(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
12 November 2020 (12.11.2020)

WIPO | PCT

(10) International Publication Number
WO 2020/226659 A1

(51) International Patent Classification:
G06F 9/50 (2006.01) *G06F 9/445* (2018.01)
H04L 29/08 (2006.01) *H04L 12/24* (2006.01)

(21) International Application Number:
PCT/US2019/031570

(22) International Filing Date:
09 May 2019 (09.05.2019)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant: **HUAWEI TECHNOLOGIES CO., LTD.** [CN/CN]; Huawei Administration Building, Bantian, Shenzhen, GUANDONG, 518129 (CN).

(72) Inventors: **LING, Wei**; 5876 Castano Drive, San Jose, California 95129 (US). **CHUO, Wei Chung**; 473 Barkentine Lane, Redwood City, California 94065 (US). **PHAM, Peter M.**; 3698 Lucas Drive, San Jose, California 95148 (US). **DAVISON, Lawrence**; 5645 Crow Lane, San Jose, California 95123 (US).

(74) Agent: **SCHEER, Bradley W.** et al.; Schwegman, Lundberg & Woessner, P.A., P. O. Box 2938, Minneapolis, Minnesota 55402 (US).

(81) Designated States *(unless otherwise indicated, for every kind of national protection available)*: AE, AG, AL, AM, AO, AT, AU, AZ, BA, BB, BG, BH, BN, BR, BW, BY, BZ, CA, CH, CL, CN, CO, CR, CU, CZ, DE, DJ, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JO, JP, KE, KG, KH, KN, KP, KR, KW, KZ, LA, LC, LK, LR, LS, LU, LY, MA, MD, ME, MG, MK, MN, MW, MX, MY, MZ, NA, NG, NI, NO, NZ, OM, PA, PE, PG, PH, PL, PT, QA, RO, RS, RU, RW, SA,

(54) Title: FAAS WARM STARTUP AND SCHEDULING

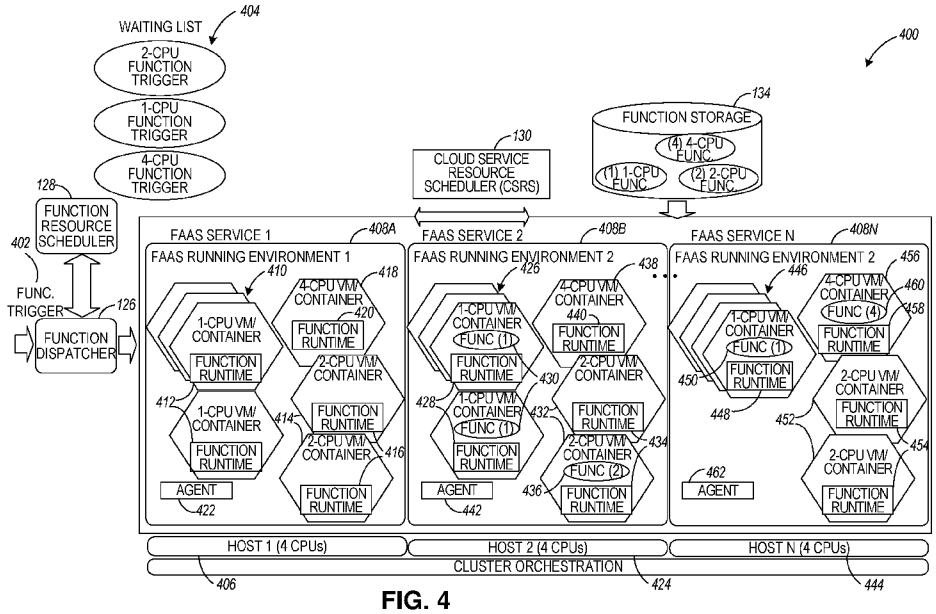


FIG. 4

(57) Abstract: Scheduling computing resources for executing functions within a FaaS service of a network architecture is disclosed, independent of FaaS service scheduling. The FaaS service is generated on a host and includes an agent configured to monitor computing resource usage and instantiate a container pool. The container pool includes a plurality of containers running within the FaaS service and using computing resources of the host, each container of the plurality of containers having a pre-determined resource size. A function trigger event for triggering a function is detected and computing resource requirements associated with the function are determining. A container is selected from the plurality of containers based on the computing resource requirements and computing resources available at the FaaS service as indicated by the computing resource usage. The function trigger event is transmitted to the selected container for executing function code of the function within the selected container.

WO 2020/226659 A1

[Continued on next page]

SC, SD, SE, SG, SK, SL, SM, ST, SV, SY, TH, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VC, VN, ZA, ZM, ZW.

(84) **Designated States** *(unless otherwise indicated, for every kind of regional protection available)*: ARIPO (BW, GH, GM, KE, LR, LS, MW, MZ, NA, RW, SD, SL, ST, SZ, TZ, UG, ZM, ZW), Eurasian (AM, AZ, BY, KG, KZ, RU, TJ, TM), European (AL, AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HR, HU, IE, IS, IT, LT, LU, LV, MC, MK, MT, NL, NO, PL, PT, RO, RS, SE, SI, SK, SM, TR), OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, KM, ML, MR, NE, SN, TD, TG).

Declarations under Rule 4.17:

— *of inventorship (Rule 4.17(iv))*

Published:

— *with international search report (Art. 21(3))*

FAAS WARM STARTUP AND SCHEDULING

TECHNICAL FIELD

[0001] The present disclosure is related to serverless computing in network-based service infrastructure. Some aspects relate to function as a service (FaaS) warm startup and scheduling techniques.

BACKGROUND

[0002] Serverless computing is associated with a network architecture where a user of the network architecture relies on network-based servers, network infrastructure, and operating systems and functions provided as a service and managed by a network provider. The user of the network architecture can take advantage of serverless computing by creating, managing, and deploying applications that can scale on demand and using computing resources managed by the network provider.

[0003] In serverless computing, the compute platform automatically manages and creates the underlying compute resources to host and run the function code of a serverless function. One of the challenges in serverless computing is reducing the startup latency of services offered by the network provider, including reducing FaaS startup latency.

[0004] FaaS enables a software program (e.g., a stateless function), to run on a cloud platform with a trigger-action-based execution model. Unlike traditional long lasting micro-services (e.g., as provided by the above-mentioned serverless compute platforms), the life cycle of a FaaS function is considered short-lived, (e.g., 5 minutes if there is no new trigger). However, due to security considerations in a multi-tenant cloud environment, a separate host with a separate container or a virtual machine is needed to isolate functions from each other and from the host running environment. This results in function startup overhead of 3-10 seconds (or even more), which greatly reduces FaaS use cases, particularly when quick reactions are needed.

SUMMARY

[0005] Various examples are now described to introduce a selection of concepts in a simplified form that is further described below in the detailed description. The Summary is not intended to identify key or essential features of the claimed subject matter, nor is it intended to be used to limit the scope of the claimed subject matter.

[0006] According to a first aspect of the present disclosure, there is provided a method for scheduling computing resources for executing functions. The method includes generating, by a function scheduling device, a Function as a Service (FaaS) running environment (or service) on a host. The FaaS service includes a FaaS service agent configured to monitor computing resource usage within the FaaS service and instantiate a virtual machine (VM)/container pool. The VM/container pool includes a plurality of VMs/containers running within the FaaS service and using computing resources of the host, each VM/container of the plurality of VMs/containers having a pre-determined resource size. A function trigger event for triggering a function is detected by the function scheduling device. Computing resource requirements associated with the function are determined by the function scheduling device. A VM/container is selected, by the function scheduling device, from the plurality of VMs/containers based on the computing resource requirements associated with the function and computing resources available at the FaaS service as indicated by the computing resource usage. The function trigger event is transmitted, by the function scheduling device, to the selected VM/container for executing function code of the function within the selected VM/container.

[0007] Generating a standby VM/container pool with sufficient (i.e., more than currently needed) number of VM/containers of different resource subscriptions (or differently sized VM/containers), results in increased VM/container size choices as well as reduced resource segmentation and resource partition issues. Additionally, using the agent to monitor (and report) computing resource usage within the FaaS running environment (separately from a cloud orchestration scheme), results in more efficient VM/container selection (for function execution) and VM/container pool management in each FaaS running environment.

[0008] In a first implementation form of the method according to the first aspect as such, information indicative of the computing resource usage within the FaaS service is received by the function scheduling device. At least one additional FaaS service is generated, by the function scheduling device, based on the information indicative of the computing resource usage, on at least one corresponding additional host within the network architecture. The at least one additional FaaS service includes a second VM/container pool with a plurality of additional VMs/containers.

[0009] In a second implementation form of the method according to the first aspect as such or any preceding implementation form of the first aspect, the selecting of the VM/container is further based on selecting, by the function scheduling device, the VM/container from the plurality of VMs/containers of the VM/container pool and the plurality of additional VMs/containers instantiated within the at least one additional FaaS service of the second VM/container pool, based on the computing resource requirements of the function, the pre-determined resource size for each VM/container of the plurality of VMs/containers, and a pre-determined resource size for each VM/container of the plurality of additional VMs/containers.

[0010] In a third implementation form of the method according to the first aspect as such or any preceding implementation form of the first aspect, a FaaS service of the at least one additional FaaS service having available computing resources to satisfy the computing resource requirements associated with the function is determined. An IP address of the determined FaaS service is obtained. The function trigger event is forwarded (i.e., transmitted) to the determined FaaS service using the IP address. A VM/container is selected of the plurality of additional VMs/containers within the determined FaaS service that matches the computing resource requirements of the function. The function code is retrieved from a function code repository for mounting within the selected VM/container. The function code is executed within the selected VM/container after the mounting.

[0011] In a fourth implementation form of the method according to the first aspect as such or any preceding implementation form of the first aspect, usage of the computing resources by the plurality of VMs/containers within the

host is monitored. A usage value indicative of the computing resources usage and remaining resources within the host is determined. The usage value is compared to a threshold. Generation of an additional FaaS service is triggered when the usage value exceeds the threshold.

[0012] In a fifth implementation form of the method according to the first aspect as such or any preceding implementation form of the first aspect, information identifying the function associated with the function trigger event and the selected VM/container is cached. Upon receiving a new function trigger event associated with the function, the cached information is retrieved. The new function trigger event is forwarded to the selected VM/container.

[0013] In a sixth implementation form of the method according to the first aspect as such or any preceding implementation form of the first aspect, the selecting of the VM/container from the plurality of VMs/containers includes determining a likelihood operator of the function for each computing resource of the computing resources. The likelihood operator indicates whether a corresponding computing resource value of the computing resource satisfies the computing resource requirements associated with the function. For each VM/container of the plurality of VMs/containers of the VM/container pool, a capability parameter of the function is determined based on the likelihood operator for each of the computing resources.

[0014] In a seventh implementation form of the method according to the first aspect as such or any preceding implementation form of the first aspect, the selecting the VM/container further includes selecting the VM/container based on capability parameters of the plurality of VMs/containers.

[0015] In an eighth implementation form of the method according to the first aspect as such or any preceding implementation form of the first aspect, the selecting the VM/container further includes selecting the VM/container when a corresponding capability parameter of the VM/container is above a threshold value.

[0016] In a ninth implementation form of the method according to the first aspect as such or any preceding implementation form of the first aspect, a validity operator indicative of one or more VM/containers of the plurality of

VM/containers that are still available for function assignment is determined, after the selecting of the VM/container.

[0017] According to a second aspect of the present disclosure, there is provided a system including a memory that stores instructions and one or more processors in communication with the memory. The one or more processors execute the instructions to generate a Function as a Service (FaaS) running environment (or FaaS service) on a host within a network architecture. The FaaS service includes a FaaS service agent configured to monitor computing resource usage within the FaaS service and instantiate a VM/container pool. The VM/container pool includes a plurality of VMs/containers running within the FaaS service and uses computing resources of the host, each VM/container of the plurality of VMs/containers having a pre-determined resource size. A function trigger event for triggering a function is detected. Computing resource requirements associated with the function are determined. A VM/container is selected from the plurality of VMs/containers based on the computing resource requirements associated with the function and computing resources available at the FaaS service as indicated by the computing resource usage. The function trigger event is transmitted to the selected VM/container for executing function code of the function within the selected VM/container.

[0018] In a first implementation form of the system according to the second aspect as such, the one or more processors execute the instructions to determine an IP address of the selected VM/container, forward the function trigger event to the selected VM/container using the IP address, retrieve the function code from a function code storage for mounting within the selected VM/container, and execute the function code within the selected VM/container after the mounting.

[0019] In a second implementation form of the system according to the second aspect as such or any preceding implementation form of the second aspect, the one or more processors execute the instructions to monitor usage of the computing resources by the plurality of VM/containers within the host, determine a usage value indicative of the computing resources usage, compare the usage value to a threshold, and trigger generation of an additional FaaS service when the usage value exceeds the threshold.

[0020] In a third implementation form of the system according to the second aspect as such or any preceding implementation form of the second aspect, for selecting the VM/container from the plurality of VM/containers, the one or more processors execute the instructions to determine a likelihood operator of the function for each of the computing resources. The likelihood operator indicates whether a corresponding computing resource of the computing resources satisfies the computing resource requirements associated with the function.

[0021] In a fourth implementation form of the system according to the second aspect as such or any preceding implementation form of the second aspect, the one or more processors execute the instructions to determine for each VM/container of the plurality of VMs/containers of the VM/container pool, a capability parameter of the function based on the likelihood operator for each of the computing resources.

[0022] In a fifth implementation form of the system according to the second aspect as such or any preceding implementation form of the second aspect, for the selecting of the VM/container from the plurality of VMs/containers, the one or more processors execute the instructions to select the VM/container based on capability parameters of the plurality of VMs/containers.

[0023] In a sixth implementation form of the system according to the second aspect as such or any preceding implementation form of the second aspect, for the selecting of the VM/container from the plurality of VMs/containers, the one or more processors execute the instructions to select the VM/container when a corresponding capability parameter of the VM/container is above a threshold value.

[0024] According to a third aspect of the present disclosure, there is provided a non-transitory computer-readable medium storing instruction for scheduling computing resources for executing functions within a Function as a Service (FaaS) service of a network architecture, that when executed by one or more processors, cause the one or more processors to perform operations. The operations include generating the FaaS service on a host. The FaaS service includes a FaaS service agent configured to monitor computing resource usage within the FaaS service and instantiate a VM/container pool. The VM/container

pool includes a plurality of VMs/containers running within the FaaS service and using computing resources of the host, each VM/container of the plurality of VMs/containers having a pre-determined resource size. A function trigger event for triggering a function is detected. Computing resource requirements associated with the function are determined. A VM/container is selected from the plurality of VMs/containers based on the computing resource requirements associated with the function and computing resources available at the FaaS service as indicated by the computing resource usage. The function trigger event is transmitted to the selected VM/container for executing function code of the function within the selected VM/container.

[0025] In a first implementation form of the non-transitory computer-readable medium according to the third aspect as such, where upon execution, the instructions further cause the one or more processors to perform operations including receiving information indicative of the computing resource usage within the FaaS service. At least one additional FaaS service is generated on at least one corresponding additional host within the network architecture. The at least one additional FaaS service includes a second VM/container pool with a plurality of additional VMs/containers, the generating based on the information indicative of the computing resource usage.

[0026] In a second implementation form of the non-transitory computer-readable medium according to the third aspect as such, where upon execution, the instructions further cause the one or more processors to perform operations including selecting the VM/container from the plurality of VMs/containers of the VM/container pool and the plurality of additional VMs/containers instantiated within the second VM/container pool, based on the computing resource requirements of the function and the resource size for each VM/container of the plurality of VMs/containers and each VM/container of the plurality of additional VMs/containers.

[0027] Anyone of the foregoing examples may be combined with any one or more of the other foregoing examples to create a new embodiment within the scope of the present disclosure.

BRIEF DESCRIPTION OF THE DRAWINGS

[0028] In the drawings, which are not necessarily drawn to scale, like numerals may describe similar components in different views. The drawings illustrate generally, by way of example, but not by way of limitation, various embodiments discussed in the present document.

[0029] FIG. 1 is a high-level system overview of a network architecture using a function scheduling device, according to some example embodiments.

[0030] FIG. 2 is a block diagram illustrating a single-level function scheduling in a network architecture, according to some example embodiments.

[0031] FIG. 3 is a block diagram illustrating a new VM/container assignment for each host and function loading using a function resource scheduler in a network architecture, according to some example embodiments.

[0032] FIG. 4 is a block diagram illustrating a warm container/virtual machine (VM) management and function assignment using two-level scheduling in a network architecture, according to some example embodiments.

[0033] FIG. 5 illustrates graphs of example likelihood calculations associated with different computing resources, according to some example embodiments.

[0034] FIG. 6 is a block diagram illustrating an example processing flow for function scheduling and VM/container selection in a network architecture, according to some example embodiments.

[0035] FIG. 7 is a flowchart of a method suitable for executing functions within a FaaS running environment of a network architecture, according to some example embodiments.

[0036] FIG. 8 is a block diagram illustrating a representative software architecture, which may be used in conjunction with various device hardware described herein, according to some example embodiments.

[0037] FIG. 9 is a block diagram illustrating circuitry for a device that implements algorithms and performs methods, according to some example embodiments.

DETAILED DESCRIPTION

[0038] It should be understood at the outset that although an illustrative implementation of one or more embodiments is provided below, the disclosed systems and/or methods described with respect to FIGS. 1-9 may be implemented using any number of techniques, whether currently known or not yet in existence. The disclosure should in no way be limited to the illustrative implementations, drawings, and techniques illustrated below, including the exemplary designs and implementations illustrated and described herein, but may be modified within the scope of the appended claims along with their full scope of equivalents.

[0039] In the following description, reference is made to the accompanying drawings that form a part hereof, and in which are shown, by way of illustration, specific embodiments which may be practiced. These embodiments are described in sufficient detail to enable those skilled in the art to practice the inventive subject matter, and it is to be understood that other embodiments may be utilized, and that structural, logical, and electrical changes may be made without departing from the scope of the present disclosure. The following description of example embodiments is, therefore, not to be taken in a limiting sense, and the scope of the present disclosure is defined by the appended claims.

[0040] As used herein, the term "network-based service infrastructure" includes a plurality of network devices (also referred to as hosts, nodes, or servers) providing on-demand computing capacity (e.g., via one or more virtual machines or other virtual resources running on the network devices) and storage capacity as a service to a community of end-recipients (e.g., customers of the service infrastructure), where the end recipients are communicatively coupled to the network devices within the service infrastructure via a network. The customers of the service infrastructure can use one or more computing devices (or customer devices) to access and manage the services (including any FaaS) provided by the service infrastructure via the network. The customer devices, the network, and the network-based service infrastructure can be collectively referred to as a "network architecture." The customers of the service infrastructure can also be referred to as "users."

[0041] As used herein, the term "function trigger" (or "function trigger event") refers to a function invocation request (e.g., as communicated by a customer device and received at the service infrastructure) for executing a function on a virtual machine (VM) or a container (e.g., a warm container, as defined below, running in a virtual private cloud or virtual private cloud (VPC) of a customer) in connection with a FaaS provided by the network-based service infrastructure.

[0042] As used herein, the term "cold container" refers to an application container which is instantiated (i.e., started on an instance such as a virtual machine or a virtual private cloud running on a computing device) when a trigger event that uses functionalities of an application running on the container takes place. When the cold container is instantiated for a given application in response to the trigger even taking place, the container includes both function code associated with functionalities performed by the application as well as the function runtime (e.g., the runtime language libraries and binaries for the function code). In some examples, latency for cold containers is in the range of 10 seconds.

[0043] As used herein, the term "hot container" refers to an application container which is up and running with the user function code and its function runtime before the event trigger arrives. The hot containers can remain active for a certain time and then released after the function code execution completes. The hot containers may use a significant amount of wasted resources as the container with the function code and runtime library is active even when it has not been requested and it is not needed.

[0044] As used herein, the term "warm container" refers to an application container which can be instantiated and is running only with a function runtime before the trigger even arrives. The function runtime can include a runtime library for a programming language, such as Java Runtime library, Go Runtime library, Python Runtime library, and so forth. The warm containers do not have any function code when instantiated. The user function code can be dynamically mounted to the warm container when the trigger event arrives and unmounted from the warm container when the function (or application) associated with the mounted function (or application) code of the

FaaS completes execution. In this regard, a warm container can run a User A's function at one time and run a User B's function at another time.

[0045] As used herein, the term "resource subscription" refers to a maximum size/amount of a computing resource that can be used by a virtual machine or a container. As used herein, the term "resource usage" is synonymous with "computing resource usage" and indicates the computing resources that are being utilized by a virtual machine or a container within a FaaS service. A computing resource can include one or more of the following resources of a host: central processing unit (CPU) resources, graphical processing unit (GPU) resources, memory resources, and other host resources. Additionally, computing resource usage can be monitored and can change dynamically, or it can be adjusted dynamically.

[0046] As used herein, the term "x-CPU container" (where x is an integer greater than or equal to 1) indicates an application container with maximum resource usage of x number of CPUs when a function is executed inside the container. A host resource other than CPU (e.g., GPU, memory, and so forth) can be used in a similar fashion when describing a container (e.g., y-GPU container, and so forth). Additionally, the term "x-CPU container" further indicates that the container subscribes to the usage of a certain resources (e.g., x number of CPUs) or the container is of a "size" (or a "resource size") of x CPUs, or the container has a resource subscription of x number of CPUs. In this regard, the term "differently sized containers" refers to a plurality of containers that are of different sizes (i.e., the containers subscribe to resources of different sizes).

[0047] As used herein, the term "virtual machine" (or VM) is used interchangeably with the term "container" in connection with executing function code associated with FaaS provided by a network-based service architecture. More specifically, function code and function runtime can be hosted at (and executed from) a container (e.g., a warm container) or a VM instantiated on a host device within the service architecture.

[0048] As used herein, the term "VM/container pool" refers to a plurality of VMs, a plurality of containers, or a plurality of VMs and containers.

[0049] Prior art serverless computing solutions use cold containers (containers started when an event occurs) or hot containers (containers already running with user's code). However, both of these approaches have drawbacks. For example, cold containers have a large startup time whilst hot containers cannot scale.

[0050] In order to reduce FaaS function startup latency, some FaaS providers use warm startup container approach (e.g., starting a function container or virtual machine of variant size as a service on a separate network node prior to loading the function) using one-level scheduling by a cloud service orchestration scheme. This way, the container or virtual machine startup time is not counted in the function startup time, which reduces the function startup time to sub-second ranges. However, due to the limitation of cloud infrastructure and the cloud service orchestration scheme, the one-level scheduling results in creating warm containers where each container is using actual host resources. Consequently, only a limited number/combination of warm containers can be created via one-level scheduling since each created container is occupying (reserving) actual host resources, regardless of whether the container is empty or not. In this regard, since one-level scheduling results in creating containers that occupy host resources, only a limited number of containers that subscribe up to the maximum host resource capacity can be created (e.g., a host with 4 CPUs can only include containers that subscribe to a total of 4 CPUs).

[0051] Additionally, once a function is dispatched to the FaaS service and is running inside the VM/container of the service, it can cause resource segmentation, resource under-subscription, or resource oversubscription. For example, one-level scheduling can result in resource segmentation when one or more smaller capacity containers are created (e.g., 1-CPU containers) along with larger capacity containers (e.g., 2-CPU containers), and all incoming functions only require the use of the larger capacity containers, leaving the smaller capacity containers unutilized (but still taking host resources). One-level scheduling techniques can create multiple containers in a host in an oversubscription fashion (e.g., create containers that do not explicitly reserve any host resources but only specify the upper limit resource usage at the host in order to prevent possible segmentation issues). However, containers created via one-

level scheduling are visible to (and can be used by) the cloud service orchestration scheme, which may result in overutilization of the host resources (i.e., resource oversubscription) and moving of containers to other hosts. As a result, expensive and interruptive resource re-distribution and function migration may be necessary with one-level scheduling techniques.

[0052] Two-level scheduling techniques disclosed herein can be used to improve the function startup time as well as resource partitioning and resource use efficiency associated with using the warm startup approach for executing functions in a FaaS running environment. More specifically, techniques disclosed herein use a 2-level resource scheduling of resources for executing functions in a FaaS running environment.

[0053] As a first level of resource scheduling, a Cloud Service Resource Scheduler (CSRS) is used to load a FaaS running environment (or a FaaS service) that contains a pool of empty virtual machines (VMs)/containers of different sizes, which are not visible by the CSRS, on a physical server/node. The FaaS service reserves all the resources on the server, but the service does not actually use these resources if there is no function running in it. In this regard, the first level of resource scheduling can also be referred to as cloud-level resource scheduling. As used herein, the terms FaaS service and a FaaS running environment are used interchangeably.

[0054] A second level of resource scheduling uses a function dispatcher (FD), that receives a function trigger event, and a function resource scheduler (FRS) that is configured to perform function scheduling and select an optimal FaaS service from a plurality of available FaaS services, or a VM/container from a plurality of available containers in the plurality of available FaaS services for executing the function associated with the received function trigger event. The second level of resource scheduling can also be referred to as function-level resource scheduling.

[0055] After a FaaS service (or a container within a FaaS service) is selected, the function trigger event is communicated to the FaaS service or container (e.g., transmitted using an IP address of the service or the container). A function (e.g., function code) corresponding to the function trigger event is loaded (e.g., from function code storage) to be executed in the VM/container

based on the function trigger event. In order to host functions with different resource usage requirements, a sufficient number (or more than currently needed) of VMs/containers (e.g., warm containers) of different resource subscriptions (sizes) may be pre-created in the container pool within the FaaS service running on each node (e.g., to prevent any warm container shortage during function level scheduling). As a result, total resource subscription by these VMs/containers exceeds available physical resources on a server if all the VM/containers are loaded (creating a potential resource "oversubscription"). However, when the VMs/containers are not loaded with the function code of functions, the containers consume minimal host resources, such as CPU resources, GPU resources, and memory resources. In this regard, VMs/containers created via two-level scheduling are not visible to first level resource scheduler (e.g., CSRS), and the CSRS does not interfere with overall resource partition. Therefore, oversubscription of the node resources is acceptable and can be used to benefit resource optimization when function scheduling happens.

[0056] In comparison to existing solutions using a single level scheduling where a cloud service orchestration scheme creates warm containers where each container is using/reserving actual host resources, disclosed techniques use two-level scheduling with the following distinguishing features and benefits: (a) static standby VM/container pool with sufficient (i.e., more than currently needed) number of VMs or containers of different resource subscriptions (or differently sized VMs or containers); VMs/containers in each FaaS running environment using minimal cloud resources, resulting in increased VM/container size choices as well as reduced resource segmentation and resource partition issue; (b) the two-level scheduling allows for over-subscription of physical (e.g., CPU and memory) resources of each host in a planned manner, making cloud resource usage more efficient; (c) decoupled two-level scheduling (cloud service level and function level) can be used, with the function level scheduling being responsible for handling function and VM/container binding, monitoring available resources, determining valid (useable) VM/containers, and making proper over-subscription decisions, and the cloud service level scheduling being responsible for FaaS running environment-to-physical server binding and scaling (and not being responsible

for creating containers within the FaaS running environment); and (d) distributed VM/container pool management in each FaaS running environment to increase system resilience.

[0057] FIG. 1 is a high-level system overview of a network architecture 100 using a serverless function container VPC, according to some example embodiments. Referring to FIG. 1, the network architecture 100 can include a plurality of devices (e.g., user devices) 102A, …, 102N (collectively, devices 102) communicatively coupled to a network-based service infrastructure 114 via a network 112. The devices 102A, …, 102N are associated with corresponding users 106A, …, 106N and can be configured to interact with the network-based service infrastructure 114 using a network access client, such as one of clients 104A, …, 104N. The network access clients 104A, …, 104N can be implemented as web clients or application (app) clients.

[0058] Users 106A, …, 106N may be referred to generically as "a user 106" or collectively as "users 106." Each user 106 may be a human user (e.g., a human being), a machine user (e.g., a computer configured by a software program to interact with the devices 102 and the network-based service infrastructure 114), or any suitable combination thereof (e.g., a human assisted by a machine or a machine supervised by a human). The users 106 are not part of the network architecture 100 but are each associated with one or more of the devices 102 and may be users of the devices 102 (e.g., the user 106A may be an owner of the device 102A, and the user 106N may be an owner of the device 102N). For example, the device 102A may be a desktop computer, a vehicle computer, a tablet computer, a navigational device, a portable media device, or a smartphone belonging to the user 106A. Users 106A, …, 106N can use devices 102A, …, 102N to access services (e.g., serverless computing services or FaaS) provided by the network-based service infrastructure 114. In this regard, users 106 can also be referred to as "customers 106" or "tenants 106" of the network-based service infrastructure 114. The serverless computing services can include instantiating and using, e.g., virtual machines (VMs), virtual private clouds (VPCs), and application containers in connection with FaaS-related functionalities.

[0059] The network-based service infrastructure 114 can include a plurality of computing devices 116, …, 118. One or more of the computing devices within the network-based service infrastructure 114 (e.g., the computing device 116) can include a FaaS service 120. The FaaS service 120 can use an agent 121 to instantiate one or more containers (such as warm containers 122, …, 124), virtual machines, or other computing resources in connection with providing FaaS-related functionalities to user 106. The agent 121 is also configured to monitor the usage of the computing resources (e.g., how many of the containers or VMs within the FaaS service are occupied and what is the resource subscription of such occupied containers), and communicate the resource usage information to one or more network entities outside of the FaaS service 120.

[0060] Any of the devices shown in FIG. 1 may be implemented in a general-purpose computer modified (e.g., configured or programmed) by software to be a special-purpose computer to perform the functions described herein for that machine, database, or device. As used herein, a "database" is a data storage resource that stores data structured as a text file, a table, a spreadsheet, a relational database (e.g., an object-relational database, a NoSQL database, a network or graph database), a triple store, a hierarchical data store, or any suitable combination thereof. Additionally, data accessed (or stored) via an application programming interface (API) or remote procedure call (RPC) may be considered to be accessed from (or stored to) a database. Moreover, any two or more of the devices or databases illustrated in FIG. 1 may be combined into a single machine, database, or device, and the functions described herein for any single machine, database, or device may be subdivided among multiple machines, databases, or devices.

[0061] The network 112 may be any network that enables communication between or among machines, databases, and devices (e.g., devices 102A, …, 102N and devices 116, …, 118 within the network-based service infrastructure 114). Accordingly, the network 112 may be a wired network, a wireless network (e.g., a mobile or a cellular network), or any suitable combination thereof. The network 112 may include one or more

portions that constitute a private network, a public network (e.g., the Internet), or any suitable combination thereof.

[0062] In some aspects, one or more of the users 106 can communicate function triggers 108, …, 110 to trigger FaaS-related functionalities provided by the network-based service infrastructure 114 and using one or more of the warm containers 122, …, 124 or other containers running on one or more of the computing devices 116, …, 118.

[0063] In some aspects, the network-based service infrastructure 114 can include a function scheduling device 132 which can be used for providing FaaS - related functionalities as described herein. More specifically, the function scheduling device 132 can include a function dispatcher (FD) module 126, a function resource scheduler (FRS) module 128, and a cloud service resource scheduler (CSRS) module 130.

[0064] The CSRS module 130 may comprise suitable circuitry, logic, interfaces and/or code and is configured to schedule one or more FaaS running environments on one or more hosts within the network-based service infrastructure 114. The CSRS module 130 can be configured to receive computing resource utilization information associated with host resources used by a FaaS running environment, and generate another FaaS running environment on a different host based on the resource utilization information. In some aspects, the CSRS module 130 is configured to load and execute a FaaS service image onto a host in order to generate the FaaS running environment. The agent 121 can be part of the FaaS service image so that the agent 121 is part of the FaaS running environment when the environment is generated and running on the host. In some aspects, the agent 121 is configured to instantiate a container pool that includes a plurality of warm containers as part of the function level scheduling, as further discussed in connection with FIG. 4.

[0065] The FD module 126 may comprise suitable circuitry, logic, interfaces, and/or code and is configured to receive a function trigger event, identify a function associated with the function trigger event, and communicate information identifying the function as well as computing resource requirements associated with the function to the FRS module 128. The FD module 126 also receives destination information (e.g., an IP address) identifying a container

within a FaaS running environment, and forwards the received function trigger to the identified container using the destination information. After the received function trigger is communicated to the identified container, the function runtime within the container loads the function code associated with the function (e.g., from function code storage 134) and executes the function code once it is mounted in the container.

[0066] The FRS module 128 may comprise suitable circuitry, logic, interfaces, and/or code and is configured to receive (e.g., from the FD module 126) information identifying a function associated with a function trigger as well as computing resource requirements of the function, and select a container from a plurality of available containers within one or more FaaS running environments in the network-based service infrastructure 114 based at least on the received information. Additional functionalities of the FD module 126, the FRS module 128, and the CSRS module 130 are discussed hereinbelow in connection with FIG. 2 – FIG. 9. In some aspects, the agent 121 is configured to perform the selection of the container from the plurality of available containers in the FaaS running environment.

[0067] FIG. 2 is a block diagram illustrating a single-level function scheduling in a network architecture, according to some example embodiments. Referring to FIG. 2, diagram 202 illustrates an initial stage of a network architecture where a CSRS module 130 starts FaaS services of varying sizes using hosts 204 and 224 (each host having four CPUs). In some aspects, the CSRS module 130 can start one-CPU FaaS services 206 and 212, and a 2-CPU FaaS service 218 on host 204. The CSRS module 130 can also start a 4-CPU FaaS service 226 on host 224. The FaaS services 206, 212, 218, and 226 can include a 1-CPU warm container 208, a 1-CPU warm container 214, a 2-CPU warm container 220, and a 4-CPU warm container 228, respectively. Each of the containers 208, 214, 220, and 228 can include corresponding function runtime modules 210, 216, 222, and 230. As used herein, the term "x-CPU container" (where x is an integer greater than or equal to 1) refers to a container that can accommodate and execute a function (or functions) with a total CPU requirement of n (or the function uses, or subscribes to, a total of n number of CPUs).

[0068] Diagram 232 illustrates subsequent dispatching of functions by the FD module 126, as managed by the CSRS module 130. Specifically, the FD module 126 can dispatch (e.g., using functions identified by function triggers in the waiting list 240) a 1-CPU function 234 into container 208, a 2-CPU function 236 into container 220, and a 2-CPU function 238 into container 228.

[0069] In some aspects, due to the limitation of cloud infrastructure and the orchestration scheme used by the CSRS module 130, the pre-started empty FaaS services occupy (reserve) actual computing resources regardless of whether the service is empty or not. Once the FD module 126 dispatches a function to a container within a FaaS service and the function is running inside the container of the service, it can cause resource segmentation, resource under-subscription, or resource oversubscription. As a result, expensive and interruptive resource re-distribution or function migration may be necessary. For example, and as seen in diagram 232, resource under-subscription takes place since a 2-CPU function 238 is assigned to a 4-CPU service 226 since a 2-CPU service is not available. Resource segmentation takes place since a 1-CPU container cannot be used if no small (e.g., 1-CPU) function is in the waiting list 240. Resource oversubscription takes place when 1) function services do not reserve resource during creation, and the total resource usage on a host ends up exceeding the resource upper limit; or 2) a function of a larger CPU requirement is assigned to a 1-CPU FaaS service in hoping that the other function on the same host may not use all of their assigned resources. Then it is possible that the FaaS service together with its container cannot work properly on the current server, and the FaaS service may need to be migrated to other servers (which may cause service interruption).

[0070] FIG. 3 is a block diagram illustrating a new VM/container assignment for each host and function loading using a function resource scheduler in a network architecture, according to some example embodiments. Referring to FIG. 3, diagram 302 illustrates a 4-CPU FaaS service (or FaaS running environment) 306 generated on a 4-CPU host 304 (e.g., by the CSRS module 130). The FaaS running environment 306 can include a pool of empty containers (e.g., warm containers) of different sizes and with a sufficient number of container instances, as shown in diagram 302. For example, the FaaS running

environment 306 can include 1-CPU containers 308, 310, 312, and 314 (i.e., four 1-CPU containers are sufficient for a 4-CPU host), 2-CPU containers 316 and 318 (i.e., two 2-CPU containers are sufficient for a 4-CPU host), and a 4-CPU container 320 (the number and maximum resource utilization capacity of specific containers 308-320 is simply for illustration). Since these containers are not generated by (and not visible to) the first-level scheduler CSRS, there is no limit on the number and sizes of the containers within each FaaS running environment.

[0071] In comparison to FIG. 2, the CSRS module 130 can generate the FaaS running environment 306 and reserve all the resources on the host 304, but FaaS running environment 306 does not actually use the resources when no function is running in any of the containers. A function can be dispatched (e.g., as illustrated by diagram 336) to be executed in a container of the FaaS running environment 306 per outside function trigger at a later time. Even though the total resource subscription by the containers within the container pool (e.g., containers 308 – 320 require a total resource subscription of 12 CPUs on a 4-CPU host) exceeds the available host resources, not all containers will be loaded with functions and minimal host resources (e.g., CPU, GPU, and memory resources) are being consumed by empty containers. In this regard, since the containers within a FaaS running environment are not visible to the first-level scheduler CSRS, the oversubscription of host resources by the container pool within the FaaS running environment is acceptable and can be used for resource optimization when function scheduling takes place, as illustrated in diagram 336.

[0072] Diagram 336 illustrates the use of the FRS module 128 in communication with the FD module 126 in order to dispatch functions associated with received function event triggers to containers within the FaaS running environment 306. More specifically, the FRS module 128 can use techniques disclosed herein to select a container within the FaaS running environment 306, which can be used by the FD module 126 for dispatching the received function event trigger (or function code associated with a function identified by the function event trigger) for execution within the selected container. For example, the FD module 126 can receive function trigger events identifying a 1-CPU function 338, a 1-CPU function 340, and a 2-CPU function

342. The FRS module 128 can select corresponding containers 308, 310, and 316 for executing functions 338, 340, and 342 respectively. The FRS module 128 can send container identification information (e.g., the container's IP address) so that the FD module 126 can communicate the received function trigger to the selected container. The function runtime within the selected container can retrieve the function code (e.g., from function code storage) and executed the function within the container. Additionally, and as illustrated in FIG. 3, one 4-CPU container (320), one 2-CPU container (318), and one 1-CPU container are not assigned any functions (this is because the FRS 128 is aware that CPU resources on host 304 have all been utilized).

[0073] FIG. 4 is a block diagram illustrating a warm container/virtual machine (VM) management and function assignment using two-level scheduling in a network architecture 400, according to some example embodiments. Referring to FIG. 4, the network architecture 400 is configured for two-level scheduling (cloud service level and function level) using the FD module 126, the FRS module 128, and the CSRS module 130. The CSRS module 130 performs the cloud service level scheduling and is responsible for creating and scaling up/down FaaS services that contain FaaS running environments. For example, the CSRS module 130 generates FaaS running environments 408A, 408B, …, 408N on corresponding hosts (e.g., 4-CPU hosts 406, 424, and 444). When the FaaS running environment 408A is generated on host 406, a function scheduling agent 422 (which is part of the function level scheduling and is separate from the service level scheduling performed by the CSRS) as a component of the FaaS running environment. The agent 422 is configured to create a container pool inside the running environment 408A, including, for example, 1-CPU containers 410, 2-CPU containers 414, and a 4-CPU container 418 (other combinations of containers can be used as well). The containers within the container pool can be warm containers and can initially include function runtime (without function code), such as function runtime 412, 416, and 420. The agent 422 is further configured to sync up with the FRS module 128 to provide the FRS module 128 with information related to the FaaS running environment 408A, including size and availability of containers within the container pool as well as current containers subscription information (e.g., how much of the computing resources

(e.g., CPU, GPU, memory) of the host 406 are utilized by functions currently running in one or more of the containers in the container).

[0074] When the FaaS running environment 408B is generated on host 424, a function scheduling agent 442 is also created inside the FaaS running environment 408B. The agent 442 is configured to create a container pool, including, for example, 1-CPU containers 426, 2-CPU containers 432, and a 4-CPU container 438. The containers within the container pool of the FaaS running environment 408B can be warm containers and can initially include function runtime (without function code), such as function runtime 428, 434, and 440.

[0075] When the FaaS running environment 408N is generated on host 444, a function scheduling agent 462 is automatically generated inside the FaaS running environment 408N, as a component of the FaaS running environment 408N. The agent 462 is configured to create a container pool of differently sized containers, including, for example, 1-CPU containers 446, 2-CPU containers 452, and a 4-CPU container 456. The containers within the container pool of the FaaS running environment 408N can be warm containers and can initially only include function runtime (without function code), such as function runtime 448, 454, and 458. Even though only CPU-based resource considerations are mentioned in connection with the generation of the warm containers, the disclosure is not limited in this regard, and other types of resources (e.g., memory, GPU, and so forth) can also be considered when the warm containers are generated.

[0076] In operation, the CSRS module 130 determines a usage value indicative of computing resources of a host (e.g., 406) used by an existing FaaS running environment 408A (e.g., based on information indicative of the computing resource usage as reported by the agent 422) as well as remaining resources within the host. The CSRS module 130 compares the usage value to a threshold. If the usage value exceeds the threshold, the CSRS module 130 triggers the generation of a new FaaS running environment, such as 408B on host 424, with a function scheduling agent (e.g., 442) automatically generated inside the environment. For example, The function scheduling agent 442 generates a container pool of different size containers 426, 432, and 438 inside

the FaaS running environment 408B. In some aspects, the container pool is pre-configured for each FaaS running environment. The function scheduling agent 442 further informs the FRS module 128 of the size and availability of the containers available within the FaaS running environment 408B.

[0077] After the FD module 126 receives a function trigger 402, the FD module 126 identifies a function associated with the function trigger as well as resource requirements for the identified function (e.g., minimum CPU, GPU, and memory requirements necessary for the function to execute inside a FaaS running environment.) In aspects when the trigger was not previously served, the trigger will be sent to the FRS module 126 to obtain (i.e., select) a warm container for the function associated with the trigger. In aspects when the trigger was previously processed (and a container was previously selected for the trigger, with the container selection getting stored/cached), the FD module 126 can retrieve the cached information about the container used at the prior processing of the function and then dispatch the function trigger directly to the identified container (without the need for the FRS module 126 to make a new container selection). The FD module 126 forwards the identified function and resource requirements to the FRS module 128. In some aspects, the FD module 126 can forward the function trigger 402 to the FRS module 128, and the FRS module 128 can obtain information identifying the function associated with the function trigger as well as computing resource requirements for the identified function. The FRS module 128 can use one or more of the techniques disclosed herein (e.g., the techniques discussed in connection with FIG. 5 in FIG. 6) to select a FaaS running environment or a container from a pool of available containers within one or more FaaS running environments available within the network architecture 400. In this regard and in aspects where at least two FaaS running environments are available, a container is selected from a plurality of containers of a container pool within the first FaaS running environment as well as a plurality of containers within the second FaaS running environment. Such selection is based on the computing resource requirements of the function as well as the pre-determined resource size for each container within the available container pools (e.g., the container pools within the first and second FaaS running environments).

[0078] After the FaaS running environment (or a container within a FaaS running environment) is selected, FaaS running environment (or container) identification information (e.g., the FaaS running environment or the container IP address) is communicated to the FD module 126. The FRS module 128 further instructs the FD module 126 to forward (e.g., transmit) the received function trigger 402 (and/or information identifying the function associated with the function trigger) to the selected FaaS running environment (or container) (e.g., using the FaaS running environment or container IP address). In aspects when the function trigger event is communicated to the FaaS running environment, the agent within the environment can select an available container from the container pool in its running environment that matches the computing resource requirements of the function associated with the function trigger. Once the FaaS running environment (or container) receives the function trigger, the function runtime within the container can retrieve the function code of the associated function from a function code repository (e.g., function code storage 134) for mounting the function code inside the container for execution. In some aspects, received function triggers can be stored in a waiting list 404, pending selection of an appropriate container based on overall container availability as well as computing resource availability within the network architecture 400

[0079] For example, and as illustrated in FIG. 4, a 4-CPU FaaS service is created on a 4-CPU host 406 as a FaaS running environment 408A. The function scheduling agent 422 in the FaaS running environment creates a set of containers with the following sizes: four 1-CPU containers 410, two 2-CPU containers 414, and one 4-CPU container418 for a total of 9 potential CPU subscription. This is to ensure that for a 4-CPU host: if there are multiple 1-CPU functions that need to be triggered, there is a sufficient number of 1-CPU containers available, and if there are multiple 2-CPU functions that need to be triggered, there is a sufficient number of 2-CPU containers available. Even though each of the FaaS running environments illustrated in FIG. 4 uses a container pool of nine containers with different sizes, the disclosure is not limited in this regard and container pools with a different number of containers in different container resource combinations can be used.

[0080] The CSRS module 130 can scale up the FaaS services as long as there are enough physical hosts. The FRS module 128 together with the FD module 126 can assign two small functions (e.g., 430) to two 1-CPU containers (e.g., 426 in FaaS running environment 408B) and one large function (e.g., 436) to a 2-CPU container (e.g., 432). The remaining four containers within the FaaS running environment 408B can be kept unloaded, without using any computing resources of the host 424.

[0081] In some aspects, the FRS module 128 can determine whether there are enough computing resources on a given host to achieve oversubscription with functions executing in containers running on the host. For example, at FaaS running environment 408N, the FRS module 128 can determine that host 444 has available computing resources and it can be oversubscribed by loading one large function (e.g., 460) to a 4-CPU container, and one small function (e.g., 450) to a 1-CPU container, which achieves the planned over-subscription.

[0082] By using the disclosed techniques for two-level scheduling, the following advantages can be realized: achieving quick function loading time (e.g., 200 ms) by using the warm container approach, ability to manage differently sized warm containers without sacrificing overall resource usage efficiency or planning (e.g., by the first-level scheduling), ability to perform function level resource over-subscription to increase overall cloud resource efficiency, and de-coupling FaaS service level scheduling and function level scheduling without changes in the cloud orchestration engine.

[0083] As described herein, the FRS module 128 is responsible for handling function and container binding (e.g., selecting a suitable container from the available pool of containers for executing the function), leaving unsuitable containers (e.g., containers of size that is unsuitable for executing a triggered function) in the service unassigned or making oversubscription decisions for purposes of oversubscribing a host with multiple functions being executed using resources of such host. The following techniques can be used in connection with performing the function level scheduling and, in particular, selecting a container from a plurality of available containers to use for executing a function.

[0084] In some aspects, the following characteristics/parameters can be defined and maintained by the FRS module 128 in connection with performing the container selection functions.

[0085] As used herein, container resource subscription vector $X_j=\{x_{1,j}, x_{2,j}, x_{i,j},\ldots x_{n,j}\}$, represents different resource subscription characteristics of container X_j. The variable x_i represents subscription value of different resource types, such as, number of CPUs, GPUs, Tensor Processing Units (TPUs), memory, and so forth.

[0086] As used herein, $Y_k= \{X_{1,k}, X_{2,k}, X_{j,k},\ldots. X_{m,k}\}$ is a vector of all pre-created containers X in FaaS Service Y_k. The value m is the total created containers in service k (multiple different containers can be created in a FaaS service and until a function is assigned, the empty container occupies little resources). This vector may not be changed once a FaaS service is created and the vector is synced to the FRS module 128 through the agent.

[0087] As used herein, $A_k=\{x_{1,k}, x_{2,k},\ldots x_{n,k}\}$ represents available resources in FaaS Service k. This information is calculated by the function scheduling agent in each FaaS Service when there is a new function assignment or function lifecycle ending.

[0088] As used herein, $Y= \{Y_1, Y_2, Y_k,\ldots. Y_l\}$ is a vector of all available FaaS services in a cluster orchestration (e.g., a collection of hosts in a network-based service infrastructure). This vector can be scaled up or down by the CSRS module 130.

[0089] FIG. 5 illustrates graphs of example likelihood calculations associated with different computing resources, according to some example embodiments. As used herein, the term "likelihood operator" of a function (e.g., likelihood operators $L_{i,q}$ of function q) is used in connection with a computing resource to indicate whether a measure of the computing resource (or a computing resource value) satisfies the computing resource requirements associated with the function. More specifically, FIG. 5 illustrates graphs of example likelihood operators $L_{i,q}$ of function q. For example, graph 502 illustrates the likelihood operator $L_{1,q}(x_1)$, which can be used to perform likelihood calculation towards resource type x_1 for function q. Graph 502 shows

that x_1 represents CPU resources. The $L_{1,q}$ operator is constructed in such way that with a 1-CPU container, the likelihood (R value) will be 1, which is optimal. The likelihood value will be 0 for a container with other CPU subscription numbers.

[0090] Graph 504 illustrates the likelihood operator $L_{2,q}(x_2)$, which can be used to perform likelihood calculation towards resource type x_2 for function q. Graph 504 shows that x_2 represents GPU resources. The $L_{2,q}$ operator is constructed in such way that with a 1-GPU container, the likelihood (R value) will be 1, which is optimal. The likelihood value will be 0.5 if there is a 2-GPU subscription in a container, and the likelihood value will be zero if there is no GPU subscription for a container.

[0091] Graph 506 illustrates the likelihood operator $L_{3,q}(x_3)$, which can be used to perform likelihood calculation towards resource type x_3 for function q. Graph 506 shows that x_3 represents memory resources. The $L_{3,q}$ operator is constructed in such way that with 500MB memory subscription in a container, the likelihood (R value) will be 1, which is optimal. The likelihood value will decrease if there are more than 500MB memory subscribed by a container, and the likelihood value will be zero if there is less than 500MB memory subscribed by a container.

[0092] In some aspects, the following functions scheduling algorithm can be used by the FRS module 128 or the function scheduling agent in order to select a container from a pool of available containers for executing the function.

[0093] In some aspects, one or more of the functionalities associated with container selection and described (e.g., in connection with FIGS. 5-6) as being performed by the FRS module 128 can be performed by the function scheduling agent inside a FaaS running environment. In this regard, processing associated with the selection of a container for a received trigger event can be shared between the function scheduling agent and the FRS module 128.

[0094] For example, the FRS module 128 can use available resources reported by the agent to send a received function trigger to the FaaS service associated with the agent. The function scheduling agent inside the FaaS service, after receiving the trigger, selects a container to serve the function

trigger. In this regard, the agent knows the exact mapping between the function trigger and the container, while the FRS module only knows the mapping of the function trigger and the FaaS services/or agent. In this case, the FRS module does not need to select a container and know the container IP address in advance in order to dispatch the function trigger; instead, the trigger is simply dispatched to the FaaS service and the agent inside the service performs the container selection.

[0095] In some aspects, the following capability parameter can be determined: $\text{Capable}_q(X_j)=\{L_{1,q}(x_{1,j})+ L_{2,q}(x_{2,j})+L_{i,q}(x_{i,j}),\ldots +L_{n,q}(x_{n,j})\}/n$ produces a normalized capable value of the container X_j if function q is executed in it (i.e., the capable parameter can be smaller than or equal to 1 and it indicates whether container Xj is capable of performing the function). The value $L_{i,q}$is the likelihood operator that calculates preference of resource type x_i of function q. Capable = 1 means the container is fully capable of executing function q (with Capable being ≤ 1).

[0096] In some aspects, $Capable_q(Y) \geq T$ can be used to select a container for a function. The value T is the threshold that a function can be assigned to a container. If no over-subscription is allowed, $T=1$, otherwise T can be 0.8 or 0.9. In some aspects, T can be adjusted dynamically, when overall the resource is in shortage.

[0097] In some aspects, when a new function needs to be scheduled, the FRS module can loop through all available containers in all FaaS services running in the network architecture. The FRS module can be configured to determine the first $Capable_q(Y) \geq T$, and the function q will be assigned to the corresponding container that satisfies this condition.

[0098] In some aspects, $A_k=\{x_{1,k}, x_{2,k},\ldots x_{n,k}\}$ represents available resources in a FaaS Service k. The available resources can be updated when a container is loaded with a function or when a function is unloaded at the end of its lifecycle.

[0099] In some aspects, the FRS module and/or the function scheduling agent can determine $V_k(X_j)$ as a validity operator that indicates if container X_j in service k is valid. Such determination uses A_k to measure if containers X_j in a

service k can still obtain its targeted resources $X_j=\{x_{1,j}, x_{2,j}, x_{i,j},\ldots x_{n,j}\}$ based on the current resource availability A_k in service k. If one type of resource for a container is not available, the container is no longer valid and its value V_k=0. If all types of resources are still available for a given container, then such container is valid and available, indicated by V_k=1.

[00100] In some aspects, the following operation can be defined and used by the FRS module and/or the function scheduling agent during container selection: $\text{Valid}_k(Y_k)=\{V_k(X_{1,k}), V_k(X_{2,k}), V_k(X_{j,k}),\ldots, V_k(X_{m,k})\}$, which applies the validity operator to all unassigned containers in FaaS service Y_k. The output of the $\text{Valid}_k(Y_k)$ operation is a new valid and available container list of FaaS service Y_k. The containers on this new container list can be used for future function assignment. An example container selection process is illustrated in connection with FIG. 6.

[00101] FIG. 6 is a block diagram illustrating an example processing flow 600 for function scheduling and VM/container selection in a network architecture, according to some example embodiments. In some aspects, the example processing flow 600 can be performed by the FRS module 128 and/or the function scheduling agent in connection with selecting a container from a pool of available containers for executing the function. Referring to FIG. 6, at 602, the likelihood operators $L_{i,q}$ of the listed functions (e.g., F1, F2, F3, F4, Fx) can be determined for the indicated host resources (e.g., GPU, CPU, and memory requirements for each function). At operation 604, the initially available resources A_K in FaaS service Y_K can be determined (e.g., by each agent). For example, FaaS service Y1 has 1 GPU, 1 CPU, and 500MB of memory available, while FaaS service Yk has 1 GPU, 6 CPUs, and 8 GB of memory available.

[00102] At operation 606, the valid and available container list $Valid_K$ in FaaS service Y_K can be determined and sent to the FRS module. As illustrated in FIG. 6, service Y_K includes different containers (other combinations of containers may also be used). Similarly, the FRS module can obtain the valid and available resource list for all other remaining FaaS services within the network architecture from each agent.

[00103] At operation 608, assuming a function trigger identifying function Fx for placement has arrived, the FRS module 128 and/or the function scheduling agent determines the Capable parameter for all available containers in the container pools of the network architecture to locate container $X_{j,k}$ that can execute function F_X with parameter $T=0.9$. Since container $X_{3,k}$ has the highest Capable parameter (e.g., 1), this container can be selected by the FRS module for executing function F_X. In this regard, the FRS module uses the table illustrated at 604 (FIG. 6) to determine overall resources of one FaaS service are sufficient, and if resources are sufficient, the trigger can be communicated to that FaaS service (since there is a sufficient number of differently sized empty containers available in each service). The agent inside the FaaS service can perform the container selection and mapping between the function and the container.

[00104] At operation 610, the available resources A_K in FaaS service Y_K can be re-calculated after function Fx is assigned to (and executing from) container $X_{3,K}$ in service Y_K. At operation 612, a new valid and available container list $Valid_K$ in FaaS service Y_K is calculated (e.g., by the agent in the FaaS service), which list can be used for subsequent container selection within each FaaS Service.

[00105] FIG. 7 is a flowchart of a method 700 suitable for executing functions within a FaaS running environment of a network architecture, according to some example embodiments. The method 700 includes operations 702, 704, 706, 708, and 710. By way of example and not limitation, the method 700 is described as being performed by the function scheduling device 132 using the modules 126, 128, and 130 (corresponding to modules 860, 862, and 864 of FIG. 8, or modules 960, 965, and 970 of FIG. 9).

[00106] At operation 702, a FaaS running environment (or FaaS service) is generated on a host within a network architecture. For example, the CSRS module 130 can generate the FaaS running environment 408A on host 406. The FaaS running environment includes a FaaS service agent (e.g., agent 422) configured to monitor computing resource usage within the FaaS running environment and report the resource usage to one or more modules outside of the FaaS running environment (e.g., to the CSRS for determining how occupied host resources are within the FaaS running environment and whether another FaaS

running environment may need to be created, or to the FRS module for purposes of container selection for executing a function associated with a received function trigger). The FaaS service agent is also configured to instantiate a VM/container pool within the FaaS running environment 408A. The VM/container pool includes a plurality of containers (with two or more of the containers being of different sizes) running within the FaaS running environment on the host, each container indicating maximum computing resources that a function executing in the container can use (e.g., a 4-CPU container can be used for running a function with a maximum CPU usage of 4 CPUs). For example, and as illustrated in FIG. 4, the VM/container pool in the FaaS running environment 408A includes 1-CPU containers 410, 2-CPU containers 414, and a 4-CPU container 418 (and may also include other possible container size combinations as well as a plurality of VM instances). Each container of the plurality of containers can have a resource size not exceeding (or based on over-subscription rules) the computing resources of the host. In this regard, by using the disclosed techniques for two-level scheduling, multiple differently sized containers or a sufficient number of VM instances can be created on a host (regardless of the total available host resources). The creation of the multiple containers takes place without disrupting the first level scheduling by the CSRS associated with creating the FaaS running environment/FaaS service. The created containers are then used by the second level scheduler (i.e., the FRS module, the FD module, and the agent) to select a container for running a function associated with a received function trigger.

[00107] At operation 704, a function trigger event for triggering a function is detected. At operation 706, computing resource requirements associated with the function can be determined. For example, after a function trigger 402 is received by the FD module 126, the FD module can determine the function associated with the function trigger as well as the computing resource requirements of the function, which information can be forwarded to the FRS module 128.

[00108] At operation 708, a container is selected from the plurality of containers based on the computing resource requirements associated with the function and computing resources available at the FaaS service. For example,

the agent within the FaaS service can report container availability within one or more FaaS running environments to the FRS module (e.g., how many containers as well as the container capacity are available in each FaaS service). The FRS module selects, based on the overall available resource of each FaaS service as reported by the agent as well as the computing resource requirements of the function (e.g. the specific container subscription, or size, that is needed by the function), a FaaS service which can be used by the FD module for dispatching the function trigger. After the function trigger is dispatched to the selected FaaS service, the agent within the selected FaaS service (or the FRS module 128) can select a container from the container pool within the FaaS running environment 408A using the techniques disclosed herein (e.g., as described in connection with FIG. 5 in FIG. 6), or the FRS module can select a container directly.

[00109] At operation 710, the function trigger event is transmitted to the selected container for executing function code of the function within the selected container. For example, the FD module 126 can receive information identifying the selected FaaS service from the FRS module 128 and can forward the function trigger 402 to the selected FaaS Service, and the agent in the FaaS service selects a container for mounting and executing the function code of the function associated with the trigger. Alternatively, the FRS module can select a FaaS service and a container, and forward information identifying the container to the FD module for communicating the trigger to the identified container.

[00110] FIG. 8 is a block diagram illustrating a representative software architecture 800, which may be used in conjunction with various device hardware described herein, according to some example embodiments. FIG. 8 is merely a non-limiting example of a software architecture 802 and it will be appreciated that many other architectures may be implemented to facilitate the functionality described herein. The software architecture 802 may be executing on hardware such as computing device 900 of FIG. 9 that includes, among other things, processor 905, memory 910, storage 915 and 920, and I/O components (or interfaces) 925 and 930. A representative hardware layer 804 is illustrated and can represent, for example, the computing device 900 of FIG. 9. The representative hardware layer 804 comprises one or more processing units 806 having associated executable instructions 808. Executable instructions 808

represent the executable instructions of the software architecture 802, including implementation of the methods, modules and so forth of FIGS. 1-7. Hardware layer 804 also includes memory and/or storage modules 810, which also have executable instructions 808. Hardware layer 804 may also comprise other hardware 812, which represents any other hardware of the hardware layer 804, such as the other hardware illustrated as part of computing device 900.

[00111] In the example architecture of FIG. 8, the software architecture 802 may be conceptualized as a stack of layers where each layer provides particular functionality. For example, the software architecture 802 may include layers such as an operating system 814, libraries 816, frameworks/middleware 818, applications 820 and presentation layer 844. Operationally, the applications 820 and/or other components within the layers may invoke application programming interface (API) calls 824 through the software stack and receive a response, returned values, and so forth illustrated as messages 826 in response to the API calls 824. The layers illustrated in FIG. 8 are representative in nature and not all software architectures 802 have all layers. For example, some mobile or special purpose operating systems may not provide frameworks/middleware 818, while others may provide such a layer. Other software architectures may include additional or different layers.

[00112] The operating system 814 may manage hardware resources and provide common services. The operating system 814 may include, for example, a kernel 828, services 830, drivers 832, an FD module 860, an FRS module 862, and a CSRS module 864. The kernel 828 may act as an abstraction layer between the hardware and the other software layers. For example, kernel 828 may be responsible for memory management, processor management (e.g., scheduling), component management, networking, security settings, and so on. The services 830 may provide other common services for the other software layers. The drivers 832 may be responsible for controlling or interfacing with the underlying hardware. For instance, the drivers 832 may include display drivers, camera drivers, Bluetooth® drivers, flash memory drivers, serial communication drivers (e.g., Universal Serial Bus (USB) drivers), Wi-Fi® drivers, audio drivers, power management drivers, and so forth, depending on the hardware configuration.

[00113] In some aspects, the FD module 860, the FRS module 862, and the CSRS module 864 can be the same as (and perform the same functionalities as) the FD module 126, the FRS module 128, and the CSRS module 130, respectively.

[00114] The libraries 816 may provide a common infrastructure that may be utilized by the applications 820 and/or other components and/or layers. The libraries 816 typically provide functionality that allows other software modules to perform tasks in an easier fashion than to interface directly with the underlying operating system 814 functionality (e.g., kernel 828, services 830, drivers 832, and/or modules 860-864). The libraries 816 may include system libraries 834 (e.g., C standard library) that may provide functions such as memory allocation functions, string manipulation functions, mathematic functions, and the like. In addition, the libraries 816 may include API libraries 836 such as media libraries (e.g., libraries to support presentation and manipulation of various media format such as MPEG4, H.264, MP3, AAC, AMR, JPG, PNG), graphics libraries (e.g., an OpenGL framework that may be used to render 2D and 3D in a graphic content on a display), database libraries (e.g., SQLite that may provide various relational database functions), web libraries (e.g., WebKit that may provide web browsing functionality), and the like. The libraries 816 may also include a wide variety of other libraries 838 to provide many other APIs to the applications 820 and other software components/modules.

[00115] The frameworks/middleware 818 (also sometimes referred to as middleware) may provide a higher-level common infrastructure that may be utilized by the applications 820 and/or other software components/modules. For example, the frameworks/middleware 818 may provide various graphical user interface (GUI) functions, high-level resource management, high-level location services, and so forth. The frameworks/middleware 818 may provide a broad spectrum of other APIs that may be utilized by the applications 820 and/or other software components/modules, some of which may be specific to a particular operating system 814 or platform.

[00116] The applications 820 include built-in applications 840 and/or third-party applications 842. Examples of representative built-in applications

840 may include but are not limited to, a contacts application, a browser application, a book reader application, a location application, a media application, a messaging application, and/or a game application. Third-party applications 842 may include any of the built-in applications 840 as well as a broad assortment of other applications. In a specific example, the third-party application 842 (e.g., an application developed using the Android™ or iOS™ software development kit (SDK) by an entity other than the vendor of the particular platform) may be mobile software running on a mobile operating system such as iOS™, Android™, Windows® Phone, or other mobile operating systems. In this example, the third-party application 842 may invoke the API calls 824 provided by the mobile operating system such as operating system 814 to facilitate functionality described herein.

[00117] The applications 820 may utilize built-in operating system functions (e.g., kernel 828, services 830, drivers 832, and/or modules 860-864), libraries (e.g., system libraries 834, API libraries 836, and other libraries 838), and frameworks/middleware 818 to create user interfaces to interact with users of the system. Alternatively, or additionally, in some systems, interactions with a user may occur through a presentation layer, such as presentation layer 844. In these systems, the application/module "logic" can be separated from the aspects of the application/module that interact with a user.

[00118] Some software architectures utilize virtual machines. In the example of FIG. 8, this is illustrated by virtual machine 848. A virtual machine creates a software environment where applications/modules can execute as if they were executing on a hardware machine (such as the computing device 900 of FIG. 9, for example). A virtual machine 848 is hosted by a host operating system (operating system 814 in FIG. 8) and typically, although not always, has a virtual machine monitor 846, which manages the operation of the virtual machine 848 as well as the interface with the host operating system (i.e., operating system 814). A software architecture 802 executes within the virtual machine 848 such as an operating system 850, libraries 852, frameworks/middleware 854, applications 856, and/or presentation layer 858. These layers of software architecture executing within the virtual machine 848

can be the same as corresponding layers previously described or may be different.

[00119] FIG. 9 is a block diagram illustrating circuitry for a device that implements algorithms and performs methods, according to some example embodiments. All components need not be used in various embodiments. For example, clients, servers, and cloud-based network devices may each use a different set of components, or in the case of servers, for example, larger storage devices.

[00120] One example computing device in the form of a computer 900 (also referred to as computing device 900, computer system 900, or computer 900) may include a processor 905, memory 910, removable storage 915, non-removable storage 920, input interface 925, output interface 930, and communication interface 935, all connected by a bus 940. Although the example computing device is illustrated and described as the computer 900, the computing device may be in different forms in different embodiments.

[00121] The memory 910 may include volatile memory 945 and non-volatile memory 950 and may store a program 955. The computer 900 may include – or have access to a computing environment that includes – a variety of computer-readable media, such as the volatile memory 945, the non-volatile memory 950, the removable storage 915, and the non-removable storage 920. Computer storage includes random-access memory (RAM), read-only memory (ROM), erasable programmable read-only memory (EPROM) and electrically erasable programmable read-only memory (EEPROM), flash memory or other memory technologies, compact disc read-only memory (CD ROM), digital versatile disks (DVD) or other optical disk storage, magnetic cassettes, magnetic tape, magnetic disk storage or other magnetic storage devices, or any other medium capable of storing computer-readable instructions.

[00122] Computer-readable instructions stored on a computer-readable medium (e.g., the program 955 stored in the memory 910) are executable by the processor 905 of the computer 900. A hard drive, CD-ROM, and RAM are some examples of articles including a non-transitory computer-readable medium such as a storage device. The terms "computer-readable medium" and "storage device" do not include carrier waves to the extent that carrier waves are deemed

too transitory. "Computer-readable non-transitory media" includes all types of computer-readable media, including magnetic storage media, optical storage media, flash media, and solid-state storage media. It should be understood that software can be installed in and sold with a computer. Alternatively, the software can be obtained and loaded into the computer, including obtaining the software through a physical medium or distribution system, including, for example, from a server owned by the software creator or from a server not owned but used by the software creator. The software can be stored on a server for distribution over the Internet, for example. As used herein, the terms "computer-readable medium" and "machine-readable medium" are interchangeable.

[00123] The program 955 may utilize modules discussed herein, such as an FD module 960, an FRS module 965, and a CSRS module 970. In some aspects, the FD module 960, the FRS module 965, and the CSRS module 970 can be the same as (and perform the same functionalities as) the FD module 126, the FRS module 128, and the CSRS module 130, respectively.

[00124] Any one or more of the modules described herein may be implemented using hardware (e.g., a processor of a machine, an application-specific integrated circuit (ASIC), field-programmable gate array (FPGA), or any suitable combination thereof). Moreover, any two or more of these modules may be combined into a single module, and the functions described herein for a single module may be subdivided among multiple modules. Furthermore, according to various example embodiments, modules described herein as being implemented within a single machine, database, or device may be distributed across multiple machines, databases, or devices.

[00125] In some aspects, one or more of the modules 960-970 can be integrated as a single module, performing the corresponding functions of the integrated modules.

[00126] Although a few embodiments have been described in detail above, other modifications are possible. For example, the logic flows depicted in the figures do not require the particular order shown, or sequential order, to achieve desirable results. Other steps may be provided, or steps may be eliminated, from the described flows, and other components may be added to, or

removed from, the described systems. Other embodiments may be within the scope of the following claims.

[00127] It should be further understood that software including one or more computer-executable instructions that facilitate processing and operations as described above with reference to any one or all of steps of the disclosure can be installed in and sold with one or more computing devices consistent with the disclosure. Alternatively, the software can be obtained and loaded into one or more computing devices, including obtaining the software through physical medium or distribution system, including, for example, from a server owned by the software creator or from a server not owned but used by the software creator. The software can be stored on a server for distribution over the Internet, for example.

[00128] Also, it will be understood by one skilled in the art that this disclosure is not limited in its application to the details of construction and the arrangement of components set forth in the description or illustrated in the drawings. The embodiments herein are capable of other embodiments and capable of being practiced or carried out in various ways. Also, it will be understood that the phraseology and terminology used herein is for the purpose of description and should not be regarded as limiting. The use of "including," "comprising," or "having" and variations thereof herein is meant to encompass the items listed thereafter and equivalents thereof as well as additional items. Unless limited otherwise, the terms "connected," "coupled," and "mounted," and variations thereof herein are used broadly and encompass direct and indirect connections, couplings, and mountings. In addition, the terms "connected" and "coupled" and variations thereof are not restricted to physical or mechanical connections or couplings. Further, terms such as up, down, bottom, and top are relative and are employed to aid illustration, but are not limiting.

[00129] The components of the illustrative devices, systems, and methods employed in accordance with the illustrated embodiments can be implemented, at least in part, in digital electronic circuitry, analog electronic circuitry, or in computer hardware, firmware, software, or in combinations of them. These components can be implemented, for example, as a computer program product such as a computer program, program code or computer instructions tangibly

embodied in an information carrier, or in a machine-readable storage device, for execution by, or to control the operation of, data processing apparatus such as a programmable processor, a computer, or multiple computers.

[00130] A computer program can be written in any form of programming language, including compiled or interpreted languages, and it can be deployed in any form, including as a stand-alone program or as a module, component, subroutine, or another unit suitable for use in a computing environment. A computer program can be deployed to be executed on one computer or on multiple computers at one site or distributed across multiple sites and interconnected by a communication network. Also, functional programs, codes, and code segments for accomplishing the techniques described herein can be easily construed as within the scope of the claims by programmers skilled in the art to which the techniques described herein pertain. Method steps associated with the illustrative embodiments can be performed by one or more programmable processors executing a computer program, code or instructions to perform functions (*e.g.*, by operating on input data and/or generating an output). Method steps can also be performed by, and apparatus for performing the methods can be implemented as, special purpose logic circuitry, *e.g.*, an FPGA (field programmable gate array) or an ASIC (application-specific integrated circuit), for example.

[00131] The various illustrative logical blocks, modules, and circuits described in connection with the embodiments disclosed herein may be implemented or performed with a general-purpose processor, a digital signal processor (DSP), an ASIC, a FPGA or other programmable logic device, discrete gate or transistor logic, discrete hardware components, or any combination thereof designed to perform the functions described herein. A general-purpose processor may be a microprocessor, but in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. A processor may also be implemented as a combination of computing devices, *e.g.*, a combination of a DSP and a microprocessor, a plurality of microprocessors, one or more microprocessors in conjunction with a DSP core, or any other such configuration.

[00132] Processors suitable for the execution of a computer program include, by way of example, both general and special purpose microprocessors, and any one or more processors of any kind of digital computer. Generally, a processor will receive instructions and data from a read-only memory or a random-access memory or both. The required elements of a computer are a processor for executing instructions and one or more memory devices for storing instructions and data. Generally, a computer will also include, or be operatively coupled to receive data from or transfer data to, or both, one or more mass storage devices for storing data, e.g., magnetic, magneto-optical disks, or optical disks. Information carriers suitable for embodying computer program instructions and data include all forms of non-volatile memory, including by way of example, semiconductor memory devices, *e.g.*, electrically programmable read-only memory or ROM (EPROM), electrically erasable programmable ROM (EEPROM), flash memory devices, and data storage disks (*e.g.*, magnetic disks, internal hard disks, or removable disks, magneto-optical disks, and CD-ROM and DVD-ROM disks). The processor and the memory can be supplemented by, or incorporated in special purpose logic circuitry.

[00133] Those of skill in the art understand that information and signals may be represented using any of a variety of different technologies and techniques. For example, data, instructions, commands, information, signals, bits, symbols, and chips that may be referenced throughout the above description may be represented by voltages, currents, electromagnetic waves, magnetic fields or particles, optical fields or particles, or any combination thereof.

[00134] As used herein, "machine-readable medium" (or "computer-readable medium") means a device able to store instructions and data temporarily or permanently and may include, but is not limited to, random-access memory (RAM), read-only memory (ROM), buffer memory, flash memory, optical media, magnetic media, cache memory, other types of storage (e.g., Erasable Programmable Read-Only Memory (EEPROM)), and/or any suitable combination thereof. The term "machine-readable medium" should be taken to include a single medium or multiple media (*e.g.*, a centralized or distributed database, or associated caches and servers) able to store processor instructions. The term "machine-readable medium" shall also be taken to

include any medium (or a combination of multiple media) that is capable of storing instructions for execution by one or more processors 905, such that the instructions, when executed by one or more processors 905, cause the one or more processors 905 to perform any one or more of the methodologies described herein. Accordingly, a "machine-readable medium" refers to a single storage apparatus or device, as well as "cloud-based" storage systems or storage networks that include multiple storage apparatus or devices. The term "machine-readable medium" as used herein excludes signals per se.

[00135] In addition, techniques, systems, subsystems, and methods described and illustrated in the various embodiments as discrete or separate may be combined or integrated with other systems, modules, techniques, or methods without departing from the scope of the present disclosure. Other items shown or discussed as coupled or directly coupled or communicating with each other may be indirectly coupled or communicating through some interface, device, or intermediate component whether electrically, mechanically, or otherwise. Other examples of changes, substitutions, and alterations are ascertainable by one skilled in the art and could be made without departing from the scope disclosed herein.

[00136] Although the present disclosure has been described with reference to specific features and embodiments thereof, it is evident that various modifications and combinations can be made thereto without departing from the scope of the disclosure. For example, other components may be added to, or removed from, the described systems. The specification and drawings are, accordingly, to be regarded simply as an illustration of the disclosure as defined by the appended claims, and are contemplated to cover any and all modifications, variations, combinations or equivalents that fall within the scope of the present disclosure. Other aspects may be within the scope of the following claims.

CLAIMS

What is claimed is:

1. A method for scheduling computing resources for executing functions, the method comprising:

generating, by a function scheduling device, a Function as a Service (FaaS) service on a host, the FaaS service including a FaaS service agent configured to monitor computing resource usage within the FaaS service and instantiate a container pool, the container pool including a plurality of containers running within the FaaS service and using computing resources of the host, each container of the plurality of containers having a pre-determined resource size;

detecting, by the function scheduling device, a function trigger event for triggering a function;

determining, by the function scheduling device, computing resource requirements associated with the function;

selecting, by the function scheduling device, a container from the plurality of containers based on the computing resource requirements associated with the function and computing resources available at the FaaS service as indicated by the computing resource usage; and

transmitting, by the function scheduling device, the function trigger event to the selected container for executing function code of the function within the selected container.

2. The method of claim 1, further comprising:

receiving, by the function scheduling device, information indicative of the computing resource usage within the FaaS service; and

generating, by the function scheduling device and based on the information indicative of the computing resource usage, at least one additional FaaS service on at least one corresponding additional host, the at least one additional FaaS service including a second container pool with a plurality of additional containers.

3. The method of claim 2, wherein the selecting of the container is further based on:

selecting, by the function scheduling device, the container from the plurality of containers of the container pool and the plurality of additional containers instantiated within the at least one additional FaaS service of the second container pool, based on the computing resource requirements of the function, the pre-determined resource size for each container of the plurality of containers, and a pre-determined resource size for each container of the plurality of additional containers.

4. The method of claim 2, further comprising:

determining a FaaS service of the at least one additional FaaS service having available computing resources to satisfy the computing resource requirements associated with the function, and an IP address of the determined FaaS service;

forwarding the function trigger event to the determined FaaS service using the IP address;

selecting a container of the plurality of additional containers within the determined FaaS service that matches the computing resource requirements of the function;

retrieving the function code from a function code repository for mounting within the selected container; and

executing the function code within the selected container after the mounting.

5. The method of claims 1 to 4, further comprising:

monitoring usage of the computing resources by the plurality of containers within the host;

determining a usage value indicative of the computing resources usage and remaining resources within the host;

comparing the usage value to a threshold; and

triggering generation of an additional FaaS service when the usage value exceeds the threshold.

6. The method of claims 1 to 5, further comprising:

caching information identifying the function associated with the function trigger event and the selected container;

upon receiving a new function trigger event associated with the function, retrieving the cached information; and

forwarding the new function trigger event to the selected container.

7. The method of claims 1 to 6, wherein the selecting of the container from the plurality of containers comprises:

determining a likelihood operator of the function for each computing resource of the computing resources, the likelihood operator indicating whether a corresponding computing resource value of the computing resource satisfies the computing resource requirements associated with the function; and

for each container of the plurality of containers of the container pool, determining a capability parameter of the function based on the likelihood operator for each of the computing resources.

8. The method of claim 7, with the selecting the container further comprising:

selecting the container based on capability parameters of the plurality of containers.

9. The method of claim 7, with the selecting the container further comprising:

selecting the container when a corresponding capability parameter of the container is above a threshold value.

10. The method of claim 9, further comprising:

determining, after the selecting of the container, a validity operator indicative of one or more containers of the plurality of containers that are still available for function assignment.

11. A system comprising:

a memory storing instructions; and

one or more processors in communication with the memory, wherein the one or more processors execute the instructions to:

generate a Function as a Service (FaaS) service on a host, the FaaS service including a FaaS service agent configured to monitor computing resource usage within the FaaS service and instantiate a container pool, the container pool including a plurality of containers running within the FaaS service and using computing resources of the host, each container of the plurality of containers having a pre-determined resource size;

detect a function trigger event for triggering a function;

determine computing resource requirements associated with the function;

select a container from the plurality of containers based on the computing resource requirements associated with the function and computing resources available at the FaaS service as indicated by the computing resource usage; and

transmit the function trigger event to the selected container for executing function code of the function within the selected container.

12. The system of claim 11, wherein the one or more processors execute the instructions to:

determine an IP address of the selected container;

forward the function trigger event to the selected container using the IP address;

retrieve the function code from a function code storage for mounting within the selected container; and

execute the function code within the selected container after the mounting.

13. The system of claims 11 to 12, wherein the one or more processors execute the instructions to:

monitor usage of the computing resources by the plurality of containers within the host;

determine a usage value indicative of the computing resources usage;

compare the usage value to a threshold; and

trigger generation of an additional FaaS service when the usage value exceeds the threshold.

14. The system of claims 11 to 13, wherein for selecting the container from the plurality of containers, the one or more processors execute the instructions to:

determine a likelihood operator of the function for each of the computing resources, the likelihood operator indicating whether a corresponding computing resource of the computing resources satisfies the computing resource requirements associated with the function.

15. The system of claim 14, wherein the one or more processors execute the instructions to:

determine for each container of the plurality of containers of the container pool, a capability parameter of the function based on the likelihood operator for each of the computing resources.

16. The system of claim 15, wherein for the selecting of the container from the plurality of containers, the one or more processors execute the instructions to:

select the container based on capability parameters of the plurality of containers.

17. The system of claim 15, wherein for the selecting of the container from the plurality of containers, the one or more processors execute the instructions to:

select the container when a corresponding capability parameter of the container is above a threshold value.

18. A computer-readable medium storing computer instructions for scheduling computing resources for executing functions within a Function as a Service (FaaS) service of a network architecture, wherein the instructions when

executed by one or more processors, cause the one or more processors to perform steps comprising:

generating the FaaS service on a host, the FaaS service including a FaaS service agent configured to monitor computing resource usage within the FaaS service and instantiate a container pool, the container pool including a plurality of containers running within the FaaS service and using computing resources of the host, each container of the plurality of containers having a pre-determined resource size;

detecting a function trigger event for triggering a function;

determining computing resource requirements associated with the function;

selecting a container from the plurality of containers based on the computing resource requirements associated with the function and computing resources available at the FaaS service as indicated by the computing resource usage; and

transmitting the function trigger event to the selected container for executing function code of the function within the selected container.

19. The computer-readable medium of claim 18, wherein the instructions further cause the one or more processors to perform steps of:

receiving information indicative of the computing resource usage within the FaaS service; and

generating at least one additional FaaS service on at least one corresponding additional host within the network architecture, the at least one additional FaaS service including a second container pool with a plurality of additional containers, the generating based on the information indicative of the computing resource usage.

20. The computer-readable medium of claim 19, wherein for the selecting of the container, the instructions further cause the one or more processors to perform steps of:

selecting the container from the plurality of containers of the container pool and the plurality of additional containers instantiated within the second container pool, based on the computing resource requirements of the function

and the resource size for each container of the plurality of containers and each container of the plurality of additional containers.

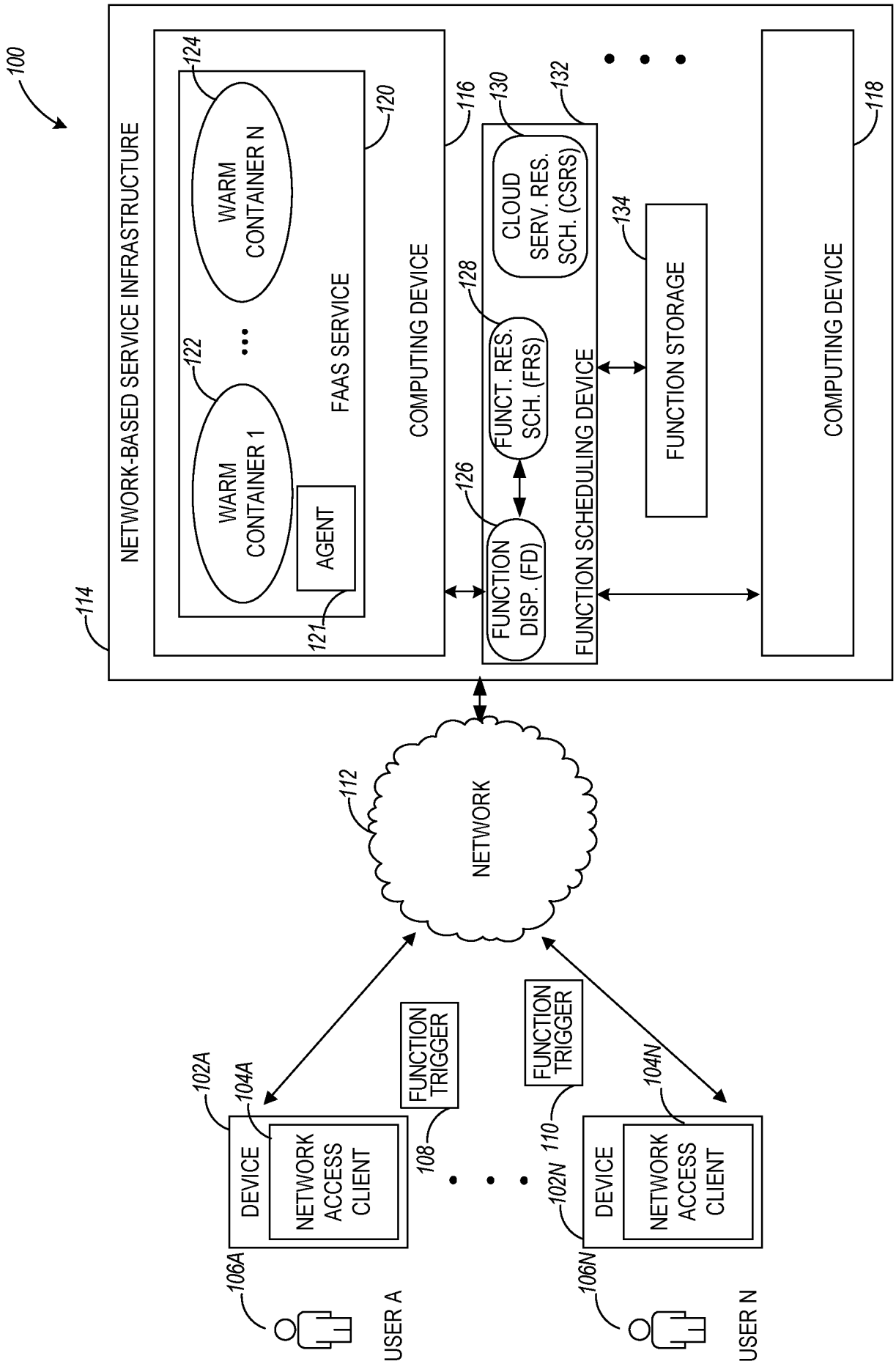

100
NETWORK-BASED SERVICE INFRASTRUCTURE
114
COMPUTING DEVICE
116
FAAS SERVICE
120
WARM CONTAINER 1
122
WARM CONTAINER N
124
AGENT
121
FUNCTION DISP. (FD)
126
FUNCT. RES. SCH. (FRS)
128
CLOUD SERV. RES. SCH. (CSRS)
130
FUNCTION SCHEDULING DEVICE
132
FUNCTION STORAGE
134
COMPUTING DEVICE
118
NETWORK
112
FUNCTION TRIGGER
108
FUNCTION TRIGGER
110
102A
104A
DEVICE
NETWORK ACCESS CLIENT
106A
USER A
102N
104N
DEVICE
NETWORK ACCESS CLIENT
106N
USER N

FIG. 1

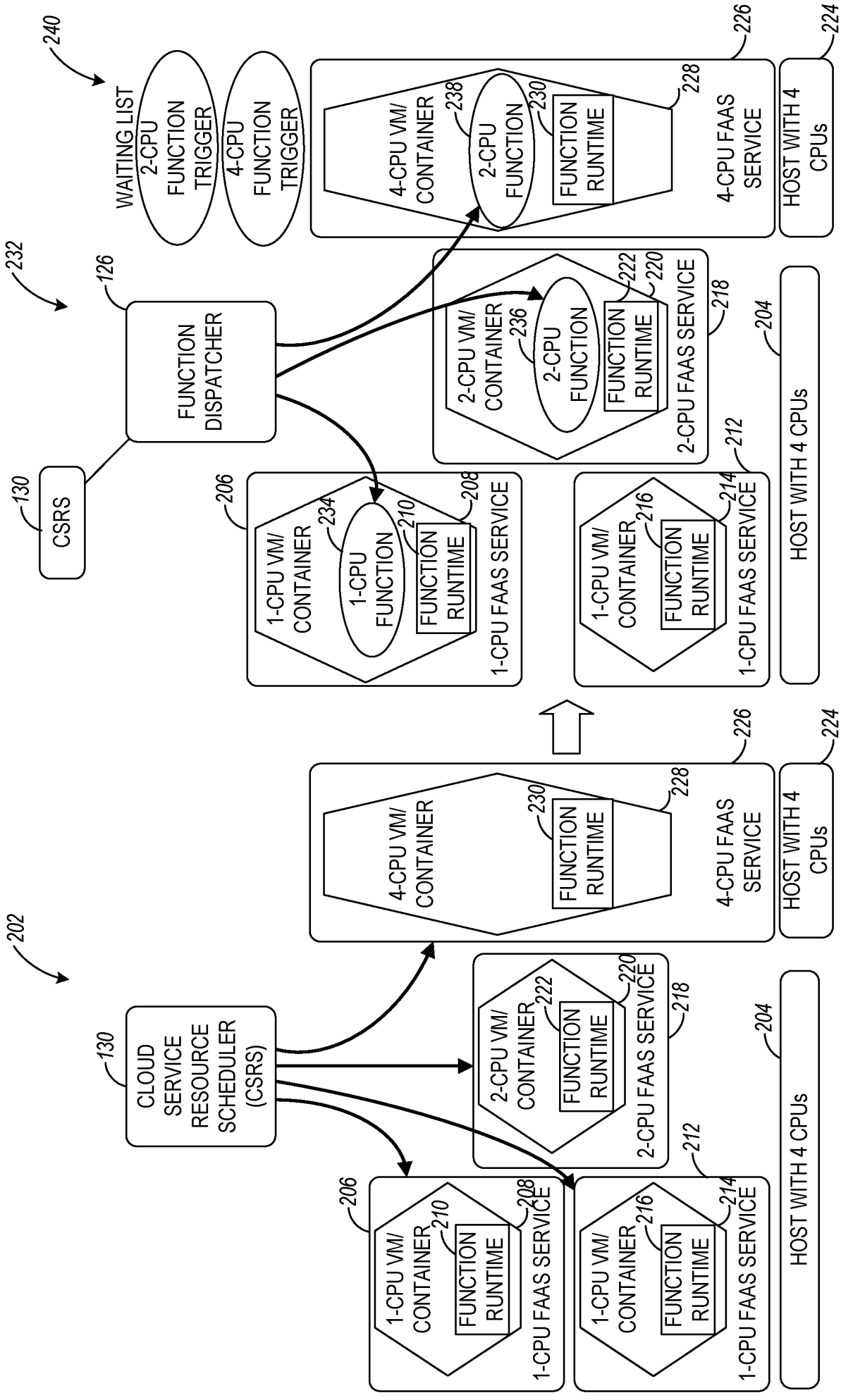

202
CLOUD SERVICE RESOURCE SCHEDULER (CSRS)
130
206
1-CPU VM/ CONTAINER
210
FUNCTION RUNTIME
208
1-CPU FAAS SERVICE
212
1-CPU VM/ CONTAINER
216
FUNCTION RUNTIME
214
1-CPU FAAS SERVICE
HOST WITH 4 CPUs
204
2-CPU VM/ CONTAINER
222
FUNCTION RUNTIME
220
2-CPU FAAS SERVICE
218
4-CPU VM/ CONTAINER
230
FUNCTION RUNTIME
228
4-CPU FAAS SERVICE
226
HOST WITH 4 CPUs
224
232
126
FUNCTION DISPATCHER
130
CSRS
206
1-CPU VM/ CONTAINER
234
1-CPU FUNCTION
210
FUNCTION RUNTIME
208
1-CPU FAAS SERVICE
212
1-CPU VM/ CONTAINER
216
FUNCTION RUNTIME
214
1-CPU FAAS SERVICE
HOST WITH 4 CPUs
204
2-CPU VM/ CONTAINER
236
2-CPU FUNCTION
222
FUNCTION RUNTIME
220
2-CPU FAAS SERVICE
218
4-CPU VM/ CONTAINER
238
2-CPU FUNCTION
230
FUNCTION RUNTIME
228
4-CPU FAAS SERVICE
226
HOST WITH 4 CPUs
224
240
WAITING LIST
2-CPU FUNCTION TRIGGER
4-CPU FUNCTION TRIGGER

FIG. 2

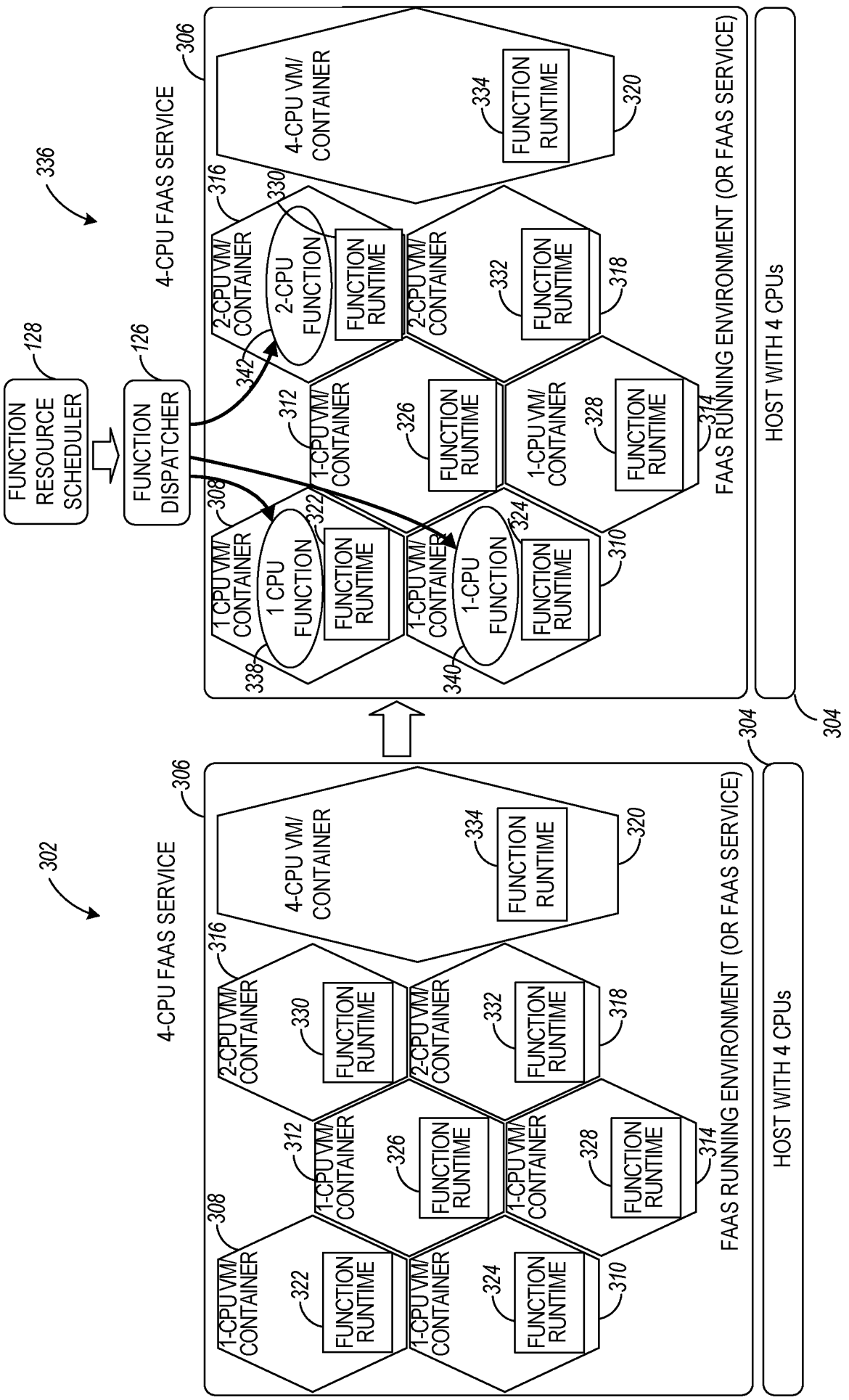

302
4-CPU FAAS SERVICE
306
FAAS RUNNING ENVIRONMENT (OR FAAS SERVICE)
304
HOST WITH 4 CPUs
4-CPU VM/ CONTAINER
FUNCTION RUNTIME
320
334
2-CPU VM/ CONTAINER
316
330
332
318
1-CPU VM/ CONTAINER
312
326
328
314
308
322
324
310
336
FUNCTION RESOURCE SCHEDULER
128
FUNCTION DISPATCHER
126
342
1 CPU FUNCTION
338
1-CPU FUNCTION
340
2-CPU FUNCTION

FIG. 3

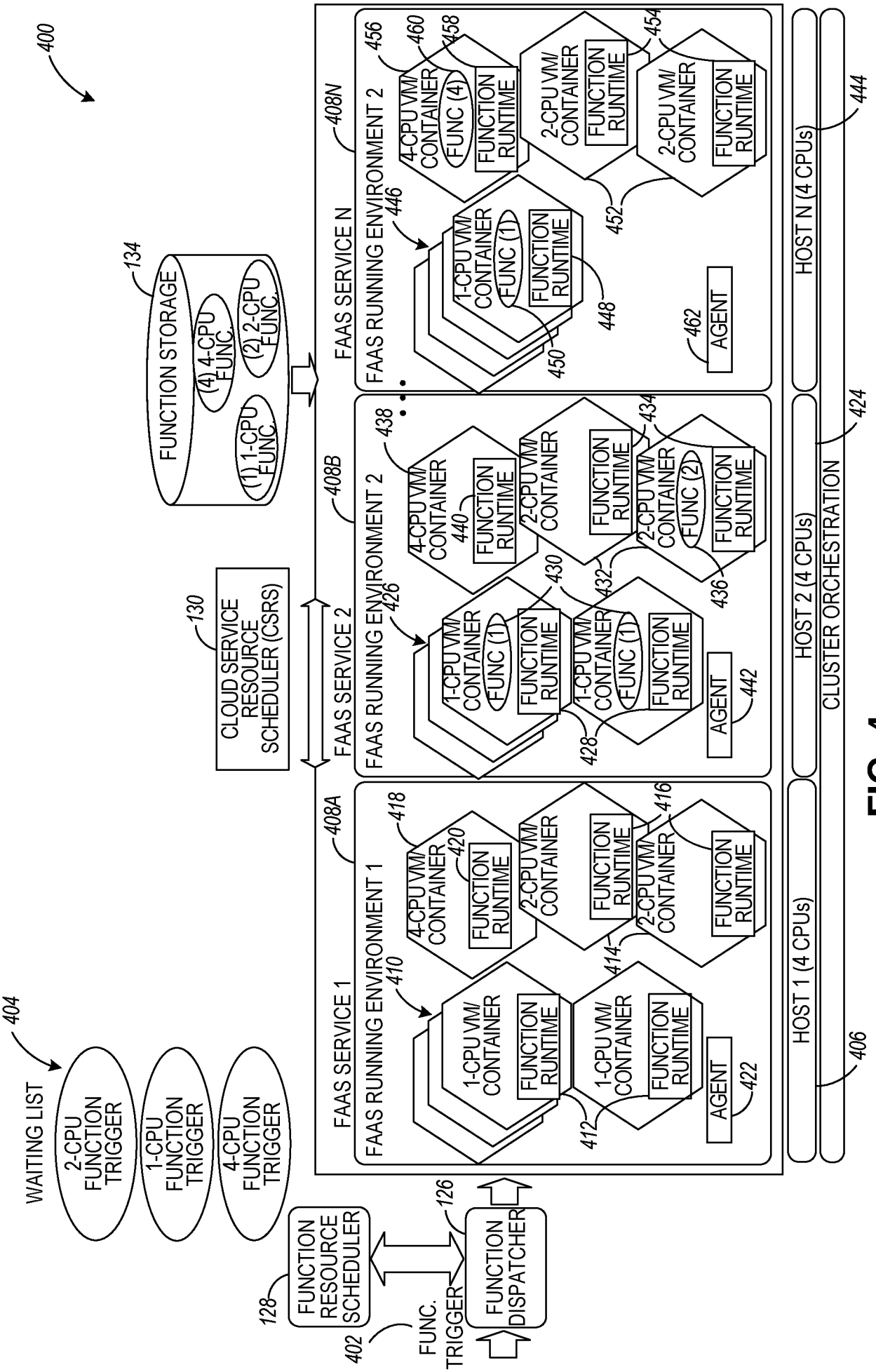

400
WAITING LIST
404
2-CPU FUNCTION TRIGGER
1-CPU FUNCTION TRIGGER
4-CPU FUNCTION TRIGGER
FUNCTION RESOURCE SCHEDULER
128
FUNC. TRIGGER
402
126
FUNCTION DISPATCHER
CLOUD SERVICE RESOURCE SCHEDULER (CSRS)
130
FUNCTION STORAGE
134
(4) 4-CPU FUNC.
(2) 2-CPU FUNC.
(1) 1-CPU FUNC.
FAAS SERVICE 1
408A
FAAS SERVICE 2
408B
FAAS SERVICE N
408N
FAAS RUNNING ENVIRONMENT 1
FAAS RUNNING ENVIRONMENT 2
FAAS RUNNING ENVIRONMENT 2
1-CPU VM/ CONTAINER
FUNCTION RUNTIME
4-CPU VM/ CONTAINER
2-CPU VM/ CONTAINER
FUNC (1)
FUNC (2)
FUNC (4)
AGENT
410
412
414
416
418
420
422
426
428
430
432
434
436
438
440
442
446
448
450
452
454
456
458
460
462
HOST 1 (4 CPUs)
HOST 2 (4 CPUs)
HOST N (4 CPUs)
CLUSTER ORCHESTRATION
406
424
444

FIG. 4

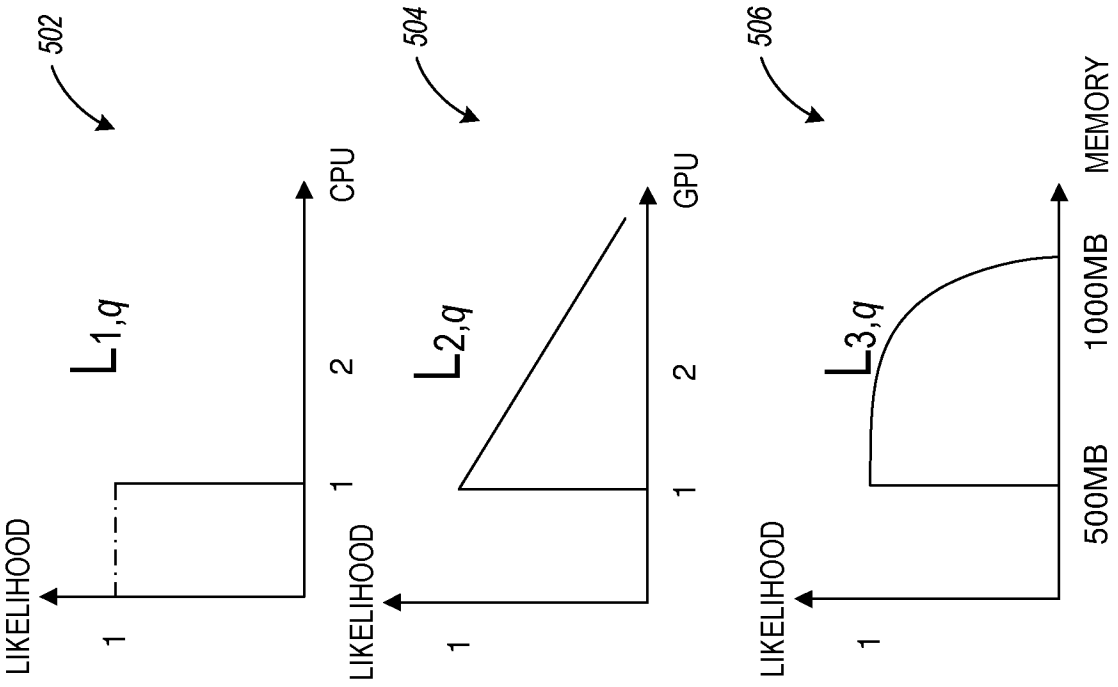

502
$L_{1,q}$
LIKELIHOOD
1
1
2
CPU
504
$L_{2,q}$
LIKELIHOOD
1
1
2
GPU
506
$L_{3,q}$
LIKELIHOOD
1
500MB
1000MB
MEMORY

FIG. 5

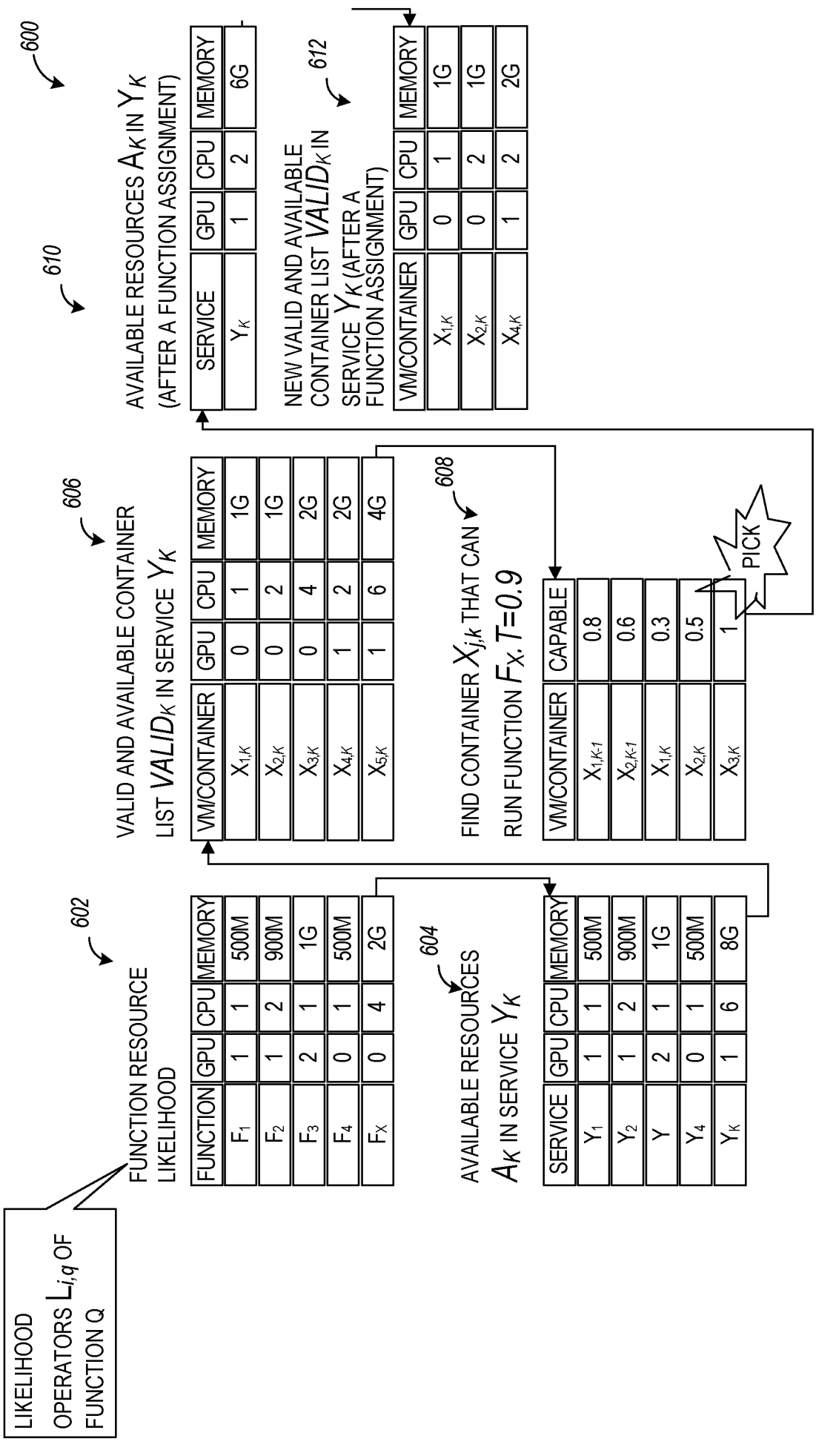

600
610
612
606
608
602
604
AVAILABLE RESOURCES A_K IN Y_K (AFTER A FUNCTION ASSIGNMENT)
SERVICE
Y_K
GPU
1
CPU
2
MEMORY
6G
NEW VALID AND AVAILABLE CONTAINER LIST $VALID_K$ IN SERVICE Y_K (AFTER A FUNCTION ASSIGNMENT)
VM/CONTAINER
$X_{1,K}$
$X_{2,K}$
$X_{4,K}$
GPU
0
0
1
CPU
1
2
2
MEMORY
1G
1G
2G
VALID AND AVAILABLE CONTAINER LIST $VALID_K$ IN SERVICE Y_K
VM/CONTAINER
$X_{1,K}$
$X_{2,K}$
$X_{3,K}$
$X_{4,K}$
$X_{5,K}$
GPU
0
0
0
1
1
CPU
1
2
4
2
6
MEMORY
1G
1G
2G
2G
4G
FIND CONTAINER $X_{j,k}$ THAT CAN RUN FUNCTION F_X, $T=0.9$
VM/CONTAINER
$X_{1,K-1}$
$X_{2,K-1}$
$X_{1,K}$
$X_{2,K}$
$X_{3,K}$
CAPABLE
0.8
0.6
0.3
0.5
1
PICK
FUNCTION RESOURCE LIKELIHOOD
FUNCTION
F_1
F_2
F_3
F_4
F_X
GPU
1
1
2
0
0
CPU
1
2
1
1
4
MEMORY
500M
900M
1G
500M
2G
AVAILABLE RESOURCES A_K IN SERVICE Y_K
SERVICE
Y_1
Y_2
Y
Y_4
Y_K
GPU
1
1
2
0
1
CPU
1
2
1
1
6
MEMORY
500M
900M
1G
500M
8G
LIKELIHOOD OPERATORS $L_{i,q}$ OF FUNCTION Q

FIG. 6

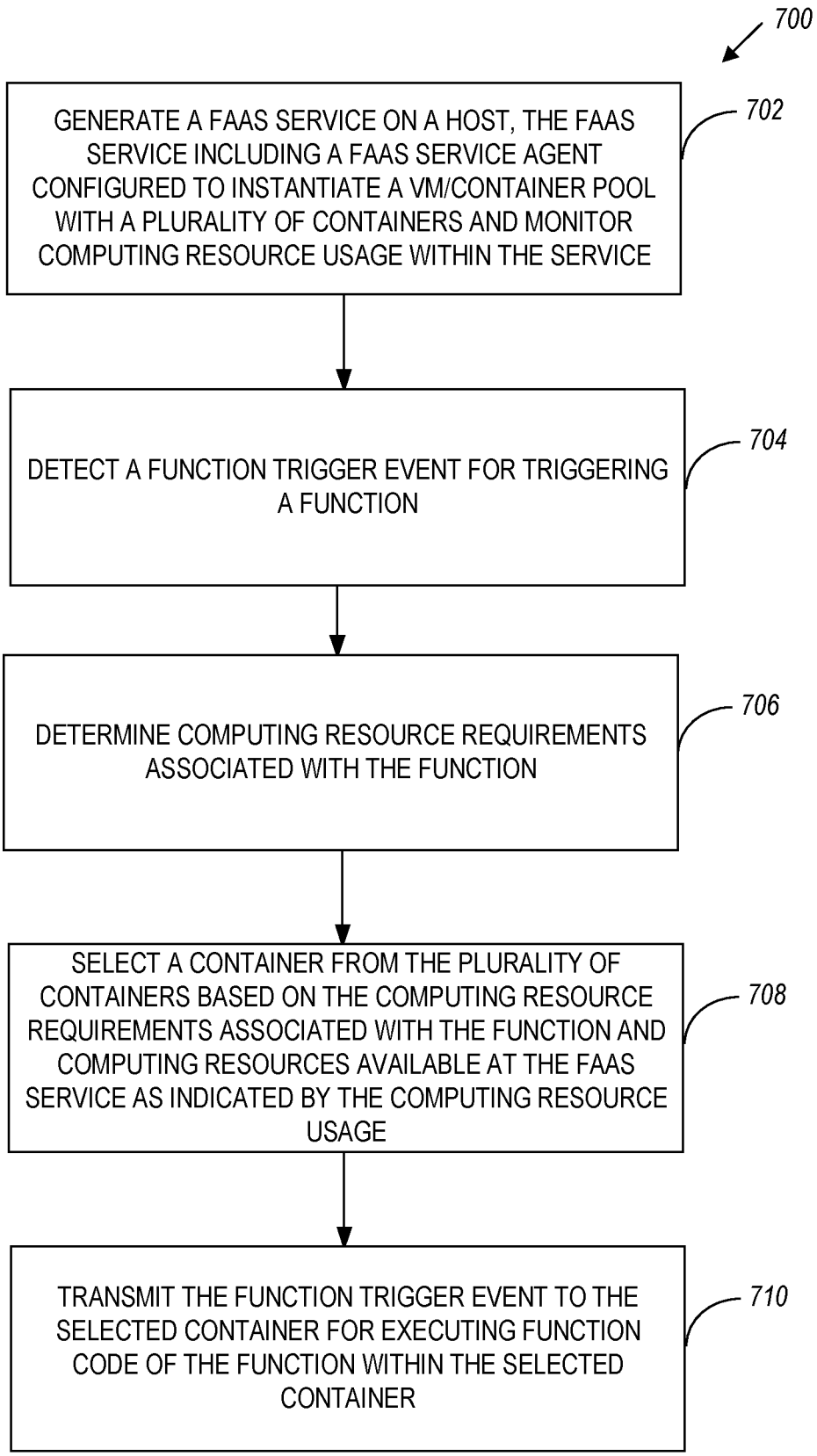

700
GENERATE A FAAS SERVICE ON A HOST, THE FAAS SERVICE INCLUDING A FAAS SERVICE AGENT CONFIGURED TO INSTANTIATE A VM/CONTAINER POOL WITH A PLURALITY OF CONTAINERS AND MONITOR COMPUTING RESOURCE USAGE WITHIN THE SERVICE
702
DETECT A FUNCTION TRIGGER EVENT FOR TRIGGERING A FUNCTION
704
DETERMINE COMPUTING RESOURCE REQUIREMENTS ASSOCIATED WITH THE FUNCTION
706
SELECT A CONTAINER FROM THE PLURALITY OF CONTAINERS BASED ON THE COMPUTING RESOURCE REQUIREMENTS ASSOCIATED WITH THE FUNCTION AND COMPUTING RESOURCES AVAILABLE AT THE FAAS SERVICE AS INDICATED BY THE COMPUTING RESOURCE USAGE
708
TRANSMIT THE FUNCTION TRIGGER EVENT TO THE SELECTED CONTAINER FOR EXECUTING FUNCTION CODE OF THE FUNCTION WITHIN THE SELECTED CONTAINER
710

FIG. 7

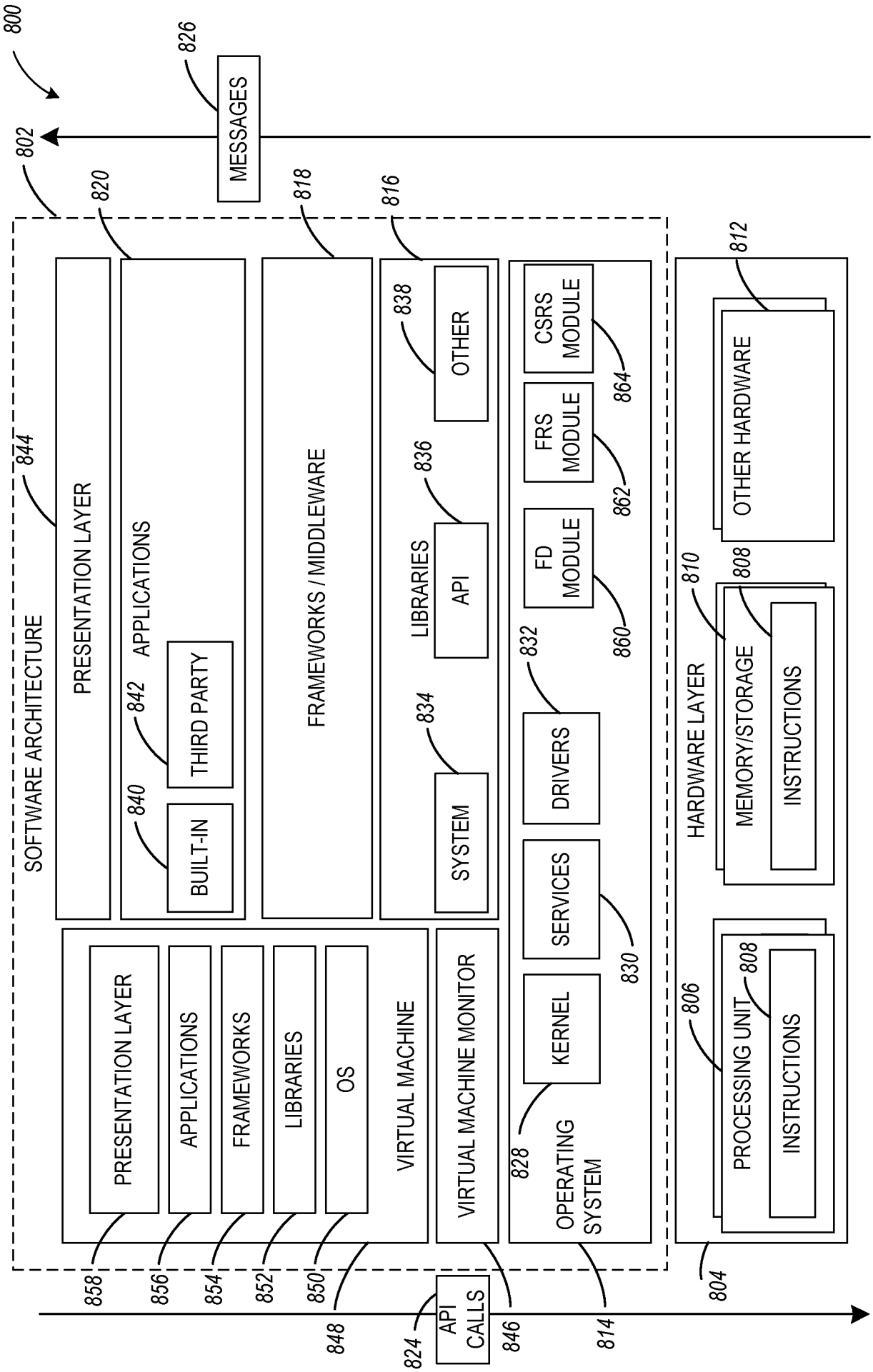

800
826
MESSAGES
802
820
818
816
SOFTWARE ARCHITECTURE
844
PRESENTATION LAYER
APPLICATIONS
842
THIRD PARTY
840
BUILT-IN
FRAMEWORKS / MIDDLEWARE
LIBRARIES
836
API
838
OTHER
834
SYSTEM
832
DRIVERS
FD MODULE
860
FRS MODULE
862
CSRS MODULE
864
VIRTUAL MACHINE
858
PRESENTATION LAYER
856
APPLICATIONS
854
FRAMEWORKS
852
LIBRARIES
850
OS
VIRTUAL MACHINE MONITOR
828
OPERATING SYSTEM
KERNEL
SERVICES
830
848
824
API CALLS
846
814
HARDWARE LAYER
812
OTHER HARDWARE
810
808
MEMORY/STORAGE
INSTRUCTIONS
806
PROCESSING UNIT
808
INSTRUCTIONS
804

FIG. 8

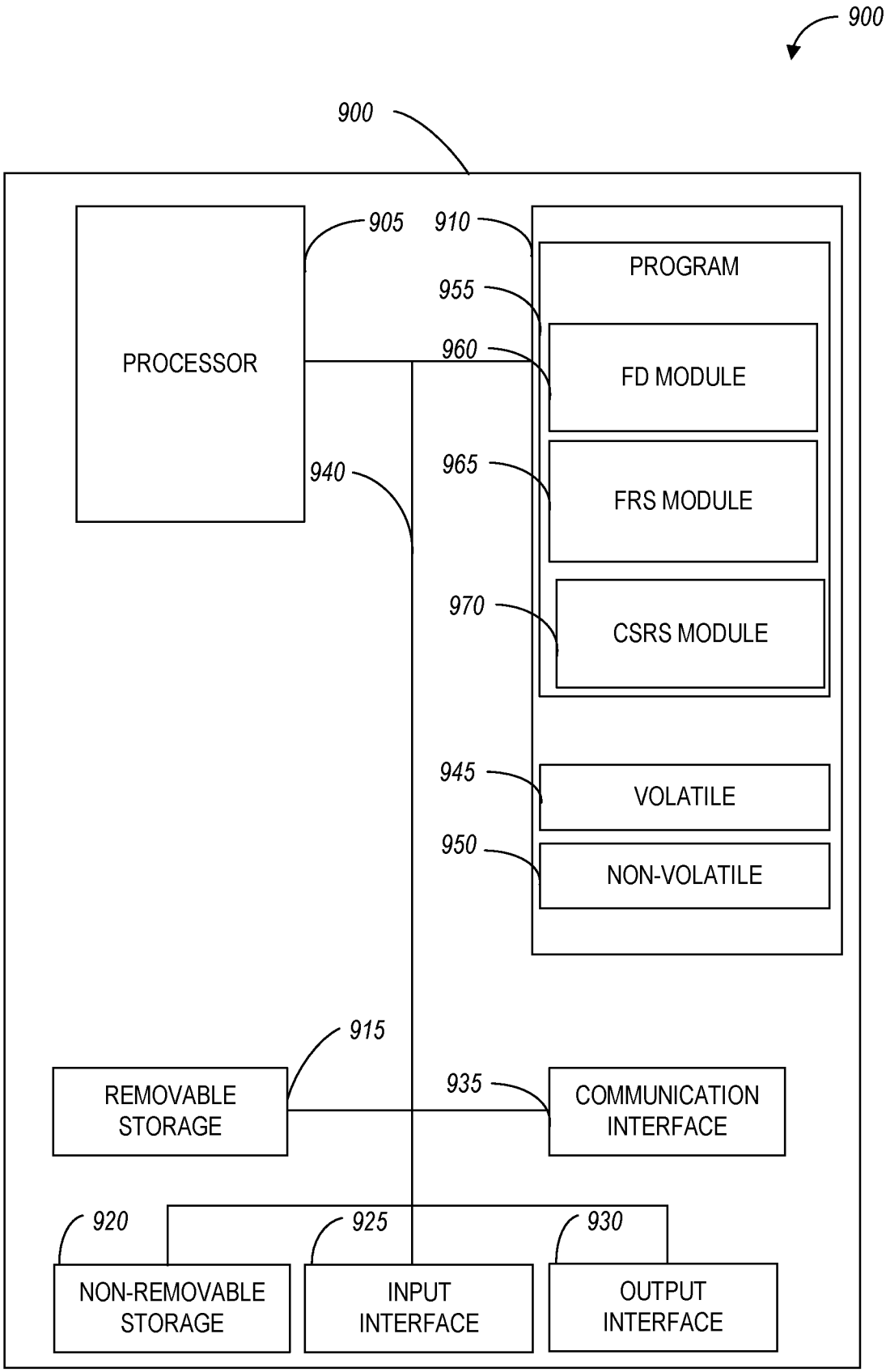

900
900
905
910
PROGRAM
955
960
PROCESSOR
FD MODULE
940
965
FRS MODULE
970
CSRS MODULE
945
VOLATILE
950
NON-VOLATILE
915
935
REMOVABLE STORAGE
COMMUNICATION INTERFACE
920
925
930
NON-REMOVABLE STORAGE
INPUT INTERFACE
OUTPUT INTERFACE

FIG. 9

INTERNATIONAL SEARCH REPORT

International application No
PCT/US2019/031570

A. CLASSIFICATION OF SUBJECT MATTER
INV. G06F9/50 H04L29/08 G06F9/445
ADD. H04L12/24

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
G06F H04L

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practicable, search terms used)
EPO-Internal, WPI Data

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	US 2017/177413 A1 (WISNIEWSKI SCOTT DANIEL [US] ET AL) 22 June 2017 (2017-06-22) paragraph [0032] - paragraph [0056]	1-20
X	WO 2016/126731 A1 (AMAZON TECH INC [US]) 11 August 2016 (2016-08-11) paragraph [0032] - paragraph [0052]	1-20
X	US 2015/120928 A1 (GUMMARAJU JAYANTH [US] ET AL) 30 April 2015 (2015-04-30)	1,11,18
A	paragraph [0024] - paragraph [0026] paragraph [0037]	2-10, 12-17, 19,20
A	US 2019/097900 A1 (RODRIGUEZ BRYAN J [US] ET AL) 28 March 2019 (2019-03-28) paragraph [0066] - paragraph [0067] paragraph [0033]	1-20

☐ Further documents are listed in the continuation of Box C.

☒ See patent family annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier application or patent but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
19 November 2019	27/11/2019

Name and mailing address of the ISA/	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Fax: (+31-70) 340-3016	Alecu, Mihail

1

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No
PCT/US2019/031570

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 2017177413 A1	22-06-2017	CN 108369534 A	03-08-2018
		EP 3394750 A1	31-10-2018
		JP 2018537776 A	20-12-2018
		US 2017177413 A1	22-06-2017
		US 2018210760 A1	26-07-2018
		US 2019227849 A1	25-07-2019
		WO 2017112526 A1	29-06-2017
WO 2016126731 A1	11-08-2016	AU 2016215438 A1	24-08-2017
		CA 2975522 A1	11-08-2016
		CN 107533470 A	02-01-2018
		EP 3254434 A1	13-12-2017
		JP 6463494 B2	06-02-2019
		JP 2018512087 A	10-05-2018
		KR 20170110647 A	11-10-2017
		US 9471775 B1	18-10-2016
		WO 2016126731 A1	11-08-2016
US 2015120928 A1	30-04-2015	NONE	
US 2019097900 A1	28-03-2019	NONE	