a further embodiment, the same volume **456** may be mounted on two or more VMs, and a block-level replication facility may be used to synchronize changes that occur in multiple places. In a third embodiment, the Volume Controller **450** acts as a block-device proxy for the Object Store **440**, and directly exports a view of one or more portions of the Object Store **440** as a volume. In this embodiment, the volumes are simply views onto portions of the Object Store **440**, and the Volume Workers **454** are part of the internal implementation of the Object Store **440**.

[0082] In one embodiment, the Network Controller **460** manages the networking resources for VM hosts managed by the compute manager **470**. Messages received by Network Controller **460** are interpreted and acted upon to create, update, and manage network resources for compute nodes within the compute service, such as allocating fixed IP addresses, configuring VLANs for projects or groups, or configuring networks for compute nodes.

[0083] In one embodiment, the Network Controller **460** may use a shared cloud controller directly, with a set of unique addresses, identifiers, and routing rules, or may use a similarly configured but separate service.

[0084] In one embodiment, the Compute Manager **470** manages computing instances for use by API users using the compute service **400**. In one embodiment, the Compute Manager **470** is coupled to a plurality of resource pools **472**, each of which includes one or more compute nodes **474**. Each compute node **474** is a virtual machine management system as described relative to FIG. **3** and includes a compute worker **476**, a module working in conjunction with the hypervisor and instruction processor to create, administer, and destroy multiple user- or system-defined logical containers and operating environments—VMs—according to requests received through the API. In various embodiments, the pools of compute nodes may be organized into clusters, such as clusters **476*a*** and **476*b***. In one embodiment, each resource pool **472** is physically located in one or more data centers in one or more different locations. In another embodiment, resource pools have different physical or software resources, such as different available hardware, higher-throughput network connections, or lower latency to a particular location.

[0085] In one embodiment, the Compute Manager **470** allocates VM images to particular compute nodes **474** via a Scheduler **478**. The Scheduler **478** is a matching service; requests for the creation of new VM instances come in and the most applicable Compute nodes **474** are selected from the pool of potential candidates. In one embodiment, the Scheduler **478** selects a compute node **474** using a random algorithm. Because the node is chosen randomly, the load on any particular node tends to be non-coupled and the load across all resource pools tends to stay relatively even.

[0086] In a second embodiment, a smart scheduler **478** is used. A smart scheduler analyzes the capabilities associated with a particular resource pool **472** and its component services to make informed decisions on where a new instance should be created. When making this decision it consults not only all the Compute nodes across the resource pools **472** until the ideal host is found.

[0087] In a third embodiment, a distributed scheduler **478** is used. A distributed scheduler is designed to coordinate the creation of instances across multiple compute services **400**. Not only does the distributed scheduler **478** analyze the capabilities associated with the resource pools **472** available to the current compute service **400**, it also recursively consults the schedulers of any linked compute services until the ideal host is found.

[0088] In one embodiment, either the smart scheduler or the distributed scheduler is implemented using a rules engine **479** (not shown) and a series of associated rules regarding costs and weights associated with desired compute node characteristics. When deciding where to place an Instance, rules engine **479** compares a Weighted Cost for each node. In one embodiment, the Weighting is just the sum of the total Costs. In a second embodiment, a Weighting is calculated using an exponential or polynomial algorithm. In the simplest embodiment, costs are nothing more than integers along a fixed scale, although costs can also be represented by floating point numbers, vectors, or matrices. Costs are computed by looking at the various Capabilities of the available node relative to the specifications of the Instance being requested. The costs are calculated so that a "good" match has lower cost than a "bad" match, where the relative goodness of a match is determined by how closely the available resources match the requested specifications.

[0089] In one embodiment, specifications can be hierarchical, and can include both hard and soft constraints. A hard constraint is a constraint is a constraint that cannot be violated and have an acceptable response. This can be implemented by having hard constraints be modeled as infinite-cost requirements. A soft constraint is a constraint that is preferable, but not required. Different soft constraints can have different weights, so that fulfilling one soft constraint may be more cost-effective than another. Further, constraints can take on a range of values, where a good match can be found where the available resource is close, but not identical, to the requested specification. Constraints may also be conditional, such that constraint A is a hard constraint or high-cost constraint if Constraint B is also fulfilled, but can be low-cost if Constraint C is fulfilled.

[0090] As implemented in one embodiment, the constraints are implemented as a series of rules with associated cost functions. These rules can be abstract, such as preferring nodes that don't already have an existing instance from the same project or group. Other constraints (hard or soft), may include: a node with available GPU hardware; a node with an available network connection over 100 Mbps; a node that can run Windows instances; a node in a particular geographic location, etc.

[0091] When evaluating the cost to place a VM instance on a particular node, the constraints are computed to select the group of possible nodes, and then a weight is computed for each available node and for each requested instance. This allows large requests to have dynamic weighting; if 1000 instances are requested, the consumed resources on each node are "virtually" depleted so the Cost can change accordingly.

[0092] Turning now to FIG. **5**, a diagram showing one embodiment of the process of instantiating and launching a VM instance is shown as diagram **500**. At time **502**, the API Server **510** receives a request to create and run an instance with the appropriate arguments. In one embodiment, this is done by using a command-line tool that issues arguments to the API server **510**. In a second embodiment, this is done by sending a message to the API Server **510**.

[0093] In one embodiment, the API to create and run the instance includes arguments specifying a resource type, a resource image, and control arguments. A further embodiment includes requester information and is signed and/or encrypted for security and privacy. At time **504**, API server **510** accepts the message, examines it for API compliance, and relays a message to Compute Controller **520**, including the information needed to service the request. In an embodiment in which user information accompanies the request, either explicitly or implicitly via a signing and/or encrypting key or certificate, the Compute Controller **520** sends a message to Auth Manager **530** to authenticate and authorize the request at time **506** and Auth Manager **530** sends back a response to Compute Controller **520** indicating whether the request is allowable at time **508**. If the request is allowable, a message is sent to the Compute Manager **570** to instantiate the requested resource at time **510**. At time **512**, the Compute Manager selects a Compute Worker **576** and sends a message to the selected Worker to instantiate the requested resource. At time **514**, Compute Worker identifies and interacts with Network Controller **560** to get a proper VLAN and IP address. At time **516**, the selected Worker **576** interacts with the Object Store **540** and/or the Image Service **544** to locate and retrieve an image corresponding to the requested resource. If requested via the API, or used in an embodiment in which configuration information is included on a mountable volume, the selected Worker interacts with the Volume Controller **550** at time **518** to locate and retrieve a volume for the to-be-instantiated resource. At time **519**, the selected Worker **576** uses the available virtualization infrastructure to instantiate the resource, mount any volumes, and perform appropriate configuration. At time **522**, selected Worker **556** interacts with Network Controller **560** to configure routing. At time **524**, a message is sent back to the Compute Controller **520** via the Compute Manager **550** indicating success and providing necessary operational details relating to the new resource. At time **526**, a message is sent back to the API Server **526** with the results of the operation as a whole. At time **599**, the API-specified response to the original command is provided from the API Server **510** back to the originally requesting entity. If at any time a requested operation cannot be performed, then an error is returned to the API Server at time **590** and the API-specified response to the original command is provided from the API server at time **592**. For example, an error can be returned if a request is not allowable at time **508**, if a VLAN cannot be created or an IP allocated at time **514**, if an image cannot be found or transferred at time **516**, etc. Such errors may be one potential source of mistakes or inconsistencies in periodic system status notifications discussed below.

[0094] Having described an example of a distributed application and operation within a distributed network system, various embodiments of methods and systems for verification of records of system change events in a distributed network system are described with references to FIGS. **6-11**. As used herein, a distributed network system may relate to one or more services and components, and in particular cloud ser-

vices. Various embodiments of the methods and systems disclosed herein may permit verification of records of system change events in a distributed network system providing cloud services.

[0095] FIG. **6** illustrates an environment **600** in which predictive monitoring may be useful. An example of environment **600** is cloud system **130** which may include many notifications **605** communicated between components of system **130** by message service **110**. Monitoring service **107** may be coupled to message service **110** and configured to observe the notifications **605** as system objects progress from one state to another within system **130**. Occasionally, monitoring service **107** may also observe that objects within system **130** may progress to a failure or other artifact of operation. Indeed, monitoring service **107** may observe many different artifacts **610-630** which may occur as a result of object being at one of various possible operational states. Advantageously, monitoring service **107** may gather information related to the operational states of the objects in system **130** and generate a predictive state model. Monitoring service **107** may generate predictions of one or more artifacts **610-630** in response to the various active states of the object. Further embodiments and details of implementation of monitoring service **107** are described below, with reference to FIGS. **7-11**.

[0096] FIG. **7** illustrates one embodiment of a monitoring service **107**. In one embodiment, monitoring service **107** includes an observer **705**, a state determination unit **710**, and a prediction engine **715**. One of ordinary skill in the art may recognize additional or alternative modules which may be adapted for use with the present embodiments of monitoring service **107**.

[0097] In one embodiment, observer **705** may be coupled to message service **110** and configured to observe notifications **605** communicated between components of system **130** regarding the state of one or more system objects. In various embodiments, observer **705** may include one or more switches or routers, implemented in either hardware or software, for direction and redirection of notifications **605**.

[0098] State determination module **710** may determine the state of an object in response to a received notification **605**. For example, state determination module **710** may include a database or other correlation of notification content and object states. A notification **605** may include, for example, an identifier associated with the object, a time the particular action associated with the notification **605** occurs, and a description of the action. Based upon this information, state determination module **710** may lookup the state of the object, or otherwise decipher the state of the object in response to the notification **605**.

[0099] Prediction engine **715** may include a predictive model of the object associated with the notification. For example, prediction engine **715** may generate and store model date in a database associated with the object identifier. In one embodiment, prediction **715** may track the state of the object in real-time, or near real-time, and then update the state of the object in response to each notification **605**. In a further embodiment, the prediction engine **715** may also update the predictive model to reflect the state information obtained from the notification **605** such that the predictive model is dynamic and reflects current conditions as well as historical state data. In still further embodiments, prediction engine **715** may use weights to give a certain preference for state data obtained within a first time period, while still giving consideration to state data obtained in a second time period. For example, more recent data may be assigned a relatively higher weight value than older state data.

[0100] FIG. **8** illustrates one embodiment of a method **800** for predictive object modeling which may be implemented by an embodiment of monitoring service **107**. In one embodiment, the method **800** starts when observer **705** observes a notification **605** communicated by message service **110** in system **130** as illustrated at block **805**. The notification **605** may be associated with a state of an object hosted or managed by system **130**.

[0101] State determination module **710** may determine the state of the object in response to the notification **605** as illustrated in block **810**. The prediction engine **715** may then reference a predictive object state model to predict occurrence of an artifact in response to the state of the object. For example, if the object is a VM to be built or updated on the system **130**, the notification **605** may be associated with an API request for setting up the VM. State determination module **710** may determine the state of the VM setup process in response to information in the notification **605**. For example, the notification **605** may be a disk attach notification, and based upon prior VM setup data, the state determination module **710** may determine which step or state of the VM setup process is associated with a disk attach operation.

[0102] Prediction engine **715** may then reference a predictive object state model to predict occurrence of an artifact **610-630**, such as a failure, in response to the state of the object as shown at block **815**. In a further embodiment, prediction engine may use additional information to generate the prediction. For example, prediction engine **715** may use timing information, such as a duration of time spent at each state, to enhance the prediction of an artifact.

[0103] By way of illustration, FIG. **9** illustrates an example of a process state flow **900** that may be associated with an object. In the illustrated example, the process state flow **900** starts at step **905** when an API request is received. The API request may include a request for a variety of operations. For example, a VM may be created or modified in response to an API request.

[0104] In the depicted example, one of three states may result from an API request **905**: the system **130** may return a 5XX response at state **910**, a server-side exception may be issued at state **920**, or a process may be started at state **915**. In the depicted example, the process is called process XYZ, and is a hypothetical process described merely as an example. If Process XYZ is started at state **915**, it may proceed to state 'X' **925** to perform an operation. State X **925** may result in one of two states, either state 'Y' **940** or state 'Z' **930**. If the process proceeds to state Z **930**, then the process will perform an operation and proceed to completion at state **935**. Alternatively, the process may move to state Y **940** to perform an operation. The operation at state Y **940** may yield one of three possible states: error state failure **945**, state 'F' **950** or completion at state **935**. If state F **950** is the result, another operation may be performed. The operation at state F **950** may yield one of two states: either a time threshold failure **955** or completion at state **935**.

[0105] In one embodiment, the state flow diagram **900** may be generated in response to observations or historical data related to the results of the API request at state **905**. In one embodiment, the state flow diagram **900** may be generated manually in response to process code, process design specifications, or observations of historical data. Alternatively, the

process flow diagram **900** may be generated automatically through observations of the procession of operations on system **130**.

[0106] FIG. **10** illustrates one embodiment of a prediction model **1000**. In the depicted embodiment, the left column represents the present state of process flow, and the corresponding row represents historical data associated with the number of times the one of a plurality of subsequent states resulted. In one embodiment, the model **1000** may be updated in response to the outcome of each state transition observed by monitoring service **107**. Thus, the model may be dynamic such that the accuracy of the model may improve over time as the number of state transitions recorded increases.

[0107] In the depicted example, the model **1000** may include data representing one hundred API requests received at state **905**. Of the one hundred API requests, one resulted in a 5XX response at the state transition **910**. Two API requests resulted in server-side exceptions at state **920**. Ninety-seven API requests resulted in process XYZ starting at state transition **915**. Similarly, the data represented in the model **1000** represents hypothetical state transitions during performance of process XYZ.

[0108] In one embodiment, the data in model **1000** may be used to determine predictions of the next state transition for each state in process flow **900**. For example, the model prediction model **1100** illustrates the probability of each state transition occurring within the entire process state flow **900** based on the date in model **1000** of FIG. **10**. In certain embodiments, the prediction model **1100** may be represented as a Markov chain.

[0109] In the described example, if the process XYZ starts at state **915**, then there is a 100% chance that the process with progress to state X **925** according to the data in the model. Accordingly there would initially be no need for concern from a system administration perspective. Upon completion of State X, however, the process may transition to one of two different states, state Y **940** or state Z **930**. There would still be no need for concern if the process proceeds to state Z **930**, because there is still a 100% probability that the process will complete at state **935** if transitioning from state Z **930**.

[0110] The prediction engine **715** may generate an alert if the process progresses to state Y **940**, however, because there is a substantial likelihood that one or more artifacts may occur. In this example, an error state failure **945** may be an artifact, and a time threshold failure **955** may also be an artifact. The prediction engine may, for example, generate an electronic communication to a system administrator indicated that process XYZ has entered state Y **940**. The prediction engine **715** may further include a probability of an artifact occurring in the communication. For example, at state Y, there is a 10% chance that an error state failure **945** will occur.

[0111] The prediction engine **715** may also generate an alert if the process transitions to state F **950** because there is a 40% likelihood that a time threshold failure **955** may occur as a result of the process transitioning to state F.

[0112] The following is an example of an API request and process XYZ which may proceed according to the model **1100** in some embodiments. This example is presented merely by way of illustration, and is not intended to be limiting.

[0113] In one embodiment, the observer **705** observes a notification **605** that system **130** has received an API request to attach a 50 GB disk to an instance of a VM. In this example, there is a 1% chance that a 5XX response is provided **910** and a 2% chance that a server-side exception **920** occurs, but there is a 97% chance that a disk attach process is launched at state **915**. At state **925** a disk initialization function occurs. According to the predictive model, there is a 69% chance that the disk will be successfully initialized and a configuration process may occur at **930** before returning a process complete notification at state **935**.

[0114] On the other hand, there is a 31% chance that the first disk initialization attempt will fail and a retry state **940** will occur. In one embodiment, the prediction engine may generate an alert to a system administrator, such as an electronic communication, indicating that the disk initialization failed its first attempt, and that a retry state **940** has been initiated. The alert may also indicate that there is a 10% chance that an error state failure **945** will occur. At that point, the system administrator may still have time to facilitate the retry at state **940** to avoid or mitigate error state failure **945**. If the retry is successful, then the disk may be configured at state **950**. In this example, there is a 40% chance that if a retry occurred at state **940**, then there will not be sufficient time for the configuration at state **950** and a time threshold failure **955** may result. Otherwise, the disk attaches successfully at state **935**. As mentioned above, this example is merely a non-limiting illustration of one embodiment, and any inaccuracies or simplifications in the description of a disk attach process are not intended to limit the scope of the claims in any way.

[0115] In one embodiment, predictive monitoring is implemented as an electrical circuit or as software instructions to be used in conjunction with a hardware processor to create a hardware-software combination that implements the specific functionality described herein. To the extent that one embodiment includes computer-executable instructions, those instructions may include software that is stored on a computer-readable medium. Further, one or more embodiments have associated with them a buffer. The buffer can take the form of data structures, a memory, a computer-readable medium, or an off-script-processor facility. For example, one embodiment uses a language runtime as an instruction processor, running as a discrete operating environment, as a process in an active operating environment, or can be run from a low-power embedded processor. In a second embodiment, the instruction processor takes the form of a series of interoperating but discrete components, some or all of which may be implemented as software programs. In another embodiment, the instruction processor is a discrete component, using a small amount of flash and a low power processor, such as a low-power ARM processor. In a further embodiment, the instruction processor includes a rule engine as a submodule as described herein.

[0116] Although illustrative embodiments have been shown and described, a wide range of modification, change and substitution is contemplated in the foregoing disclosure and in some instances, some features of the embodiments may be employed without a corresponding use of other features. Accordingly, it is appropriate that the appended claims be construed broadly and in a manner consistent with the scope of the embodiments disclosed herein.

What is claimed is:

1. A method of predictive monitoring of processes in a distributed network system providing cloud services, the method comprising:

observing one or more update messages sent and received among components of the distributed network system,

the update messages comprising information associated with a state of an object on the distributed network system;

determining the state of the object in response to the one or more update messages; and

referencing a predictive object state model to predict occurrence of an artifact in response to the state of the object.

2. The method of claim **1**, further comprising generating the predictive object state model in response to observations of one or more artifact occurrences associated with an object state.

3. The method of claim **2**, further comprising generating the predictive object state model in response to observations of one or more artifact occurrences associated with a sequence of sequential object states leading to the object state.

4. The method of claim **2**, wherein generating the predictive object state model comprises defining relationships between states and artifacts prior to observing the one or more updated messages.

5. The method of claim **2**, wherein generating the predictive object state model comprises dynamically defining relationships between states and artifacts in response to a plurality of update messages observed over time.

6. The method of claim **1**, further comprising updating the predictive object state model in response to the one or more update messages observed.

7. The method of claim **1**, further comprising assigning a probability that the one or more artifacts will occur in response to the predictive object state model and the state of the object.

8. The method of claim **1**, further comprising generating a notification in response to predicting the occurrence of the artifact.

9. A distributed network system comprising:

a plurality of service components;

an Application Program Interface (API) configured to receive a request for a system state change; and

an monitoring service component configured to:

observe one or more update messages sent and received among components of the distributed network system, the update messages comprising information associated with a state of an object on the distributed network system;

determine the state of the object in response to the one or more update messages; and

reference a predictive object state model to predict occurrence of an artifact in response to the state of the object.

10. The distributed network system of claim **9**, wherein the monitoring service is further configured to generate the predictive object state model in response to observations of one or more artifact occurrences associated with an object state.

11. The distributed network system of claim **10**, wherein the monitoring service is further configured to generate the predictive object state model in response to observations of one or more artifact occurrences associated with a sequence of sequential object states leading to the object state.

12. The distributed network system of claim **10**, wherein generating the predictive object state model comprises defining relationships between states and artifacts prior to observing the one or more updated messages.

13. The distributed network system of claim **10**, wherein generating the predictive object state model comprises dynamically defining relationships between states and artifacts in response to a plurality of update messages observed over time.

14. The distributed network system of claim **9**, wherein the monitoring service is further configured to update the predictive object state model in response to the one or more update messages observed.

15. The distributed network system of claim **9**, wherein the monitoring service is further configured to assign a probability that the one or more artifacts will occur in response to the predictive object state model and the state of the object.

16. The distributed network system of claim **9**, wherein the monitoring service is further configured to generate a notification in response to predicting the occurrence of the artifact.

17. A non-transitory computer-accessible storage medium storing program instructions that, when executed by a data processing device, cause the data processing device to implement operations for failure monitoring in a distributed network system providing cloud services, the operations comprising:

observe one or more update messages sent and received among components of the distributed network system, the update messages comprising information associated with a state of an object on the distributed network system;

determine the state of the object in response to the one or more update messages; and

reference a predictive object state model to predict occurrence of an artifact in response to the state of the object.

18. The computer-accessible storage medium of claim **17**, further comprising generating the predictive object state model in response to observations of one or more artifact occurrences associated with an object state.

19. The computer-accessible storage medium of claim **18**, further comprising generating the predictive object state model in response to observations of one or more artifact occurrences associated with a sequence of sequential object states leading to the object state.

20. The computer-accessible storage medium of claim **18**, wherein generating the predictive object state model comprises defining relationships between states and artifacts prior to observing the one or more updated messages.

21. The computer-accessible storage medium of claim **18**, wherein generating the predictive object state model comprises dynamically defining relationships between states and artifacts in response to a plurality of update messages observed over time.

22. The computer-accessible storage medium of claim **17**, further comprising updating the predictive object state model in response to the one or more update messages observed.

23. The computer-accessible storage medium of claim **17**, further comprising assigning a probability that the one or more artifacts will occur in response to the predictive object state model and the state of the object.

24. The computer-accessible storage medium of claim **17**, further comprising generating a notification in response to predicting the occurrence of the artifact.

* * * * *