



(51) International Patent Classification:

G06F 9/48 (2006.01) G06F 9/445 (2006.01)
G06F 9/50 (2006.01) G06F 9/44 (2006.01)
G06F 9/54 (2006.01)

(21) International Application Number:

PCT/IB2015/001288

(22) International Filing Date:

21 May 2015 (21.05.2015)

(25) Filing Language:

English

(26) Publication Language:

English

(30) Priority Data:

62/001,457	21 May 2014 (21.05.2014)	US
62/028,145	23 July 2014 (23.07.2014)	US
62/041,566	25 August 2014 (25.08.2014)	US
62/137,007	23 March 2015 (23.03.2015)	US

(71) Applicant: SOCIETAL INNOVATIONS IPCO LIMITED [GB/GB]; 14 New Street, London EC2M 4HE (GB).

(72) Inventors: STANDLEY, Douglas, A.; 2801 Pennsylvania Ave. #104, Boulder, CO 80303 (US). DODGE, Matthew, R.; 3137A Divisadero Street, San Francisco, CA 94123 (US). BYE, Randall, E.; 843 Trail Ridge Drive, Louisville, CO 80027 (US).

(74) Agent: MANKE, Lars; Uexküll & Stolberg, Beselerstr. 4, 22607 Hamburg (DE).

(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, DK, DM, DO, DZ, EC, EE, EG, ES, FI, GB, GD, GE, GH, GM, GT, HN, HR, HU, ID, IL, IN, IR, IS, JP, KE, KG, KN, KP, KR, 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, 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,

[Continued on next page]

(54) Title: SYSTEM AND METHOD FOR FULLY CONFIGURABLE REAL TIME PROCESSING

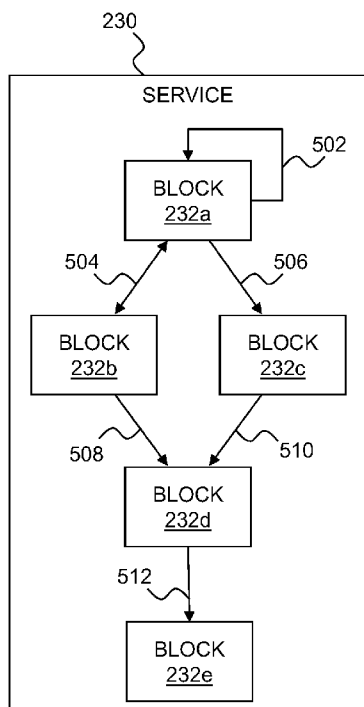


FIG. 5C

(57) Abstract: Described are various embodiments of a neutral input/output (NIO) platform that includes a core that supports one or more services. While the platform itself may technically be viewed as an executable application in some embodiments, the core may be thought of as an application engine that runs task specific applications called services. The services are constructed using defined templates that are recognized by the core, although the templates can be customized to a certain extent. The core is designed to run the services, and the services in turn run blocks that provide processing functionality to their respective service. Due to the structure and flexibility provided by the platform's core, services, and blocks, the platform is configurable to asynchronously process any input signals from one or more sources in real time.



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).

Published:

- *with international search report (Art. 21(3))*
- *before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments (Rule 48.2(h))*

5

SYSTEM AND METHOD FOR FULLY CONFIGURABLE REAL TIME PROCESSING

10

BACKGROUND

[0001] The proliferation of devices has resulted in the production of a tremendous amount of data that is continuously increasing. Current processing methods are unsuitable for processing this data. Accordingly, what is needed are systems and methods that address
15 this issue.

BRIEF DESCRIPTION OF THE DRAWINGS

[0002] For a more complete understanding, reference is now made to the following description taken in conjunction with the accompanying Drawings in which:

[0003] FIG. 1A illustrates one embodiment of a neutral input/output (NIO) platform with customizable and configurable processing functionality and configurable support functionality;

[0004] FIG. 1B illustrates one embodiment of a data path that may exist within a NIO platform instance based on the NIO platform of FIG. 1A;

[0005] FIGS. 1C and 1D illustrate embodiments of the NIO platform of FIG. 1A as part of a stack;

[0006] FIGS. 1E-1I illustrate embodiments of environments in which the NIO platform of FIG. 1A may be deployed;

[0007] FIGS. 1J-1L illustrate embodiments of systems on which the NIO platform of FIG. 1A may be run;

[0008] FIG. 2A illustrates a more detailed embodiment of the NIO platform of FIG. 1A;

[0009] FIGS. 2B and 2C illustrate embodiments of hierarchical arrangements that may exist within NIO platform instances;

[0010] FIG. 3 illustrates one embodiment of a service that may exist within a NIO platform instance;

[0011] FIG. 4A illustrates another embodiment of the NIO platform of FIG. 2A;

[0012] FIG. 4B illustrates one embodiment of a NIO platform instance based on the NIO platform of FIG. 4A;

[0013] FIG. 4C illustrates one embodiment of a diagram showing the operation of a core process and multiple services of the NIO platform instance of FIG. 4B plotted against time;

[0014] FIGS. 5A-5C illustrate various embodiments of the service of FIG. 3;

[0015] FIGS. 5D and 5E illustrate embodiments of how blocks within the service of FIG. 3 are decoupled from the service and other blocks within the service;

[0016] FIG. 5F illustrates an embodiment of the service of FIG. 3;

[0017] FIG. 5G illustrates an embodiment of multiple block threads corresponding to blocks within the service of FIG. 5F;

[0018] FIG. 6A illustrates one embodiment of a method that may be used with the NIO platform of FIGS. 1A, 2A, and 4A;

[0019] FIGS. 6B-8 illustrate more detailed embodiments of various steps of FIG. 6A;

[0020] FIGS. 9A and 9B illustrate another embodiment of the NIO platform of FIG. 2A from a platform perspective and a stack perspective, respectively;

[0021] FIG. 9C illustrates one embodiment of a hierarchy that may exist within the NIO platform of FIGS. 9A and 9B;

5 [0022] FIG. 9D illustrates one embodiment of a service that may exist within the NIO platform of FIGS. 9A and 9B;

[0023] FIG. 10 illustrates one embodiment of communications that may occur within the NIO platform of FIGS. 9A and 9B;

10 [0024] FIGS. 11A and 11B illustrate embodiments of a messaging module that may be used in the NIO platform of FIGS. 9A and 9B;

[0025] FIG. 12 illustrates one embodiment of the interaction between various components within the NIO platform of FIGS. 9A and 9B;

[0026] FIG. 13 illustrates one embodiment of interactions among various components of the NIO platform of FIGS. 9A and 9B when an instance is launched;

15 [0027] FIGS. 14 and 15 illustrate embodiments of methods that may be used during the interactions of FIG. 13;

[0028] FIGS. 16A-16D illustrate embodiments of methods that may be used by a service of the NIO platform of FIGS. 9A and 9B;

20 [0029] FIG. 16E illustrates one embodiment of a method that may be used by a block within the NIO platform of FIGS. 9A and 9B;

[0030] FIG. 17 illustrates one embodiment of an environment showing a user's perspective of the NIO platform 900 of FIGS. 9A and 9B;

[0031] FIG. 18 illustrates one embodiment of a workflow that may be used to create and configure a NIO platform;

25 [0032] FIG. 19A illustrates one embodiment of a service configuration environment within which a service is configured at runtime;

[0033] FIG. 19B illustrates one embodiment of a block configuration environment within which a block is configured at runtime;

30 [0034] FIGS. 20A and 20B illustrate embodiments of block classes that may be used within the block configuration environment of FIG. 19B;

[0035] FIG. 21 illustrates one embodiment of an environment within which configuration information is used to configure two blocks based on the same block class in different ways;

[0036] FIG. 22 illustrates one embodiment of an environment within which configuration information is used to configure two services based on the same service class in different ways;

5 [0037] FIG. 23 illustrates one embodiment of an environment with a base block class and multiple blocks based on the base block class;

[0038] FIGS. 24 and 25 illustrate embodiments of a service built using blocks from the environment of FIG. 23;

[0039] FIG. 26 illustrates an embodiment of an environment in which a NIO platform running the services of FIGS. 24 and 25 is coupled to external sources and/or destinations;

10 [0040] FIGS. 27 and 28 illustrate embodiments of a method that may be executed by the NIO platform of FIG. 26 to create and configure a block and a service, respectively;

[0041] FIG. 29 illustrates one embodiment of an environment in which a real time publishing system can operate;

15 [0042] FIG. 30 illustrates one embodiment of functionality provided by a NIO platform that forms part of the real time publishing system of FIG. 29;

[0043] FIG. 31 illustrates one embodiment of services that may be used within the NIO platform of FIG. 30;

[0044] FIGS. 32 and 33A illustrate more detailed embodiments of one of the services of FIG. 31;

20 [0045] FIG. 33B illustrates one embodiment of a method that may be used within the NIO platform of FIG. 31;

[0046] FIG. 34 illustrates one embodiment of functionality provided by a server that forms part of the real time publishing system of FIG. 29;

25 [0047] FIG. 35 illustrates one embodiment of a matrix that may be created by the real time publishing system of FIG. 29;

[0048] FIGS. 36A-36E illustrate various embodiments of tiles that may be used with the matrix of FIG. 35; and

[0049] FIGS. 37-40E illustrate embodiments of methods that may be used with the real time publishing system of FIG. 29.

DETAILED DESCRIPTION

[0050] The present disclosure is directed to a system and method for fully configurable real time processing. It is understood that the following disclosure provides many different embodiments or examples. Specific examples of components and arrangements are described below to simplify the present disclosure. These are, of course, merely examples and are not intended to be limiting. In addition, the present disclosure may repeat reference numerals and/or letters in the various examples. This repetition is for the purpose of simplicity and clarity and does not in itself dictate a relationship between the various embodiments and/or configurations discussed.

[0051] The present disclosure describes various embodiments of a neutral input/output (NIO) platform that includes a core that supports one or more services. While the platform itself may technically be viewed as an executable application in some embodiments, the core may be thought of as an application engine that runs task specific applications called services. The services are constructed using defined templates that are recognized by the core, although the templates can be customized to a certain extent. The core is designed to manage and support the services, and the services in turn manage blocks that provide processing functionality to their respective service. Due to the structure and flexibility of the runtime environment provided by the NIO platform's core, services, and blocks, the platform is able to asynchronously process any input signal from one or more sources in real time.

[0052] Referring to FIG. 1A, one embodiment of a NIO platform 100 is illustrated. The NIO platform 100 is configurable to receive any type of signal (including data) as input, process those signals, and produce any type of output. The NIO platform 100 is able to support this process of receiving, processing, and producing in real time or near real time. The input signals can be streaming or any other type of continuous or non-continuous input.

[0053] When referring to the NIO platform 100 as performing processing in real time and near real time, it means that there is no storage other than possible queuing between the NIO platform instance's input and output. In other words, only processing time exists between the NIO platform instance's input and output as there is no storage read and write time, even for streaming data entering the NIO platform 100.

[0054] It is noted that this means there is no way to recover an original signal that has entered the NIO platform 100 and been processed unless the original signal is part of the output or the NIO platform 100 has been configured to save the original signal. The original signal is received by the NIO platform 100, processed (which may involve changing and/or destroying the original signal), and output is generated. The receipt, processing, and

generation of output occurs without any storage other than possible queuing. The original signal is not stored and deleted, it is simply never stored. The original signal generally becomes irrelevant as it is the output based on the original signal that is important, although the output may contain some or all of the original signal. The original signal may be
5 available elsewhere (e.g., at the original signal's source), but it may not be recoverable from the NIO platform 100.

[0055] It is understood that the NIO platform 100 can be configured to store the original signal at receipt or during processing, but that is separate from the NIO platform's ability to perform real time and near real time processing. For example, although no long term (e.g.,
10 longer than any necessary buffering) memory storage is needed by the NIO platform 100 during real time and near real time processing, storage to and retrieval from memory (e.g., a hard drive, a removable memory, and/or a remote memory) is supported if required for particular applications.

[0056] The internal operation of the NIO platform 100 uses a NIO data object (referred to
15 herein as a niogram). Incoming signals 102 are converted into niograms at the edge of the NIO platform 100 and used in intra-platform communications and processing. This allows the NIO platform 100 to handle any type of input signal without needing changes to the platform's core functionality. In embodiments where multiple NIO platforms are deployed, niograms may be used in inter-platform communications.

[0057] The use of niograms allows the core functionality of the NIO platform 100 to
20 operate in a standardized manner regardless of the specific type of information contained in the niograms. From a general system perspective, the same core operations are executed in the same way regardless of the input data type. This means that the NIO platform 100 can be optimized for the niogram, which may itself be optimized for a particular type of input for a
25 specific application.

[0058] The basic structure of a niogram is designed to allow the information within the niogram to change as it moves through the NIO platform 100. For example, different types of information can be added to or removed from a niogram (e.g., a niogram containing a radio frequency identifier (RFID) value from an RFID tag can be modified to include an
30 expiration date of a corresponding product from which the tag was read). Furthermore, multiple niograms can be merged into a single niogram and a single niogram can be divided into multiple niograms. The NIO platform 100 can also create and destroy niograms as needed. The ability to create niograms, destroy niograms, and change the internal information of a niogram enables information to be enriched contextually in many different

ways by the NIO platform 100 without changing the niogram's basic structure and this enrichment can occur in real time or near real time.

[0059] The NIO platform 100 is designed to process niograms in a customizable and configurable manner using processing functionality 106 and support functionality 108. The processing functionality 106 is generally both customizable and configurable by a user. Customizable means that at least a portion of the source code providing the processing functionality 106 can be modified by a user. In other words, the task specific software instructions that determine how an input signal that has been converted into one or more niograms will be processed can be directly accessed at the code level and modified. Configurable means that the processing functionality 106 can be modified by such actions as selecting or deselecting functionality and/or defining values for configuration parameters. These modifications do not require direct access or changes to the underlying source code and may be performed at different times (e.g., before runtime or at runtime) using configuration files, commands issued through an interface, and/or in other defined ways.

[0060] The support functionality 108 is generally only configurable by a user, with modifications limited to such actions as selecting or deselecting functionality and/or defining values for configuration parameters. In other embodiments, the support functionality 108 may also be customizable. It is understood that the ability to modify the processing functionality 106 and/or the support functionality 108 may be limited or non-existent in some embodiments.

[0061] The support functionality 108 supports the processing functionality 106 by handling general configuration of the NIO platform 100 at runtime and providing management functions for starting and stopping the processing functionality. The resulting niograms can be converted into any signal type(s) for output(s) 104.

[0062] Referring to FIG. 1B, one embodiment of a NIO platform instance 101 illustrates a data path that starts when the input signal(s) 102 are received and continues through the generation of the output(s) 104. The NIO platform instance 101 is created when the NIO platform 100 of FIG. 1A is launched. A NIO platform may be referred to herein as a “NIO platform” before being launched and as a “NIO platform instance” after being launched, although the terms may be used interchangeably for the NIO platform after launch. As described above, niograms are used internally by the NIO platform instance 101 along the data path.

[0063] In the present example, the input signal(s) 102 may be filtered in block 110 to remove noise, which can include irrelevant data, undesirable characteristics in a signal (e.g.,

ambient noise or interference), and/or any other unwanted part of an input signal. Filtered noise may be discarded at the edge of the NIO platform instance 101 (as indicated by arrow 112) and not introduced into the more complex processing functionality of the NIO platform instance 101. The filtering may also be used to discard some of the signal's information while keeping other information from the signal. The filtering saves processing time because core functionality of the NIO platform instance 101 can be focused on relevant data having a known structure for post-filtering processing. In embodiments where the entire input signal is processed, such filtering may not occur. In addition to or as alternative to filtering occurring at the edge, filtering may occur inside the NIO platform instance 101 after the signal is converted to a niogram.

[0064] Non-discarded signals and/or the remaining signal information are converted into niograms for internal use in block 114 and the niograms are processed in block 116. The niograms may be converted into one or more other formats for the output(s) 104 in block 118, including actions (e.g., actuation signals). In embodiments where niograms are the output, the conversion step of block 118 would not occur.

[0065] Referring to FIG. 1C, one embodiment of a stack 120 is illustrated. In the present example, the NIO platform 100 interacts with an operating system (OS) 122 that in turn interacts with a device 124. The interaction may be direct or may be through one or more other layers, such as an interpreter or a virtual machine. The device 124 can be a virtual device or a physical device, and may be standalone or coupled to a network.

[0066] Referring to FIG. 1D, another embodiment of a stack 126 is illustrated. In the present example, the NIO platform 100 interacts with a higher layer of software 128a and/or a lower layer of software 128b. In other words, the NIO platform 100 may provide part of the functionality of the stack 126, while the software layers 128a and/or 128b provide other parts of the stack's functionality. Although not shown, it is understood that the OS 122 and device 124 of FIG. 1C may be positioned under the software layer 128b if the software 128b is present or directly under the NIO platform 100 (as in FIG. 1C) if the software layer 128b is not present.

[0067] Referring to FIG. 1E, in one embodiment, an environment 130 is illustrated with the NIO platform 100 of FIG. 1A. As illustrated, the NIO platform 100 supports instance to instance or device and system, which enables a NIO platform 100 to subscribe from and publish to one or more other platforms broadly or specifically to a single application logic/context requirement. The NIO platform 100 provides functionality needed to receive, process, and/or act on any input signal received from one or more external sources 132 (as

represented by arrow 102) and/or from the NIO platform 100 itself (as represented by arrow 134). Once the input signal is handled as defined by the configuration of the NIO platform 100, the input signal and/or other signals that may result from processing may be output to one or more external destinations 132 (as represented by arrow 104) and/or to the NIO platform 100 itself (as represented by arrow 134).

[0068] In the present embodiment, the input signals are not stored except for queuing (if needed), and no database or other permanent storage mechanism need be used for information handled by the NIO platform 100. Queuing may be handled in a variety of ways, including the use of memory/random access memory (RAM) and/or other mechanisms such as a persistence layer (e.g., an SQLite persistence layer). As the received input signals are handled in real time or near real time and the NIO platform 100 is not constrained by database access limitations, throughput can occur at a much higher rate than can be achieved in systems that rely on database access or that require storing data prior to processing.

[0069] In some embodiments, the NIO platform 100 allocates and de-allocates functionality as needed, thereby minimizing the platform's footprint.

[0070] The NIO platform 100 provides its functionality via a generic architecture that may be configured to address specific needs. Although the architecture may be implemented as an application specific integrated circuit (ASIC) or as another application specific embodiment, the architecture itself is configurable and, accordingly, highly flexible. Furthermore, the architecture is able to process signals in one platform instance and then pass the processed signals back to itself for further processing in the same or another platform instance. This ability to internally link to itself enables the architecture to take advantage of multiple concurrently executing related platform instances. With each platform instance being fully configurable, the architecture is able to rapidly process large amounts of information while also providing highly customizable outputs.

[0071] The NIO platform 100 may be implemented in a manner that is relatively OS independent. For example, the NIO platform 100 may be implemented using a language such as Python. To provide additional OS neutrality, design decisions may include avoiding the use of library calls that are OS specific and/or may avoid the inclusion of modules that are OS specific.

[0072] The NIO platform 100 may provide self-awareness functional capability through services configured to support platform instances, advanced dynamic context artificial intelligence, and/or system monitoring. With respect to instances, pre-configured services may be based on specific indices relating to signal type and source, level of awareness

function, and actions. The services may be utilized for device and/or system diagnostics and quality control. With respect to advanced dynamic context artificial intelligence, custom developed composite context within a process resident within the NIO platform 100 may be specific to a use case, business process, system of devices or signal producers, or a single device specification.

[0073] With respect to system monitoring, the NIO platform 100 may be used to monitor the state or condition of itself (i.e., the NIO platform 100) as a self-aware system. To accomplish this monitoring, niograms may be generated that correspond to the current state of the NIO platform 100. Details contained in such niograms may range from the amount of central processing unit (CPU) usage of the NIO platform 100 to an error generated from one aspect of the NIO platform 100. These niograms can then be processed by services and combined with internal actions to create a self-aware and proactive system monitoring solution. Additionally, a separate instance of the NIO platform 100 can be set up to apply this system monitoring logic and the niograms from the internal monitoring service can be sent there.

[0074] The NIO platform 100 may be stored and executed on the device 124. The NIO platform 100 may be an application residing on the device 124 and/or may be embedded in the device 124. Examples of the device 124 include single-board computing (SBC) and onboard computing (OBC) platforms, cellular telephones (including smart phones), personal digital assistants (PDAs), netbooks, tablets, laptops, desktops, workstations, servers, equipment (e.g., manufacturing equipment, monitoring equipment, and security equipment), home appliances (e.g., refrigerators, stoves, ovens, coffee makers, stereos, and televisions), vehicles and other mobile systems (e.g., air, land, sea, and/or space vehicles, whether manned or autonomous), and any other device that is able to execute instructions and support some or all of the architecture of the NIO platform 100.

[0075] Communications to and from the NIO platform 100 may be direct (e.g., via a peer-to-peer network, an ad hoc network, or using a direct connection), indirect, such as through a server or other proxy (e.g., in a client-server model or a wireless network), or may use a combination of direct and indirect communications.

[0076] Referring to FIG. 1F, one embodiment of an environment 140 is illustrated where the functionality provided by the NIO platform 100 is distributed as represented by NIO platforms 100a and 100b on devices 124a and 124b, respectively. Although only two NIO platforms 100a and 100b are shown, it is understood that the functionality may be distributed across many devices. The distributed NIO platforms 100a and 100b may communicate with

one another (as represented by arrows 142 and 144). Each distributed NIO platform 100a and 100b may communicate with the external source/destination 132, only a particular one of the NIO platforms 100a and 100b may be configured for communication with the external source/destination 132, or one NIO platform 100a or 100b may be configured to receive communications from the external source 132, while the other of the NIO platforms may be configured to send communications to the external destination 132.

[0077] In another embodiment of FIG. 1F, each NIO platform 100a and 100b may be a complete platform with full functionality, and may be configured to communicate with one another and/or with the external source/destination 132 in order to accomplish designated tasks. In such embodiments, one NIO platform 100a or 100b may be offline unless needed (e.g., if the other platform fails or becomes overloaded). In other embodiments, although each NIO platform 100a and 100b provides full functionality, some functionality on one or both platforms may not be used. This enables the same NIO platform to be used on multiple devices while still allocating particular functionality to one or more specific devices.

[0078] Referring to FIG. 1G, one embodiment of an environment 150 is illustrated where some or all of the functionality provided by the NIO platform 100 is provided by one or more storage and/or processing systems 152 that provides services from one or more remote locations, such as is provided by cloud computing. It is understood that the storage and/or processing systems 152 may have distributed control, with functionality provided by different entities and combined within the NIO platform 100.

[0079] Referring to FIG. 1H, one embodiment of an environment 160 is illustrated where some or all of the functionality provided by the NIO platform 100 is used only within the device 124. In the present embodiment, the device 124 does not communicate with external source/destinations 132 unless needed for purposes such as installation, maintenance, and/or configuration.

[0080] Referring to FIG. 1I, one embodiment of an environment 165 is illustrated where multiple NIO platforms 100a and 100b are running on a single device 124. Although only two NIO platforms 100a and 100b are shown, it is understood that many instances of the NIO platform 100 may be deployed on a single device. The NIO platforms 100a and 100b may communicate with one another (as represented by arrows 142 and 144). Each distributed NIO platform 100a and 100b may communicate with the external source/destination 132, only a particular one of the NIO platforms 100a and 100b may be configured for communication with the external source/destination 132, or one NIO platform 100a or 100b may be configured to receive communications from the external source 132, while the other

of the NIO platforms may be configured to send communications to the external destination 132.

5 [0081] It is understood that the environments of Figs. 1E-1I may be combined in various ways. For example, the functionality of the NIO platform 100 may be distributed between the device 124a of FIG. 1F and the cloud 152 of FIG. 1G.

[0082] Referring to FIG. 1J, one embodiment of a system 170 is illustrated. The system 170 is one possible example of a portion or all of the device 124 of FIGS. 1D-1F, 1H, and 1I, and/or the external source(s)/destinations 132 of FIGS. 1E-1G and 1I. The system 170 may include a controller (e.g., a processor/central processing unit ("CPU")) 172, a memory unit 10 174, an input/output ("I/O") device 176, and a network interface 178. The components 172, 174, 176, and 178 are interconnected by a data transport system (e.g., a bus) 180. A power supply (PS) 182 may provide power to components of the system 170 via a power transport system 184 (shown with data transport system 180, although the power and data transport systems may be separate).

15 [0083] It is understood that the system 170 may be differently configured and that each of the listed components may actually represent several different components. For example, the CPU 172 may actually represent a multi-processor or a distributed processing system; the memory unit 174 may include different levels of cache memory, main memory, hard disks, and remote storage locations; the I/O device 176 may include monitors, keyboards, and the 20 like; and the network interface 178 may include one or more network cards providing one or more wired and/or wireless connections to a network 186. Therefore, a wide range of flexibility is anticipated in the configuration of the system 170, which may range from a single physical platform configured primarily for a single user or autonomous operation to a distributed multi-user platform such as a cloud computing system.

25 [0084] The system 170 may use any operating system (or multiple operating systems), including various versions of operating systems provided by Microsoft (such as WINDOWS), Apple (such as Mac OS X), UNIX, and LINUX, and may include operating systems specifically developed for handheld devices (e.g., iOS, Android, Blackberry, and/or Windows Phone), personal computers, servers, and other computing platforms depending on the use of 30 the system 170. The operating system, as well as other instructions (e.g., for telecommunications and/or other functions provided by the device 124), may be stored in the memory unit 174 and executed by the processor 172. For example, if the system 170 is the device 124, the memory unit 174 may include instructions for providing the NIO platform 100 and for performing some or all of the methods described herein.

[0085] The network 186 may be a single network or may represent multiple networks, including networks of different types, whether wireless or wireline. For example, the device 124 may be coupled to external devices via a network that includes a cellular link coupled to a data packet network, or may be coupled via a data packet link such as a wide local area network (WLAN) coupled to a data packet network or a Public Switched Telephone Network (PSTN). Accordingly, many different network types and configurations may be used to couple the device 124 with external devices.

[0086] Referring to FIG. 1K, one embodiment of a system 180 is illustrated. The system 180 is another possible example of a portion or all of the device 124 of FIGS. 1D-1F, 1H, and 1I, and/or the external source(s)/destinations 132 of FIGS. 1E-1G and 1I. The system 180 may be similar to the system 170 of FIG. 1J, but may contain only the CPU 172 and memory 174. Other components, such as the power supply 182 and I/O 176, may be external. In the present example, the system 180 may have no network capability. In other embodiments, the system 180 may access a network such as the network 186 using a network interface such as the network interface 178.

[0087] Referring to FIG. 1L, one embodiment of a system 190 is illustrated. The system 190 is another possible example of a portion or all of the device 124 of FIGS. 1D-1F, 1H, and 1I, and/or the external source(s)/destinations 132 of FIGS. 1E-1G and 1I. The system 190 may be similar to the system 170 of FIG. 1J, but may contain only the CPU 172. The memory 174 and other components, such as the power supply 182 and I/O 176, may be external. For example, the system 190 may rely on an external drive. In the present example, the system 190 may have no network capability. In other embodiments, the system 190 may access the memory 174 via a network such as the network 186 using a network interface such as the network interface 178.

[0088] Referring to FIG. 2A, a NIO platform 200 illustrates a more detailed embodiment of the NIO platform 100 of FIG. 1A. In the present example, the NIO platform 200 includes two main components: service classes 202 for one or more services that are to provide the configurable processing functionality 106 and core classes 206 for a core that is to provide the support functionality 108 for the services. Each service corresponds to block classes 204 for one or more blocks that contain defined task specific functionality for processing niograms. The core includes a service manager 208 that will manage the services (e.g., starting and stopping a service) and platform configuration information 210 that defines how the NIO platform 200 is to be configured, such as what services are available when the instance is launched.

[0089] When the NIO platform 200 is launched, a core and the corresponding services form a single instance of the NIO platform 200. It is understood that multiple concurrent instances of the NIO platform 200 can run on a single device (e.g., the device 124 of FIG. 1D). Each NIO platform instance has its own core and services. The most basic NIO platform instance is a core with no services. The functionality provided by the core would exist, but there would be no services on which the functionality could operate. Because the processing functionality of a NIO platform instance is defined by the executable code present in the blocks and the services are configured as collections of one or more blocks, a single service containing a single block is the minimum configuration required for any processing of a niogram to occur.

[0090] It is understood that FIG. 2A illustrates the relationship between the various classes and other components. For example, the block classes are not actually part of the service classes, but the blocks are related to the services. Furthermore, while the service manager is considered to be part of the core for purposes of this example (and so created using the core classes), the core configuration information is not part of the core classes but is used to configure the core and other parts of the NIO platform 200.

[0091] With additional reference to FIGS. 2B and 2C, embodiments of two NIO platform instances 222 and 224, respectively, are illustrated as hierarchies that are created when a NIO platform, such as the NIO platform 200 of FIG. 2A, is launched. The NIO platform instance 222 is an instance of a NIO platform #1 and the NIO platform instance 224 is an instance of a NIO platform #2. The NIO platforms #1 and #2 may be different platforms or may be different configurations of the same platform. Multiple NIO platform instances can be launched and executed concurrently, either on a single device or on separate devices, and each instance can be separately closed. The NIO platform instances can be based on the same NIO platform, in which case the instances will have identical functionality when launched. Alternatively, the NIO platform instances can be based on different NIO platform configurations, in which case the instances will have different functionality when launched with the functionality of a particular instance being based on the particular configuration of the underlying NIO platform.

[0092] Each NIO platform instance 222 and 224 contains a core 228 and one or more services 230, and each service 230 contains one or more blocks 232. Each NIO platform instance 222 and 224 may have different numbers of services 230 running and those services 230 may use different numbers of blocks 232. The services 230 running within a single NIO platform instance 222 or 224 can be identical in functionality, different in functionality,

related (e.g., one service may perform one task in a series of tasks and then another service may perform the next task), and/or unrelated.

[0093] This hierarchical structure enables the configurability of the NIO platform 200 to be accessed at different levels, each of which offers a different level of granularity in the configuration process. Each NIO platform instance 222 and 224 can be configured by adding, removing, and/or modifying the services 230 that form the instance. A service 230 can be configured by adding, removing, and/or modifying the blocks 232 that form the service 230, by modifying the arrangement of the blocks 232 to change the data path through the blocks 232, by setting various configuration parameters corresponding to the service, and/or by enabling the service 230 to use functionality provided by the core 228. In some embodiments, a service 230 can also be customized by adding, removing, and/or modifying the instructions (e.g., the source code) contained in the corresponding service class 202. A block 232 can be customized by adding, removing, and/or modifying the instructions (e.g., the source code) contained in the corresponding block class 204. A block 232 can also be configured by setting various configuration parameters corresponding to the block.

[0094] It is understood that, once configured using the platform configuration information 210, the NIO platform 200 may be reduced to a minimal footprint. This may, in some embodiments, involve removing or otherwise limiting the configurable and/or customizable functionality, with only defined service classes 202 and/or block classes 204 remaining with respect to the processing functionality 106. Similarly, unused modules and/or core components (discussed later) may also be removed from the support functionality 108. Even if the NIO platform 200 is made to be no longer customizable or configurable by a user, it is understood that updates and other changes may still be made to existing service classes 202 and/or block classes 204 in some embodiments.

[0095] Removing and/or otherwise limiting the configurability and/or customizability may be used to ensure that the existing functionality is not changed once defined as desired. It is understood that the support functionality 108 would still remain in the NIO platform 100 to manage the services 230 and/or blocks 232 in a running instance. Accordingly, the NIO platform 200 may be reduced to a more conventional application-style format for distribution, installation, and/or use for a targeted purpose, although services 230 and/or blocks 232 would still run via the core environment provided by the NIO platform 200.

[0096] With additional reference to FIG. 3, one embodiment of a service 230 (based on one of the service classes 202 of FIG. 2A as launched in one of the instances 222 of FIG. 2B or 224 of FIG. 2C) is illustrated with multiple blocks 232a, 232b, ..., and 232M (based on

block classes 204), with M being the total number of blocks contained by the service 230. The service 230 may be viewed as the framework that is responsible for assembling the contained group of blocks (e.g., blocks 232a-232M) into a workflow to define the logical path that a niogram within the service 230 will follow. While the blocks 232a-232M may not be literally linked within the service 230, the service 230 may manage the order in which blocks 232 are called for processing and, in so doing, direct a niogram within the service 230 to the appropriate next block. Starting the service 230 enables an instance of the NIO platform 200 to use the blocks' functionality to process niograms. Accordingly, the value of M is greater or equal to one, as a service 230 that has no blocks 232 would provide no functionality. It is understood that the service 230 may include blocks that are not always used, such as blocks that are only executed if a particular conditional branch occurs.

[0097] While entirely custom service classes 202 and block classes 204 may be created and used with the NIO platform 200 for custom services 230 and blocks 232, the functionality of some or all services 230 and/or blocks 232 may be predefined, either in whole or in part. For example, a service 230 directed to receiving and forwarding a particular type of message may be defined in the corresponding service class 202 with the exception of configuration parameters for source and destination. Starting the service 230 would automatically start the blocks 232 to be used with the service and provide an instance of the NIO platform 200 with the service's receive and forward functionality. The source and destination can be added via the platform configuration information 210 or in another way, such as through the use of commands issued to the service 230 and/or blocks 232 through an interface.

[0098] In a more complex example, a largely pre-defined service 230 may be modifiable by defining the behavior of one or more blocks classes 204 for the blocks 232 within the service 230, either by providing original code for the block class 204 or by selecting a predefined block class 204. For example, the service 230 may process the received message and forward the message to one of various different destinations depending on the message's contents. This conditional functionality may be achieved by writing instructions for inclusion in a new block class 204 for a new block to be used in the service 230, by writing instructions for inclusion in one of the block classes 204 for a block 232 that is already part of the service 230, and/or by configuring an existing block class 204 that already enables such conditional functionality.

[0099] A single defined service 230 may be used in multiple platform instances and a single defined block 232 may be used in multiple services. It is understood that modifying a

service 230 by customizing and/or reconfiguring the service class 202/service 230 may require a restart of any platform instances currently using that service 230 for the modification to take effect, as the platform instances may otherwise continue running the previous version of the service 230 that was instantiated when launched. Similarly, modifying a block 232 by customizing and/or reconfiguring the block class 204/service 232 may require a restart of any services 230 currently using that block 232 for the modification to take effect, as the services 230 may otherwise continue running the previous version of the block 232 that was instantiated when launched.

[0100] With additional reference to FIGS. 4A and 4B, another embodiment of the NIO platform 200 of FIG. 2A is illustrated as a NIO platform 400 prior to being launched (FIG. 4A) and as a NIO platform instance 402 after being launched (FIG. 4B). FIG. 4A illustrates the NIO platform 400 with core classes 206, service classes 202, block classes 204, and configuration information 210 that are used to create and configure the core 228, services 230a-230N, and blocks 232a-232M of the NIO platform instance 402. It is understood that, although not shown in FIG. 4B, the core classes 206, service classes 202, block classes 204, and configuration information 210 generally continue to exist as part of the NIO platform instance 402.

[0101] Referring specifically to FIG. 4B, the NIO platform instance 402 may be viewed as a runtime environment within which the core 228 creates and runs the services 230a, 230b, ..., and 230N. Each service 230a-230N may have a different number of blocks. For example, service 230a includes blocks 232a, 232b, and 232c. Service 230b includes a single block 232d. Service 230N includes blocks 232e, 232f, ..., and 232M.

[0102] One or more of the services 230a-230N may be stopped or started by the core 228. When stopped, the functionality provided by that service will not be available until the service is started by the core 228. Communication may occur between the core 228 and the services 230a-230N, as well as between the services 230a-230N themselves.

[0103] In the present example, the core 228 and each service 230a-230N is a separate process from an operating system/hardware perspective. Accordingly, the NIO platform instance 402 of FIG. 4B would have N+1 processes running, and the operating system may distribute those across multi-core devices as with any other processes. It is understood that the configuration of particular services may depend in part on a design decision that takes into account the number of processes that will be created. For example, it may be desirable from a process standpoint to have numerous but smaller services in some embodiments, while it may be desirable to have fewer but larger services in other embodiments. The

configurability of the NIO platform 400 enables such decisions to be implemented relatively easily by modifying the functionality of each service 230a-230N.

[0104] In other embodiments, the NIO platform instance 402 may be structured to run the core 228 and/or services 230a-230N as threads rather than processes. For example, the core
5 228 may be a process and the services 230a-230N may run as threads of the core process.

[0105] Referring to FIG. 4C, one embodiment of a diagram 410 illustrates the core process 228 and the services 230a, 230b, and 230N of FIG. 4B plotted against an axis 412 representing time. More specifically, the diagram 410 illustrates examples of stop and start times for the core process 228 and services 230a, 230b, and 230N relative to one another,
10 with multiple services running simultaneously at some times and with no services running at other times.

[0106] At time t₀, the NIO platform instance 402 is not running. At time t₁, the NIO platform instance 402 is started with the core process 228 as indicated by line 414. The core process 228 generally remains running the entire time the NIO platform instance 402 is running, as the core process 228 is needed for the NIO platform instance 402 to exist. It is
15 understood that, if the core process 228 crashes, the services 230a, 230b, and 230N may continue to run as long as they are separate processes from the core process 228. However, the core process 228 can no longer be used to communicate with the service processes or otherwise control them, and so the NIO platform instance 402 no longer exists when the core
20 process ends even if the service processes continue to run. If the services 230a, 230b, and 230N are threads of the core process 228, then the services 230a, 230b, and 230N will typically end if the core process 228 crashes.

[0107] At time t₂, the service 230a is started as indicated by line 416. At time t₃, the service 230a is stopped, but the core process 228 continues running even though no services
25 are running at this time. At time t₄, the service 230N is started as indicated by line 422. At time t₅, the service 230a is restarted as indicated by line 418. At time t₆, the service 230b is started as indicated by line 420. At time t₇, the service 230b is stopped. At time t₈, the services 230a and 230N are stopped. At time t₉, the core process 228 is stopped, which shuts down the NIO platform instance 402. If the core process 228 is stopped while services are
30 still running, the core process 228 may shut down the running services before stopping.

[0108] Referring to FIGS. 5A-5C, embodiments of the service 230 of FIG. 3 are illustrated. Any service 230 defined in the processing functionality 106 of FIG. 2A may be altered by adding, removing, and/or modifying one or more of the blocks 232 that form that service. Each block 232 within the service 230 contains instructions. The instructions within

a particular block 232 may be simple or complex. Accordingly, a particular service 230 that is configured to provide a particular function may be implemented in many different ways depending on the blocks 232 used and the instructions contained within each block 232. In this respect, the service 230 may be viewed as an executable program that can be written in many different ways and then executed by the NIO platform instance 402.

[0109] Referring specifically to FIG. 5A, one embodiment of the service 230 is formed from a single block 232. This is the minimal requirement for a service 230 having functionality. Referring specifically to FIG. 5B, another embodiment of the service 230 is formed from two blocks 232a and 232b that are linked together to form a “chain” of blocks. In this embodiment, block 232a feeds into block 232b. It is understood that the blocks 232a and 232b may not be literally linked within the service 230, but that the service 230 “links” the blocks from an operational perspective by directing that a niogram that is an output of block 232a becomes an input to block 232b.

[0110] Referring specifically to FIG. 5C, yet another embodiment of the service 230 is illustrated with a more complicated block structure. Although the service 230 is made up of a “chain” of blocks 232a-232e, it is understood that the blocks 232a-232e may not be executed in a linear manner. In other words, the blocks 232a-232e may include instructions that vary the execution order within the service 230. For example, a block may contain an instruction that is conditional, with the next block being selected and executed depending on how the condition is met. Furthermore, it is understood that the blocks 232a-232e need not be literally arranged in an order of execution, and that the platform configuration information 210 is used to define the service 230 and, in some embodiments, the various dependencies between the blocks 232a-232e.

[0111] The blocks 232a-232e and the arrangement of those blocks may be optimized for use with a particular service 230. For example, the blocks 232a-232e may be optimized for a particular pattern of execution that corresponds to the service’s processing needs.

[0112] The blocks 232a-232e of FIG. 5C serve to illustrate various possible combinations that may occur within the service 230, although it is understood that other combinations may be used that are not illustrated. In the present embodiment, block 232a is able to call or otherwise return to itself (as indicated by arrow 502). Block 232a feeds into block 232b (as indicated by arrow 504) and/or block 232c (as indicated by arrow 506). For example, block 232a may split a niogram into two niograms. One niogram may be passed to block 232b and the other niogram may be passed to block 232c. In another example, the block 232a may contain instructions for a conditional branch and an evaluation of the niogram may determine

whether the niogram is sent to block 232b or block 232c. Alternatively, the block 232a may send the niogram to both block 232b and block 232c. One or both of the blocks 232b and 232c may then filter the niogram after receiving it or another block (not shown) may be added before one or both of the blocks 232b or 232c to filter the niogram before it reaches the
5 block.

[0113] One or both blocks 232b and 232c may feed into block 232d (as indicated by arrows 508 and 510, respectively), which illustrates how multiple blocks can feed into a single block. For example, a niogram from each of blocks 232b and 232c may be passed to block 232d, where the niograms are combined or otherwise processed. In another example, a
10 niogram may be received from only one of blocks 232b and 232c, depending on factors such as the processing within those blocks and whether a conditional statement executed in block 232a passed a niogram to only one of the blocks 232b and 232c. Block 232d feeds into block 232e (as indicated by arrow 512).

[0114] It is understood that a block may be called without being a link in the overall chain. For example, block 232a may use block 232b to obtain a value (e.g., perform a calculation and return a value) and then continue execution with block 232c after the value is returned. In this case, arrow 508 would not exist. The manner in which the block 232b returns the value depends on the particular implementation of the blocks. As will be described below, the blocks may be implemented so that the output of a block is routed to
15 another block by the service, may be implemented to support direct calls between blocks, or may be implemented with one or more commands used to access a block. Regardless of the implementation, from this perspective, the blocks 232a-232e may be viewed as modular components of a program that can be called within the program based on the desired functionality provided by a particular block.

[0115] It is understood that, although complex arrangements of blocks 232 may be used to create the service 230, it may be desirable in some embodiments to arrange the blocks in a single chain to create an optimized execution path. Furthermore, the service 230 of FIG. 5C may be created as a single block in some embodiments.

[0116] While input to the service 230 is often fed first to the initial block in a chain (e.g.,
30 block 232a), input may be handled by any block within the service 230 based on how the service 230 and/or the blocks 232a-232e are configured. For example, input may enter block 232b, and be passed to block 232a and/or 232d. In turn, block 232b may remain as part of the output path for block 232a. This allows a great deal of flexibility within a particular service 230.

[0117] It is understood that when the present disclosure refers to a service 230 receiving input, the input is actually being received by a block 232 within the service 230. Similarly, output is produced by a block 232 within the service 230. However, as the service 230 includes the blocks 232 used within the service 230, the service 230 may be described generally as receiving input and producing output.

[0118] Accordingly, a single service 230 may be configured in many different ways. While optimizations may indicate a particular desirable service configuration for a particular use, a service 230 can be fine-tuned based on factors such as the desired functionality of the service and the characteristics of the device 124 on which the NIO platform 200 is to run (e.g., memory limitations such as cache size and throughput, processor speed, number of processor cores, and/or data rates of input data streams). This configurability enables the NIO platform 200 to be optimized for many different environments if needed by changing the structure of the services 230 and blocks 232 when possible, rather than the core 228.

[0119] Referring to FIGS. 5D and 5E, embodiments of services 230a and 230b, respectively, are illustrated. In the present embodiments, each block 232 is decoupled from both the service 230 by which it is being used and from the other blocks 232 that are used by the service 230. Each block 232 is a self-contained entity that asynchronously receives input, processes that input in a manner defined by the block's internal code, and produces an output. The block's asynchronicity is possible because the block 232 has no awareness that it is part of a service 230 or that it is associated with other blocks as part of a functional chain. The block 232 simply handles data as it arrives and creates an output.

[0120] This decoupling of the block's functionality from other blocks 232 in the service 230 enables each block 232 to be reused without having to alter the code within the block 232 itself. For example, if a block 232 was configured to call a specific destination block after finishing its own processing, this call would likely have to be modified for each service 230 in which the block 232 was used and also if the block 232 was moved to a different location of the functional path within the same service 230. Instead, because the block 232 is decoupled and the service 230 is configured to manage the block's output, the block 232 can remain unchanged for different services or during moves within the same service even though its output may be routed differently.

[0121] By making the service 230 responsible for the data flow path between blocks 232, the actual block structure is simplified and need not be concerned with anything except its own processing and, at the end, notifying the service 230 that output is available. Decoupling the block's functionality from the service 230 and other blocks 232 in this manner also

enables the block to be swapped out for another block with different functionality, which makes service modification easier.

[0122] This decoupling is illustrated in FIGS. 5D and 5E, which represent embodiments of services 230a and 230b, respectively. Each service 230a and 230b has the same number of blocks 232. However, the order of the blocks is different and the final output is different. For example, the service 230a of FIG. 5D has the following functional chain: block 232a (input), block 232b (filter), block 232c (convert to niograms), block 232d (compare niograms), block 232e (combine niograms), and block 232f (output as actuation). In contrast, the service 230b of FIG. 5E has the following functional chain: block 232a (input), block 232b (filter), block 232c (convert to niograms), block 232e (combine niograms), block 232d (compare niograms), and block 232g (output as email).

[0123] These functional chains are illustrated below in Table 1 (for service 230a) and Table 2 (for service 230b) in the form of source and destination information. For example, each service 230a and 230b may manage a list, table, or other data storage structure with such information. It is understood that the actual order of the blocks in the data storage structure may vary as long as the services 230a and 230b can tell which destination block(s) are to be serviced with output from a particular source block.

[0124]

Source Block	Destination Block
Block 232a	Block 232b
Block 232b	Block 232c
Block 232c	Block 232d
Block 232d	Block 232e
Block 232e	Block 232f

Table 1

[0125]

Source Block	Destination Block
Block 232a	Block 232b
Block 232b	Block 232c
Block 232c	Block 232e
Block 232e	Block 232d
Block 232d	Block 232g

Table 2

[0126] As can be seen, the order of execution of blocks 232d and 232e in the service 230a has been altered in the service 230b. Because the blocks are decoupled, the alteration can be accomplished simply by modifying the order of blocks (e.g., the source/destination pairs) in the service 230b and requires no changes to the blocks themselves or to the blocks
5 preceding or succeeding the blocks 232d and 232e.

[0127] Another difference between the two services 230a and 230b is that the output block of FIG. 5E sends an email rather than the actuation of FIG. 5D. One way to perform this change in functionality is to swap out the block 232f of FIG. 5D for the block 232g of FIG. 5E. This may be done when the blocks 232f and 232g contain only the specific
10 functionality required for their purpose. Another way to perform this change in functionality would be to modify the block 232f to give it the desired functionality of the block 232g or to create a new block with the desired functionality. Yet another way would be to set configuration parameters for the block 232f to select the desired functionality, assuming that the block 232f already contains the functionality for producing both actuations and email as
15 outputs. Regardless of the way in which the functionality is implemented for the blocks 232f and 232g, the decoupled aspect of the blocks means that the preceding block 232e (FIG. 5D) or 232d (FIG. 5E) is unaware of the changes and is not affected.

[0128] With additional reference to FIGS. 5F and 5G, embodiments of a service 230 (FIG. 5F) and multiple block threads (FIG. 5G) are illustrated. In the present example, the
20 service 230 includes five blocks 232a-232e, which may also be referred to as Blocks 1-5, respectively. Due to the asynchronous nature of the blocks 232a-232e, some or all of the blocks 232a-232e in the service 230 can process niograms concurrently. In other words, different niograms may be concurrently processed by the different blocks 232a-232e of the single service 230. Furthermore, a single block 232a-232e may process multiple niograms
25 concurrently due to threading.

[0129] The ability to concurrently process multiple niograms may increase the throughput of the service 230. The ability to replicate one or more of the blocks 232a-232e by executing the block's functionality in multiple concurrent threads enables the service 230 to dynamically scale its processing capacity as needed. For example, this allows the service 230
30 to increase its processing capacity to handle input surges and then reduce its processing capacity once the surge subsides.

[0130] Each block 232a-232e takes a period of time (e.g., a second or a fraction of a second) to process a single signal or niogram, although it is understood that the actual rate of a particular block may vary depending on such factors as the complexity of the block, the

content of the niograms, the processor speed of the device on which the service 230 is running, the number of processor cores, and/or the size of the cache. For purposes of example, the block 232a is able to process a signal and produce an output of a niogram in 0.5 seconds, the block 232b is able to process a niogram in 1.5 seconds, the block 232c is able to process a niogram in 1.0 seconds, the block 232d is able to process a niogram in 0.5 seconds, and the block 232e is able to process a niogram in 1.0 seconds.

[0131] If only one thread can exist for each block, the block that takes the longest time to process a signal or niogram would determine the throughput of the service 230, assuming that the input rate to the service 230 is higher than the slowest block's processing rate. In this case, the block 232b would be the bottleneck and would limit the service's throughput to one niogram every 1.5 seconds if only one thread exists for each block 232a-232e. The blocks 232c and 232e would be bottlenecks if not for the block 232b. The blocks 232a and 232d would not be bottlenecks as they can process more quickly than the other blocks. With only one thread for each block 232a-232e, the service 230 would be processing five niograms every 1.5 seconds (e.g., one niogram in each block 232a-232e) with an output of one niogram every 1.5 seconds, but this would not be efficient as four of the five blocks would spend time waiting on block 232b.

[0132] One way to address this problem would be to construct the blocks 232a-232e so that all of the blocks in the service 230 process niograms at approximately the same rate. However, this is a relatively inefficient way to handle the problem as any changes might affect the processing rate of one or more blocks and each new service would have to be balanced in this manner. This approach would complicate the design and implementation of services within the NIO platform 200.

[0133] Accordingly, as shown in FIG. 5G, the NIO platform 200 is designed so that the service 230 can make use of concurrent threads for a single block, which allows the service 230 to dynamically scale its processing capacity as needed. Continuing the previous example, Block 1 is executing at a speed of 0.5 seconds per niogram, so it is processing two signals per second and outputting two niograms per second. It is assumed that Block 1 outputs its niograms at a steady rate of one niogram every 0.5 seconds. This and other simplifications, such as a block execution speed measured in seconds, are used in the present example for purposes of illustration and it is understood that inputs may be irregular, processing may occur much more rapidly, and threading within an actual executing service may be far more complicated.

[0134] Block 2 has three threads 1, 2, and 3 running. It is noted that each thread of Block 2 is only processing one niogram and a total of three concurrent threads will be needed to process the three niograms output by Block 1 during the 1.5 second processing time required by Block 2. Block 3 has two threads 1 and 2 running. Block 4 has only thread 1 running.
5 Block 5 has two threads 1 and 2 running.

[0135] If the input lessens, the number of threads may lessen. If the input rate drops to one signal or less every 1.5 seconds, only a single thread of each block would exist. If the input rate increases, the number of threads may increase, and Blocks 1 and 5 may also have additional threads. Accordingly, the NIO platform 200 is able to dynamically scale its
10 processing capacity to handle scenarios such as varying input rates.

[0136] In cases where race conditions or similar thread collision issues may present a problem between concurrently executing threads, resource locking and unlocking may be used. For example, if the Block 5 threads 1 and 2 are to write to a file as output, it may be undesirable to allow them both to write to the file concurrently. Accordingly, one thread may
15 acquire a lock, write to the file, and then release the lock. The other thread cannot write to the file until the lock is released. In other cases, race conditions or other thread collision issues may not be a concern.

[0137] It is noted that while intra-service interactions are decoupled in the present embodiment, certain blocks 232 may be aware of sources and/or destinations outside of the
20 service 230. For example, the block 232a may be configured to poll or otherwise pull data from a data source and would need to be aware of the source (e.g., the source's network address) in order to accomplish its purpose. Likewise, the blocks 232e and 232f may be configured with an awareness of the destination.

[0138] It is understood that some embodiments may use blocks 232 that are implemented
25 in such a way that the blocks are responsible for calling the next block. In other words, the blocks 232 may be coupled, with a block being aware of the next block to which it should pass its output. For example, a block 232 may contain instructions identifying another block to which output is to be sent. The block 232 may then execute those instructions when output is available and directly call the next block without needing the service 230 to handle the
30 output. The blocks 232 may still execute in an asynchronous manner in such embodiments. While this implementation is within the scope of the present application, it is not used in this embodiment due to the additional complexity and lack of reusability that would result from such an implementation.

[0139] Referring to FIG. 6A, a method 600 illustrates one embodiment of a process that may be executed by the NIO platform instance 101 of FIG. 1B or the NIO platform instance 402 of FIG. 4B. In step 602, a platform instance is launched. In step 604, one or more services and/or blocks may be configured. For example, a block 232 may be configured with customized code and/or configuration parameters, and a service 230 may be configured by assigning various blocks to it and defining a data flow among those blocks, and/or by setting various configuration parameters. In step 606, one or more of the configured service(s) 230 and block(s) 232 are started. In step 608, input data is processed as defined by the service(s) 230 and block(s) 232 in the instance. In other embodiments, one or more services and/or blocks may be configured before the platform instance is launched.

[0140] Referring to FIG. 6B, a method 610 illustrates a more detailed embodiment of step 604 of FIG. 6A. In step 612, one or more blocks 232 are selected or defined for use in the service 230. For example, predefined blocks 232 may be selected for use in their current form or may be selected and modified. Custom blocks 232 may also be created by inserting instructions into a blank block and saving the block as a new block.

[0141] In step 614, the service 230 is defined by creating a new service and configuring it with information identifying the blocks 232 to be used and the arrangement (e.g., the data flow) of those blocks 232. Configuration parameters may also be set for the service 230. In other embodiments, an existing service 230 may be configured by setting configuration parameters associated with the service 230 and/or by adding, removing, and/or modifying blocks, as well as by modifying the data flow defined for the blocks 232. In step 616, the service 230 is saved.

[0142] It is understood that if an existing service 230 is to be used without modification, the method 610 would simply involve selecting that service 230 for use, as the corresponding blocks 232 would already be associated with the service 230. In some embodiments, the service 230 may be created before the blocks 232 are selected or defined, and the service may then be updated with blocks and other information.

[0143] Referring to FIG. 7, a method 700 illustrates a more detailed embodiment of steps 602 and 606 of FIG. 6A. In step 702, a platform instance is launched, which includes launching a core process. In step 704, the core process discovers the platform instance's services 230 using the available configuration information. Among other items, the configuration information identifies whether any services 230 are configured to auto-start as represented by step 706.

[0144] If one or more services 230 are configured to auto-start, the method 700 moves to step 708. In step 708, the service or services are started without needing additional start commands and the method 700 then moves to step 710. If no services 230 are configured to auto-start, the method 700 moves from step 706 to step 710.

5 [0145] In step 710, a determination is made as to whether any commands have been received. If no command has been received, step 710 may repeat until a command is received. If a command has been received, the method 700 moves to step 712.

[0146] In step 712, a determination is made as to whether any service start/stop commands have been received. In other words, whether a command has been received to
10 start a service 230 or to stop a currently running service 230. If the determination of step 712 indicates that a start or stop command has been received, the method 700 continues to step 714. In step 714, a determination is made as to whether the command was a start command or a stop command. If the determination of step 714 indicates that a stop command was received, the method 700 continues to step 716, where the service or services identified in the
15 stop command are stopped. If the determination of step 714 indicates that a start command was received, the method 700 continues to step 718, where the service or services identified in the start command are started. Following the execution of either step 716 or step 718, the method 700 returns to step 710.

[0147] If the determination of step 712 indicates that a start or stop command has not
20 been received, the method 700 continues to step 720. In step 720, a determination is made as to whether a command has been received to close the platform instance. If the determination of step 720 indicates that a close command was received, the method 700 continues to step 722, where the platform instance is closed. This may entail closing any running services 230 prior to closing the platform instance. If the determination of step 720 indicates that no close
25 command was received, the method 700 continues to step 724.

[0148] In step 724, the command (which is not a service stop/start command or a close instance command) is executed. The method 700 then returns to step 710. The method 700 may repeat from step 710 multiple times, starting and stopping services and/or executing other commands if requested until the platform instance is closed. It is understood that the
30 order of some steps in FIG. 7 may vary, such as the order of steps 712, 720, and 724. Furthermore, some steps, such as steps for handling errors (e.g., requests for invalid commands) are not shown.

[0149] Referring to FIG. 8, a method 800 illustrates a more detailed embodiment of step 608 of FIG. 6A. In step 802, one or more input signal streams are received as defined by the

NIO platform instance's configuration. For example, the NIO platform instance 402 may be configured to pull and/or receive data from one or more streaming sources. These signal streams may contain any type of analog and/or digital signal and may contain data. In step 804, the signals are extracted from the signal stream in real time as defined by the NIO platform instances's configuration. This step may include filtering to separate noise from relevant signal information. In step 806, the extracted signals are converted into niograms for internal processing as defined by the NIO platform instance's configuration.

[0150] In step 808, the internal processing is performed, which may include real time context enrichment. The particular processing and context enrichment (if applicable) that occurs depends on the configuration of the NIO platform instance 402. In step 810, output is produced by the NIO platform instance 402 as defined by the platform instance's configuration. The output may be in the form of a niogram and/or any type of signal, including data of any format type and/or actuation signals.

[0151] Referring to FIGS. 9A and 9B, a NIO platform 900 illustrates a more detailed embodiment of the NIO platform 100 of FIG. 1A, the NIO platform 200 of FIG. 2A, or the NIO platform 400 of FIG. 4A from a platform perspective (FIG. 9A) and a stack perspective (FIG. 9B). For purposes of example, the NIO platform 900 is written in the programming language Python, but it is understood that any suitable programming language can be used, including but not limited to languages such as C++. The NIO platform 900 is built on a core 906. When launched, a core process creates a core server (generally referred to herein as the core), which forms the underlying structure of the NIO platform 900.

[0152] Service components 902 include services 916 and blocks 918 from a functional perspective, even though the services 902 and blocks 918 are illustrated separately in the stack of FIG. 9B. As previously described, the service components 902 are responsible for user defined functionality by enabling block and service functionality to be defined and changed. Much of the functionality in a service component 902 can be user specific, which allows for a high level of customization.

[0153] In the present example, the service components 902 are provided as service classes 917 that define how services 916 are created and executed. The execution of services 916 includes routing signals, executing commands, and defining class structures. Some or all of the service classes 917 that form a service component 902 can be extended to define new functionality. This provides a large amount of flexibility in a neutral manner, as a user can define whatever functionality is desired through the service components 902 and that functionality will be executed by the NIO platform 900.

[0154] Generally, the service components 902 in one platform instance have no dependency or awareness of another platform instance's service components, which allows for each particular platform instance to be configured without having to take into account how the configuration would affect other platform instances. Furthermore, changing
5 functionality in a service component 902 has no effect on the core 906. This ensures that the core 906 does not have to be modified to be compatible with the service components 902.

[0155] In the present example, from a functional perspective, the service components 902 include blocks 918, block classes 917, block instances (also referred to simply as "blocks"), block groups, commands, services 916, and niograms.

10 [0156] In the NIO platform 900, blocks classes 919 may include classes for both custom blocks 934 and blocks having predefined functionality such as RFID block(s) 924, short message service (SMS) block(s) 926, sensor block(s) 928, programmable logic controller (PLC) block(s) 930, and global positioning satellite (GPS) block(s) 932. Although not shown, it is understood that many other blocks 918 may be defined for use with systems
15 using Electronic Product Codes (EPCs) (a trademark of EPCglobal Inc. of Lawrenceville, New Jersey), Low Level Reader Protocol (LLRP) information, email (e.g., simple mail transfer protocol (SMTP)), hypertext transfer protocol (HTTP) documents, and/or any other protocols.

[0157] Blocks 919 are classes that specify the metadata template and computational
20 functionality of block instances. In the present example, blocks 918 are built from block classes 919 that extend a BaseBlock class and can specify custom behavior by overriding any of the following five basic methods provided by the BaseBlock class: BaseBlock.initialize, BaseBlock.configure, BaseBlock.start, BaseBlock.stop, and BaseBlock.processSignals. These methods are used by the service 916 that corresponds to the blocks 918.

25 [0158] The BaseBlock.initialize method is called to instantiate the block 918 using the corresponding block class 919. The BaseBlock.configure method is called to configure the block 918 after initialization using a saved block configuration. The BaseBlock.start method is called to start the block 918 after instantiation and configuration. The BaseBlock.stop method is called to stop the block 918 (e.g., when the containing service 916 has been
30 stopped). The BaseBlock.processSignals contains the main processing functionality provided by the block 918. The BaseBlock.processSignals method processes a (possibly empty) list of incoming signals and notifies the service 916 when done (e.g., via a notifySignals method, which is discussed below).

[0159] A block instance is created when a block 918 is instantiated from a block class 919. A block instance may be viewed as the fundamental unit of computation in the NIO platform 900 and may be customized and configured as prescribed by the block class 919 being instantiated. A block instance only exists inside a service 916. Accordingly, when a service 916 is started or stopped, the blocks 918 inside that service 916 are also started or stopped. In the present example of the NIO platform 900, there is no concept of a block 918 running outside a service 916.

[0160] Block configurations, which are used to configure blocks 918, can be reused in different services 916 and may be viewed as saved configurations of blocks 918. When the configuration of a block 918 is changed, it will be changed for all blocks 918 in all services 916 that contain it. However, if a service 916 is running, the configuration of the running block instance may only be updated after the service 916 is restarted.

[0161] In other embodiments, a block instance may be updated without restarting the service 916. For example, if the block instance is not currently in use by the service 916, the block instance may be stopped, reconfigured with the new block configuration, and restarted. Alternatively, if not in use, the block instance may be destroyed and a new block instance may be instantiated with the new block configuration. In such embodiments, the service 916 may continue running or may be paused, rather than stopped and restarted.

[0162] Outside agents (e.g., other services and/or external APIs) may modify the behavior of specific blocks 918 via a command API (discussed below). Within the command API, block instances may be referenced by a service level block alias and/or a block group level. For this reason, globally unique block identifiers are not necessary in the present example, although they may be used in some embodiments.

[0163] Block instances can directly receive and send signals without going through the service 916. In this respect, a block 918 can serve as an interface through which signals can enter the NIO platform 900 and be sent from the NIO platform 900.

[0164] With additional reference to FIG. 9C, one embodiment of a hierarchy 990 is illustrated with a single service 916, blocks 918, and block groups 992 and 994. Block groups provide re-usable assemblies of block configurations that can be dropped into a service 916 like other blocks. The configuration of a block group resembles a service 916 in that there is a list of blocks and block mappings to define the flow of niograms between the blocks 918. Whenever one of these inner blocks produces niograms, the block group's logic handles the routing of those niograms to the next blocks in the chain, similar to how a service 916 would handle such routing. For example, the block group may include one or more

controllers (e.g., one or more blocks or other control mechanisms) containing the logic to handle routing within the block group to control the data flow within the block group. In such embodiments, the service 916 may treat the block group's control block as simply another block within the service 916 and may communicate with the control block just as with other non-grouped blocks.

[0165] In the present embodiment, the block groups 992 and 994 specifically define entry and exit points for niograms. In contrast, niograms may enter and leave the service 916 through any properly configured block 918. In other embodiments, the block groups 992 and 994 may not specifically define such entry and exit points.

[0166] Block groups can be nested in some embodiments, with a block group forming part of another block group. This is shown in FIG. 9C with block group 994 nested inside of block group 992. A nested block group may only communicate with the service 916 through the higher level block group. For example, block group 994 may only communicate with the service 916 through block group 992, rather than directly. In other embodiments, the service 916 may have direct access to nested block groups.

[0167] Since a block group is to be used inside a service 916, the service 916 may need to access commands (discussed later) of the inner blocks 918. In the present example of the NIO platform 900, this may be accomplished by defining command mappings inside the block group's configuration. After these mappings are defined, the block group knows to pass the command down to one of the inner blocks when a certain command is called on the block group. It is understood that not every inner block command may need to be exposed to the service 916 in the block group's command mappings. Accordingly, access may be provided based on such factors as the particular functionality of the block group and how that functionality is used by a service 916.

[0168] While blocks 918 have specific entry and exit points for niograms, block groups may have more than one entry and/or exit point. More specifically, a block 918 only has one possible entry point for use by previous blocks 918 in a service chain. In contrast, a block group has many blocks 918 that could be considered the starting point for niogram processing. As a result, one of the configuration options of the block group is defining one or more starting points for niogram processing. Then, when a service 916 passes niograms to the block group, those niograms will be forwarded to each block 918 in the list of starting points. Similarly, the block group can be configured to define a list of end points. When blocks 918 specified in this list produce niograms, the block group's logic handles the process of notifying the parent service 916 and/or the parent block group of those niograms.

[0169] For purposes of example, one embodiment of a block group may include various configuration options such as blocks, blockMappings, commandMappings, startPoints, and endPoints. An object “blocks” provides a mapping of the block names of all blocks 918 within a block group to the unique block alias of each block 918. An object “blockMappings” provides a mapping of the block aliases to a list of the next block aliases that fall in the chain of the block group (i.e., which block 918 feeds into which block). An object “commandMappings” provides a mapping of commands to inner-block commands.

[0170] An array “startPoints” is an array of strings that defines the block aliases that should be used as the starting points of the block group. When niograms are delivered from the service 916 to the block group, the blocks 918 listed in the startPoints array will receive the niograms for processing. An array endPoints is an array of strings that defines the block aliases that should be used as the ending points of the block group. When blocks 918 listed in the endPoints array produce niograms, a notification will propagate up outside of the block group.

[0171] With additional reference to FIG. 9D, as described previously, services 916 are the main organizational component of the NIO platform’s configurable processing functionality. Each service 916 maintains a block router 921, a block controller 922 for each block 918a-918M, its own configuration parameters, a list of block instances for that service, and an associative list of block aliases. In the present embodiment where the service 916 is a process, the service process may handle all communications between the service 916 and the service manager 914. In other embodiments, the block router 921 or another component within the service 916 may handle such communications.

[0172] The block controllers 922a-922M serve as intermediaries between the block router 921 and their respective blocks 918a-918M. In performing this intermediary function, the block controllers 922a-922M mimic both the block router 921 and the blocks 918a-918M. For example, the block router 921 may instantiate the block controller 922a, which in turn instantiates the block instance 918a. In other embodiments, the block router 921 may instantiate the block controller 922a and the block instance 918a. After instantiation, the block router 921 communicates with the block controller 922a as though the block controller 922a is the block 918a. Similarly, the block 918a communicates with the block controller 922 as though the block controller 922 is the block router 921. Accordingly, removal of the block controllers 922a-922M would not prevent communications between the block router 921 and the blocks 918a-918M, but would remove the functionality provided by the block

controllers 922a-922M from the service 916 unless that functionality was included elsewhere in the service (e.g., in the block router 921 and/or the blocks 918a-918M).

[0173] The block controllers 922a-922M may be configured to perform error handling and/or other functions for their respective blocks 918a-918c. Generally, only functions that are likely needed by many or all blocks may be provided by the block controllers 922a-922M. This enables a generic block controller to be used for a block 918 regardless of the functionality of that particular block. Accordingly, each block controller 922a-922M is identical in the present example. In other embodiments, block controllers having different configurations may be used for different blocks based on the need of a particular block and/or other criteria.

[0174] The block controllers 922a-922M may be configured to make certain decisions about whether to pass information to the block router 921. For example, when the block 918a throws an error, the error is caught by the block controller 922a. The block controller 922a may then decide how to handle the error, including passing the error up to the block router 921, ignoring the error, and/or taking other action. For example, if the error indicates that the block instance 918a has stopped working, the block controller 922a may proactively notify the block router 921 or may wait to notify the block router 921 until the block router 921 attempts to use the block instance. Removal of the block controller 922a would remove this error handling functionality so that when the block 918a throws the error, the block router 921 would catch it.

[0175] The block router 921 handles data flow among the blocks 918a-918M by defining the flow of niograms between blocks 918a-918M within the service 916. More specifically, communication between block instances within the service 916 is managed by the block router 921 via a Blockrouter.notifySignals() method and a processSignals() method. The Blockrouter.notifySignals() call is issued by a block 918 that has output ready. The Blockrouter.notifySignals() method identifies the source block and contains the niogram(s) forming the output. For example, the Blockrouter.notifySignals() may be implemented as Blockrouter.notifySignals(source block identifier, niogram(s)).

[0176] In the current embodiment, this call is made whenever a block 918 within the service 916 has output and the block need not be aware of the service at all. In other words, the block 918 receives input, processes, the input, calls Blockrouter.notifySignals(), and is done without even knowing that it is part of a service. In other embodiments, the block 918 may know the service 916 of which it is a part, which enables the block 918 to notify the signal to the particular service 916. Although the output itself is passed as a parameter in the

method call in the present embodiment, it is understood that other processes may be used to transfer the output. For example, a pointer to the output may be passed rather than the output itself.

[0177] When Blockrouter.notifySignals() is invoked, the block router 921 looks up the source block 918 in the routing table to determine the destination block(s) 918 to which the output should be directed. The block router 921 then calls processSignals() on each of the next blocks in succession. The processSignals() method identifies the destination block and contains the niogram(s) to be processed (e.g., the niograms that were the output of the source block). For example, the processSignals() method may be implemented as processSignals(destination block identifier, niogram(s)). Although the niogram(s) themselves are passed as a parameter in the method call in the present embodiment, it is understood that other processes may be used to transfer the niogram(s). For example, a pointer to the niogram(s) may be passed rather than the niogram(s) themselves. The block router 921 may, with each call for processSignals(), launch the called block instance in a new thread of the service process.

[0178] In the present example, the blocks 918 operate asynchronously (i.e., each block 918 executes independently of other blocks). When a block 918 publishes a niogram to another block 918, the receiving block executes immediately. This means that there is no buffering of niograms between blocks 918 except as needed (e.g., buffering may occur if a thread pool is used and there is no currently available thread for the receiving block) and data passes through the service 916 as quickly as the blocks 918 can process the data. The processing speed for a given block 918 may depend on the complexity of the block's instructions, as well as on factors outside of a block's control, such as the speed of the device's processor and the amount of processor time allocated to the block's thread.

[0179] Services 916 are started and stopped by commands issued through a service API. When a service 916 receives the start command, it "starts" all blocks 918 contained by the service 916. Similarly, when a service 916 receives the stop command, it stops all blocks 918 contained by the service 916. It is noted that the blocks 918 may not actually be "started," but simply notified that the service 916 encapsulating them has been started. If desired, the blocks 918 can then use the notification hook to execute some functionality (e.g., a block 918 that polls an external API and needs to know when to start polling could use the notification as the polling trigger).

[0180] In some embodiments, stopping a service 916 may result in the loss of any information (e.g., the local state) in any corresponding block instances. For example, in the

current example that uses Python objects for block instances, block objects can be wiped out by calling the `Blockinstance.destroy()` method. In other embodiments, it may be desirable to maintain the local state after a service 916 is stopped. For example, instead of wiping out the local state of instantiated blocks when a service 916 is stopped, the service 916 can instead be
5 paused to stop the service's execution temporarily without losing potentially valuable data. This may be accomplished by issuing the stop command to all the blocks 918 in the service 916 without doing the normally associated cleanup (e.g., without calling `Blockinstance.destroy()`) and/or in other ways.

[0181] Commands are used to interact with blocks 918 and must be reachable from
10 outside the blocks 918. Accordingly, how a block 918 defines and exposes a command needs to be known. For example, a block 918 may be used to provide SMS functionality. To accomplish this, the block 918 may be configured to expose a command "sendSMS." For the block 918 to function within the NIO platform 900, the method for actually sending an SMS would be written in the block 918 in executable instructions, and then the method would have
15 to be declared as a command to make it reachable through, for example, a REST API. A command to call the method may be formatted in various ways depending on the particular implementation of the block structure, such as a name (e.g., the block's method name), title (e.g., a descriptive name), and arguments. It is noted that this may be the same command structure used to start/stop services.

[0182] As previously described, the niogram is the primary mechanism for intra-service
20 data transmission (e.g., between blocks/block groups). All blocks 918 may accept and emit generic niograms of a base niogram class. The base niogram class generally has no required fields and does not require validation. The base niogram class simply exposes a way to add or remove attributes, and serialize/de-serialize the niogram into different forms (e.g.,
25 JavaScript Object Notation (JSON)). In the present example, an instance of the base niogram can add or remove attributes freely.

[0183] The base niogram can be subclassed for use in a block 918. However, in the present embodiment, the NIO platform 900 will not maintain any awareness of these subclasses and other blocks 918 will expect base niograms. In general, blocks 918
30 should not rely on processing a sub-class of the base niogram unless it is mandatory. Using only the base niogram class ensures that blocks 918 can be reused in different services with minimal impact. Filtering of blocks 918 should generally be done via a type attribute of the niogram, rather than the class type. Accordingly, while the generic niogram class can be

extended for convenience and/or encapsulation, only the attributes of an incoming niogram should be taken into account by a receiving block 918.

[0184] Another benefit of using the base class of niograms is to enable real-time cross referencing. For example, a niogram could start out containing data from one source and
5 then have its information enriched using data from another source. The resulting niogram would contain information from both sources, rather than having to carry around multiple niogram types.

[0185] With continued reference to FIG. 9B, the modules 904 are modules containing predefined code that the NIO platform 900 may use itself and that blocks 918 may also use.
10 The modules 904 may provide functionality defined with respect to each module, such as a logging module 938, a security module 940, a threading module 942, a communication module 944, a scheduler module 946, a persistence module 948, and/or a web server module 950. Some or all of the modules 904 are designed so that they can be exchanged for different implementations of the same functionality without affecting existing blocks or platform
15 functionality. A role of the modules 904 within the NIO platform 900 is to provide swappable functionality for different platform instances without affecting the blocks 918 and the core 906. The modules 904 provide APIs that can be called by blocks 918 and the core 906. The result of the API call is defined by the functionality of the called module 904.

[0186] The functionality defined in the modules 904 spans an entire platform instance.
20 Accordingly, when the functionality within a module 904 is changed, the entire platform instance will use the new version of the module. For example, if the logging module 938 is changed to log to a remote database instead of a local file, all logging calls (in the core 906 and in the services 916) will start logging accordingly. However, such changes may require a platform instance restart to take effect.

[0187] The modules 904 support the ability of the NIO platform 900 to run within
25 different environments without having to modify the core design of the NIO platform 900. For example, if a particular environment does not support some needed feature, the module 904 responsible for that feature can be reconfigured or replaced with functionality that is supported by the environment. Accordingly, by changing modules 904 as needed, platform
30 instances may be run in varied environments that have different needs.

[0188] Depending on the functionality of the particular module 904, a module 904 may need to initialize its functionality based on variable data. For example, the logging module 938 may need a file name where the information is saved, while the communication module 944 may need a list of current publishers in the platform instance. In

order to accomplish this, both the core 906 and the services 916 initialize the modules 904 by calling a setup method and passing context information with this data.

[0189] For services 916, the module's initialization data may come directly or indirectly as part of the service's initialization data. For example, the data may be provided indirectly
5 by providing the name of the configuration file where the data for the module 904 resides. For the core 906, the data may reside in a system wide configuration file that can be read during start up and then used for initializing the module 904.

[0190] The logging module 938 is used to provide logging functionality and, like all of the modules 904, may provide a customized solution or may use an existing solution, such as
10 Python's built-in logging module. At initialization, the logging module 938 receives parameters detailing adapters that are active in the NIO platform 900, which may include adapters for logging to a local file, for logging to a shared database (e.g., MySQL), and/or for creating a niogram and publishing the niogram through the NIO platform 900. In the present example, the logging module 938 exposes two classes, one for logging and one to
15 retrieve logged information. This enables the core 906 and services 916, which may be separate processes, to log to a single environment.

[0191] The security module 940 enables blocks 918 to interface with internal or external security applications. In the present example, the security module 940 provides an authentication method and an authorization method, both of which may be overridden. The
20 authentication method enables the security module 940 to authenticate a user. This method can take arguments and will return an instance of the SecureUser class (see below). Examples of authentication include username/password, OAuth Secure Token, and MAC Address. The authorization method enables a consumer of the security module 940 to be able to authorize an authenticated user (e.g., a SecureUser) against a certain set of SecureTasks. The existing
25 forms of authentication include access control lists, role based security, and User/Group/Other Permissions (e.g., 755). This enables the blocks 918 to use the same security implementation as the core 906 without being concerned about how the security is actually implemented.

[0192] In addition to overriding the methods of the security module 940, a secure
30 implementation may also override two objects that can be secure, which are SecureUser and SecureTask. SecureUser is a class that represents a user. There are no required fields and these objects will be returned by the authentication method. When implementing the security module 940, this class should be overridden to map to users in the secure system. SecureTask is a class that represents something to be done. In general, a SecureUser either

can or cannot perform a SecureTask. These tasks will be passed to the authentication method and SecureUsers will be authenticated against them. The security module 940 should override this class with tasks that it wishes to secure.

[0193] The threading module 942 provides threading support and may provide one or more threading options. For example, two threading modules may be available, with non-native threading used only when needed. In the present embodiment, the main NIO platform process may not need any thread customization and can run under Python's regular threading module. Services 916 and blocks 918, however, may benefit from having a large number of light threads and the ability to launch asynchronous tasks in a short amount of time. Accordingly, the NIO platform 900 can provide a wrapper for Python's threading functionality with the objective of making it transparent to a developer and allowing switching from one threading mode to another. The threading module that will be in effect for a particular service 916 may be specified through a setting.

[0194] The communication module 944 enables services 916 within a platform to subscribe and publish niograms. The niograms can be transported within the platform instance or between platform instances. The communication module 944 may use ZeroMQ or a similar library as a publish and subscribe mechanism. It is noted that queuing may occur between services 916 if needed and such queuing may be handled by the communication module 944 via ZeroMQ or another selected library.

[0195] The communication module 944 exposes two classes, a publisher class and a subscriber class. Each class may include a list of arguments that are treated as flags, and a subscriber matches a publisher when all flags are matched. All functionality handling the subscription/publication mechanism is controlled within the individual communication module 944 that the NIO platform 900 is running. When a service 916 wants to publish, it simply publishes to the communication module 944 and the communication module 944 determines where the published niograms will go.

[0196] For example, assume there are four publishers A-D as follows: (A) publisher (type=RFID, source=Dock Door); (B) publisher (type=RFID, source=Conveyor); (C) publisher (type=BarCode, source=Conveyor); and (D) publisher (type=RFID, source=Shelf).

A subscriber may designate the publishers to which it is subscribing as follows. A subscriber (type=[RFID]) would receive publications from A, B, and D. A subscriber (type=[RFID], source=[Dock Door, Conveyor]) would receive publications from A and B. A subscriber (source=[Conveyor]) would receive publications from B and C.

[0197] The scheduler module 946 facilitates the execution of tasks at scheduled intervals or at a single point in the future. The scheduler module 946 may be included in the NIO platform 900 so that the scheduler can be replaced if issues arise with a particular scheduler in a given environment. The scheduler module 946 operates via instances of a “job” class, which will be needed only to cancel the job at the end. The implementation of the scheduler module 946 is responsible for initializing and terminating the underlying scheduler.

[0198] The persistence module 948 enables blocks 918 and core components to “persist” certain information relevant to them that will survive through a platform instance restart. The persistence module 948 can choose to save wherever and however it wants (e.g., in a flat text file or a local database). It exposes several methods that can be used within a block 918 to access this functionality, such as save, load, and clear.

[0199] The web server module 950 enables services 916 and/or blocks 918 to expose a web server for interacting on an isolated port. In addition, the core 906 may use the web server module 950 to expose a web server that hosts the API 908. The web server module 950 provides an interface for each to do so using several methods, such as open, handle, and close. The web server module 950 may use a library such as the CherryPy library in Python. This removes the core dependency of CherryPy and allows block writers to utilize the same web server functionality as the core 906. This not only allows other libraries to be substituted, but also allows block writers to easily expose web servers without having to worry about conflicting with the core’s API 908.

[0200] Services 916, which operate as different processes in the present example, can ease the load on the core process by receiving data directly through their own web server. Without this, blocks/services use commands to receive data through HTTP, but those commands are regulated and passed through the core 906. By using the web server module 950, the blocks 918 can listen directly to a port for incoming HTTP requests and handle the requests accordingly without loading the core process.

[0201] In the present example, the core 906 includes an API 908, a service manager 914, and a configuration manager 910. The configuration manager 910 includes configurations 911, a loader 952, and discovery functionality 954, which may be part of the loader 952 in some embodiments. In other embodiments, the configuration manager 910 may not exist as a component, but the loader/discovery functionality and the configurations may continue to exist within the core 906 (e.g., as part of the service manager 914 or elsewhere). The core 906 may also include core components 912 in some embodiments. The core 906 maintains

the services 916 provided by the NIO platform 900. The core 906 is not directly exposed to the service components 902 and can use the modules 904.

[0202] The API 908 represents multiple APIs, but it is understood that blocks 918 and block groups may be able to receive and/or send information without passing through the API 908 in the core 906. For example, a block may be able to send and receive SMS messages without using the API 908. It is understood that many different APIs and API calls may be defined, and that the examples described below are only for the purpose of illustrating how various components of the NIO platform 900 may be accessed and managed. In the present example, the API 908 includes a block API, a block configuration API, a command API, a mechanism for providing custom APIs, and a service API.

[0203] The block API enables a user to alter the state of the blocks 918 loaded in the NIO platform 900. For example, the block API enables a user to add, reload, and/or remove blocks 918 without having to restart the instance in which the blocks 918 are located. For purposes of example, the block API follows the create, read, update, delete (CRUD) model, exposing four methods to interact with blocks 918, as well as an instances endpoint to interact with a block's instances.

[0204] A create method adds a new block 918 to an instance and may be accomplished in multiple ways. For example, a file, module, and/or package may be attached for use as the block 918, a file name where the block code is loaded may be referenced, a remotely hosted block may be referenced, and/or a class may be specified and the NIO platform 900 may be configured to locate and retrieve the class's code.

[0205] A read method returns a list of blocks 918 and therefore exposes the functionality of the NIO platform 900. In addition to the list of blocks 918, the read method may return other block meta information, such as version, dependencies, and install time.

[0206] An update method refreshes a block 918 in the NIO platform 900. This may include reloading the block's code, re-validating, and updating references. The update method may not update the block code for block instances that are currently in running services 916. In such cases, the service 916 may have to be restarted to realize the block code. In other embodiments, a block instance may be updated without having to restart the service 916.

[0207] A delete method enables a block 918 to be deleted from the NIO platform 900. Any block instances of the block 918 will also be deleted. Any blocks 918 that are in running services 916 will continue to run, but when the service 916 is restarted, an error will be

thrown and the service 916 will not be able to start unless the service 916 is updated to reflect the deletion.

[0208] An instances method enables interaction with the instances of a block 918. For example, “instances” may be viewed as a custom endpoint that is essentially an alias for
5 /instances?block=BlockName. The instances method allows a user to modify the block instances associated with a given block 918. This will be discussed in greater detail below with respect to the block instance API.

[0209] The block configuration API enables a user to alter the state of the block instances loaded in the NIO platform 900. Because block configurations are configured instances of
10 blocks 918, some API calls can happen through the previously described block API. For purposes of example, the block configuration API follows the CRUD model, but may also define some alternative methods.

[0210] A create method adds a new block configuration. To create a block configuration, a relevant block 918 must exist for the configuration. As a result, configuration creation can
15 go through the specified block's API endpoint within the block API. Configuration creation can also go through the NIO platform's block configuration API as long as a valid block 918 is specified.

[0211] A read method returns a list of block configurations, although there may be multiple ways to see the block configurations that are configured within the NIO platform
20 900. For example, by hitting the main block configurations endpoint, all configurations in the NIO platform 900 will be returned. Further refinement can be achieved by specifying a block name as a parameter or issuing the GET to the block configuration's endpoint. The GET calls will return the configuration's name as well as the configuration defined within the block 918.

25 [0212] An update method updates the configuration of a block configuration on the NIO platform 900. Blocks 918 that are part of a currently running service 916 will not have their configuration updates realized until the service 916 is restarted.

[0213] A delete method enables a block configuration to be deleted from the NIO platform 900. This removes a block configuration from the NIO platform 900, but not the
30 block 918 itself. If the block 918 is part of a running service 916, the service 916 will continue to run with the original block code. When the service 916 is restarted, an error will be thrown indicating the block 918 cannot be found.

[0214] The command API enables a user to interact with previously described command handlers that have been defined to expose commands for blocks 918. Services 916 and

blocks 918 can both be commanded. However, in the present embodiment, because blocks 918 do not stand alone but exist within a service 916, the caller must go through the service 916 to command a block 918. Depending on the particular implementation, a command may be called in many different ways, including hypertext transfer protocol (HTTP) methods such as GET and POST. The block 918 being called should define the proper handling for each type of allowed call.

[0215] A command method can be used to command a block 918 inside a service 916. For example, the method may be structured as `/services/ServiceName/BlockAlias/commandName`. The root of this API call is the service 916, since the block 918 inside of that service 916 is what will be commanded. If the specified service 916 does not exist, an error will be thrown. The next component in the method is the BlockAlias. By default, this will be the block configuration name. However, if a service builder wishes to include more than one of the same blocks 918 within a service 916, a block alias can be defined for each configuration of that block 918. The final component is the command name. This command must be a valid command as defined by the block 918 connected to BlockAlias.

[0216] The mechanism for defining custom APIs leverages the ability of blocks 918 to define custom command handlers. Because of this, custom APIs can be written as blocks 918 and implemented as block configurations within a service 916. For example, a service builder can drop an API block 918 into any point in a service 916. The API block 918 does not affect the operation of the service 916, but does provide a new API endpoint that can be used to leverage attributes of the service 916 at the point where the block 918 is inserted.

[0217] The service API enables a user to alter the state of the services 916 in the NIO platform 900. For purposes of example, the service API follows the CRUD model, as well as a command model that allows a user to start/stop a service 916.

[0218] A create method adds a new service 916 to the NIO platform 900. The specification of the service 916 (e.g., blocks and block mappings) may be included in the body of the call.

[0219] A read method returns a list of services 916 and their configuration. This information may include the blocks 918 within a service 916, the state of the service 916 (e.g., running or stopped), and any other configuration options specified when the service 916 was created.

[0220] An update method updates a service's configuration. If the service 916 is currently running, the configuration update will be accepted, but the changes will not be realized until the service 916 is restarted.

5 [0221] A delete method removes a service 916 from the NIO platform 900. If the service 916 is currently running, this call will return an error. The service 916 should be stopped before being deleted.

[0222] A command method is used to start or stop a service 916. If a problem exists with the configuration of a service 916 (e.g., there are non-existent blocks 918, block instances with an invalid block 918, and/or other validation issues), the call will return an error.

10 [0223] With continued reference to FIGS. 9A and 9B, in the present embodiment, the configuration manager 910 manages configurations 911 for the current instance of the NIO platform 900, loads services 916 and blocks 918 for inspection if needed, and performs auto-discovery. Ideally, the core 902 has no dependency on its functionality (e.g., the blocks 918) or its configuration (e.g., the block instances and services 916). This lack of dependency
15 enables the use of relocatable instance configurations, such as one or more directories specified by a user. Then, when an instance of the NIO platform 900 is launched, the location of the instance configuration will be identified and the NIO platform 900 will load the instance's blocks 918, services 916, and other needed components from that location. This enables a user to version control their configurations, create multiple configurations on
20 the same machine, and easily share and inspect their configurations.

[0224] Configurations may be represented within the NIO platform 900 in many different ways. For example, block instances and services 916 may use JSON flat files, SQLite databases, and/or zip files, while blocks 918 may use python files or python module directories. It is understood that these are merely examples and that many different formats
25 may be used to represent configuration information.

[0225] The NIO platform 900 may include different types of configurations depending on what part of the NIO platform 900 is being described. Examples include a core configuration, a platform configuration, a core components configuration, a service configuration, and a block configuration. It is understood that these configurations may be
30 stored as separate files or may be combined. Furthermore, any of these configurations may be divided into multiple configurations or combined in many different ways.

[0226] The core configuration is directed to settings related to the core 906. These values may be private to the core 906 and visible to the services 902. The platform configuration is directed to settings for the entire NIO platform 900. These include all settings that are visible

to the core 906 and to the services 902. The core components configuration is directed to settings related to a specific core component. The service configuration is directed to settings related to a specific service 916. The block configuration is directed to settings related to a specific block 918.

5 **[0227]** The NIO platform 900 may use a configuration data file that details what is included in the NIO platform 900. This data file may be different from what is actually inside the configuration directory. For example, if a user copies a block file into a block directory, the block file may not be picked up by an instance until the block file is loaded via the block API. At this point, the instance may load that block 918 into the configuration data
10 file. Similarly, block instance configurations may be copied to the directory, but may not be recognized until the instance is restarted. In other embodiments, an instance restart may not be needed in order for the block instance configurations to be recognized.

[0228] In some embodiments, the data may reside at a remote location (e.g., in a remote database or a data structure server), which allows definitions to be shared among different
15 platform instances. In such embodiments, the handler to use in loading a particular configuration may be specified through a platform setting. The NIO platform 900 would then instantiate the specified handler and use it to fetch the instance configuration. One example of an instance configuration directory for the NIO platform 900 is illustrated below, with comments in parentheses.

20 **[0229]** / (The root of the configuration, which can be any directory on a device.)
 /blocks/ (A directory containing the blocks in the instance.)
 /blocks/"specific block name"/ (An example of a directory block.)
 /blocks/blockGroup.py
 /instances/ (A directory containing the block instances in the instance.)
 /instances/"specific block name".db (An example where the
25 blocks are a database.)
 /instances/blockGroup/ (An example where the blocks are
 grouped into a folder.)
 /instances/blockGroup/BigBlockGroup.json
 /instances/blockGroup/LittleBlockGroup.json
 30 /services/ (A directory containing the services in the instance.)
 /services/EmailNotificationService.json
 /services/WeatherSMSService.db

config.db (A set of data representing what is configured in the instance.)

[0230] With continued reference to FIGS. 9A and 9B, the core components 912 are modules containing predefined code that the NIO platform 900 may use. The core components 912 provide functionality to the NIO platform 900 and may include modules such as a monitoring module 956, a messaging module 958, a communication manager module 960, and/or an instance distribution module 962.

[0231] The core components 912 are somewhat different from core functionality provided by the configuration manager 910 and service manager 914. While core functionality is generally hidden from block writers and required for operation of the NIO platform 900, core components 912 are swappable components (similar to the modules 904) that are positioned within the core 906 and provide functions usable by the core 906. Like the core functionality, the core components 912 are hidden from block writers (unlike the modules 904). Unlike the core functionality, the core components 912 are not required for the NIO platform 900 to run. However, it is understood that certain implementations of the NIO platform 900 may rely on the core components 912 due to the platform's configuration, in which case the functionality of one or more of the core components 912 would be needed to provide the desired functionality. In other words, the NIO platform 900 might run without the needed core components 912, but would be unable to accomplish certain tasks. In other embodiments, the NIO platform 900 may not start without the needed core components 912.

[0232] With additional reference to FIG. 10, one embodiment of an environment 1000 that is internal to the NIO platform 900 illustrates the communication manager module 960 serving as a broker for all processes that participate in publish/subscribe operations. To accomplish this, the communication manager module 960 interacts with the previously described communication module 944. When a service 916 containing publishing/subscribing functionality is launched, the communication manager module 960 assigns two ports to the service 916. One port is for subscribing to management data and the other port is for publishing management data. The communication module 944 that is being used by the service 916 tracks these ports for the service.

[0233] The communication manager module 960 subscribes to the publisher in the service 916 (e.g., the communication module 944 corresponding to the service). When the communication manager module 960 receives a message from the service 916, the communication manager module 960 broadcasts the message to existing services. This aids in maintaining a synchronized state among all services 916 regarding the publishers in the

NIO platform 900. In some embodiments where multiple platform instances are running concurrently, the communication manager modules 960 of the different platform instances may share an internal channel to aid in communications among the various instances. In other embodiments where multiple platform instances are running concurrently, one
5 communication manager module 960 may act as a broker for the other communication manager modules.

[0234] The basic communication process provided by the communication module 944 and the communication manager module 960 is illustrated in FIG. 10. For purposes of illustration, each step is labeled in Fig. 10 as a number in a circle. In step 1, the
10 communication manager module 960 populates a service context 1002 received from the service manager 914 with channel information for a new service 916a. This includes the subscribe/publish ports introduced previously. In step 2, the service manager 914 passes the context with the channel information during the initialization of the new service 916a.

[0235] In step 3, the new service's communication manager 916a opens a publisher via
15 the communication module 944a that is being used by the service 916a and provides its own information to the communication manager module 960. The new service's communication module 944a also opens a subscriber for receiving information about other services. In step 4, the communication manager module 960 broadcasts the publisher information received in step 3 to the other services, such as a service 916b. In step 5, the other service 916b uses its
20 communication module 944b to open a subscriber to the publisher of the new service 916a. The service 916b can then receive communications to which it is subscribed directly from the service 916b.

[0236] It is understood that the described communication process may be accomplished in many different ways. For example, while the present embodiment illustrates a processes
25 that uses the communication manager module 960 to aid in establishing direct communications between the communication modules 944a and 944b, the communication manager module 960 may continue to receive and broadcast messages published by the communication modules 944a and 944b. In other embodiments, the communication modules 944a and 944b may not communicate directly and may subscribe and publish only to the
30 communication manager module 960.

[0237] Referring again to FIGS. 9A and 9B, the instance distribution module 962 may be used when more than one platform instance is sharing the services 916. For example, in a distributed system, where multiple platform instances work together with the purpose of sharing the load of running intended services 916, information about the platform instances

currently in the distributed system has to be maintained. This information enables each platform instance in the distributed system to be able to find out if the other instances are running. For example, each platform instance may ping the other instances and, when an instance is detected to be non-functioning, may remove any existing dependencies to the non-functioning instance.

[0238] The instance distribution module 962 maintains a list of other platform instances in the distributed system, as well as metadata about each platform instance (e.g., how long each instance has been running, how many services 916 are running in each instance, communication ports, and/or other information). In some embodiments, an instance distribution module 962 will reside on every platform instance within the distributed system and self-update when platform instances are added or removed. Whenever this list is updated, relevant information will be published through the communication module 944 for blocks and core components to consume.

[0239] To accomplish this, the instance distribution module 962 in each platform instance subscribes to the instance distribution module 962 in each of the other platform instances. When the instance distribution module 962 of an instance detects a change (e.g., a new instance or a dead instance), it publishes that relevant information to the remaining instance distribution modules 962. The instance distribution modules 962 receiving this information then update their respective lists accordingly. Publishing may be centralized or distributed.

[0240] Because load balancing tends to be dependent on the way services 916 are built and the metrics of each service 916 (e.g., throughput), the actual balancing process of launching new platform instances and tearing down platform instances may be performed within a service 916. This places the responsibility of load balancing on the services 916. In other embodiments, the core 906 may take a larger role in load balancing.

[0241] Referring to FIGS. 11A and 11B, the messaging module 958 provides a way for external systems to send and receive information from the NIO platform 900. It is understood that such communications may be accomplished in many different ways and may vary depending on the implementation of a particular external system. For example, some external systems may inquire about the data available and formulate a request based on the response, while other external systems may know exactly what data to request and what data to provide. In some embodiments, the messaging module 958 may be replaced by using commands in conjunction with the web server module 950.

[0242] FIG. 11A illustrates an embodiment where an external system 1102 inquires about the data available and formulates a request based on the response. For purposes of

illustration, each step is labeled in Fig. 11A as a number in a circle. An inquiry message (e.g., a request) from the external system 1102 is received by the messaging module 958 via the REST interface 964/API 908 in step 1. The messaging module 958 accepts the inquiry, adds and/or registers the service, and sends the service's information (e.g., a uniform resource locator (URL)) to the external system 1102 in step 2. The external system 1102 uses the received information to send a message (e.g., an HTTP call) to the service manager 914 in step 3. The service manager 914 then places the message in the channel for publication to the service 916 in step 4.

[0243] FIG. 11B illustrates an embodiment where an external system 1104 knows exactly what data to request and what data to provide. For purposes of illustration, each step is labeled in Fig. 11B as a number in a circle. A request from the external system 1104 is received by the messaging module 958 via the REST interface 964/API 908 in step 1. The messaging module 958 forwards the request to an external data receiver service (which may be part of or separate from the service manager 914) in step 2. The service manager 914 processes the call to publish a specific message to the service 916 in step 3.

[0244] With continued reference to FIGS. 9A and 9B, the service manager 914 handles the interaction of the core 906 with the services 916 running in a platform instance. The service manager 914 handles starting and stopping services 916, and may also manage a service's incoming commands (e.g., commands received via the REST interface 964/API 908). The service manager 914 may use functionality provided by the modules 904 and core components 912. The service manager 914 may be accessed from outside the NIO platform 900 via the API 908.

[0245] Referring to FIG. 12, one embodiment of an environment 1200 that is internal to the NIO platform 900 illustrates a service 916 and the core 906. As previously described, the service process handles communications with the service manager 914, such as commands received through the REST API 908. The block router 921 handles intra-service communications between the blocks 918 via the corresponding block controllers 922 (in embodiments where the block controllers 922 are present). The loader 952 and discovery functionality 954 may be used by the service manager 914 to load service and/or block classes for discovery purposes. External sources/destinations 1202 and 1204 can communicate via blocks 918 and/or via the REST API 908.

[0246] Referring to FIG. 13, one embodiment of an environment 1300 that is internal to the NIO platform 900 illustrates various components of the NIO platform 900 of FIGS. 9A

and 9B interacting to launch a platform instance and a service 916. For purposes of illustration, each step is labeled in Fig. 13 as a number in a circle.

[0247] After a new core process 906 is launched, the core process accesses the configuration manager 910 and retrieves one or more configurations 911 in step 1. The configuration provides information as to which, if any, modules 904 and core components 912 are available within the NIO platform 900. The core process 906 then, for each module 904 that is available, creates and initializes the module 904 for use in step 2. In step 3, the core process 906, for each core component 912 that is available, creates and initializes the core component 912.

10 [0248] In step 4, the service manager 914 is started. In step 5, the service manager 914 accesses the service configuration information via the configuration manager 910. In step 6, the service manager 914 then calls the loader 952/discovery functionality 954 to discover all the services 916 and blocks 918 available in the NIO platform 900. The loader 952 may access a repository 1302 in which the services 916 and blocks 918 are stored and, if needed, load each service 916 and/or block 918 to examine them and discover their characteristics. This process may include examining each block 918 to see if the block requires a particular module. For example, a block 918 may explicitly define the module(s) that it needs or the loader may examine the block's code to determine if it uses any modules.

[0249] In step 7, for each service 916 configured to be auto-started, the service manager 914 launches the service 916 with the configuration information corresponding to that service. Although not shown, the service manager 914 may also expose the handling for the REST interface 964/API 908 through the web server module 950 during this process.

[0250] Referring to FIG. 14, a method 1400 illustrates one embodiment of a process that may be executed by the NIO platform 900 of FIGS. 9A and 9B when an instance of the platform is launched. In the present example, a core process has been launched and is running prior to step 1402, but no modules or services have yet been launched. The method 1400 is a more detailed embodiment of a process that may occur during steps 1-4 of FIG. 13.

25 [0251] In step 1402, the core process accesses the configuration information and, in step 1404, uses the configuration information to identify the available modules 904 and/or core components 912. Step 1404 may use the loader 952/discovery functionality 954 if needed during the discovery process of step 1404, in which case at least part of step 1412 would be executed prior to step 1404. In step 1406, the core process creates a context to store the information for the various modules.

[0252] In step 1408, the core process starts and configures each available module 904. It is noted that a module 904 may be instantiated prior to step 1408 or when started, but the configuration occurs when each module is started. During configuration, the configuration information and/or context is passed to the module 904 being configured. The module 904
5 then updates the context with its own information, such as information that may be needed by the core process and/or another of the modules 904 to call the current module.

[0253] In step 1410, the core process starts and configures each available core component 912. It is noted that a core component 912 may be instantiated prior to step 1408 or when started, but the configuration occurs when each module is started. During configuration, the
10 configuration information and/or context is passed to the core component 912 being configured. The core component 912 then updates the context with its own information, such as information that may be needed by the core process, one of the modules 904, and/or another of the core components 912 to call the current module.

[0254] It is noted that the current embodiment starts the modules 904 before starting the core components 912. As the modules 904 cannot use the core components 912, but the core components 912 can use the modules 904, this order of steps enables the modules 904 to update the context before the context is passed to the core components 912. As the modules 904 have no need for the context from the core components 912, this is a more efficient process. However, in other embodiments, the steps may be reversed or even mixed, with
15 modules 904 and core components 912 being started in some other order. In such embodiments, all modules may receive the context for all other modules.

[0255] In step 1412, the core process starts and configures other core components, such as the service manager 914 and the loader 952/discovery functionality 954. The service manager 914 may then start services, as described in FIG. 15.

[0256] Referring to FIG. 15, a method 1500 illustrates one embodiment of a process that may be executed by the NIO platform 900 of FIGS. 9A and 9B after the service manager 914 is started and configured by the core process. In the present example, the modules 904 and core components 912 have been launched and a context exists with the information about those modules. The method 1500 is a more detailed embodiment of a process that may occur
25 during steps 5-7 of FIG. 13.

[0257] In step 1502, the service manager 914 accesses the configuration information for the services 916 and, in step 1504, calls the loader 952/discovery functionality 954 to discover the services 916 and blocks 918 available to the NIO platform 900. In step 1506, the service manager 914 may obtain a context (e.g., the context with the information for the

modules 904 and core components 912) or may create a new context for use with the services 916.

5 [0258] In step 1508, the service manager 914 starts and configures each available service 916 that is configured to auto-start. It is noted that a service 916 may be instantiated prior to step 1508 or when started, but the configuration occurs when each service 916 is started in the present embodiment. In other embodiments, some or all of the configuration may occur when the service 916 is instantiated. During configuration, the configuration information and/or context is passed to the service 916 being configured. The service 916 then updates the context with its own information.

10 [0259] In step 1510, the service manager 914 exposes the handling for the REST interface 964/API 908 through the web server module 950, although this step may occur earlier in the method 1500. In step 1512, the service manager 914 waits for instructions such as stop and start commands for services 916.

15 [0260] Referring to FIG. 16A, a method 1600 illustrates one embodiment of a process that may be executed by a service process of the NIO platform 900 of FIGS. 9A and 9B after the corresponding service 916 is started. As stated previously, when a service 916 is started, the service context is passed to the service process by the service manager 914.

[0261] In step 1602, the service process receives the service context. In step 1604, the service process initializes any modules needed by the service process itself and modules
20 needed by the blocks 918 used in the service 916. The service process may directly use modules such as the logging module 938 and the threading module 942. Some modules, like the communications module 944, may be used by the service process to aid a block 918 in setting up a communications channel. Other modules may be used directly by a block 918. In step 1606, the block router 921 launches the blocks 918 belonging to the service 916.

25 [0262] Referring to FIG. 16B, a method 1610 illustrates one embodiment of a process that may be executed by a service process of the NIO platform 900 of FIGS. 9A and 9B after the corresponding service 916 is started. For example, the method 1610 may be a more detailed embodiment of step 1606 of FIG. 16A and may occur for each block 918. In step 1612, the block 918 is initialized to create a block instance. In step 1618, the block instance
30 is configured, which may include passing information such as module information needed by the block instance to use the module.

[0263] In step 1616, the block instance is started. As previously described, starting a block instance may include notifying the block instance that the corresponding service 916 has been started. If configured to do so, the block instance can then execute instructions

using the notification as a trigger. Without this notification, the block instance may remain idle until called by processSignals().

[0264] Referring to FIG. 16C, a method 1620 illustrates one embodiment of a process that may be executed by the block router 921 of the NIO platform 900 of FIGS. 9A and 9B.

5 In step 1622, the block router 921 receives an output notification from a block 918. In step 1624, the block router 921 looks up the next block(s) in the routing table and, in step 1626, calls the next block(s) to process the output. As described previously, this process may involve calls such as Blockrouter.notifySignals() and processSignals().

[0265] As described previously, due to the asynchronous nature of the blocks 918, some or all of the blocks 918 in the service 916 can process niograms concurrently. Furthermore, a single block 918 may process multiple niograms concurrently by using threading to execute the block's functionality in multiple concurrent threads. The method 1620 does not need to be changed for this process, as each processSignals() call launches a new thread. In the present example, the thread may spawn with a target of block.processSignals(). If needed,

10 the blocks 918 may include instructions for acquiring and releasing locks to prevent problems between concurrently executing threads. These instructions may be included on a block by block basis, as race conditions or other thread collision issues may not be a concern for some blocks.

[0266] Referring to FIG. 16D, a method 1630 illustrates one embodiment of a process that may be executed by the block router 921 of the NIO platform 900 of FIGS. 9A and 9B.

20 In step 1632, the block router 921 receives a message. The message may be from a block 918 and intended for the REST API 908/964, or may be from the REST API and directed to the block. In step 1634, the block router 921 passes the message on to the destination (e.g., via the REST API 908/964, by publishing the message to a channel, or via another transfer mechanism).

25

[0267] Referring to FIG. 16E, a method 1640 illustrates one embodiment of a process that may be executed by a block 918 of the NIO platform 900 of FIGS. 9A and 9B. In step 1642, the block 918 receives one or more signals and/or niograms. The block 918 may receive signals if the block is directly coupled to an external source and may receive niograms from another block (directly or via the block router 921 depending on the particular implementation). It is understood that receiving includes active processes through which the block 918 may obtain data, such as pulling data from an external source.

30

[0268] In step 1644, the block 918 performs processing based on the block's internal instructions. As described previously, such instructions may range from simple to complex,

depending on the particular configuration of the block 918. In step 1646, once the processing of step 1644 produces an output, the block 918 may issue a notification that there is output (e.g., by notifying the block router 921) and/or may send the output directly to an external source, depending on the block's internal instructions.

5 [0269] Referring again to FIGS. 9A and 9B, the NIO platform 900 includes a REST interface 964 that may be part of the API 908. The REST interface 964 aids in communicating with external tools and systems, such as a console 972, enterprise monitoring tools 974, enterprise applications 976, and external devices 978 (e.g., mobile devices, servers, databases, machinery, manufacturing equipment, and/or any other device, system, and/or
10 application with which the NIO platform 900 is configured to communicate).

[0270] The NIO platform 900 may use a runtime environment 966 for a particular language (e.g., Python) and also interacts with an operating system 968 on whatever device is running the NIO platform 900.

[0271] Referring to FIG. 17, one embodiment of an environment 1700 illustrates a user's
15 perspective of the NIO platform 900 of FIGS. 9A and 9B with external devices, systems, and applications 1706. For example, the external devices, systems, and applications 1706 may be similar or identical to the external source(s)/destination(s) 132 of FIGS. 1E-1G and 1I.

[0272] From the user's perspective, much of the core's functionality (not shown) is hidden. The user has access to some components of the NIO platform 900 from external
20 systems and applications 1706 via the REST API 908, 964. The external devices, systems, and applications 1706 may include mobile devices 1708, enterprise applications 1710, an administration console 1712 for the NIO platform 900, and/or any other external systems and applications 1714 that may access the NIO platform 900 via the REST API.

[0273] Using the external devices, systems, and applications 1706, the user can issue
25 commands 1704 (e.g., start and stop commands) to services 916, which in turn either process or stop processing niograms 1702. As described above, the services 916 use blocks 918, which may receive information from and send information to various external devices, systems, and applications 1706. The external devices, systems, and applications 1706 may serve as signal sources that produce signals using sensors (e.g., motion sensors, vibration
30 sensors, thermal sensors, electromagnetic sensors, and/or any other type of sensor) 1716, the web 1718, RFID 1720, voice 1722, GPS 1724, SMS 1726, RTLS 1728, PLC 1730, and/or any other analog and/or digital signal source 1732 as input for the blocks 918. The external devices, systems, and applications 1706 may serve as signal destinations for any type of

signal produced by the blocks 918, including actuation signals. It is understood that the term “signals” as used herein includes data.

[0274] Referring to FIG. 18, a diagram 1800 illustrates one embodiment of a workflow that runs from creation to launch of a NIO platform 1802 (which may be similar or identical to the NIO platform 100 of FIG. 1A, 200 of FIG. 2A, 400 of FIG. 4A, and/or 900 of FIGS. 9A and 9B). The workflow begins with a library 1804. The library 1804 includes core classes 206 (that include the classes for any core components and modules in the present example), a base service class 202, a base block class 1806, and block classes 204 that are extended from the base block class 1806. Each extended block class 204 includes task specific code. A user can modify and/or create code for existing blocks classes 204 in the library 1804 and/or create new block classes 204 with desired task specific functionality. Although not shown, the base service class 202 can also be customized and various extended service classes may exist in the library 1804.

[0275] The configuration environment 1808 enables a user to define configurations for the core classes 206, the service class 202, and the block classes 204 that have been selected from the library 1804 in order to define the platform specific behavior of the objects that will be instantiated from the classes within the NIO platform 1802. The NIO platform 1802 will run the objects as defined by the architecture of the platform itself, but the configuration process enables the user to define various task specific operational aspects of the NIO platform 1802. The operational aspects include which core components, modules, services and blocks will be run, what properties the core components, modules, services and blocks will have (as permitted by the architecture), and when the services will be run. This configuration process results in configuration files 210 that are used to configure the objects that will be instantiated from the core classes 206, the service class 202, and the block classes 204 by the NIO platform 1802.

[0276] In some embodiments, the configuration environment 1808 may be a graphical user interface environment that produces configuration files that are loaded into the NIO platform 1802. In other embodiments, the configuration environment 1808 may use the REST interface 908, 964 (FIGS. 9A and 9B) of the NIO platform 1802 to issue configuration commands to the NIO platform 1802. Accordingly, it is understood that there are various ways in which configuration information may be created and produced for use by the NIO platform 1802.

[0277] When the NIO platform 1802 is launched, each of the core classes 206 are identified and corresponding objects are instantiated and configured using the appropriate

configuration files 210 for the core, core components, and modules. For each service that is to be run when the NIO platform 1802 is started, the service class 202 and corresponding block classes 204 are identified and the services and blocks are instantiated and configured using the appropriate configuration files 210. The NIO platform 1802 is then configured and
5 begins running to perform the task specific functions provided by the services.

[0278] Referring to FIG. 19A, one embodiment of a service configuration environment 1900 within which a service 230 is configured at runtime is illustrated. Within the NIO platform 1802, each service 230 is created using a class file 202 and configuration information 1902. The configuration information includes predefined information that exists
10 before runtime (e.g., as part of the platform configuration information 210 of FIG. 2A) and information that is dynamically generated at runtime. The dynamically generated information is not known until the NIO platform 1802 is launched and may include information described with respect to the environment 1300 of FIG. 13.

[0279] The class file 202 may be used by multiple services, but the configuration information 1902 is unique to the particular service 230 being created. The configuration information 1902 may be in a separate file for each service 230 or may be in a larger file from which a particular service's configuration information is extracted. At runtime, the class file 202 is instantiated and then the configuration information 1902 is applied to the instantiated service object.
15

[0280] Referring to FIG. 19B, one embodiment of a block configuration environment 1904 within which a block 232 is configured at runtime is illustrated. Within the NIO platform 1802, each block 232 is created using a class file 204 and configuration information 1906. The configuration information includes predefined information that exists before runtime (e.g., as part of the platform configuration information 210 of FIG. 2A) and
20 information that is dynamically generated at runtime. The dynamically generated information is not known until the NIO platform 1802 is launched and may include information described with respect to the environment 1300 of FIG. 13.

[0281] The class file 204 may be used by multiple blocks, but the configuration information 1906 is unique to the particular block 232 being created. The configuration information 1906 may be in a separate file for each block 232 or may be in a larger file from which a particular block's configuration information is extracted. At runtime, the class file 204 is instantiated and then the configuration information 1906 is applied to the instantiated block object.
25
30

[0282] Referring to FIGS. 20A and 20B, embodiments of class files 204a and 204b for blocks 232a and 232b (not shown), respectively, are illustrated. Within the NIO platform 1802, the service class files 202 and block class files 204 are based on a base service template (for services 230) and a base block template (for blocks 232), respectively. These base
5 templates include NIO platform specific behavior that is inherited by any class that extends them. This means that each service class 202 and block class 204 inherits NIO platform specific behavior that allows the corresponding service 230 or block 232 to work within the NIO platform architecture. Without this NIO platform specific behavior, the class files 202 and 204 would not be recognized within the NIO platform architecture and so the
10 corresponding services 230 and classes 232 could not be created. In addition to the NIO platform specific behavior, each block class 204 contains executable instructions that provide particular task specific functionality.

[0283] Referring specifically to FIG. 20A, the class file 204a for Block Class 1 includes the standard base block code for the NIO platform and also contains custom code for
15 connecting to an external signal source, which is Twitter for purposes of example. Referring specifically to FIG. 20B, the class file 204b for Block Class 2 includes the standard base block code for the NIO platform and also contains custom code for sending email.

[0284] If there is not an existing block class that contains the code needed to perform a particular task, either a new block class can be created using the base block template or an
20 existing block class 204 can be modified. While service classes 202 can also include custom code, they rarely do so because the base service template generally provides all the functionality needed for a service 230. However, it is understood that service classes 202 can also be customized.

[0285] Referring to FIG. 21, one embodiment of an environment 2100 within which
25 configuration information is used to configure two blocks 232 based on the same block class 204 in different ways is illustrated. The configuration information 1906 (FIG. 19B) allows configuration of a particular block 232 at runtime by setting the values of configurable parameters defined within the block class 204. This means that the same block 232 can be configured in different ways depending on the values in the configuration information 1906
30 that is used to configure the block 232.

[0286] The block class 204b (as shown in FIG. 20B) contains custom code to send any information received by the block 232 to a destination email address. The code includes a configurable parameter for the destination email address to avoid having to change the underlying block class 204 each time a different email address is used. This allows the email

address to be defined in the configuration information 1906, which means that the same block class 204 can be used to create multiple blocks that send their emails to different addresses.

[0287] Accordingly, in the present example, the block class 204b is to be used to instantiate two blocks 232a (also referred to as Block #1) and 232b (also referred to as Block #2). The blocks 232a and 232b are to be configured to send email to two different addresses using configuration information 1906a (also referred to as Block #1 configuration information) and 1906b (also referred to as Block configuration information #2), respectively. When the blocks 232a and 232b are instantiated and configured, the two blocks will have the same email sending functionality, but will send their emails to different email addresses.

[0288] Referring to FIG. 22, one embodiment of an environment 2200 within which configuration information is used to configure two services 230 based on the same service class 202 in different ways is illustrated. The configuration information 1902 (FIG. 19A) allows limited configuration of a particular service 230 at runtime by defining which blocks 232 are to be executed by the service and the order of execution of the blocks 232. The configuration information 1902 may also be used to set the values of configurable parameters defined within the service class 202. This means that the same service 230 can be configured in different ways depending on the blocks 232, the order of execution, and the values in the configuration information 1902 that is used to configure the service 230.

[0289] In the present example, the configuration information 1902 for a service 230 includes source blocks and destination blocks needed to build a routing table when the service 230 is instantiated. Because the blocks 232 do not have any connection to each other except through the service 230, the service 230 uses the routing table to direct information from one block (a source block) to the next block (a destination block). The service 230 receives the source and destination blocks as configuration information after the service 230 is instantiated, so the same underlying service class 202 can be used for different services 230. This means that the services 230 can have different functionality based on the particular blocks 232 and block execution order defined in their configuration information 1902.

[0290] Accordingly, in the present example, a service class 202 is to be used to instantiate two services 230a (also referred to as Service #1) and 230b (also referred to as Service #2). The services 230a and 230b are to be configured using different blocks and different orders of execution using configuration information 1902a (also referred to as Service #1 configuration information) and 1902b (also referred to as Service configuration information

#2), respectively. When the services 230a and 230b are instantiated and configured, the two services will have different functionality.

[0291] In the present example, the fact that a service 230 is made up of a service class 202 and configuration information 1902 means that, prior to instantiation, there is no service class 202 that can be examined to determine the execution order of blocks 232, or even the blocks 232 that are to be used, within the service 230. To determine the behavior of the service 230, the configuration information 1902 would have to be examined.

[0292] Referring to FIG. 23, one embodiment of an environment 2300 is illustrated with a base block class 1806 that is extended to create various customized block classes (not shown), such as those in the library 1804 of FIG. 18. The customized block classes can then be instantiated as described previously to form various blocks 232a-232j. As described above, a NIO platform operates by using a service 230 to organize the appropriate blocks 232 to perform a particular task. In the present example, the blocks 232 do not have any connection to each other except through the service 230. This organizational structure provides benefits such as asynchronicity in block execution, dynamic expansion and retraction of block resources in response to input changes, and the ability to modify services 230 and blocks 232 without having to restart the NIO platform 1802.

[0293] For example, as shown in FIG. 23, the environment 2300 includes a block library that contains the ten blocks 232a-232j. Each of the blocks 232a-232j is built from the base block template, so each block is compatible with the NIO platform architecture. The blocks 232a-232j have no connection to each other except that all of them can operate within the NIO platform architecture. Each block 232a-232j contains task specific code that allows that block to perform a particular function. For example, the block 232a connects to Twitter, the block 232b sends an email containing any information received from another block, the block 232c connects to a machine in an assembly line, the block 232d filters any input received from another block for one or more defined text strings, the block 232e sends a signal to turn off the machine on the assembly line, and so on.

[0294] Assume that a user wants to create two different services 230a and 230b using the ten blocks 232a-232j. Service 230a is to monitor an external source (e.g., Twitter) Twitter for the words "company name" and send an email to user1@companyname.com if such a tweet is detected. Service 230b will monitor an assembly line machine for the occurrence of certain error codes and send an email to user2@companyname.com if an error is detected. Service 230b will also shut the machine down if an error is detected. Services 230a and 230b

are to run simultaneously on a single NIO platform and perform their tasks asynchronously and in real time without any data storage.

[0295] With additional reference to FIG. 24, one embodiment of the service 230a is illustrated using blocks from the environment 2300 of FIG. 23. Service 230a is created by identifying the needed block classes and defining their order of execution. For example, the block 232a (connecting to Twitter) will be followed by the block 232d (filtering for “company name”), and then the block 232b will send an email to user1@companyname if block 232d identifies any tweets with “company name.” The block classes include configurable parameters that allow them to be customized without needing to open the block classes and change their code. FIG. 24 illustrates the configured appearance of the service 230a from a functional perspective.

[0296] The routing table for the service 230a defines the destination block for any output from a source block. If a block does not send output to another block (i.e., the block 232b), there is no entry in the routing table. There is no source block for block 232a because block 232a is connecting directly to Twitter. Table 3 illustrates an example of a routing table for the service 230a.

[0297]

Service 230a	
Source Block	Destination Block
Block 232a	Block 232d
Block 232d	Block 232b

Table 3

[0298] The decoupled nature of the blocks and the flexibility provided by the routing table allow the service 230a to be modified or blocks swapped for other blocks relatively easily. It is understood that any configuration changes and any new blocks must be loaded into the NIO platform (assuming the new blocks are not already there) and then the service 230a must be restarted for changes to take effect. For example, if a user wants to swap the email block 232b for a text message block, block 232b can be replaced with a suitably configured block for sending texts. If the block’s name remains the same, the routing table may not even change in some embodiments. If the block’s name is different, the routing table needs to be updated, but no other change may be needed. Table 4 illustrates an example of the routing table for the service 230a with the block 232b replaced by a text message block 232g.

[0299]

Service 230a	
Source Block	Destination Block
Block 232a	Block 232d
Block 232d	Block 232g

Table 4

[0300] If the user wants to send both the text message and the email, then the text message block 232g can be added so that it exists within the service 230a alongside the email block 232b. In this case, the routing table can be updated to include the new block 232g as another destination for source block 232d. Table 5 illustrates an example of the routing table for the service 230a with both block 232a and block 232g.

[0301]

Service 230a	
Source Block	Destination Block
Block 232a	Block 232d
Block 232d	Block 232b, 232g

Table 5

[0302] With additional reference to FIG. 25, one embodiment of the service 230b is illustrated using blocks from the environment 2300 of FIG. 23. Service 230b is created by identifying the needed block classes and defining their order of execution. For example, the block 232c (connecting to the machine) will be followed by the block 232d (filtering against an error list). If an error is detected, the block 232b will send an email to user2@companyname and the block 232e will shut down the machine. The block classes include configurable parameters that allow them to be customized without needing to open the block classes and change their code. FIG. 25 illustrates the configured appearance of the service 230b from a functional perspective. Table 6 illustrates an example of a routing table for the service 232b.

[0303]

Service 230b	
Source Block	Destination Block
Block 232c	Block 232d
Block 232d	Block 232b, 232e

Table 6

[0304] Referring to FIG. 26, one embodiment of the NIO platform 1802 is shown within an environment 2600. The environment 2600 includes access to Twitter 2602 and a machine 2604. As shown, the NIO platform 1802 includes a core 228 and is running the two services 230a and 230b simultaneously. Each service 230a and 230b performs its configured functions independently of the other service.

[0305] Referring to FIG. 27, a method 2700 illustrates one embodiment of a process that may be executed by the NIO platform of FIG. 26 to create and configure a block 232. In step 2702, a block class 204 is identified along with the block's corresponding configuration information and dynamically generated information needed for the block 232. In step 2704, the block 232 is instantiated from the block class 204. In step 2706, the block 232 is configured using the corresponding configuration information and dynamically generated information.

[0306] Referring to FIG. 28, a method 2800 illustrates one embodiment of a process that may be executed by the NIO platform of FIG. 26 to create and configure a service 230. In step 2802, a service class 202 is identified along with the service's corresponding configuration information and dynamically generated information needed for the service 230. In step 2804, the service 230 is instantiated from the service class 202. In step 2806, the service 230 is configured using the corresponding configuration information and dynamically generated information.

[0307] Referring to FIG. 29, one embodiment of an environment 2900 is illustrated in which a real time publishing system 2901 asynchronously captures, normalizes, filters, prioritizes, and publishes data from multiple data streams in real time or near real time. Unlike conventional systems, the real time publishing system 2901 does not need to store data in a data repository (e.g., a database) after capturing the data, but is instead able to process the data as it is received directly from the data sources and publish the data for viewing to a display. The display is updated in real time or near real time whenever additional data is captured from the sources, providing a dynamically updating display with consistently refreshing information.

[0308] In the present example, a NIO platform 2902, which may be similar or identical to the previously described NIO platform 100 (FIG. 1), NIO platform 200 (FIG. 2A), or NIO platform 900 (FIG. 9A), may be used with one or more publishing servers 2904 (e.g., a web services server for publication via a website) to provide the real time publishing system 2901. The NIO platform 100 receives data from one or more external sources 2906a, 2906b, 2906c, ..., and 2906N, where N is the total number of external sources. The external sources 2906a,

2906b, 2906c, ..., and 2906N produce data 2908a, 2908b, 2908c, ..., and 2908N, respectively, that is captured by the NIO platform 2902.

[0309] The NIO platform 2902 processes and feeds the data (in niograms or a different format) to the publishing server 2904, which handles the process of publishing the information for viewing. The publishing server 2904 may publish to one or more channels, such as the web 2910, email 2912, SMS 2914, feed 2916 (e.g., RSS feeds), post 2918 (e.g., to social media sites), voicemail 2920, and/or any other channel 2922. For purposes of example, the publishing server 2904 is a web services server that is configured to publish the information via one or more web pages.

[0310] The real time publishing system 2901 is configured to obtain data from the external sources 2906a-2906N, process the data, and publish the data in real time or near real time. The NIO platform 2902 is configured to send a relatively steady stream of data (e.g., a minimum number of niograms per second) to the publishing server 2901 for publication. The publication may occur in many different ways, but in the present example is accomplished via a matrix of tiles (described below in greater detail) that is refreshed on a relatively continuous basis.

[0311] The real time publishing system 2901 has no control over the external sources 2906a-2906N and cannot be sure that new data will be received on a regular basis. As new data may not always be available for use in refreshing the matrix, the NIO platform 2902 may cache data and use the cached data to continually refresh the matrix. Without the cached data, there would be nothing for the NIO platform 2902 to send to the publishing server 2904 if no new data had been received when it was time to send. Accordingly, new data may be sent to the matrix in real time or near real time, and cached data may be used when needed to ensure that the publishing server 2904 receives a steady supply of data to display.

[0312] Referring to FIG. 30, one embodiment of an environment 3000 illustrates the NIO platform 2902 of FIG. 29 as configured to provide various components/functions that both capture data and prepare the captured data for publishing. It is understood that the functionality provided by the components may be organized in many different ways, and may be spread across additional components or combined into fewer components.

[0313] In the present example, the functionality includes data components 3002a, 3002b, 3002c, ..., and 3002M that obtain information from external sources 2906a-2906N and then transform and filter the data. In the present example, the process of data retrieval, transformation, and filtering occurs on a per source basis. The data retrieval components

3002a-3002M may have a one to one correspondence with the external sources 2906a-2906N, or a single data component may connect to more than one external source.

[0314] The functionality also includes a prioritization component 3004 that receives the transformed and filtered data from the data components 3002a-3002M and prioritizes the data for publishing. As will be described below in greater detail, this allows particular types of data and/or particular external sources to be given priority in publishing, which in turn allows the data that is actually displayed to be tuned as desired. Although the prioritization component 3004 performs prioritization for all data components 3002a-3002M, it is understood that a separate prioritization component may exist for each of the data components 3002a-3002M.

[0315] In the present example, the prioritization component 3004 sends the prioritized data to both a queuing component 3006 and directly to a delivery component 3008. The queuing component 3006 provides one or more queues in which data is temporarily stored to ensure that the NIO platform 2902 always has data to send to the web services server 2904. The direct path to the delivery component 3008 provides real time or near real time data for the server 2904 without the delay introduced by the queue component 3006.

[0316] Referring to FIG. 31, one embodiment of the NIO platform 2902 of FIG. 30 is illustrated in greater detail with source services 3102a-3102M and a stream service 3104. The source services 3102a-3102M and stream service 3104 are services configured to operate within the NIO platform 2902 as described in previous embodiments.

[0317] The NIO platform 2902 includes multiple source services 3102a, 3102b, 3102c, ..., and 3102M. The source services 3102a-3102M are configured to obtain data from the external sources 2906a-2906N, respectively. In the present example, the source services 3102a-3102M include blocks (not shown) that perform the data retrieval, transforming, and filtering functionality described with respect to FIG. 30. The source services 3102a-3102M pass the data to the stream service 3104, which handles the prioritization, queuing, and delivery described with respect to FIG. 30.

[0318] It is understood that one or more of the source services 3102a-3102M and the stream service 3104 may be combined into a single service in other embodiments. In still other embodiments, the source services 3102a-3102M and the stream service 3104 may be further divided. For example, one or more additional services may be created and configured to handle the transforming functionality, the filtering functionality, the prioritization functionality, the queuing functionality, and/or the delivery functionality. As described

previously, the NIO platform 2902 may be configured in many different ways with many different services and/or blocks to provide the same functionality.

[0319] Referring to FIG. 32, one embodiment of the source service 3102a of FIG. 31 is illustrated. In the present example, the source service includes a connect block 3202, a transform block 3204, a white list filter block 3206, a black list filter block 3208, and a publish block 3210. When a block 3202-3210 is finished performing its configured processing, it calls Blockrouter.notifySignals() as previously described and the service's block router calls processSignals() on the next block.

[0320] The connect block 3202 is configured to connect to a specific external source, such as Twitter, Facebook, Google+, Instagram, Youtube, or another source that provides content. The type of connection may be based on the particular site, with some sites pushing data to the connect block 3202 and other sites requiring that the connect block 3202 be configured to pull data from those sites. Accordingly, the actual configuration of the connect block 3202 may vary depending on the API of the external source, the type of information being obtained (e.g., streaming data versus other non-streaming data types), whether authentication credentials are needed for access, and/or other factors.

[0321] The transform block 3204 converts the obtained data into niograms, with each niogram containing a single piece of content (e.g., a Facebook post, a tweet, an article with accompanying pictures, or a Youtube vide). In the present example, regardless of the external source and the type of data (e.g., video, audio, images, and/or text), the data will be placed into defined fields in a niogram. This deconstruction process normalizes the data for later processing. The transform block 3204 may retain all information obtained from an external source or may discard information. The transform block 3204 may also insert other information into the niogram, such as a source type (e.g., Twitter or Facebook), a username corresponding to the source data, text (e.g., relevant text for each content source), a link to the content, a unique ID (e.g., either global or on a local basis such as per type), and/or a status flag (e.g., "old," "new," or "VIP").

[0322] The filter blocks 3206 and 3208 perform filtering against the contents contained in the fields of the niogram. The white list block 3206 contains text (e.g., words or phrases) that must be present in the fields to pass the filter, and the black list block 3208 contains text that must not be present in the fields to pass the filter. The text may be in posts, image captions, user names, and/or elsewhere, and may vary depending on the particular source and/or the contents to be published. The filter blocks 3206 and 3208 enable content to be screened to

prevent false positives and to remove unwanted (e.g., objectionable, offensive, and/or irrelevant) material.

[0323] One or more custom blocks 3210 may be positioned anywhere within the service 3102 or may be omitted entirely. In the present example, no custom blocks 3210 are used.

5 [0324] The publish block 3212 publishes the transformed and filtered niograms for use by the service 3104. For example, the publish block 3212 may publish to a publisher_sources channel that is common to all of the source services 3102a-3102M.

[0325] It is understood that one or more of the blocks 3202-3212 may be combined into a single block in other embodiments. In still other embodiments, one or more of the blocks
10 3202-3212 may be further divided.

[0326] Referring to FIG. 33A, one embodiment of the stream service 3104 of FIG. 31 is illustrated. In the present example, the stream service includes a subscribe block 3302, a queue (by type) block 3304, a timestamp block 3306, a freshness block 3308, a priority block 3310, a queue (by priority) block 3312, and an output block 3314. When a block 3302-3312
15 is finished performing its configured processing, it calls Blockrouter.notifySignals() as previously described and the service's block router calls processSignals() on the next block.

[0327] The subscribe block 3302 is subscribed to subscriber_sources, which receives all niograms published by the source services 3102a-3102M via publisher_sources. The subscribe block 3302 passes received niograms to both the queue block 3304 and the
20 timestamp block 3306.

[0328] The queue (by type) block 3304, along with the queue (by priority) block 3312, may be used to provide cached data to ensure that a steady stream of data is sent for publication to the web services server 2904. In the present example, the two separate queue blocks are used to ensure that content is published based on both type and priority.

25 [0329] The queue block 3304 stores a queue for each type of external source (e.g., Twitter, Facebook, and Google+). Each queue is configured to hold a defined number of niograms. The queue block 3304 emits a niogram from each queue as defined intervals (e.g., every X milliseconds or seconds) and automatically reloads the niograms that are popped off the queue to the end of the queue to be recycled through. A new addition to the queue pops
30 the queue, discarding the front niogram and adding the new niogram to the end of the queue. The reloading ensures that the queue will remain full even without new niograms being added. Accordingly, if an external source does not produce new content frequently, the niograms in the queue for that type will typically be older than the niograms in a queue that corresponds to a frequently updated type.

[0330] The queue block 3304 may use a value (e.g., a throttle) to spread out the emitted niograms. For example, if there are eight types and the queue block 3304 is configured to emit a niogram for each type every five seconds, then eight niograms are emitted every five seconds. The throttle may be used to queue up the eight niograms so that their emission is spread relatively evenly across the five second window.

[0331] The timestamp block 3306 assigns a timestamp to each niogram that identifies a particular reference time, such as when that niogram's content was received by the timestamp block 3306. Because niograms are initially passed through to the timestamp block 3306 in real time or near real time, the timestamp will be close to the time the niogram was received by the corresponding source service.

[0332] In embodiments that use freshness, the freshness block 3308 assigns a freshness level to each niogram. Each niogram is assigned a freshness level that changes over time (e.g., a niogram loses its "freshness" over time). Different levels of freshness and different rates of decline may be applied based on content (e.g., text may lose freshness faster than images), type, and/or other factors. The freshness levels enable the stream service 3104 to tune the content to be displayed by the web services server 2904 based on how long ago the content was received.

[0333] The priority block 3310 assigns a priority to each niogram based on one or more defined criteria. For example, a priority scale of 1-5 may be used, with niograms being assigned priority based on content (e.g., images may be assigned a higher priority than text), type, and/or other criterion. Freshness may affect priority level, with the priority level of a niogram decreasing as its freshness decreases.

[0334] The queue (by priority) block 3312, which may be similar or identical to the queue block 3304 in its manner of operation, stores a queue for each priority of niogram.

[0335] The output block 3314 receives niograms and sends them to the web services server 2904. The output block 3314 may be a web socket that is able to communicate directly with the web services server 2904.

[0336] Referring to FIG. 33B, one embodiment of a method 3320 illustrates a process that may be used within the NIO platform 2902 of FIG. 29. It is noted that the method 3320 focuses on the real time or near real time aspect of publishing and does not describe queueing. In step 3322, media is obtained from an external source. In step 3324, the media is transformed and inserted into a niogram. Additional tagging may also occur in this step. In step 3326, the media is filtered based on white list and/or black list information.

[0337] In step 3328, a determination is made as to whether to discard the niogram based on the filtering of step 3326. If the niogram is to be discarded, the method 3320 moves to step 3330 and discards the niogram. If the niogram is not discarded, the method 3320 moves to step 3332. In step 3332, a timestamp is added to assign a reference time to the niogram.

5 In step 3334, a freshness level of the media is set in embodiments that use a freshness level to control the display of the media. In step 3336, a priority is assigned to the media. In step 3338, the media is output to the server 2904 for display.

[0338] Referring to FIG. 34, the web services server 2904 uses the prioritized data stream from the NIO platform 2902 to fill and refresh a matrix. In the present example, the web
10 services server 2904 has two main functions. The first function is an assignment function 3402 with which the web services server 2904 assigns a niogram to a tile. As will be described below, not all niograms may be assigned to a tile. The second function is a publication function with which the web services server 2904 publishes the niograms via the matrix for viewing on one or more device displays 3406.

15 [0339] Referring to FIG. 35, one embodiment of a matrix 3500 is illustrated. The matrix 3500 represents one example of how the real time publishing system 2901 of FIG. 29 may organize and present published data for viewing on a website, within a mobile device application, and/or elsewhere.

[0340] In the present example, the web services server 2904 determines where the
20 prioritized data belongs in the matrix 3500, which may then be published to a display. For example, the matrix 3500 may be defined using cascading style sheets (CSS). The matrix 3500 represents a portion or all of a display area (e.g., within a web browser or other viewing software) that can be displayed on an analog or digital display. As shown, the matrix 3500 is divided into sixteen tiles (e.g., cells) 3501, 3504, 3506, ..., and 3532, and each tile 3501-3532
25 represents a unique area of the matrix 3500 in which data can be displayed.

[0341] It is understood that the matrix 3500 can be of any size and shape, but is rectangular in the present embodiment for purposes of illustration. While rectangular, the matrix 3500 may have irregular columns and/or rows as shown, or the columns and/or rows may be regular. Furthermore, the matrix 3500 and/or one or more of the tiles 3501-3532 can
30 be fixed or dynamic.

[0342] With additional reference to FIGS. 25A-D, a portion of the matrix 3500 is illustrated with tiles 3502, 3504, 3510, and 3512. In the present example, tiles 3502, 3504, and 3512 are assigned changing content and tile 3510 is static (e.g., the content does not change).

[0343] Because the tiles 3502, 3504, and 3512 contain content that changes, it may be difficult for a user to read or otherwise view a tile's contents before the content changes. As the content may not appear in the matrix 3500 again or may appear in another tile at some seemingly random time from the user's perspective, the constantly changing tiles may be difficult to view. Accordingly, the tiles 3502, 3504, and 3512 may include user selectable regions 3602, 3604, and 3606, respectively. The user selectable regions 3602, 3604, and 3606 may be any shape and/or size, and may be positioned anywhere within their respective tiles 3502, 3504, and 3512.

[0344] Each user selectable region 3602, 3604, and 3606 enables the respective tile to be locked, which prevents the contents from changing until the tile is unlocked. For example, in FIG. 36A, none of the user selectable regions 3602, 3604, and 3606 are selected and the content in the respective tiles 3502, 3504, and 3512 is changing. In contrast, in FIG. 36B, the user selectable region 3602 has been selected, which locks the tile 3502 and prevents the contents of the tile 3502 from being refreshed with new content. The user selectable regions 3604 and 3606 are not selected and the content in the respective tiles 3504 and 3512 continues to change.

[0345] Deselection of the user selectable area 3602 (e.g., selecting the user selectable area a second time) will unlock the tile 3502 and allow the contents to be refreshed. When the tile 3502 is unlocked, the tile 3502 may refresh instantly or may refresh using one or more other criterion. For example, a timer corresponding to the tile 3502 may continue running while the tile 3502 is locked. In this case, if the timer expires while the tile 3502 is locked, the tile 3502 may refresh once unlocked based on the expired timer. In other embodiments, locking the tile 3502 may freeze the timer and unlocking the tile 3502 may unfreeze the timer. In this case, the timer may continue normally once unfrozen and the tile 3502 may be refreshed when the timer expires.

[0346] Referring specifically to FIGS. 36C and 36D, selection of a tile 3502, 3504, 3510, and 3512 may enlarge that tile if the tile is configured to be enlarged. For example, selection of the tile 3502 (e.g., user selection of a portion of the tile 3502) in both FIGS. 36C and 36D has enlarged the tile. In some embodiments, the contents of the enlarged tile 3502 may continue to change, as shown in FIG. 36C. In other embodiments, the contents may be locked when the tile 3502 is enlarged, as shown in 36D. The tile 3502 may be unlocked when the tile 3502 is minimized. In the case where enlarging the tile 3502 locks the tile's contents, a corresponding timer may operate as previously described.

[0347] Referring to FIG. 36E, one embodiment of the tile 3502 is illustrated in greater detail. In the present example, the tile 3502 may include a header area 3610 delineated by a line 3612. The header area 3610 may include the user selectable area 3602, a user name 3614 (if applicable) corresponding to the source of the media currently being displayed by the tile 3502, and an indicator 3616 (e.g., an icon) identifying the type of contents. The tile 3502 may also include a footer area 3618 delineated by a line 3620. The footer area 3618 may include a source link corresponding to the source of the media currently being displayed by the tile 3502, enabling a user to select the source link to load the media from the original source. For example, if the user is viewing the matrix in a browser, selecting the link may open the source in another browser tab or another browser window.

[0348] In some embodiments, the header area 3610 and/or footer area 3618 may only be displayed when a trigger occurs, such as a user mousing over or otherwise selecting the tile. In some embodiments, the header area 3610 and/or footer area 3618 may cover a portion of the media being displayed by the tile 3502, while in other embodiments the media may be fully displayed between the header area 3610 and the footer area 3618. In some embodiments, the header area 3610 and/or footer area 3618 may be partially transparent. In some embodiments, the tile may be assigned a particular color based on type or another attribute, so that text contents are displayed with different color backgrounds depending on the source of the text.

[0349] The matrix 3500 may be filtered in many different ways. For example, a user may select the username 3614 to show only content from that user. A user may also select the indicator 3616 to display only content from that source. Filters may be stacked, with selection of one or more usernames and/or sources enabling customizable displays to be created.

[0350] With additional reference to FIG. 37, a method 3700 illustrates one example of a process that may be used within the real time publishing system of FIG. 29 to assign data to various tiles 3502-3532 of the matrix 3500. It is understood that this is a real time or near real time process, with data being identified, assigned to one of the tiles 3502-3532, and published for display in real time or near real time without storing the data in a data repository.

[0351] In step 3702, content is identified for publication. In steps 3704 and 3706, the particular tile 3501-3532 to be used to display the content is determined and the content is assigned to the tile. The tile assignment occurs in a defined manner. In some embodiments, one external source may be assigned to one tile 3501-3532 of the matrix 3500 or to multiple

tiles (e.g., with data interleaved among the tiles or otherwise assigned). For example, external source 2906a may be assigned to tiles 3501, 3506, 3510, 3514, 3518, 3528, and 3532. External source 2906b may be assigned to tiles 3504, 3508, 3522, 3524, and 3530. External source 2906N may be assigned to tiles 3512, 3516, and 3526. Tile 3520 is not assigned to any external source. Tile 3520 may be used to provide articles, comments, advertisements, and/or any other desired content. In some embodiments, tile 3520 may be used for overflow if needed, such as when a spike occurs as will be described later.

[0352] Within the tiles assigned to a particular external source, the data may be updated in various ways. For example, the newest data may be assigned to the next tile to be refreshed for that external source, or may be assigned to a tile selected from an available group of tiles (e.g., a group of all tiles within five seconds of a refresh). Selecting the tile from a group may aid in providing a level of randomness to the matrix 3500.

[0353] This external source grouping of tiles for the assignment process allows advertisers or other sponsors to reserve certain tiles. This assignment process also enables the real time publishing system 2901 to reserve more tiles for external sources with a higher volume of data, which allows some level of load balancing by controlling the assignment of tiles 3501-3532.

[0354] In other embodiments, the data may be randomly assigned to tiles rather than tying a tile to a particular external source, and tile assignments may change frequently. For example, the newest data may be assigned to the next tile to be refreshed regardless of the external source from which the data was obtained, although this may depend on certain criteria, such as whether the tile is inappropriate for the content (e.g., too small to display an image or a block of text).

[0355] When randomly assigned, the tiles may be grouped so that each external source has a certain number of tiles. For example, external source 2906a may be assigned X tiles (with X being a whole number from one to sixteen), tiles forming a percentage area of the matrix 3500 (e.g., tiles forming at least fifty percent of the matrix area), a number of tiles between X and Y (with X being a whole number equal to or greater than one and less than Y, and Y being a whole number from X+1 to sixteen), or X tiles that are all larger than a specified minimum size per tile. Accordingly, the tiles may be assigned in many different ways.

[0356] Tiles may also be assigned based on content. For example, tweets may always be assigned to tiles 3506 and 3528. Images may always be assigned to tiles 3501, 3504, 3520, and 3516. Text may always be assigned to tile 3518. The other tiles may be similarly linked

to types of content or may be assigned as previously described. This enables a designer to distribute content throughout the matrix 3500 as desired in a known pattern, regardless of the source of the content.

5 [0357] Regardless of the amount of data being received, the real time publishing system 2901 is configured to publish as much new content as possible given the constraints within which the publication occurs (e.g., the number of tiles in the matrix 3500). It is understood that content may be throttled or otherwise limited based on user preferences, bandwidth limitations, and/or similar factors, and so the amount of new content that is published may vary among users.

10 [0358] In some embodiments, to determine which content should be shown, content may be prioritized based on source, type of content (e.g., images may have a higher priority than text), freshness, a relevancy weighting based on a number of factors, and/or other criteria. The priority of a piece of content may be added to the content (e.g., as tags) and, when the content is to be assigned to a tile, may be used to determine whether to replace the content
15 with other content. Accordingly, the display attributes of a tile may change based on the contents currently being displayed by that tile. This means that tiles may be prioritized for replacement based on their content.

[0359] In the present example, tile assignment prioritization is based on making sure that as much new content as possible is being published, which generally means that the real time
20 publishing system 2901 replaces older content with new content whenever new content become available. This means that tile assignment prioritization decisions can be focused on what tile is to be replaced in the matrix 3500. While simply assigning the new content to tiles containing the oldest content in the matrix 3500 (or in the group of tiles assigned to the particular source) is one approach that may be used by the real time publishing system 2901,
25 other approaches may focus on more than simply tile content age, such as the priority level of the tile's contents.

[0360] More specifically, while tile assignment prioritization may be an age based decision in part, in some embodiments, the age may not be the only factor or even the dominant factor in deciding what tile is to be replaced. For example, the real time publishing
30 system 2901 may be configured to replace newer content in the matrix 3500 before older content if the older content has a higher priority rating than the newer content. In some embodiments, age may be viewed as a sliding scale priority factor in determining replacement priority, so the priority rating of a tile's content may decrease over time as the content ages. In other words, as a tile's contents get older, the contents' priority level may

drop in some manner (e.g., continuously or based on a step function). This dropping may continue until the tile's contents fall below the priority level of newer content that originally had a lower priority level. Using this approach, the tile that will be replaced with new content is the tile with the oldest content and/or lowest priority level.

5 [0361] In step 3708, a determination may be made as to whether the assigned tile is ready to be refreshed. The tiles 3501-3532 within the matrix 3500 are refreshed when a trigger occurs, such as when new data is captured and/or when a timer expires. For example, a trigger may occur when new data is captured, but the tile may not be updated until a timer expires. The timer prevents the tile from being refreshed too rapidly when new data is being
10 received in a short amount of time. For example, if the timer is set to two seconds and five pieces of data arrive for that tile in two seconds, the timer may prevent the last four pieces of data from being displayed on arrival as a user would likely have difficulty consuming the data that quickly.

[0362] The trigger may be operate on a per tile basis (e.g., each tile may have its own
15 timer) or may operate multiple tiles or the entire matrix. Generally, some or all tiles 3501-3532 in the matrix 3500 will be regularly refreshed to maintain an appearance of dynamic data. In some embodiments, at least one tile may be refreshing at any particular time.

[0363] Different triggers may be assigned to different types of data. For example, images and tweets may have different refresh rates, which may affect the refresh rate of the tile to
20 which that content has been assigned. When the refresh rate is tied to the content itself, some tiles 3501-3532 may be omitted from selection. For example, tile 3530 may be considered too small for an image, and so may not be considered if an image is the next content to be assigned to a tile.

[0364] If the determination of step 3708 indicates that the tile is not ready, the method
25 3700 may repeat step 3708 until the tile is ready. If the tile is ready, the method 3700 moves to step 3710 and updates the tile.

[0365] A more detailed example of prioritization for contents within the matrix 3500 is illustrated below with respect to Table 7.

Priority Level	Minimum Duration	Minimum Duration (New Content)	Maximum Duration
1	7	1	20
2	15	2	30
3	20	5	40
4	30	10	60

5	45	15	240
---	----	----	-----

Table 7

[0366] Table 7 shows five priority levels, with level one being the lowest and level five the highest. Each priority level is associated with three durations, which are in seconds in the present example. Each priority level has a minimum duration, a minimum duration for new content, and a maximum duration.

[0367] The minimum duration is the shortest period of time that the content assigned to that tile will be displayed unless new content is available. This duration setting enables higher priority content to remain in the matrix 3500 longer than lower priority content. The maximum duration is the longest period of time that the content will be displayed before being refreshed, even if the refresh uses cached content. This duration setting ensures that the matrix 3500 will be periodically refreshed with different content. The minimum duration for new content is the amount of time that the current content will be shown before being refreshed with the new content. This duration setting ensures that new content is pushed to the matrix 3500 over previous content, but may also be set to prevent content from being refreshed so quickly that it cannot be consumed.

[0368] Based on Table 7, there are three time periods involved assuming no new content and/or higher priority content is available. Before the minimum duration is reached, after the maximum duration has been reached, and between the minimum duration and the maximum duration. Before the minimum duration is reached for a tile, the tile's contents will not be replaced. Past the maximum duration, the tile's contents will always be replaced, even by lower priority contents, with the oldest tile replaced first.

[0369] Any tiles falling between the minimum and maximum durations are compared based on the percentage of time remaining in their time window. For example, a priority two tile has minimum and maximum durations of fifteen and thirty seconds, respectively, for a fifteen second window. A priority three tile has minimum and maximum durations of twenty and forty seconds, respectively, for a twenty second window. Assume that the contents of the priority two tile have been displayed for twenty seconds and the contents of the priority three tile have been displayed for thirty seconds. The priority two tile is approximately thirty-three percent through its fifteen second window and the priority three tile is fifty percent through its twenty second window. In this case, the priority three tile would be replaced because it has used more of its display time than the priority two tile. In other embodiments, windows may only be compared for equal priority tiles, and lower priority tiles may always be refreshed before higher priority tiles regardless of the amount of time remaining.

[0370] If new content is available, the minimum duration for new content is applied to the tile. For example, a priority one tile may have displayed its information for five seconds when it is checked. If the available content is not new, nothing happens because the seven second minimum duration time has not expired. However, if the available content is new, the minimum duration for new content is applied and the tile is refreshed because the minimum duration is one second and the timer is at five seconds. This enables existing content to be refreshed quickly when new content becomes available, with lower priority content being replaced more rapidly than higher priority content.

[0371] In some embodiments, there may be a priority level that is always given priority. For example, a VIP level may mean that a tile is refreshed even if no tiles are ready to be refreshed. Designating content as VIP ensures that the content will be instantly published regardless of tiles states. It is understood that the tile being replaced may still be the tile that is the oldest, has the least time remaining, etc., but even the minimum duration for new content period will be overridden to display the new content. For example, assume that the next tile to be replaced is a priority five tile that has only used five seconds of its fifteen second minimum duration for new content. If VIP content is available, the remaining ten seconds will be ignored and the tile will be refreshed with the VIP content.

[0372] It is understood that the different types of tile assignment can be mixed as desired. This provides a great deal of flexibility in presenting the content and enables the matrix 3500 to be dynamically adjusted as the content changes.

[0373] In the present embodiment, content is published to the matrix 3500 and then replaced with other content in real time or near real time. While some content may be cached temporarily and delivered (e.g., to quickly fill the matrix 3500 when the matrix is first loaded) and/or redelivered (e.g., in situations where no new content has been received but a tile must be refreshed), the real time publishing system 2901 does not store the content in a data repository, and once the content is discarded by the system, it cannot be retrieved and redisplayed. In other words, there is no “back” button or other mechanism by which a user can move backwards in time through a tile’s contents. The real time aspect of the system means that information is constantly coming in and being used to replace previous information and the previous information is no longer available. It is understood that content, including a tile’s contents, may be cached or otherwise saved in other embodiments.

[0374] With additional reference to FIG. 38, a method 3800 illustrates one example of a process that may be used by the web services server 2904 within the real time publishing system of FIG. 29 to assign data to various tiles 3501-3532 of the matrix 3500. It is

understood that this is a real time or near real time process, with data being identified, assigned to one of the tiles 3501-3532, and published for display in real time or near real time. If needed, cached data may be used to prevent tiles from remaining in an unrefreshed or blank state as previously described.

5 [0375] In step 3802, media is received from the NIO platform 2902. In step 3804, a determination is made as to whether any appropriate tile timers have expired. For example, if the media is tagged as “new,” the appropriate tile timers would be the new content duration timers. If the media is not new, the appropriate tile timers would be the minimum duration timers. If no tile timers have expired, the media is discarded in step 3806 and the method
10 3800 returns to step 3802.

[0376] If an appropriate tile timer has expired, the method 3800 moves to step 3808. In step 3808, a determination is made as to whether the tile with the expired timer is an appropriate (e.g., a matching) tile for the media. For example, if the tile is a priority 1 tile and the media is priority 3 media, then there is no match. If there is no match, the media is
15 discarded in step 3806 and the method 3800 returns to step 3802. If there is a match, the media currently being displayed at the matching tile is replaced with the new media in step 3810.

[0377] It is understood that the method 3800 may occur rapidly with media being received and either assigned or discarded continuously. The actual speed depends on factors
20 such as the input rate of received media, the content of the media relative to how the assignment occurs (e.g., the ratio of high priority content to the number of high priority tiles in the matrix), the duration of the timers, the size of the matrix, and/or other factors.

[0378] Referring to FIG. 39, a method 3900 illustrates one example of a process that may be used within the real time publishing system of FIG. 29. The real time publishing system
25 2901 may cache some or all of the captured content and, if no new content has been received when a tile is due to be refreshed, the tile may be refreshed using the cached content. In other words, old content (whether previously shown or not) may be used to keep the matrix 3500 in a refreshed state. This method may be used when the cached content is injected into the media stream on an “as needed” basis instead of being automatically injected on a timed
30 basis.

[0379] Accordingly, in step 3902, a determination is made as to whether a tile is ready to be refreshed. If the tile is not ready to be refreshed, the method 3900 returns to step 3902 and waits for the tile to be ready to be refreshed. If the tile is ready to be refreshed, the method 3900 continues to step 3904. In step 3904, a determination is made as to whether there is

new content for the tile. If there is new data, the tile is updated with the new content in step 3906. If there is no new content, the tile is updated with cached content in step 3908.

[0380] It is noted that step 3908 may vary depending upon the configuration of the real time publishing system 2901. For example, if there is no new content from an external source assigned to the tile (e.g., the external source 2906a), but there is new content from another external source (e.g., the external source 2906a), the new content from external source 2906b may be used to refresh the tile. In other words, fresh content may be given priority over cached content. However, if the real time publishing system 2901 is configured to use only certain external sources for certain tiles, then only cached content will be used when a given tile is to be refreshed and no new content is available from the corresponding external source.

[0381] Referring to FIGS. 40A-40D, various methods illustrate examples of processes that may be used within the real time publishing system of FIG. 29 when a spike occurs in the input rate from one or more of the external sources 2906a-2906N. More specifically, regardless of how the tiles 3501-3532 are assigned, the real time publishing system 2901 should be prepared to handle spikes in the input rate (and therefore, the input volume) of the external sources 2906a-2906N. Because the real time publishing system 2901 is displaying content in real time or near real time, spikes may occur unexpectedly and the system must handle those spikes in a manner that maintains the real time nature of the display without refreshing the matrix 3500 so rapidly that the content cannot be consumed by someone viewing the display.

[0382] Accordingly, FIGS. 40A-40E illustrate methods that may be executed by the real time publishing system 2901 when too much data is received. The determination of what constitutes too much data may be made on a per stream basis or on an aggregate basis (i.e., multiple streams or all streams).

[0383] Referring specifically to FIG. 40A, one embodiment of a method 4000 illustrates a process by which the real time publishing system 2901 can respond to a spike in the input rate. In step 4002, a determination is made as to whether the input rate is over a defined threshold. If not, step 4002 may repeat until an input rate is detected that surpasses the threshold. If the input rate surpasses the threshold, the method 4000 continues to step 4004.

[0384] In step 4004, any content that is not displayed is discarded. This results in the discarded content not being shown at all, with the amount of lost content depending on factors such as the input rate, the number of tiles being used for that stream, and the refresh rate of those tiles.

[0385] Referring specifically to FIG. 40B, one embodiment of a method 4010 illustrates another process by which the real time publishing system 2901 can respond to a spike in the input rate. Step 4012 is identical to step 4002 of FIG. 40A and is not repeated in the present example.

5 [0386] In step 4014, tiles are added to the matrix 2900 automatically. For example, tiles may be added to the bottom of the matrix 2900, which makes the matrix longer. Since the matrix 2900 is a live display, the tiles should be added in a way that minimizes any possible disruption to someone viewing the matrix 2900 at the time the addition occurs. Adding tiles to the sides and/or top of the matrix 2900 may be done in some embodiments, but adding to
10 the bottom of the matrix 2900 may lessen the disruption to a viewer.

[0387] Once the spike ends, the extra tiles may be removed, although this may be done over time to avoid wiping out content that a user may be viewing. In some embodiments, cached content may be used in the tiles before they are removed, as this will be older content and may not be as appealing to a viewer. The viewer may then move up higher on the matrix
15 searching for new content and the bottom tiles can then be removed. In other embodiments, the added tiles may not be removed as long as they are in the visible area of the display, but may be removed then they are no longer visible.

[0388] Referring specifically to FIG. 40C, one embodiment of a method 4020 illustrates yet another process by which the real time publishing system 2901 can respond to a spike in
20 the input rate. Step 4022 is identical to step 4002 of FIG. 40A and is not repeated in the present example.

[0389] In step 4024, the content is cached and shown as the rate slows down, although this will result in a time delay on some of the content. Content past a certain age may be discarded, although the discard age may vary depending on how much data is being received.
25 For example, if the stream slows down below the threshold, some content may be cached longer so that the matrix 2900 can be refreshed with unseen content even if that content is somewhat dated. If the input rate remains above the threshold, more content may be discarded to keep up.

[0390] Referring specifically to FIG. 40D, one embodiment of a method 4030 illustrates a
30 process by which the real time publishing system 2901 can respond to a spike in the input rate. Step 4032 is identical to step 4002 of FIG. 40A and is not repeated in the present example.

[0391] In step 4034, one or more additional tiles in the matrix 2900 may be assigned to the external source(s) 2906a-2906N that is having the input rate spike. For example, if the

input rate from the external source 2906a has spiked, tiles that are exclusively assigned to the external source 2906b that is updating slowly may be assigned to the external source 2906a, or content from the external source 2906a may be interleaved with the content from the external source 2906b. Once the spike ends, the tiles may be returned to the external source to which they were originally assigned.

[0392] In configurations where the tiles are randomly assigned to external sources rather than being assigned to specific external sources, tiles may be reserved for the external source having the spike. For example, a certain percentage of the tiles may remain in use for the other external sources, while another percentage of the tiles are assigned specifically to the external source having the spike. Once the spike ends, the tiles can be allocated normally.

[0393] Referring specifically to FIG. 40E, one embodiment of a method 4040 illustrates still another process by which the real time publishing system 2901 can respond to a spike in the input rate. Step 4042 is identical to step 4002 of FIG. 40A and is not repeated in the present example. In step 4044, the refresh rate of one or more tiles assigned to the external source having the input rate spike may be changed so that more content can be shown in the same amount of time. For example, if the current refresh rate is two seconds, the refresh rate may be lowered to one second, which would double the amount of data that could be displayed using the same tile(s). The refresh rate timer may be lowered repeatedly if the input rate remains high until a floor that defines the minimum refresh rate is reached. Once the spike ends, the refresh rates can be returned to normal, although this may occur gradually.

[0394] If the refresh rate for the entire matrix 2900 is to be changed, the change may be phased in across different tiles until the entire matrix has been changed. This phased approach may be used to minimize the visual disruption that suddenly changing the entire matrix 2900 may cause for a viewer.

[0395] As stated previously, the methods of FIGS. 40A-40E may be combined, either in a cascading approach or simultaneously. The combination of methods may be based on factors such as the actual input rate, whether a single external source or multiple external sources are involved, how the tiles are being assigned (e.g., per external source or across multiple external sources), which external source is involved (e.g., whether it is an external source that is already assigned multiple tiles), historical trend data for the external source (e.g., how long the spikes typically last for this external source and what the maximum volume during a spike), and/or similar information.

[0396] For example, in a cascading approach, the content may first be cached (FIG. 40C). Then, if the input rate does not drop below the threshold within a certain period of time

and/or the input rate increases, the refresh rate may be changed (FIG. 40E). Then, if the input rate does not drop below threshold within a certain period of time and/or the input rate increases, tiles may be reassigned (FIG. 40D). Then, if the input rate does not drop below threshold within a certain period of time and/or the input rate increases, the matrix 2900 may be expanded (FIG. 40B). It is understood that this order is only for purposes of example, and that the methods of FIGS. 40A-40E may be executed in any order.

[0397] In an example of simultaneously combining the methods of FIGS. 40A-40E, content may be cached (FIG. 40C) while tiles are being reassigned (FIG. 40D), and then the reassigned tiles may be used for the cached and/or new content. In another example, the refresh rate may be changed (FIG. 40E) while the matrix 2900 is being expanded (FIG. 40B).

[0398] Accordingly, many different approaches can be used to handle spikes in the input rate from the external sources 2906a-2906N. Generally, the approach will be directed to preserving as much content as possible while displaying as much real time or near real time content as possible. The real time publishing system 2901 may be specifically configured for the external sources 2906a-2906N, thereby enabling the system to be optimized in response to changes in the external sources.

[0399] In some embodiments, users can select the data that is to be displayed in the matrix 2900. For example, a user may select only certain tiles or types of content, and the real time publishing system 2901 may then adjust to that selection.

[0400] In other embodiments, the user may adjust the tiles in the matrix 2900. For example, the user may assign content to certain tiles, moves tiles, and/or otherwise modify the visual display provided by the matrix 2900.

[0401] The matrix 2900 represents one possible display format for information to be published, and many other formats may be used. For example, histograms, Venn diagrams, bar charts, word clouds, tickers, hive plots, and/or any other format that is appropriate for the content being published may be used. In some embodiments, a three dimensional shape may be used, with information projected onto the shape. In such embodiments, the shape may be rotated to view the information.

[0402] In still other embodiments, the real time publishing system 2901 can display, in real time, information about its own performance and about the data. For example, the real time publishing system 2901 may display information about how much data is being processed per unit time, how much is being discarded, and similar information. The real time publishing system 2901 may also display how much relevant information is not being shown because, for example, the system cannot display it all because of the volume of information.

For example, the real time publishing system 2901 may show how many tweets are not being displayed. Such information may then be used to modify what the real time publishing system 2901 is publishing so that a user can narrow the focus down to particular content.

[0403] In addition to the claimed embodiments in the appended claims, the following is a
5 list of embodiments which may serve as the basis for additional claims in this application or subsequent divisional applications:

[0404] Embodiment 1: A method for execution on a digital device includes running a core process to create a platform instance of a real time processing platform, wherein the platform instance interacts with an operating system on the digital device and is configured to
10 run any service instance that is created using extendable base classes that have been defined for use within the processing platform, and wherein the core process includes a service manager configured to manage any service instance run by the platform instance; identifying, by the service manager, from configuration information corresponding to the platform instance, that the platform instance is configured to run a service instance that is defined by a
15 service class that extends a defined base service class; identifying from the configuration information, by the service manager, that the service instance is to use a plurality of block instances that are defined by a plurality of block classes, wherein each block class extends a defined base block class and contains executable instructions that provide processing functionality for the service instance; starting, by the service manager, the service instance,
20 wherein the service instance includes a block router and a routing table; starting, by the block router, the block instances, wherein each block instance has no knowledge of the service instance and the plurality of block instances other than itself; receiving, by a first block instance of the plurality of block instances, streaming input data to be processed from a source that is external to the platform instance; processing, by the first block instance, the
25 streaming input data to create processed data; sending, by the first block instance, a notification to the block router that the first block instance has the processed data ready for output; identifying, by the block router, that the processed data from the first block instance is to be directed to a second block instance of the plurality of block instances based on the routing table; directing, by the block router, the processed data from the first block instance
30 to the second block instance; repeating the steps of processing, sending, identifying, and directing for each of the plurality of block instances until a final block instance is reached, wherein the final block instance creates output data; and sending, by the final block instance, the output data to a destination that is external to the platform instance, wherein each step from the step of receiving the streaming input data by the first block instance to the step of

sending the output data from the final block instance occurs in real time without queuing the data within the platform instance.

[0405] Embodiment 1+1: The method of embodiment 1 further including receiving, by the service manager, a stop command from a source that is external to the platform instance, wherein the stop command indicates that the service instance is to be stopped; sending, by the service manager, the stop command to the block router; and stopping, by the block router, the plurality of block instances.

[0406] Embodiment 1+2: The method of any of embodiments 1 or 1+1 further including receiving, by the first block instance, second streaming input data to be processed from a second source that is external to the platform instance, wherein the second streaming input data is received immediately after the notification is sent to the block router that the first block instance has the processed data ready for output; processing, by the first block instance, the second streaming input data to create second processed data; sending, by the first block instance, a notification to the block router that the first block instance has the second processed data ready for output; identifying, by the block router, that the second processed data from the first block instance is to be directed to the second block instance based on the routing table; directing, by the block router, the second processed data from the first block instance to the second block instance; repeating the steps of processing, sending, identifying, and directing for each of the plurality of block instances until the final block instance is reached, wherein the final block instance creates second output data; and sending, by the final block instance, the second output data to a destination that is external to the platform instance, wherein each step from the step of receiving the second streaming input data by the first block instance to the step of sending the second output data from the final block instance occurs in real time without queuing the data within the platform instance.

[0407] Embodiment 2: A method for execution on a digital device includes running a core process to create a platform instance of a processing platform, wherein the platform instance interacts with an operating system on the digital device and is configured to run any service created using extendable base classes that have been defined for use within the processing platform; identifying, by the core process, from configuration information corresponding to the platform instance, that the platform instance is configured to run a service that is defined by a service class that extends a defined base service class, wherein the configuration information further identifies a plurality of block classes to be used by the service, wherein each block class extends a defined base block class and contains executable instructions that provide processing functionality for the service; instantiating the service

class to create a service instance and each of the block classes to create a plurality of block instances, wherein the block instances have no awareness of one another and wherein the service instance is configured to direct data among the block instances; and processing, by the service instance, incoming data using the block instances to provide the processing functionality to the platform instance.

[0408] Embodiment 2+1: The method of embodiment 2 further including creating, by the core process, a service context defining functionality of the platform instance that is external to the service instance and accessible by the service instance; and passing the service context to the service instance when the service class is instantiated.

[0409] Embodiment 2+2: The method of embodiment 2+1 wherein creating the service context includes passing the service context to a module, wherein the module inserts information needed by the service instance to use the module into the service context.

[0410] Embodiment 2+3: The method of any of embodiments 2 through 2+2 receiving, by the core process, a command to instantiate the service class.

[0411] Embodiment 2+4: The method of any of embodiments 2 through 2+3 further including receiving, by the core process, a command to stop the service instance; and stopping, by the core process, the service instance, wherein the core process continues running after the service instance has been stopped.

[0412] Embodiment 2+5: The method of any of embodiments 2 through 2+4 further including stopping the core process, wherein the platform instance is destroyed when the core process is stopped.

[0413] Embodiment 2+6: The method of any of embodiments 2 through 2+5 wherein the service instance is configured to direct data among the block instances based on a routing table, wherein a publication by one of the block instances is routed to another of the block instances based on the routing table.

[0414] Embodiment 2+7: The method of any of embodiments 2 through 2+6 wherein the service instance is run by the operating system as a separate process from the core process.

[0415] Embodiment 2+8: The method of any of embodiments 2 through 2+6 wherein the service instance is a thread of the core process.

[0416] Embodiment 2+9: The method of any of embodiments 2 through 2+8 further including converting all data entering the service instance into generic data objects, wherein only the generic data objects are passed between the block instances.

[0417] Embodiment 2+10: The method of any of embodiments 2 through 2+9 further including identifying, by the core process, from the configuration information corresponding

to the platform instance, that the platform instance is configured to run a second service that is defined by a second service class that extends the defined base service class, wherein the configuration information further identifies a plurality of second block classes to be used by the second service, wherein each second block class extends the defined base block class and contains executable instructions that provide second processing functionality for the second service; instantiating the second service class to create a second service instance and each of the second block classes to create a plurality of second block instances operating as part of the second service instance, wherein the second block instances have no awareness of one another and wherein the second service instance is configured to direct data among the second block instances; and processing, by the second service instance, incoming data using the second block instances to provide the second processing functionality to the platform instance.

[0418] Embodiment 2+11: The method of embodiment 2+10 wherein the service instance and the second service instance run simultaneously on the platform instance.

[0419] Embodiment 2+12: The method of any of embodiments 2+10 through 2+11 wherein the service instance and the second service instance communicate with one another.

[0420] Embodiment 2+13: The method of any of embodiments 2+10 through 2+12 wherein each of the service instance and the second service instance includes a communication manager that publishes communications for consumption by subscribed service instances.

[0421] Embodiment 3: A method for execution on a digital device includes running, by a platform instance that is configured to interact with an operating system running on the digital device, a service instance that is configured to use a routing table to direct data among a plurality of block instances that provide processing capability to the service instance, wherein each block instance executes any internal instructions contained within that block instance asynchronously upon receiving input; receiving, by a first block instance of the plurality of block instances, input data from a source that is external to the platform instance; processing, by the first block instance, the input data to create processed data; notifying, by the first block instance, the service instance of the processed data; determining, by the service instance, that the processed data is to be directed to a second block instance of the plurality of block instances based on the routing table; and transferring, by the service instance, the processed data to the second block instance for further processing.

[0422] Embodiment 3+1: The method of embodiment 3 further including accessing, by the service instance, functionality provided by a module that is part of a core process of the platform instance.

5 [0423] Embodiment 3+2: The method of any of embodiments 3 through 3+1 wherein the service instance includes a block router, wherein the service instance communicates with a core process of the platform instance using the block router, and the core process starts and stops the service instance.

[0424] Embodiment 3+3: The method of embodiment 3+2 wherein the core process communicates with at least one of the plurality of block instances through the block router.

10 [0425] Embodiment 3+4: The method of any of embodiments 3+1 through 3+3 wherein the configuration parameters of the first block instance are set by a user via the core process.

[0426] Embodiment 3+5: The method of any of embodiments 3+1 through 3+4 wherein the configuration parameters of the service instance are set by a user via the core process.

[0427] Embodiment 4: A method for using a configurable platform stored on a digital
15 device includes receiving instructions, by the digital device, defining a service to be executed by the configurable platform, wherein the service is represented within the configurable platform by a service class that extends a base service class defined for use within the configurable platform, wherein defining the service includes identifying a plurality of blocks to be used by the service, wherein each block is represented within the configurable platform
20 by a block class that extends a base block class defined for use within the configurable platform; defining instructions within at least one of the block classes, wherein the instructions configure the block class to perform functionality defined by the instructions; and defining within a routing table, for each of the plurality of block classes, any other of the plurality of blocks to which output is to be directed when that block produces output; and
25 receiving instructions, by the digital device, for saving the service in a memory accessible by the configurable platform.

[0428] Embodiment 4+1: The method of embodiment 4 wherein identifying the plurality of blocks to be used by the service includes defining a location remote from the digital device where at least one of the plurality of blocks is stored.

30 [0429] Embodiment 4+2: The method of any of embodiments 4 through 4+1 wherein identifying the plurality of blocks to be used by the service includes loading at least one of the plurality of blocks that was received as an attachment to a message.

[0430] Embodiment 4+3: The method of any of embodiments 4 through 4+2 further including receiving instructions, by the digital device, to label the service as an autostart service in a configuration file.

5 [0431] Embodiment 4+4: The method of any of embodiments 4 through 4+3 further including receiving instructions to run the configurable platform; and executing a core process to create an instance of the configurable platform.

[0432] Embodiment 4+5: The method of any of embodiments 4 through 4+4 further including receiving instructions, by the core process, to run the service; and starting, by the core process, a service instance that is an instantiation of the service class and a plurality of
10 block instances that are instantiations of the block classes.

[0433] Embodiment 4+6: The method of any of embodiments 4 through 4+5 further including providing, by the core process, a graphical user interface, wherein the graphical user interface enables a user to modify the service class and the plurality of blocks classes.

15 [0434] Embodiment 4+7: The method of any of embodiments 4 through 4+6 further including providing, by the core process, a graphical user interface that enables a user to modify the base service class and the plurality of base block classes.

[0435] Embodiment 4+8: The method of any of embodiments 4 through 4+7 further including providing, by the core process, a graphical user interface that enables a user to select and run an existing service from a plurality of existing services available on the
20 configurable platform.

[0436] Embodiment 4+9: The method embodiment 4+8 further including providing, by the core process, a graphical user interface that enables a user to modify the existing service.

[0437] Embodiment 5: A method for using a configurable platform stored on a digital device includes receiving instructions, by an operating system running on the digital device,
25 to start a core process corresponding to an instance of the configurable platform; identifying, by the core process, a service to be executed by the configurable platform, wherein the service is represented within the configurable platform by a service class that extends a base service class defined for use within the configurable platform; identifying, by the core process, a plurality of blocks to be used by the service, wherein each block is represented
30 within the configurable platform by a block class that extends a base block class defined for use within the configurable platform; starting, by the core process, the service; starting by the service, the blocks; and processing, by the blocks, data received by the configurable platform from a source that is external to the configurable platform, wherein the processing creates an output that is sent to a destination that is external to the configurable platform.

[0438] Embodiment 5+1: The method of embodiment 5 further including passing a service context from the core process to the service, wherein the core context contains configuration information needed by the service.

5 [0439] Embodiment 5+2: The method of any of embodiments 5 through 5+1 further including determining, by the service, that a module is needed by a first block of the plurality of blocks, wherein the module is provided by the core process and is configured to provide predefined functionality; initializing, by the service, the module; and passing, by the service, module information to the first block, wherein the module information is used by the first block to access the predefined functionality of the module.

10 [0440] Embodiment 5+3: The method of any of embodiments 5 through 5+2 further including determining, by the service, that a module is needed by the service, wherein the module is provided by the core process and is configured to provide predefined functionality; initializing, by the service, the module; and using, by the service, the predefined functionality of the module.

15 [0441] Embodiment 5+4: The method of any of embodiments 5 through 5+3 wherein the service includes a block router configured to: receive a notification from a source block of the plurality of blocks that the source block has an output; look up a destination block in a routing table corresponding to the source block; and call the destination block to handle the output from the source block.

20 [0442] Embodiment 5+5: The method of embodiment 5+4 wherein all communications among the plurality of blocks pass through the block router.

[0443] Embodiment 5+6: The method of any of embodiments 5 through 5+5 wherein a block that is configured to pass output to another block contains destination information identifying the other block.

25 [0444] Embodiment 5+7: The method of any of embodiments 5 through 5+6 further including converting, by a first block of the plurality of blocks, the data into a plurality of data objects corresponding to an internal data object template that is defined for use within the configurable platform, wherein all communications among the plurality of blocks are based on the internal data object template.

30 [0445] Embodiment 5+8: The method of any of embodiments 5 through 5+7 wherein the service is a first service of the plurality of services, and the method further includes receiving instructions, by the core process, to start a second service of the plurality of services to be executed by the configurable platform, wherein the second service is represented within the configurable platform by a second service class that extends the base service class;

identifying, by the core process, a plurality of second blocks to be used by the second service, wherein each second block is represented within the configurable platform by a second block class that extends the base block class; starting, by the core process, the second service; and starting by the second service, the second blocks.

5 [0446] Embodiment 5+9: The method of embodiment 5+8 further including processing, by the second blocks, data received by the configurable platform from a source that is external to the configurable platform, wherein the processing creates an output that is sent to a destination that is external to the configurable platform.

[0447] Embodiment 5+10: The method of any of embodiments 5+8 through 5+9 further
10 including processing, by the second blocks, data received from the first service, wherein the processing creates an output that is sent to a destination that is external to the configurable platform.

[0448] Embodiment 5+11: The method of any of embodiments 5+8 through 5+10 further including processing, by the second blocks, data received from the first service, wherein the
15 processing creates an output that is sent to the first service.

[0449] Embodiment 6: A method for using a configurable platform on a digital device includes providing, by the configurable platform, a runtime environment that interacts with an operating system on the digital device and within which any service that is compatible with the configurable platform can be executed, wherein the runtime environment includes a
20 service manager configured to manage any service that extends a base service class that is defined for use within the configurable platform; a configuration file that identifies any services loaded into the configurable platform for execution within the runtime environment; and an application programming interface (API) that enables a user to configure a service for use within the configurable platform.

25 [0450] Embodiment 6+1: The method of embodiment 6 further including providing, by the configurable platform, a plurality of blocks to be used by the service, wherein each block is represented within the configurable platform by a block class that extends a base block class defined for use within the configurable platform.

[0451] Embodiment 6+2: The method of any of embodiments 6 through 6+1 further
30 including providing, by the configurable platform, a plurality of core modules accessible to the service manager but not accessible to any service executed within the runtime environment.

[0452] Embodiment 6+3: The method of any of embodiments 6 through 6+2 further including providing, by the configurable platform, a plurality of functional modules

accessible to the service manager and to any service executed within the runtime environment.

[0453] Embodiment 6+4: The method of any of embodiments 6 through 6+3 further including loading a service identified in the configuration file and identifying any blocks to be used by the service, wherein each block is represented within the configurable platform by a block class that extends a base block class defined for use within the configurable platform.

[0454] Embodiment 7: A method for use by a configurable platform stored on a digital device includes creating, by a core process necessary for an instance of the configurable platform to exist on the digital device, a service context that includes information needed for a service instance to be created on the configurable platform; starting, by a service manager of the core process, a service instance to be executed by the configurable platform, wherein the service instance is an instantiation of a service class that extends a base service class defined for use by the configurable platform; and starting, by a block router of the service instance based on the service context, a plurality of block instances to be used by the service instance, wherein each block instance is an instantiation of a block class that extends a base block class defined for use by the configurable platform, and wherein each block instance contains instructions that are executed when an input is received by that block instance.

[0455] Embodiment 7+1: The method of embodiment 7 further including routing, by the block router, data among the plurality of block instances based on a routing table, wherein each block instance is unaware of the other block instances and all communication between block instances relies on the block router.

[0456] Embodiment 7+2: The method of any of embodiments 7 through 7+1 wherein no queuing occurs within the service instance.

[0457] Embodiment 7+3: The method of any of embodiments 7 through 7+2 wherein one of the block instances uses a module provided by the core process.

[0458] Embodiment 7+4: The method of embodiment 7+3 further including initializing, by the block router, the module for use based on the service context; and passing, by the block router, information about the module to the block instance that uses the module, wherein the information about the module is needed by the block instance to use the module.

[0459] Embodiment 7+5: The method of any of embodiments 7 through 7+4 wherein the block router uses a module provided by the core process.

[0460] Embodiment 7+6: The method of embodiment 7+5 further including initializing, by the block router, the module for use by the block router based on the service context.

[0461] Embodiment 8: A method includes receiving, by a block router that is part of a service instance running within a runtime environment provided by a platform instance of a configurable platform stored on a digital device, a notification from a first block instance that the first block instance has produced an output object, wherein the first block instance is one of a plurality of block instances that provide processing functionality to the service instance; accessing, by the block router, a routing table, wherein the routing table identifies a second block instance of the plurality of block instances as an output destination for output objects produced by the first block instance; and routing, by the block router, the output object to the second block instance.

[0462] Embodiment 8+1: The method of embodiment 8 wherein the block instances have no awareness of the other block instances of the plurality of block instances.

[0463] Embodiment 8+2: The method of any of embodiments 8 through 8+1 wherein the notification from the first block instance is a call to a method defined within the service instance.

[0464] Embodiment 8+3: The method of embodiment 8+2 wherein the call to the method includes the output object.

[0465] Embodiment 8+4: The method of any of embodiments 8 through 8+3 wherein routing the output object includes performing a call to the second block instance.

[0466] Embodiment 8+5: The method of embodiment 8+4 wherein the call to the second block instance includes the output object.

[0467] Embodiment 8+6: The method of any of embodiments 8+4 through 8+5 further including starting, by the block router, a thread for the second block instance.

[0468] Embodiment 8+7: The method of any of embodiments 8 through 8+6 further including receiving, by the block router, a message for the first block instance from a core process of the platform instance; and passing, by the block router, the message to the first block instance.

[0469] Embodiment 8+8: The method of any of embodiments 8 through 8+7 wherein all communications between the core process and the plurality of block instances pass through the block router.

[0470] Embodiment 8+9: The method of any of embodiments 8 through 8+8 wherein a third block instance of the plurality of block instances communicates without going through the block router with at least one of a signal source and a signal destination that is external to the platform instance.

[0471] Embodiment 8+10: The method of any of embodiments 8 through 8+9 further including receiving, by the block router, a message to stop; and stopping, by the block router, each of the plurality of block instances.

5 [0472] Embodiment 8+11: The method of any of embodiments 8 through 8+10 further including publishing, by the block router, information for consumption by subscribers of the service instance.

10 [0473] Embodiment 8+12: The method of any of embodiments 8 through 8+11 wherein the first block instance manages a data flow among a subset of the plurality of block instances based on a group routing table that identifies a data flow within the subset of block instances, and all communications between the subset of block instances and the block router pass through the first block instance.

15 [0474] Embodiment 9: A method for configuring a service within a platform on a digital device includes providing, by the configurable platform, a runtime environment that interacts with an operating system on the digital device and within which any service that is compatible with the configurable platform can be executed, wherein the runtime environment includes a service configuration file that contains information detailing an operation of a service within the runtime environment; and an application programming interface (API) that enables a user to modify the service configuration file to configure the service; receiving service configuration information about a plurality of blocks to be associated with the service, wherein the service configuration information identifies a location of each of the plurality of blocks; and identifies, for each of the plurality of blocks that produces output to be consumed by another of the plurality of blocks, the block to which the output should be directed; and saving the service configuration information in the service configuration file for use when the service is started within the runtime environment.

25 [0475] Embodiment 9+1: The method of embodiment 9 wherein the service configuration information identifies a module to be used by the service, wherein the module is provided by a core process within the runtime environment.

30 [0476] Embodiment 9+2: The method of any of embodiments 9 through 9+1 further including receiving block configuration information containing parameters for one of the blocks; and saving the block configuration information in a block configuration file for use when the block is started within the runtime environment.

[0477] Embodiment 9+3: The method of embodiment 9+2 wherein the block configuration information identifies a module to be used by the block, wherein the module is provided by a core process within the runtime environment.

[0478] Embodiment 9+4: The method of any of embodiments 9 through 9+3 further including receiving instructions to be executed by one of the blocks; and saving the instructions in a block class file that is used to instantiate the block.

[0479] Embodiment 10: A method for providing a runtime environment on a digital device includes running a platform core, wherein the platform core is a first process within an operating system on the digital device and provides the runtime environment; running a first service that can only run within the runtime environment, wherein the first service is a second process within the operating system, and wherein the first service provides first data processing functionality using a first plurality of configurable blocks; and running a second service that can only run within the runtime environment, wherein the second service runs simultaneously with the first service and is a third process within the operating system, and wherein the second service provides second data processing functionality using a second plurality of configurable blocks.

[0480] Embodiment 10+1: The method of embodiment 10 further including communicating by the first service and second service with one another within the runtime environment.

[0481] Embodiment 10+2: The method of any of embodiments 10 through 10+1 wherein any communication that occurs within the runtime environment uses a data object that complies with a data object template that is configured for use within the runtime environment.

[0482] Embodiment 10+3: The method of any of embodiments 10 through 10+2 further including using, by the first service, functionality provided by a module that is supported by the platform core.

[0483] Embodiment 10+4: The method of any of embodiments 10 through 10+3 wherein at least one block of the first plurality of blocks and the second plurality of blocks uses a single block configuration shared by the first service and the second service.

[0484] Embodiment 10+5: The method of any of embodiments 10 through 10+4 wherein the first plurality of blocks are threads of the second process and the second plurality of blocks are threads of the third process.

[0485] Embodiment 10+6: The method of any of embodiments 10 through 10+5 wherein the first service is an instantiation of a first service class that extends a base service class defined for use within the runtime environment, and the second service is an instantiation of a second service class that extends the base service class.

[0486] Embodiment 10+7: The method of embodiment 10+6 wherein the first service class is identical to the second service class.

[0487] Embodiment 10+8: The method of embodiment 10+6 wherein the first service class is different from the second service class.

5 [0488] Embodiment 10+9: The method of any of embodiments 10 through 10+8 wherein the first data processing functionality is different from the second data processing functionality.

[0489] Embodiment 10+10: The method of any of embodiments 10 through 10+8 wherein the first data processing functionality is identical to the second data processing
10 functionality.

[0490] Embodiment 10+11: The method of embodiment 10+10 further including performing load balancing by the first service and the second service to balance an amount of input data to be processed using the first processing functionality and the second processing functionality.

15 [0491] Embodiment 10+12: The method of any of embodiments 10 through 10+11 further including running a second platform core on the digital device, wherein the second platform core is a fourth process within the operating system on the digital device and provides a second runtime environment that is separate from the runtime environment provided by the first process; and running a third service that is fifth process within the
20 operating system, wherein the third service provides third data processing functionality using a third plurality of configurable blocks.

[0492] Embodiment 10+13: The method of embodiment 10+12 wherein the third service is identical to the first service, and the method further includes performing load balancing by the first service and the third service to balance an amount of input data to be processed using
25 the first processing functionality and the third processing functionality.

[0493] Embodiment 10+14: The method of any of embodiments 10+12 through 10+13 further including sending output data from the third service to the second service.

[0494] Embodiment 11: A method for configuring a service within a platform instance at runtime includes identifying, by a platform instance that is running on a digital device and
30 interacting with an operating system of the digital device, a service that is to be run by the platform instance, wherein the service is one of a plurality of available services that can be run by the platform instance, and wherein the available services correspond to a single service class; instantiating by the platform instance, the service using the service class; identifying, by the platform instance, a configuration file for the service, wherein the

configuration file contains predefined service configuration information that includes block information identifying a plurality of blocks that are to be run by the service, wherein the blocks are based on a corresponding plurality of block classes that are usable by any of the available services and wherein each of the blocks provides task specific functionality when included in one of the available services; configuring the service using the predefined service configuration information from the configuration file and dynamically generated information about the platform instance that is not known until after the platform instance has begun running, wherein the configuring prepares the service to use the blocks by providing a plurality of routing directions that enable the service to route data between the blocks; and running the service after the service is configured, wherein the task specific functionality provided by the blocks is available to the platform instance when the service is running.

[0495] Embodiment 11+1: The method of embodiment 11 wherein the service class includes at least one configurable parameter, and configuring the service includes assigning a configuration value to the configurable parameter.

[0496] Embodiment 11+2: The method of embodiment 11+1 wherein the configurable parameter defines an execution rule for the blocks.

[0497] Embodiment 11+3: The method of embodiment 11+2 wherein the execution rule determines whether the blocks will be executed synchronously or asynchronously.

[0498] Embodiment 11+4: The method of embodiment 11+1 wherein the configurable parameter identifies one of a plurality of block routers that is to be used by the service, and the block router is responsible for passing data between the blocks.

[0499] Embodiment 11+5: The method of any of embodiments 11 through 11+4 further including instantiating a first block of the plurality of blocks from a first block class of the plurality of block classes; identifying a first block configuration file for the first block; configuring a first configurable parameter of the first block with a first configuration value from the first block configuration file; and running the first block within the service.

[0500] Embodiment 11+6: The method of embodiment 11+5 further including instantiating a second block of the plurality of blocks from a second block class of the plurality of block classes; identifying a second block configuration file for the second block; configuring a second configurable parameter of the second block with a second configuration value from the second block configuration file; and running the second block within the service.

[0501] Embodiment 11+7: The method of embodiment 11+6 wherein the first block class and the second block class are the same block class, and the first block differs in functionality

from the second block due to differences between the first block configuration file and the second block configuration file.

[0502] Embodiment 11+8: The method of any of embodiments 11 through 11+7 wherein the dynamically generated information identifies a module that is external to the service and
5 part of the platform instance, wherein the module is to be used by the service.

[0503] Embodiment 11+9: The method of embodiment 11+8 further including instantiating, by the service, an instance of the module for use by the service.

[0504] Embodiment 12: A method for configuring services within a platform instance at runtime includes identifying, by a platform instance that is running on a digital device and
10 interacting with an operating system of the digital device, a first service and a second service that are to be simultaneously run by the platform instance; instantiating, by the platform instance, the first service and the second service using a single service class; identifying, by the platform instance, a first configuration file that identifies a plurality of first blocks that are to be run by the first service, wherein each of the first blocks is configured to provide task
15 specific functionality when run by the first service; identifying, by the platform instance, a second configuration file that identifies a plurality of second blocks that are to be run by the second service, wherein each of the second blocks is configured to provide task specific functionality when run by the second service; configuring the first service using the first configuration file and dynamically generated information about the platform instance that is
20 not known until after the platform instance has begun running, wherein the configuring prepares the first service to use the first blocks by providing a first plurality of routing directions that enable the first service to route data between the first blocks; configuring the second service using the second configuration file and the dynamically generated information, wherein the configuring prepares the second service to use the second blocks by
25 providing a plurality of routing directions that enable the second service to route data between the second blocks; and running the first service and the second service simultaneously.

[0505] Embodiment 12+1: The method of embodiment 12 wherein configuring the first service includes applying a first configuration value from the first configuration file to a first
30 configurable parameter of the first service, and configuring the second service includes applying a second configuration value from the second configuration file to a second configurable parameter of the second service.

[0506] Embodiment 12+2: The method of embodiment 12+1 wherein the first configurable parameter instructs a block router used by the first service to execute the first

blocks asynchronously, and the second configurable parameter instructs a block router used by the second service to execute the second blocks synchronously.

5 [0507] Embodiment 12+3: The method of any one of embodiments 12 through 12+2 wherein the dynamically generated information identifies a module that is part of the platform instance and is external to the first service and the second service.

[0508] Embodiment 12+4: The method of embodiment 12+3 further including instantiating, by the first service, an instance of the module for use by the first service.

10 [0509] Embodiment 13: A method for displaying a plurality of media objects based on priority includes providing a matrix having a plurality of tiles, wherein each tile represents a display location for a media object and is assigned one of a plurality of priority levels, and wherein each priority level is associated with a minimum duration time and a maximum duration time; tracking, for each of the tiles, a tile timer that represents how long the tile has been displaying a current media object assigned to that tile; receiving a first media object that has been assigned one of the plurality of priority levels; determining that the tile timer for a first tile of the plurality of tiles has exceeded the minimum duration time for the priority level assigned to the tile; determining that the priority level of the first media object is identical to the priority level of the first tile; and assigning the first media object to the first tile for display, wherein the current media object being displayed at the first tile is replaced by the first media object.

20 [0510] Embodiment 13+1: The method of embodiment 13 further including receiving a second media object that has been assigned one of the plurality of priority levels; determining that none of the tile timers for tiles having a priority level identical to the priority level assigned to the second media object have exceeded the minimum duration time; and discarding the second media object without assigning the second media object to any of the tiles.

25 [0511] Embodiment 13+2: The method of any of embodiments 13 or 13+1 wherein each priority level is further associated with a new content duration time that has a shorter duration than the minimum duration time, the method further comprising: receiving a second media object that has been assigned one of the plurality of priority levels and has also been assigned a tag indicating that the second media object includes new content that has not previously been displayed; determining that the tile timer for a second tile of the plurality of tiles has exceeded the new content duration timer for the priority level assigned to the tile, wherein the new content duration time is used instead of the minimum duration time due to the presence of the tag; determining that the priority level of the second media object is identical to the

priority level of the second tile; and assigning the second media object to the second tile for display, wherein the current media object being displayed at the second tile is replaced by the second media object.

5 [0512] Embodiment 13+3: The method of embodiment 13+2 wherein the tile timer for the second tile has not yet exceeded the minimum duration time when the second media object is assigned to the second tile.

10 [0513] Embodiment 13+4: The method of any of embodiments 13 through 13+3 further including receiving a second media object that has been assigned one of the plurality of priority levels and has also been assigned a tag indicating that the second media object includes very important person (VIP) content that takes precedence over any other content type; identifying each of the plurality of tiles that has a priority level that is identical to the priority level of the second media object; determining that none of the identified tiles have passed their corresponding minimum duration times; selecting a second tile from the identified tiles to use for the second media object based on at least one defined replacement
15 criterion; and assigning the second media object to the second tile.

[0514] Embodiment 13+5: The method of embodiment 13+4 wherein the defined replacement criterion is a least amount of remaining time, and wherein selecting the second tile includes determining a remaining time for each of the identified tiles.

20 [0515] Embodiment 13+6: The method of embodiment 13+5 wherein determining the remaining time includes calculating, for each identified tile, a percentage of time remaining within a time window defined between the minimum duration time and the maximum duration time.

25 [0516] Embodiment 13+7: The method of embodiment 13+5 wherein determining the remaining time includes calculating, for each identified tile, a percentage of time remaining within a window defined between the new content duration time and the maximum duration time.

30 [0517] Embodiment 13+8: The method of embodiment 13+5 wherein determining the remaining time includes calculating, for each identified tile, a percentage of time remaining within a time window defined between the minimum duration time and the maximum duration time.

[0518] Embodiment 13+9: The method of embodiment 13+5 wherein determining the remaining time includes calculating, for each identified tile, a percentage of time remaining within a window defined between the new content duration time and the maximum duration time.

[0519] Embodiment 13+10: The method of any of embodiments 13 through 13+9 further including determining that a second tile of the plurality of tiles has displayed a second media object for a period of time longer than the maximum duration time of the second tile; and replacing the second media object with a next received media object that has a priority level
5 identical to the priority level of the second tile.

[0520]

[0521] Embodiment 13+11: The method of any of embodiments 13 through 13+10 wherein each media object is assigned a priority level based on a source of the media object.

[0522] Embodiment 13+12: The method of embodiments 13+11 wherein at least one
10 source is a third party site.

[0523] Embodiment 13+13: The method of any of embodiments 13 through 13+12 wherein each media object is assigned a priority level based on at least one content type corresponding to content of the media object.

[0524] Embodiment 13+14: The method of any of embodiments 13 through 13+13
15 wherein each media object contains content from one of a plurality of media sources, and wherein each media object is in a single standardized format accepted by the matrix regardless of the content contained by the media object.

[0525] Embodiment 13+15: The method of any of embodiments 13 through 13+14 further including publishing the matrix for viewing.

[0526] Embodiment 14: A method for displaying media objects including providing a matrix having a plurality of tiles, wherein each tile represents a display location for a media object on a user viewable display screen and is assigned one of a plurality of priority levels, and wherein each priority level is associated with a maximum duration time; receiving a stream of media objects, wherein each media object has been assigned one of the plurality of
20 priority levels; determining for each tile whether the tile has exceeded the maximum duration time corresponding to the assigned priority level; and for each tile that has exceeded its maximum duration time, replacing a media object currently displayed by the tile with another one of the media objects having a priority level that matches the tile's priority level.

[0527] Embodiment 14+1: The method of embodiment 14 further including providing a user selectable region associated with a first tile of the plurality of tiles, wherein selection of the user selectable region by a user prevents the media object being displayed at the first tile from being replaced.
30

[0528] Embodiment 14+2: The method of embodiment 14+1 wherein re-selection of the user selectable region enables the media object being displayed at the first tile to be replaced.

[0529] Embodiment 14+3: The method of one of embodiments 14 through 14+2 further including providing a link associated with a first tile of the plurality of tiles, wherein the link identifies a remote location storing an original source of the media object displayed at the first tile, and wherein selection of the first tile opens the original source of the media object from the remote location.

[0530] Embodiment 14+4: The method of one of embodiments 14 through 14+3 further including providing a user selectable region covering at least a portion of a first tile of the plurality of tiles, wherein selection of the user selectable region enlarges the first tile.

[0531] Embodiment 14+5: The method of embodiment 14+4 wherein the matrix rearranges when the first tile is enlarged to ensure that all the tiles of the matrix are visible.

[0532] Embodiment 14+6: The method of embodiment 14+4 wherein the media object being displayed at the first tile is not replaceable while the first tile is enlarged.

[0533] Embodiment 14+7: The method of embodiment 14+4 further including providing a link associated with the first tile, wherein the link identifies a remote location storing an original source of the media object displayed at the first tile, and wherein selection of the first tile opens the original source of the media object from the remote location.

[0534] Embodiment 14+8: The method of embodiment 14+4 further including discarding any media objects for which no tile is available.

[0535] Embodiment 14+9: The method of embodiment 14+4 wherein at least one of the media objects has been assigned a timestamp indicating when the media object was received, wherein the method further comprises displaying a representation of the timestamp with the media object.

[0536] Embodiment 14+10: The method of embodiment 14+9 wherein the representation of the timestamp is an age of the media object.

[0537] Embodiment 15: A method including providing a matrix of repeatedly refreshing tiles for display to a user, wherein each tile represents a display location for a media object; receiving a stream of media objects representing content from a plurality of media sources; and repeatedly updating each of the tiles with different media objects from the stream of media objects.

[0538] Embodiment 15+1: The method of embodiment 15 wherein each tile displays a media object for no longer than a defined maximum amount of time before the media object is replaced with another media object.

[0539] Embodiment 15+2: The method of embodiment 15+1 wherein the media object is replaced unless the tile is locked by the user to prevent the replacement from occurring.

[0540] Embodiment 16: A software architecture for a configurable processing platform for use on a device including a core configured to interact with an operating system on the device, wherein the core is configurable to simultaneously run any of a plurality of services that are defined for the processing platform by configuration information, wherein each
5 service to be run on the processing platform is defined by a service class and the configuration information to include a set of platform specific instructions that enable the service to operate within the processing platform and a set of service specific instructions that enable the service to run a plurality of blocks that provide task specific functionality to the service, and wherein each block to be run on the processing platform is defined by a block
10 class and the configuration information to include a set of platform specific instructions that enable the block to operate asynchronously and independently from the other blocks within the processing platform and a set of task specific instructions that enable the block to perform a specific processing task for the service that uses the block, wherein an order of execution of the blocks within the service is defined by the configuration information.

15 [0541] Embodiment 17: A processing system includes a processor; and a memory coupled to the processor and containing instructions for execution by the processor, the instructions for performing any of the methods or implementing the architecture described herein.

[0542] Embodiment 18: A computer program product configured to be operable to
20 perform any of the methods or implementing the architecture described herein.

WHAT IS CLAIMED IS:

1. A method for execution on a digital device, the method characterized by:

starting a core to create a platform instance of a configurable processing platform,
5 wherein the platform instance interacts with an operating system on the digital device and is
configurable to run any of a plurality of services that are defined for the processing platform
by configuration information;

determining, based on the configuration information, that the platform instance is
configured to run a first service that is defined by a corresponding service class;

10 determining, based on the configuration information, that the first service is
configured to run a first plurality of blocks, wherein each block of the first plurality of blocks
is defined by a corresponding block class that contains a set of platform specific functions
that enables the block to operate within the platform instance and a set of task specific
functions that enables the block to perform a specific real time task for any service that is
15 configured to use the block;

starting the first service based on the corresponding service class and the
configuration information;

starting each block of the first plurality of blocks based on the corresponding block
class and the configuration information; and

20 processing a first plurality of incoming signals in real time using the task specific
functions of the first plurality of blocks, wherein an order of execution of the blocks within
the first service is defined by the configuration information.

2. The method of claim 1 characterized by converting the incoming signals into a
25 plurality of generic data objects, wherein only the generic data objects are passed between the
blocks.

3. The method of any one of claims 1 or 2 wherein the first service is configured
to direct the signals among the blocks based on a routing table that contains the order of
30 execution, wherein a publication by one of the blocks is routed to another of the blocks by the
first service based on the routing table.

4. The method of any one of claims 1 through 3 characterized by:

determining, based on the configuration information, that the platform instance is

configured to run a second service simultaneously with the first service, wherein the second service is defined by a corresponding service class;

determining, based on the configuration information, that the second service is configured to run a second plurality of blocks;

5 starting the second service based on the corresponding service class and the configuration information;

starting each block of the second plurality of blocks based on the corresponding block class and the configuration information; and

10 processing a second plurality of incoming signals using the task specific functions of the second plurality of blocks, wherein an order of execution of the blocks within the second service is defined by the configuration information.

5. The method of claim 4 wherein the first service and the second service are independent and do not communicate with each other.

15

6. The method of claim 4 wherein the first service and the second service are identical in functionality.

7. The method of claim 4 wherein the first service and the second service operate
20 together to form a processing chain, wherein an output of the first service is an input to the second service.

8. The method of claim 4 wherein the first plurality of incoming signals is from a first source and is a first type of data, and wherein the second plurality of incoming signals is
25 from a second source and is a second type of data, and wherein the first source is different from the second source, and the first type of data is different from the second type of data.

9. The method of claim 8 wherein at least one of the first plurality of incoming signals and the second plurality of incoming signals is a signal stream.

30

10. The method any of one of claims 1 through 9 characterized by:
receiving a command to stop the first service; and
stopping the first service, wherein the core and any other services that have not been stopped continue running after the first service has been stopped.

11. A processing system characterized by:

a processor; and

a memory coupled to the processor, the memory containing instructions for execution

5 by the processor for a configurable processing platform that includes:

a core configured to interact with an operating system, wherein the core is configurable to simultaneously run any of a plurality of services that are defined for the processing platform by configuration information;

10 at least one service defined by a service class and the configuration information to include a set of platform specific instructions that enable the service to operate within the processing platform and a set of service specific instructions that enable the service to run a plurality of blocks that provide task specific functionality to the service; and

15 the plurality of blocks, wherein each block is defined by a block class and the configuration information to include a set of platform specific instructions that enable the block to operate asynchronously and independently from the other blocks within the processing platform and a set of task specific instructions that enable the block to perform a specific real time processing task for the service that uses the block.

20 12. The processing system of claim 11 wherein the first service is configured to direct data among the blocks based on a routing table that contains the order of execution, wherein a publication by one of the blocks is routed to another of the blocks by the first service based on the routing table.

25 13. The processing system of any one of claims 11 or 12 wherein the processing platform is further configured to run a second service simultaneously with the first service, and wherein the second service is configured to manage a plurality of second blocks, and wherein an order of execution of the second blocks within the second service is defined by the configuration information.

30

14. The processing system of claim 12 wherein either of the first service and the second service can be started and stopped independently.

15. The processing system of any of claims 11 through 14 characterized by a plurality of services including the first service, wherein each service is coupled to a different signal source, and wherein the services asynchronously process signals received from their respective signal sources and produce an output in real time.

5

16. A software architecture for a configurable processing platform for use on a device, the architecture comprising:

a core configured to interact with an operating system on the device, wherein the core is configurable to simultaneously run any of a plurality of services that are defined for the processing platform by configuration information, wherein each service to be run on the processing platform is defined by a service class and the configuration information to include a set of platform specific instructions that enable the service to operate within the processing platform and a set of service specific instructions that enable the service to run a plurality of blocks that provide task specific functionality to the service, and wherein each block to be run on the processing platform is defined by a block class and the configuration information to include a set of platform specific instructions that enable the block to operate asynchronously and independently from the other blocks within the processing platform and a set of task specific instructions that enable the block to perform a specific processing task for the service that uses the block, wherein an order of execution of the blocks within the service is defined by the configuration information.

10
15
20



FIG. 1C

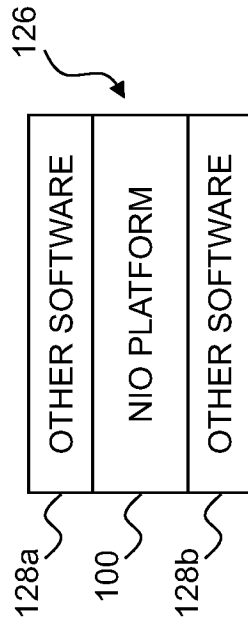


FIG. 1D

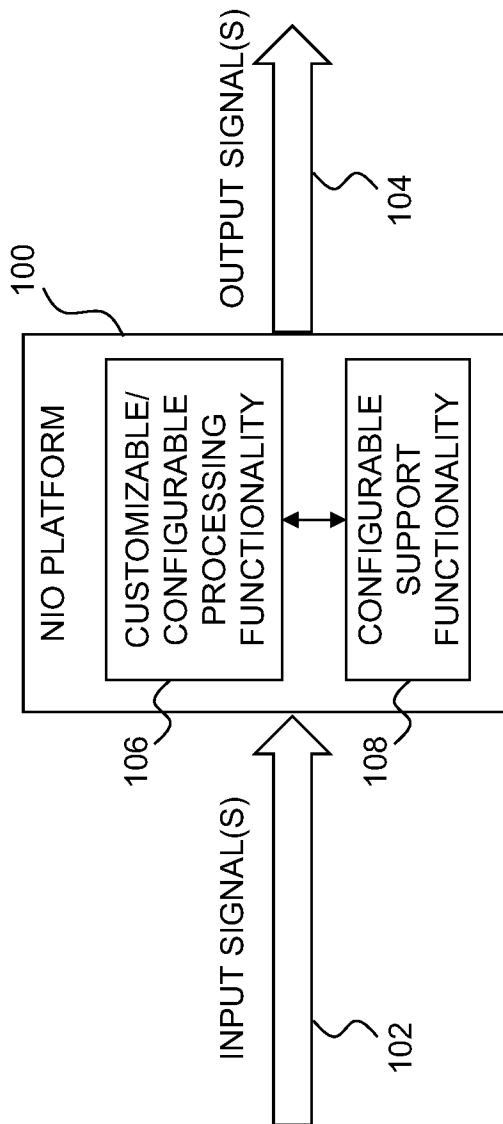


FIG. 1A

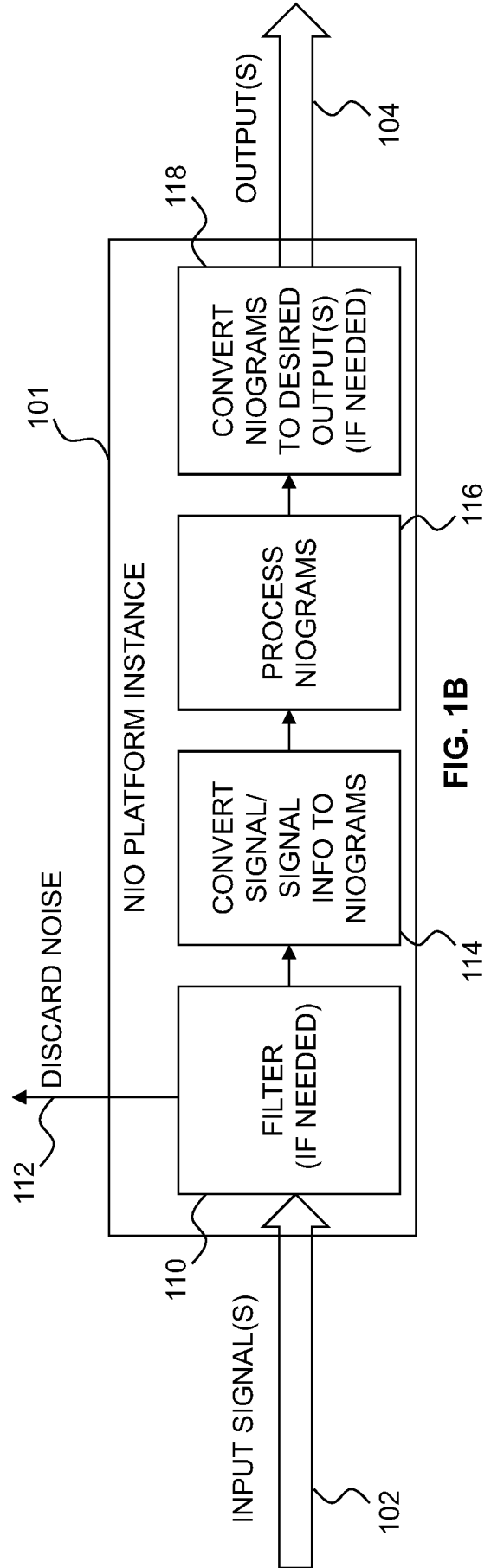


FIG. 1B

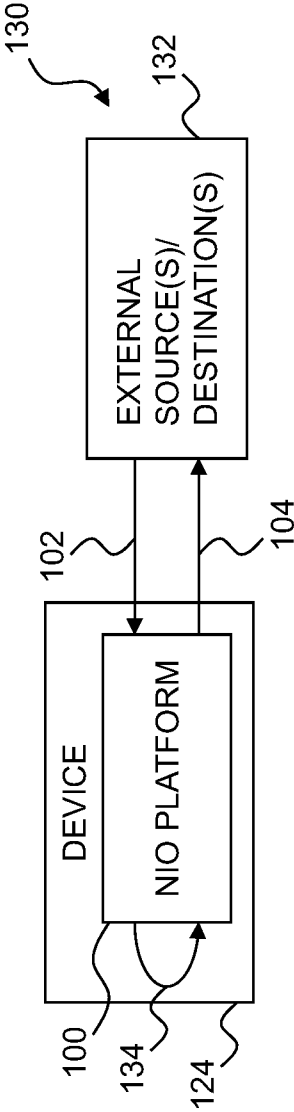


FIG. 1E

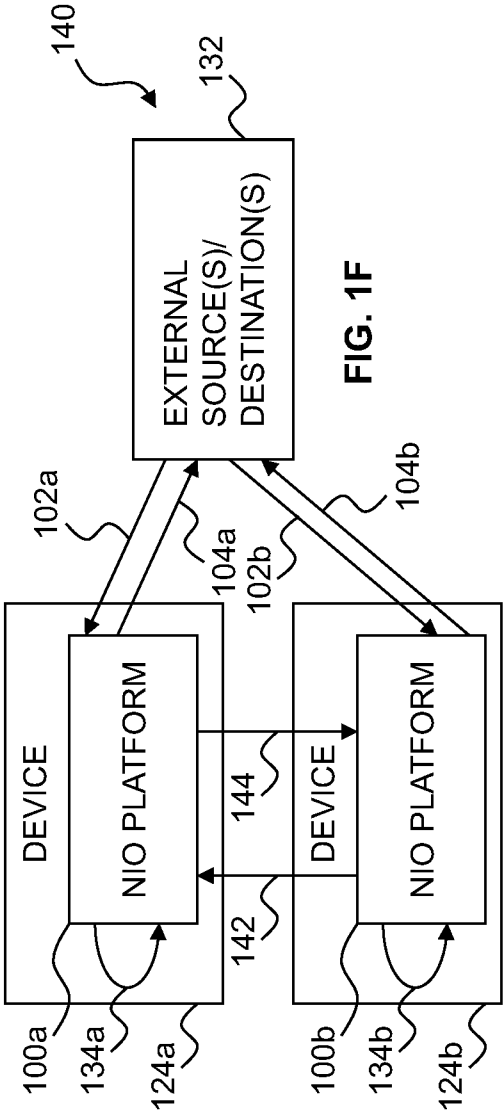


FIG. 1F

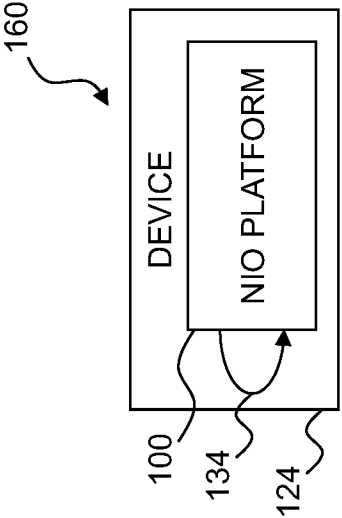


FIG. 1H

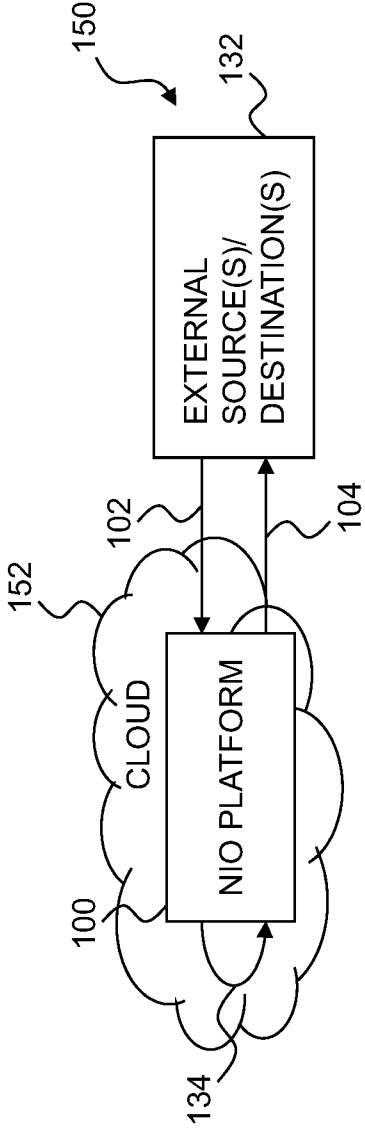


FIG. 1G

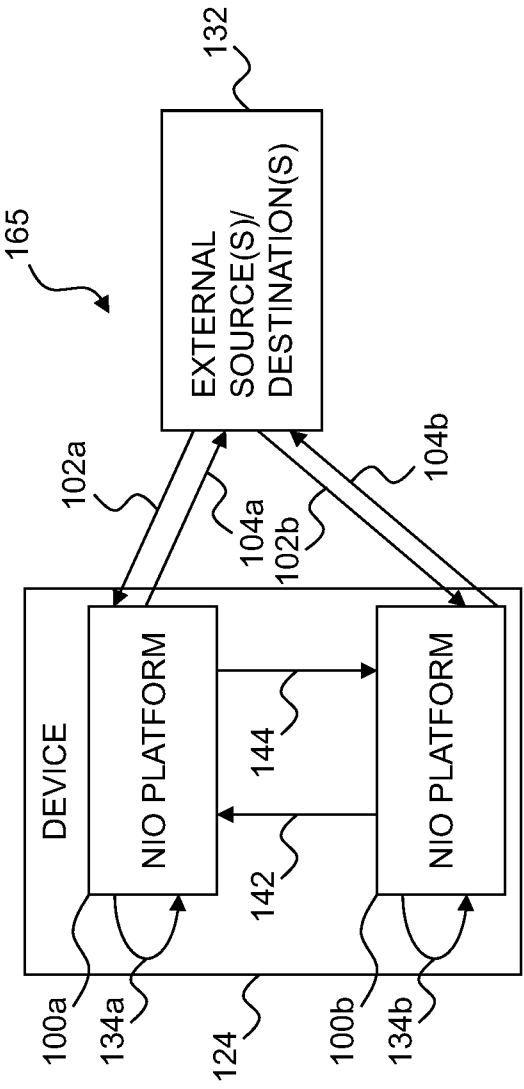


FIG. 1I

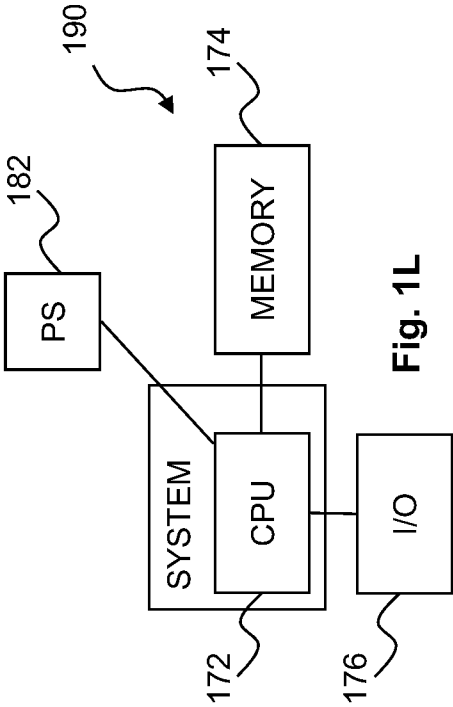


Fig. 1L

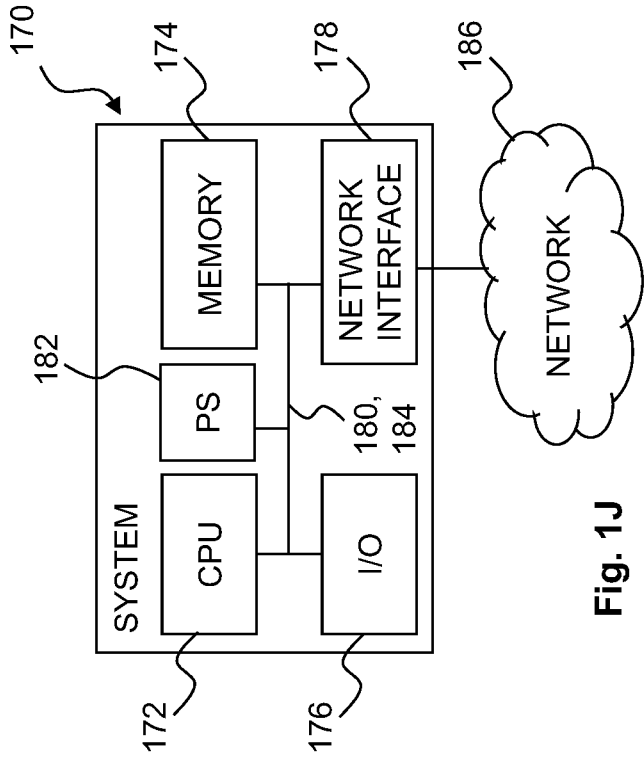


Fig. 1J

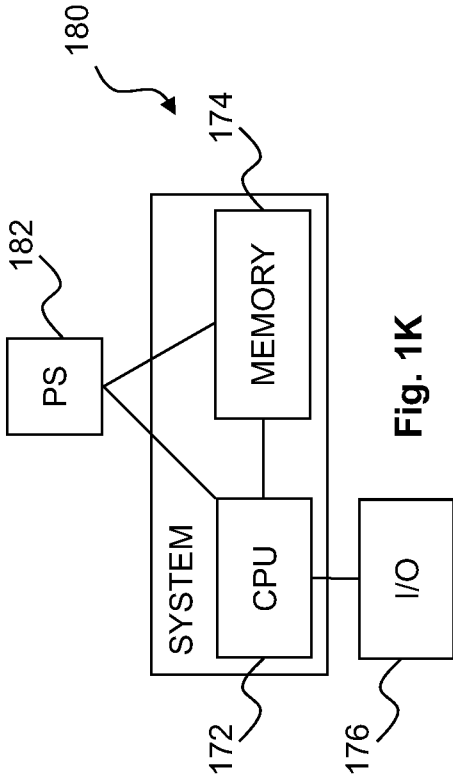


Fig. 1K

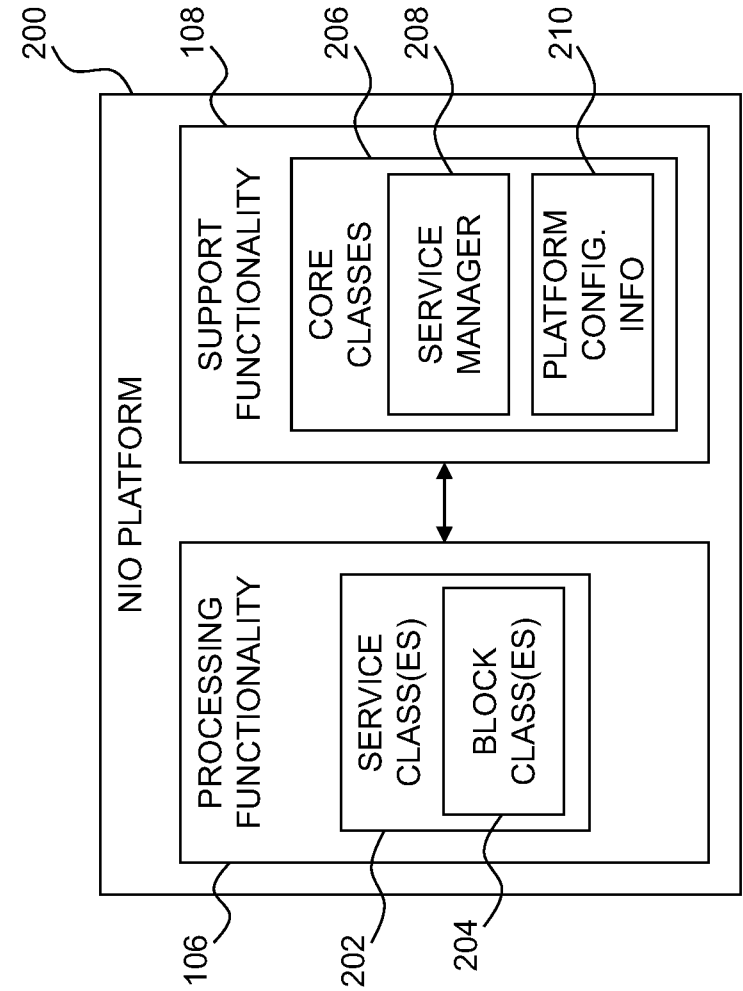


FIG. 2A

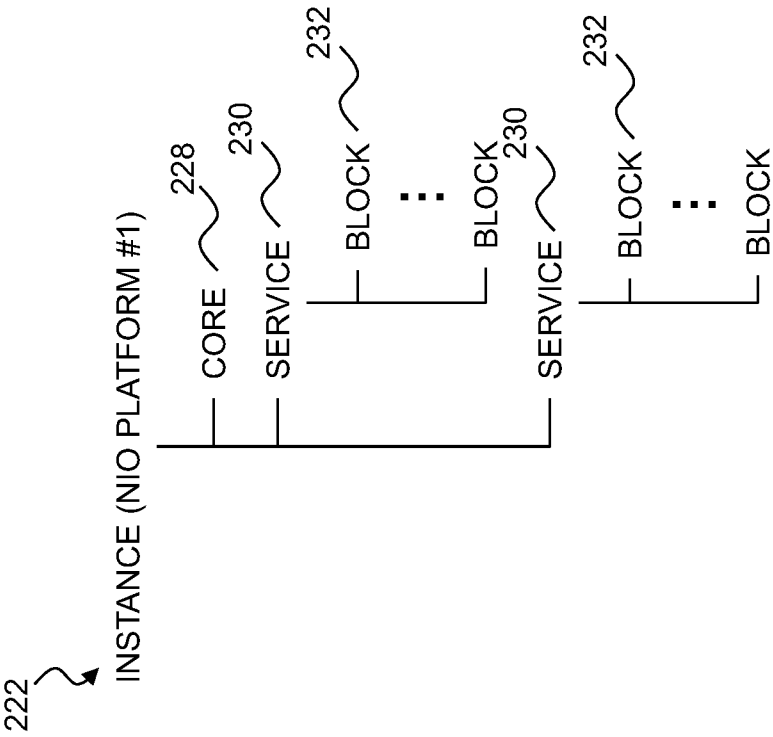


FIG. 2B

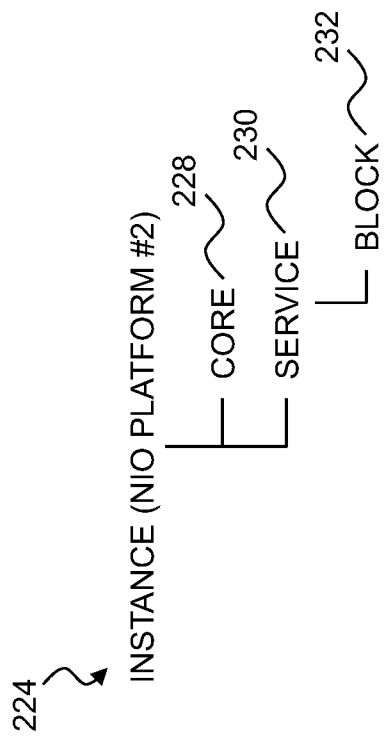


FIG. 2C

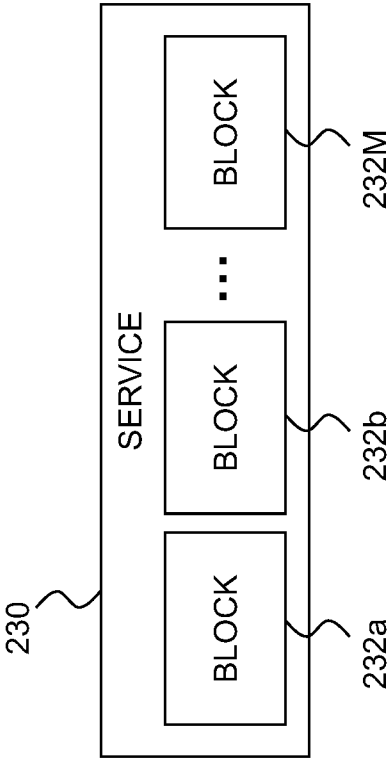


FIG. 3

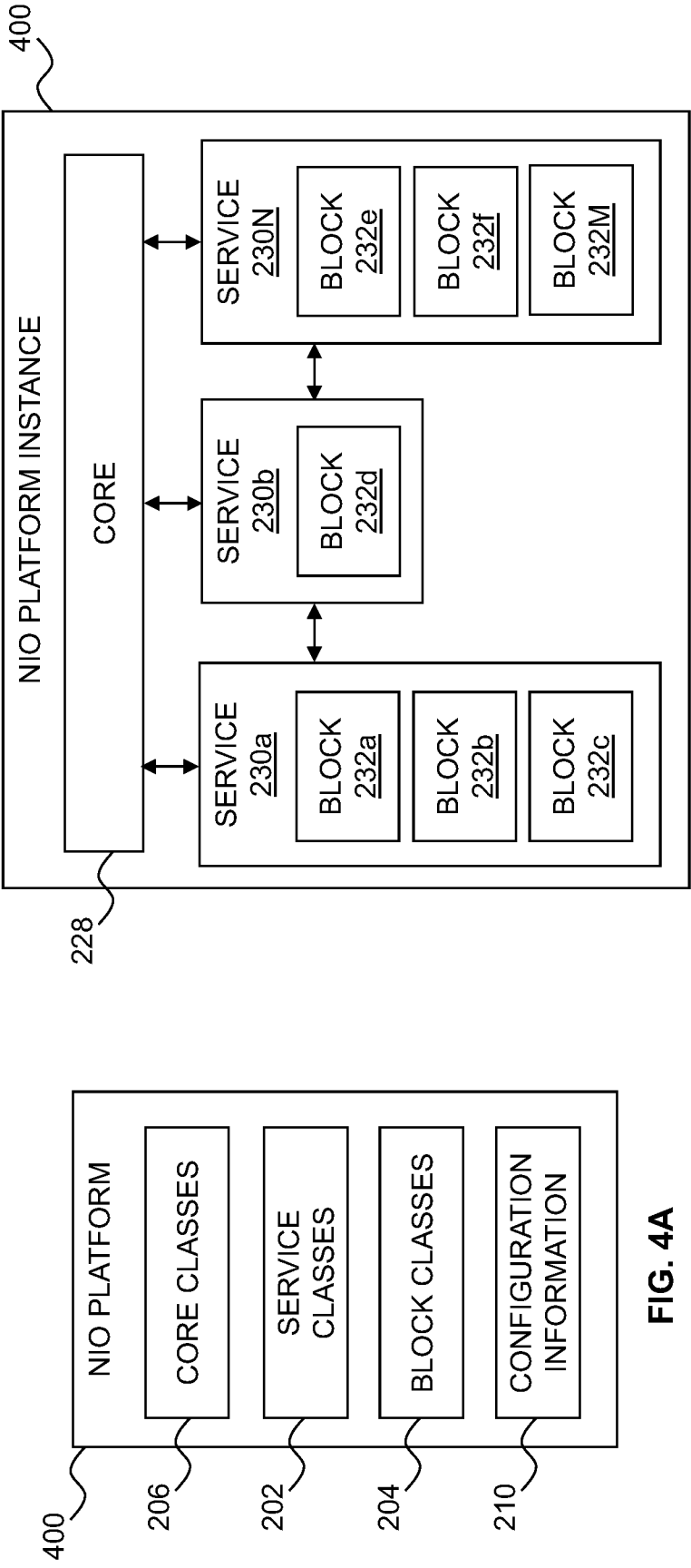


FIG. 4B

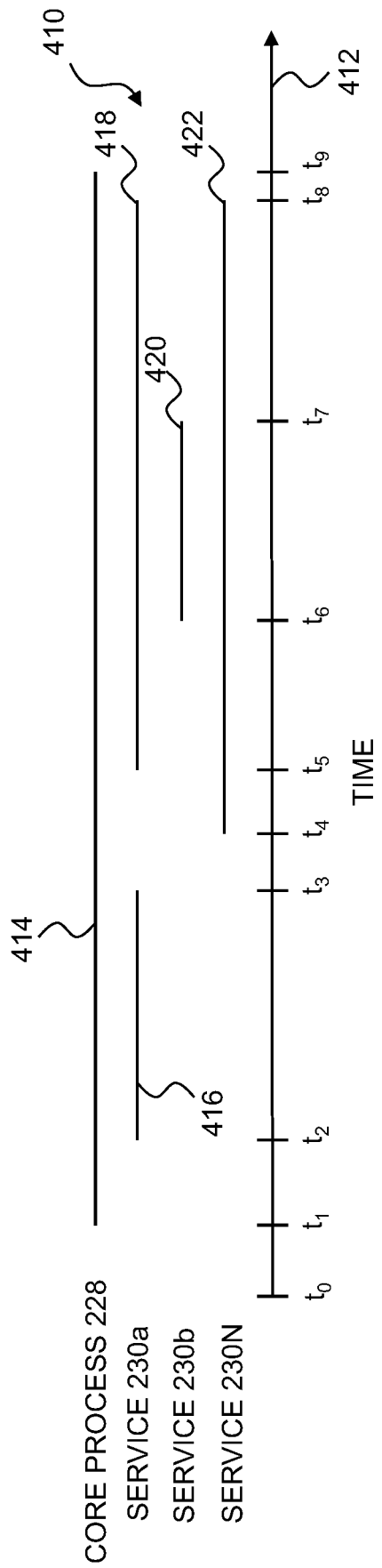


FIG. 4C

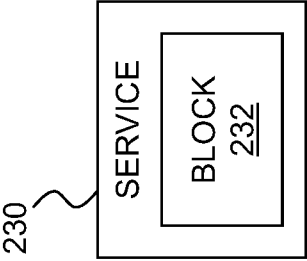


FIG. 5A

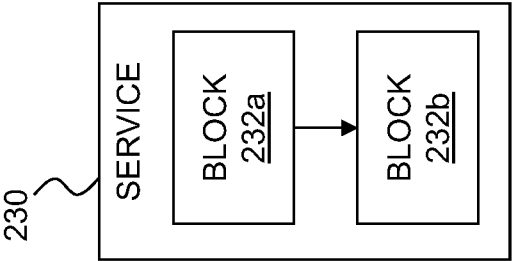


FIG. 5B

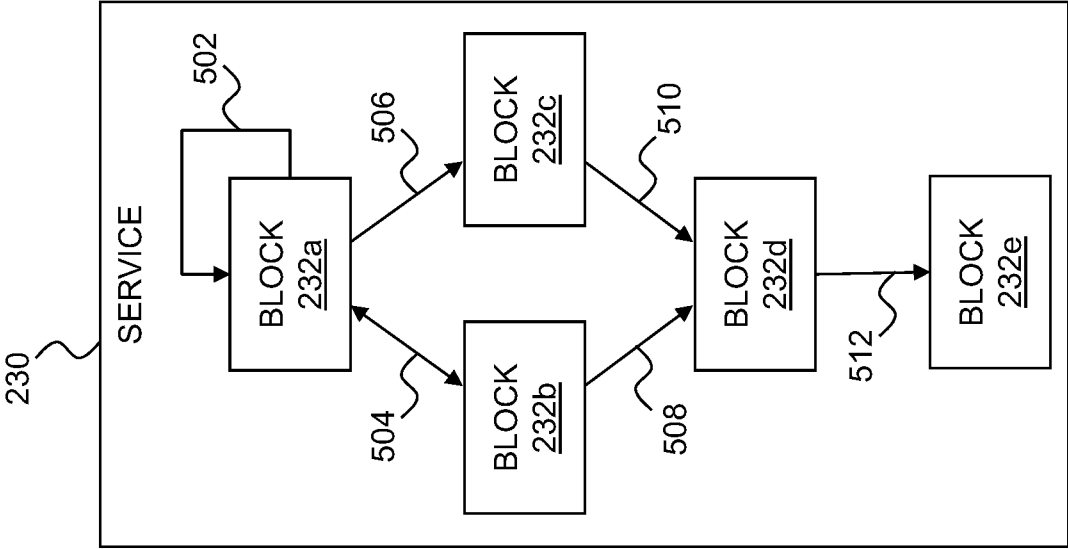


FIG. 5C

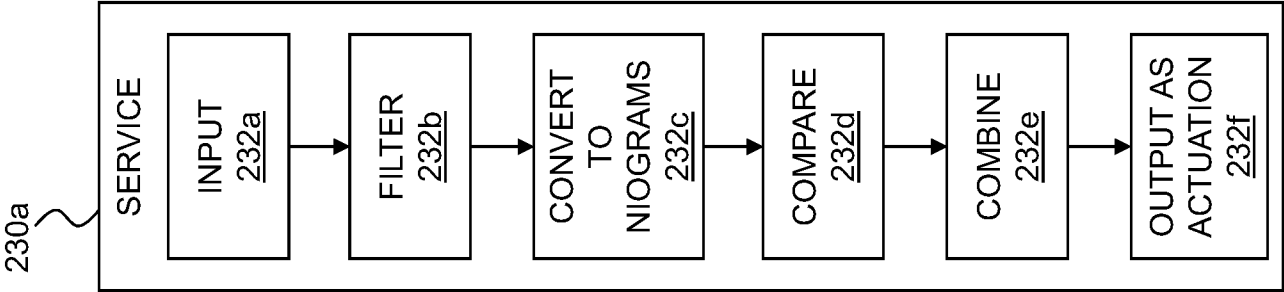


FIG. 5D

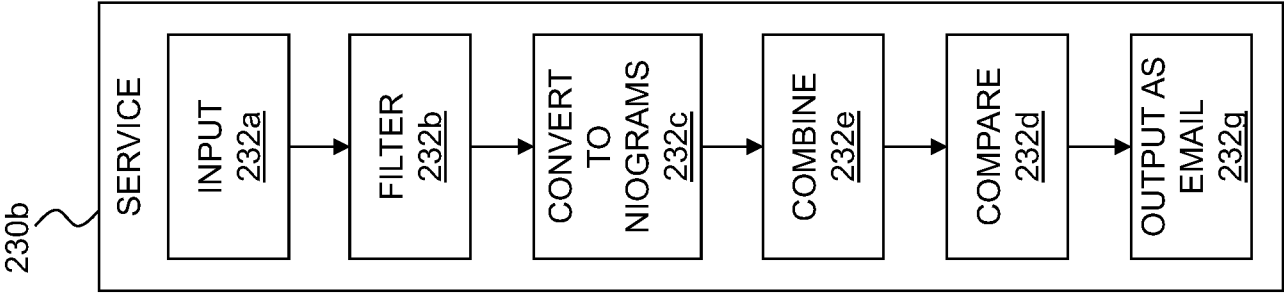


FIG. 5E

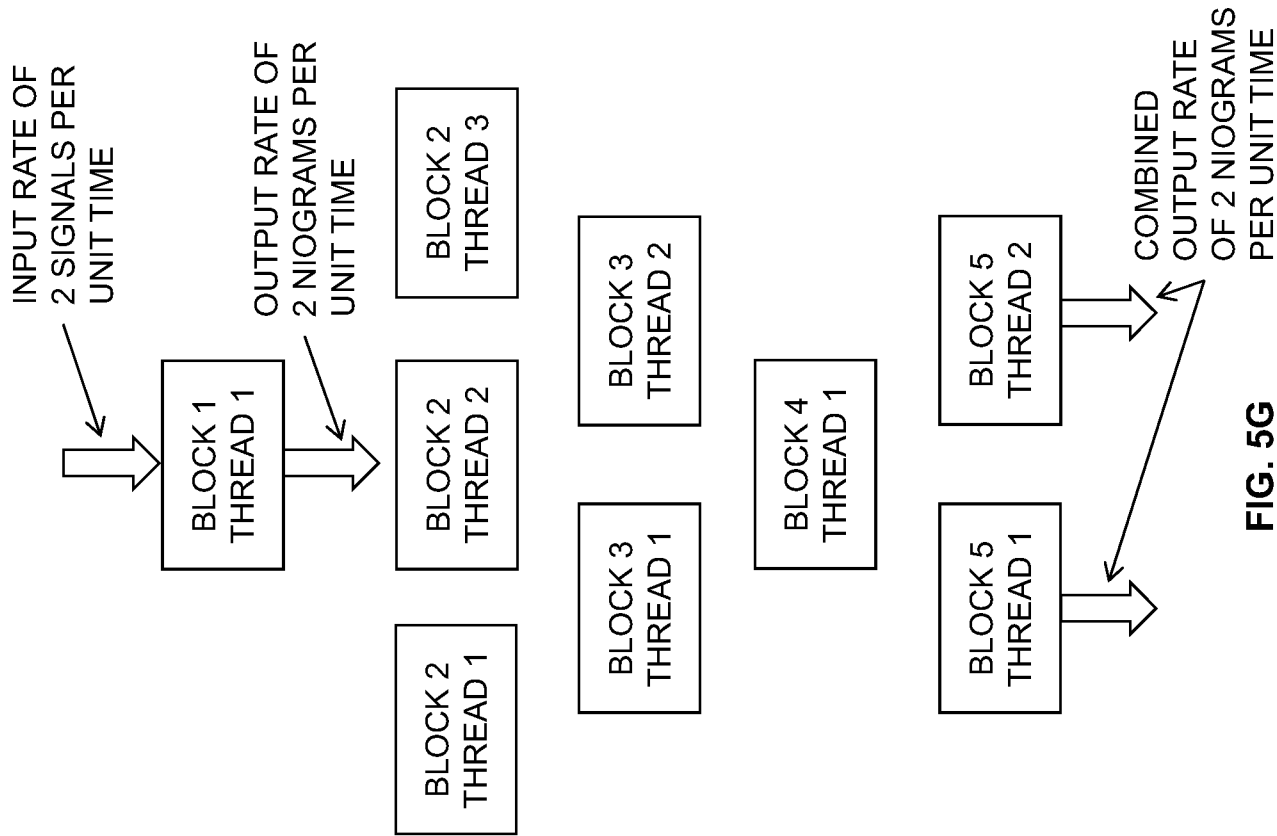


FIG. 5G

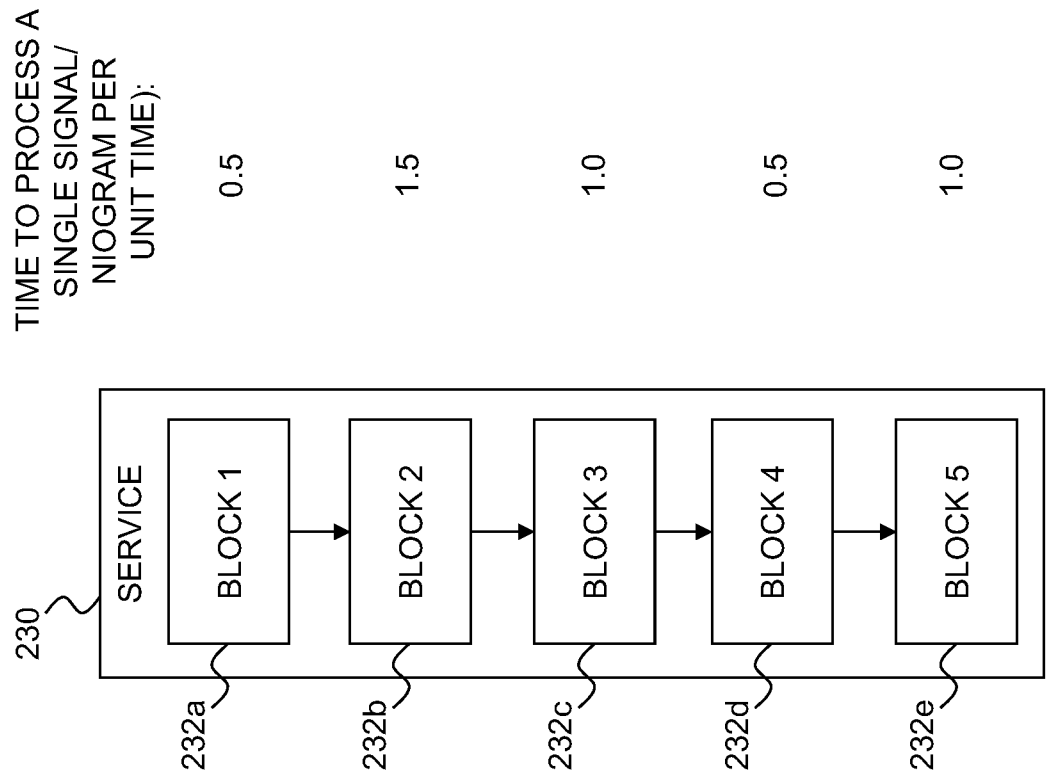


FIG. 5F

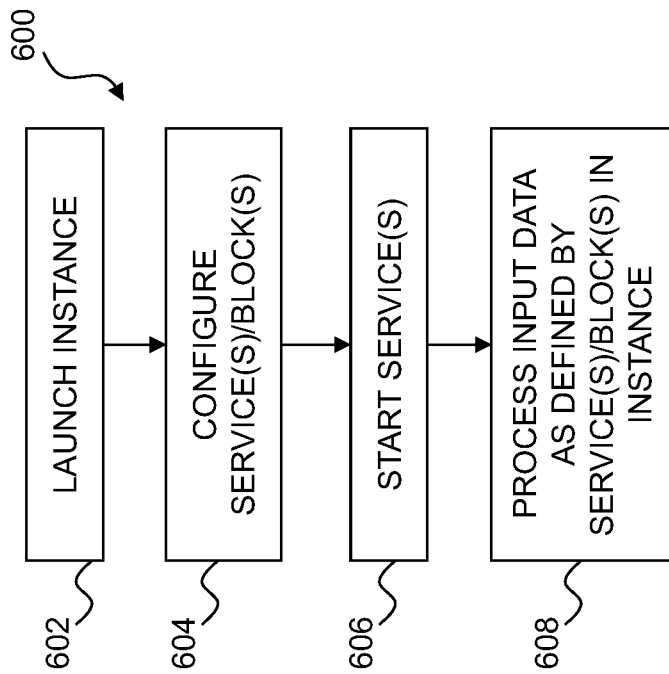


FIG. 6A

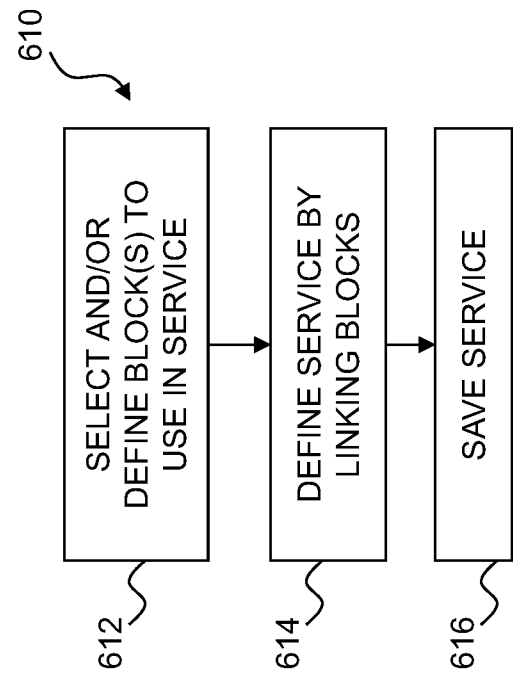


FIG. 6B

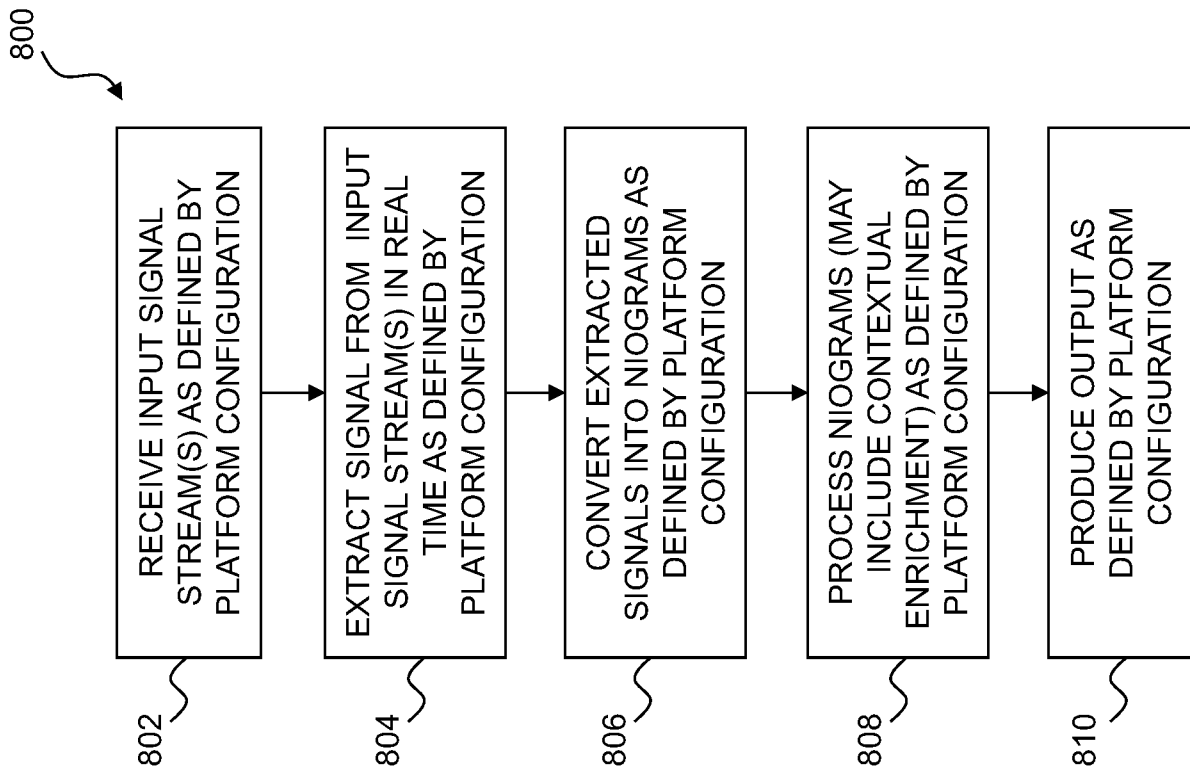


FIG. 8

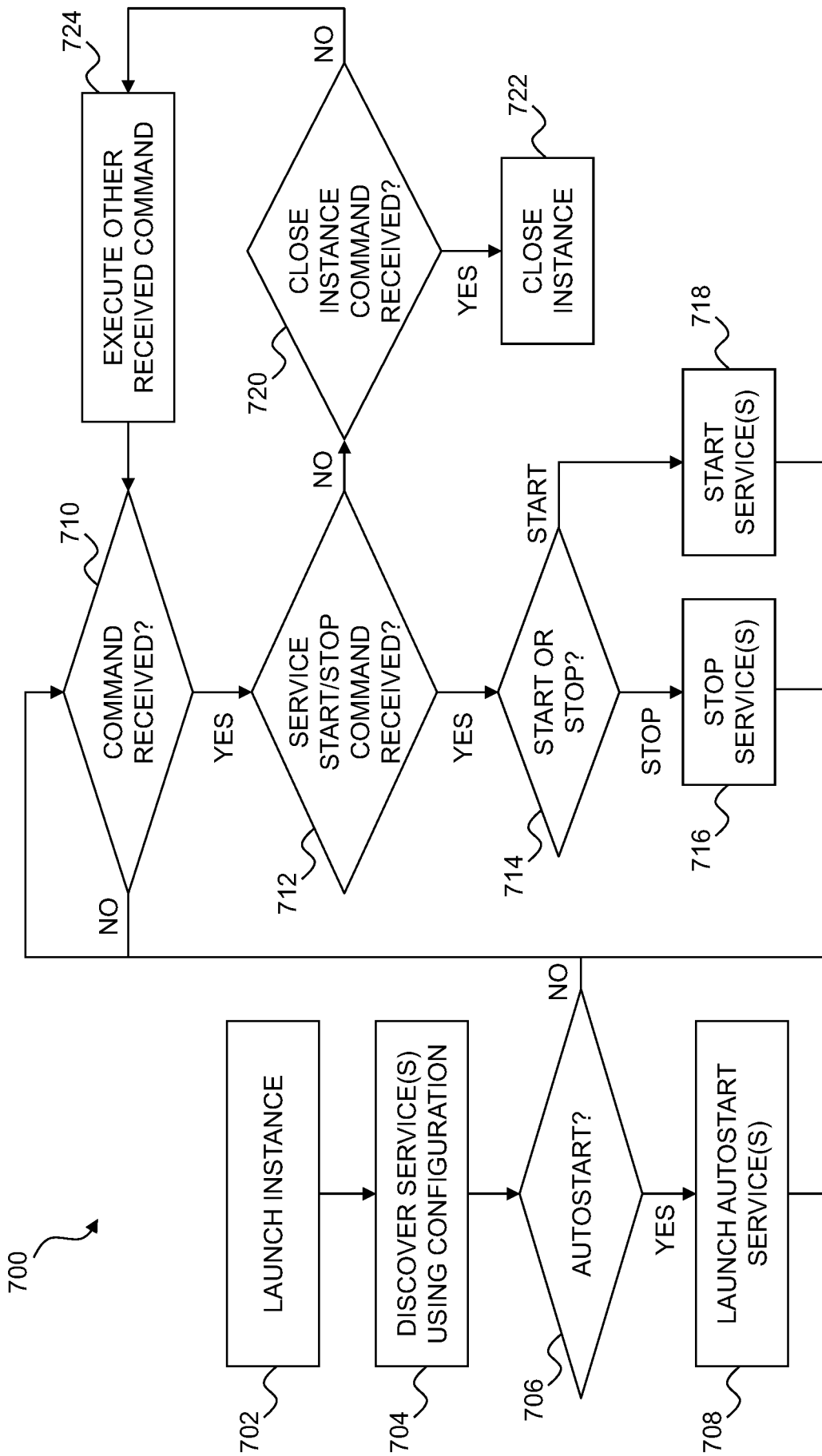


FIG. 7

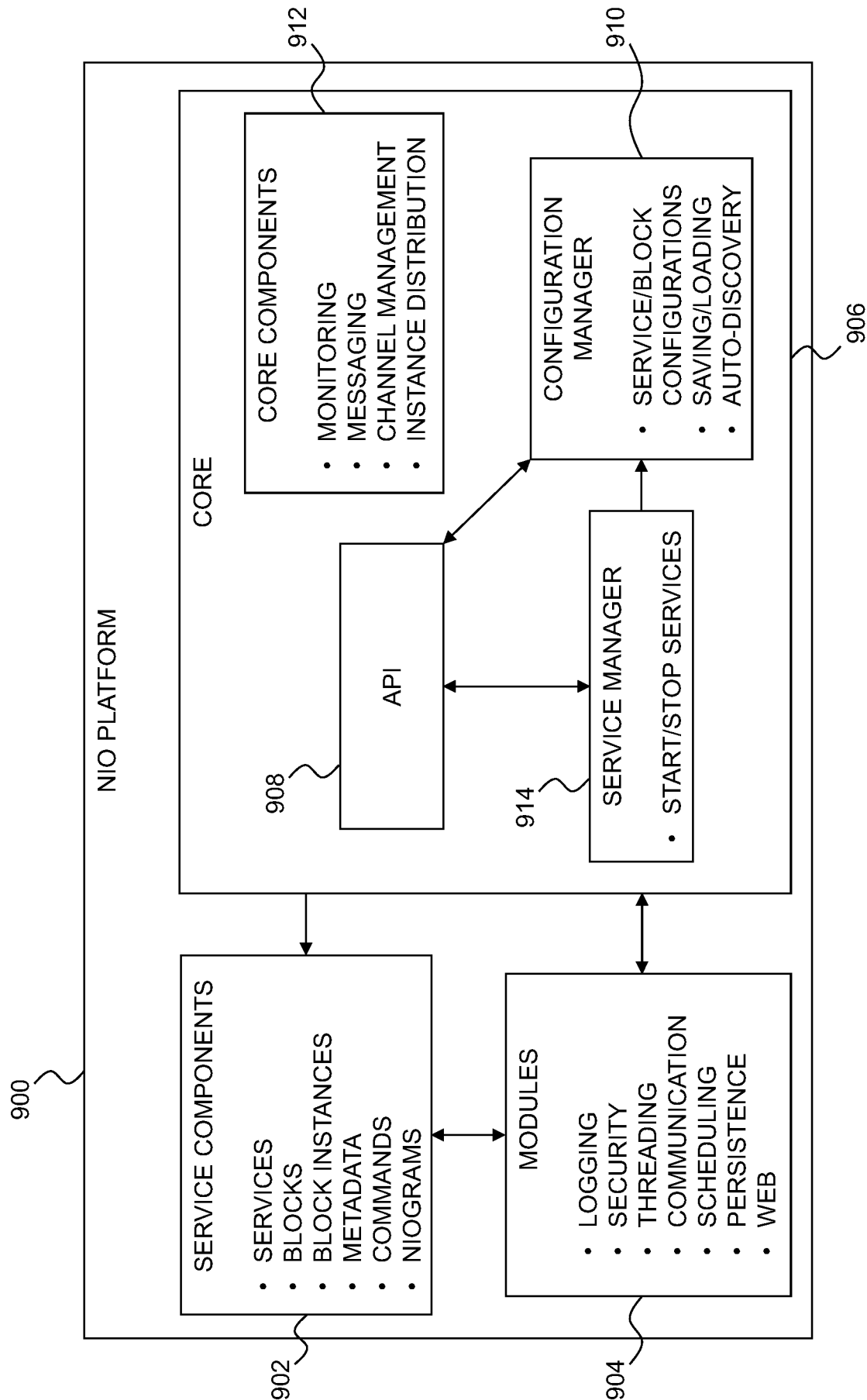


FIG. 9A

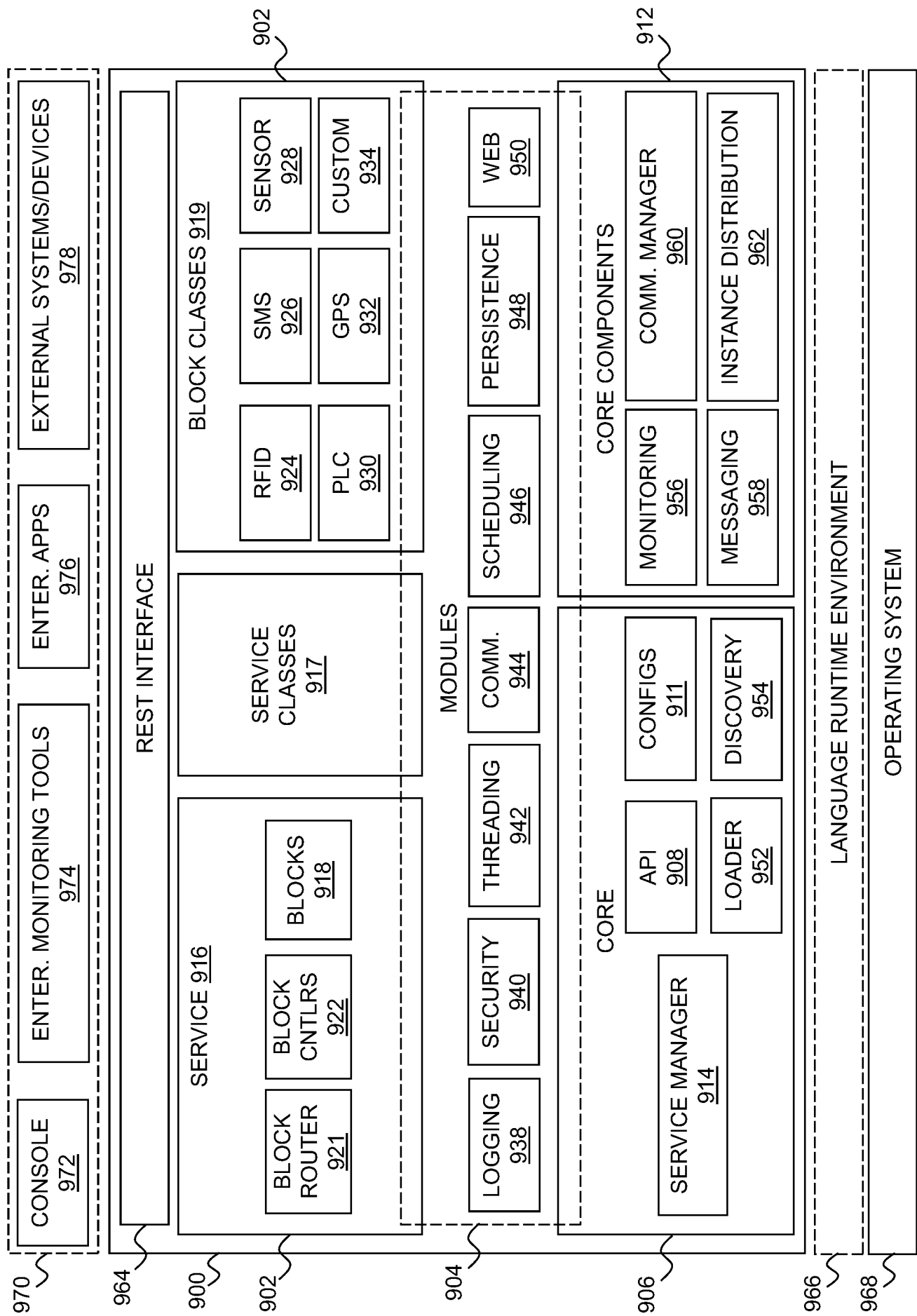


FIG. 9B

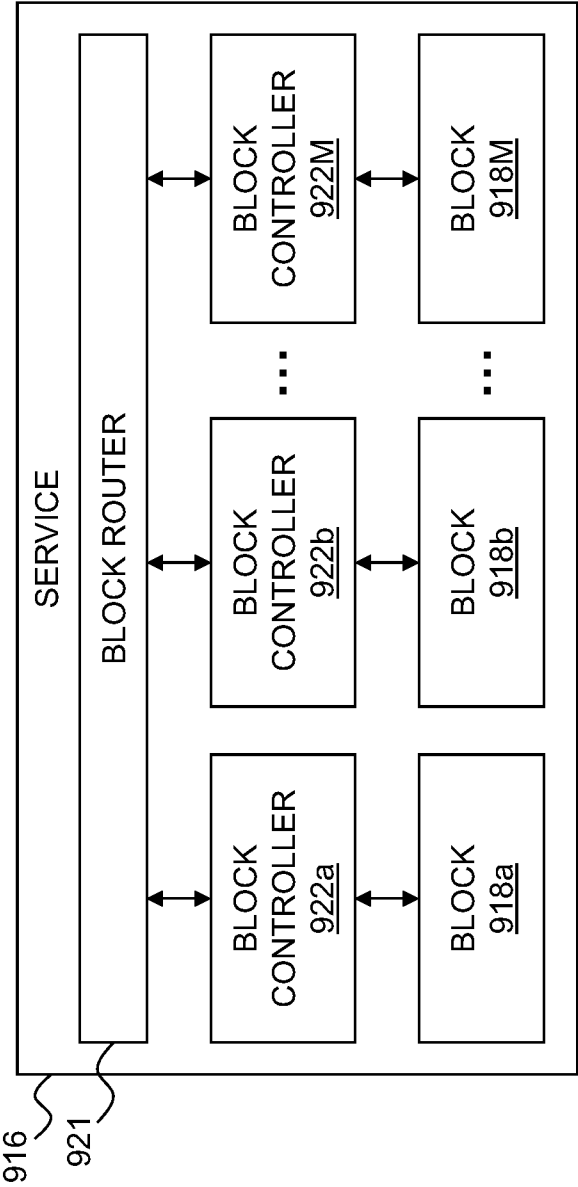


FIG. 9D

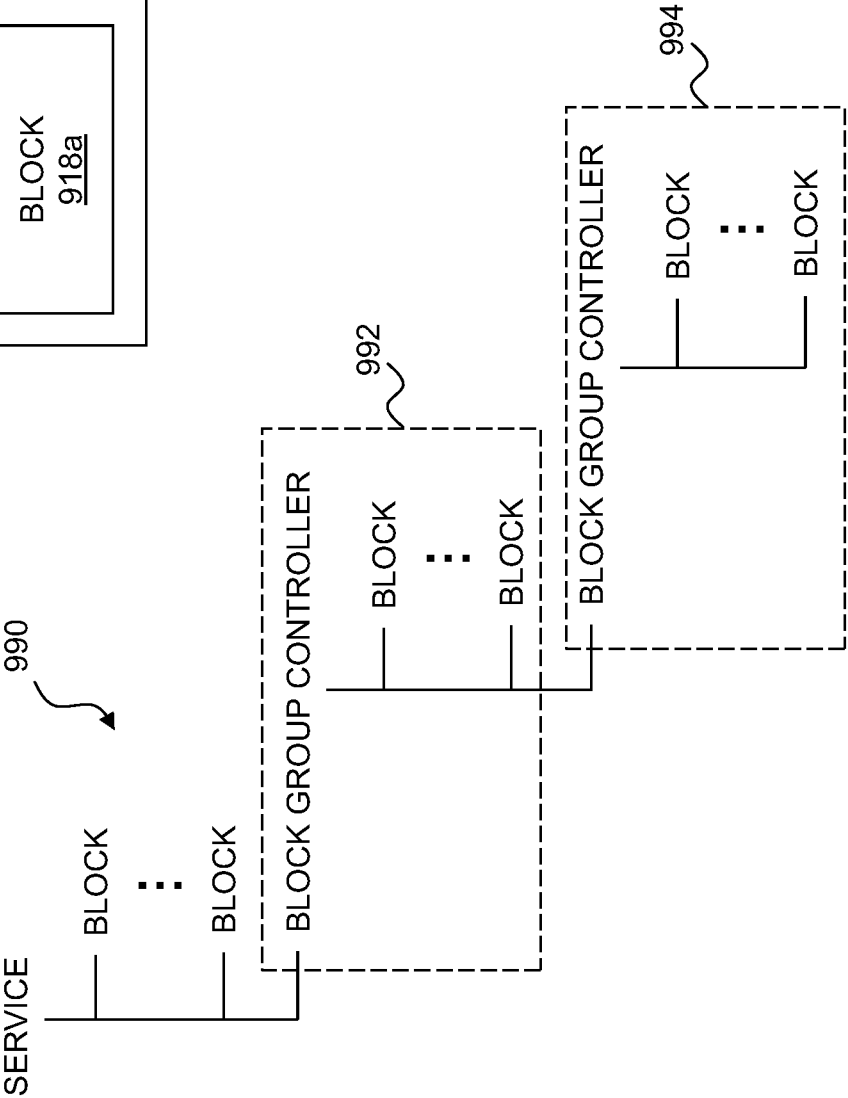


FIG. 9C

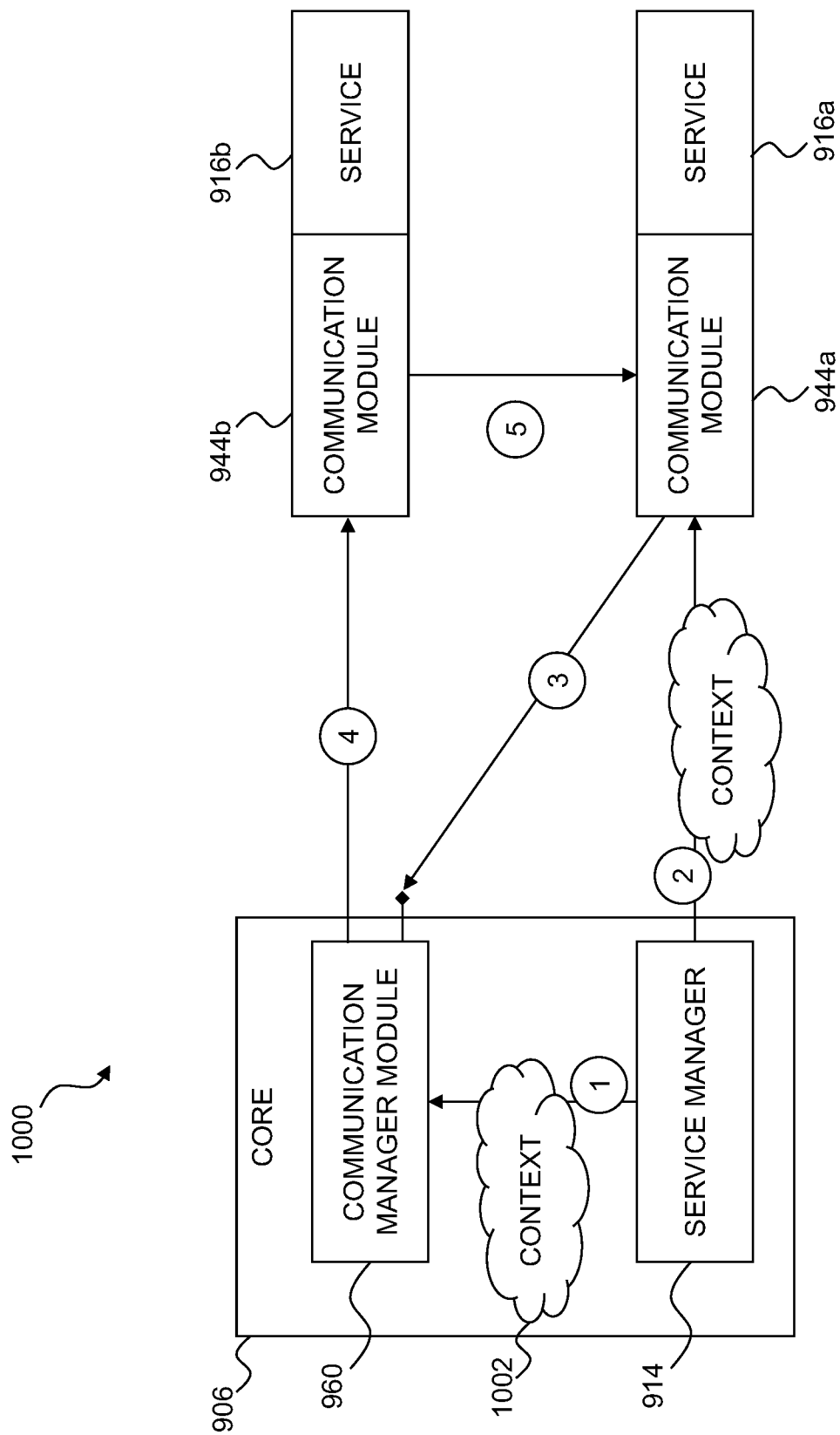


FIG. 10

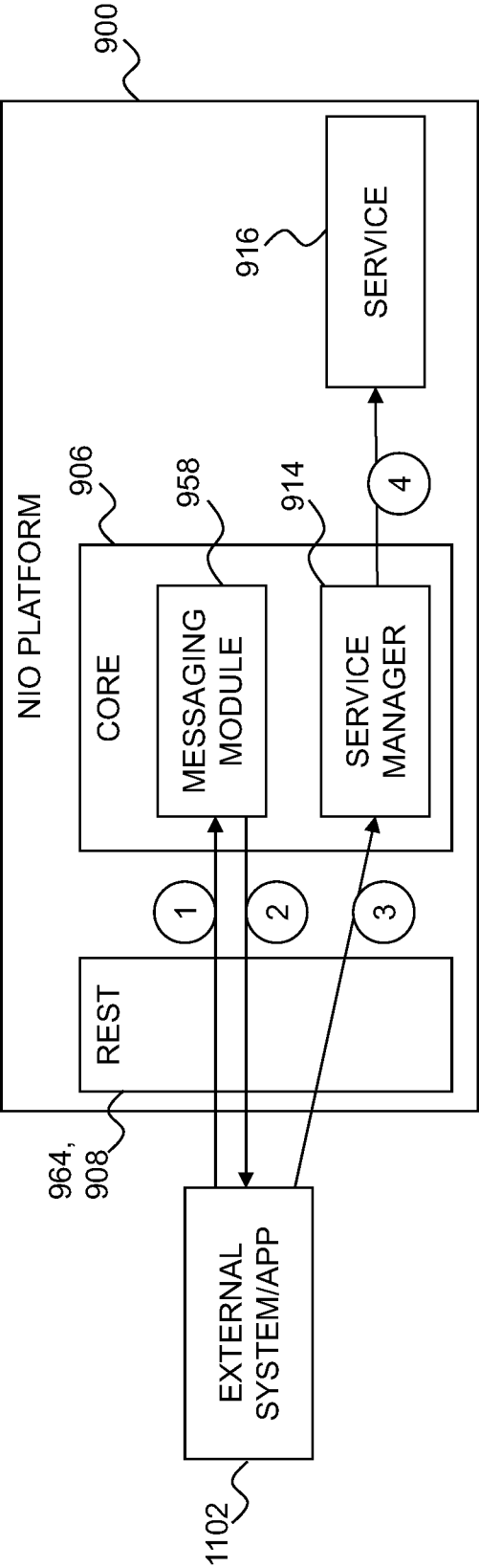


FIG. 11A

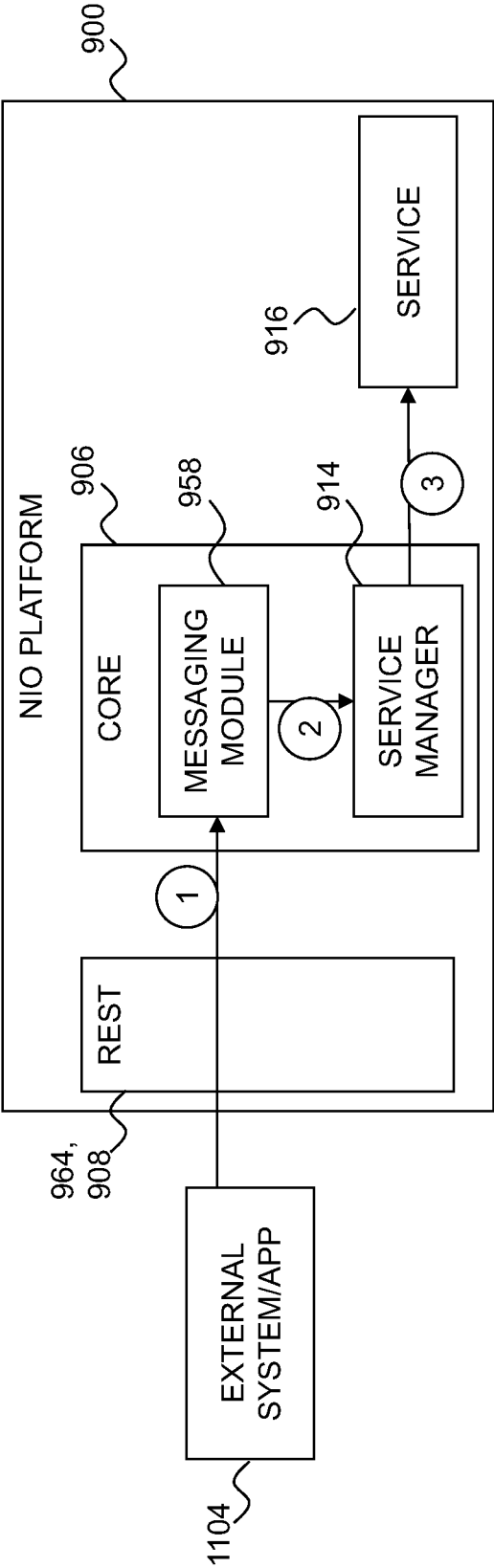


FIG. 11B

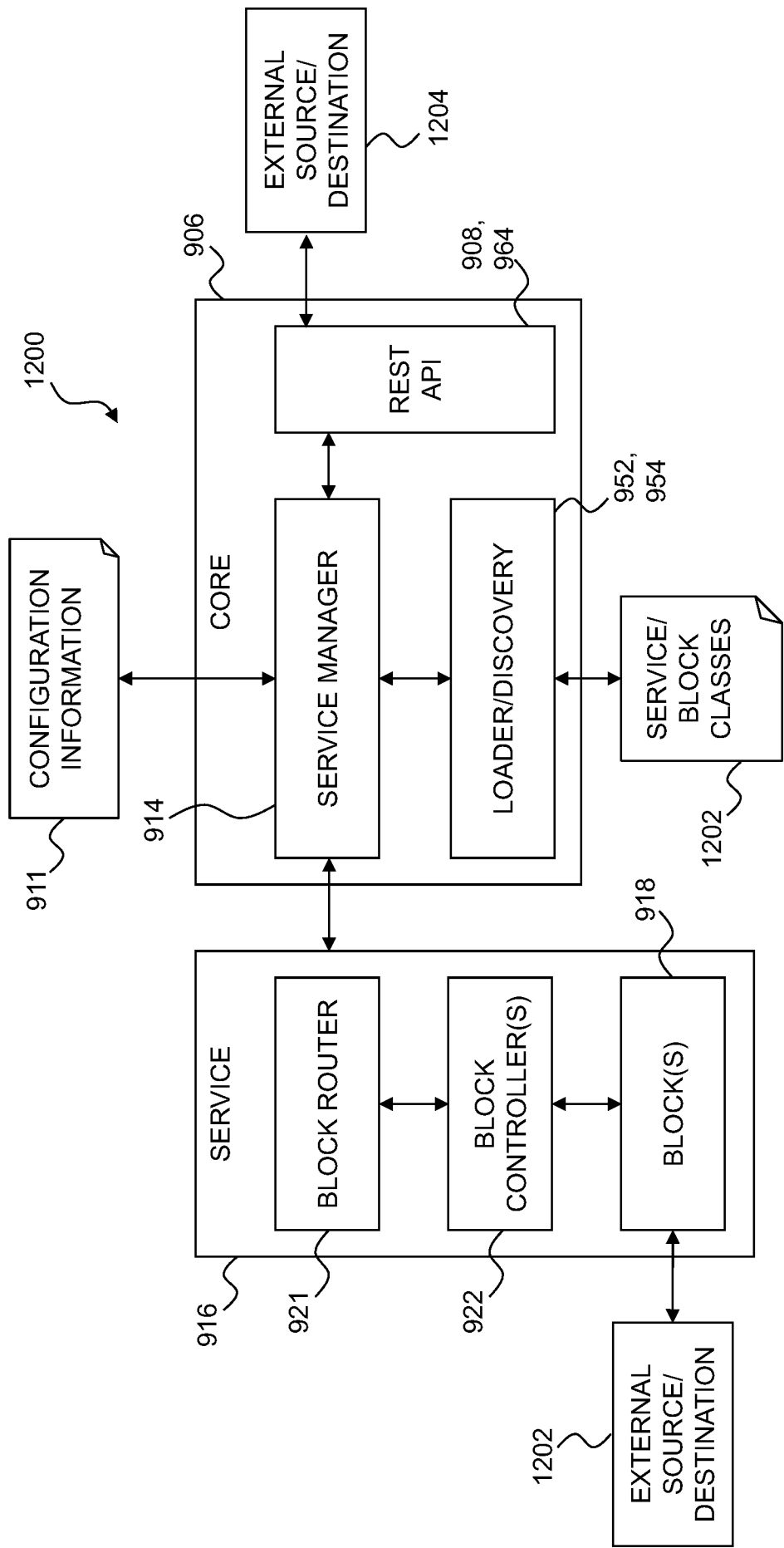


FIG. 12

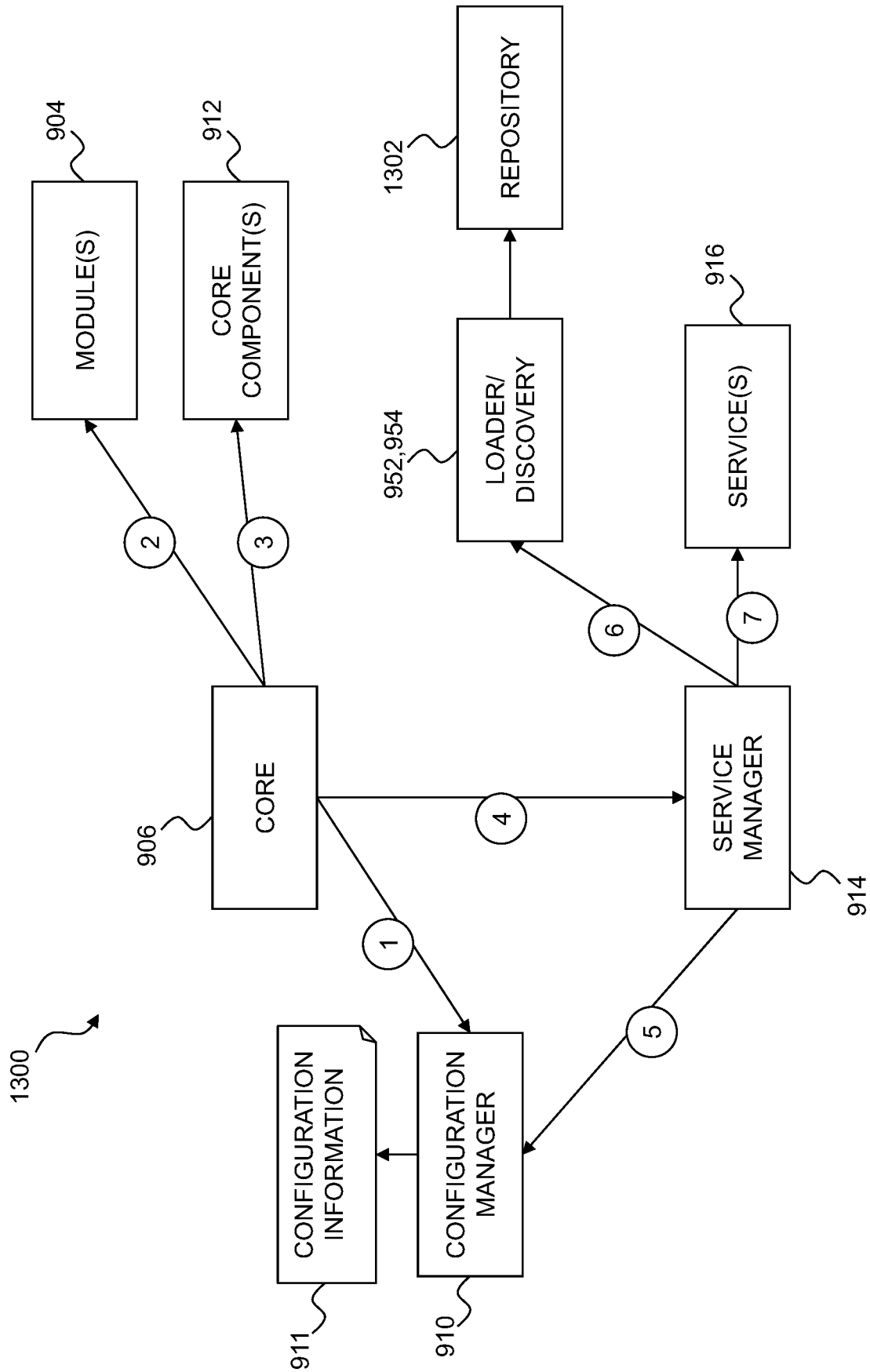


FIG. 13

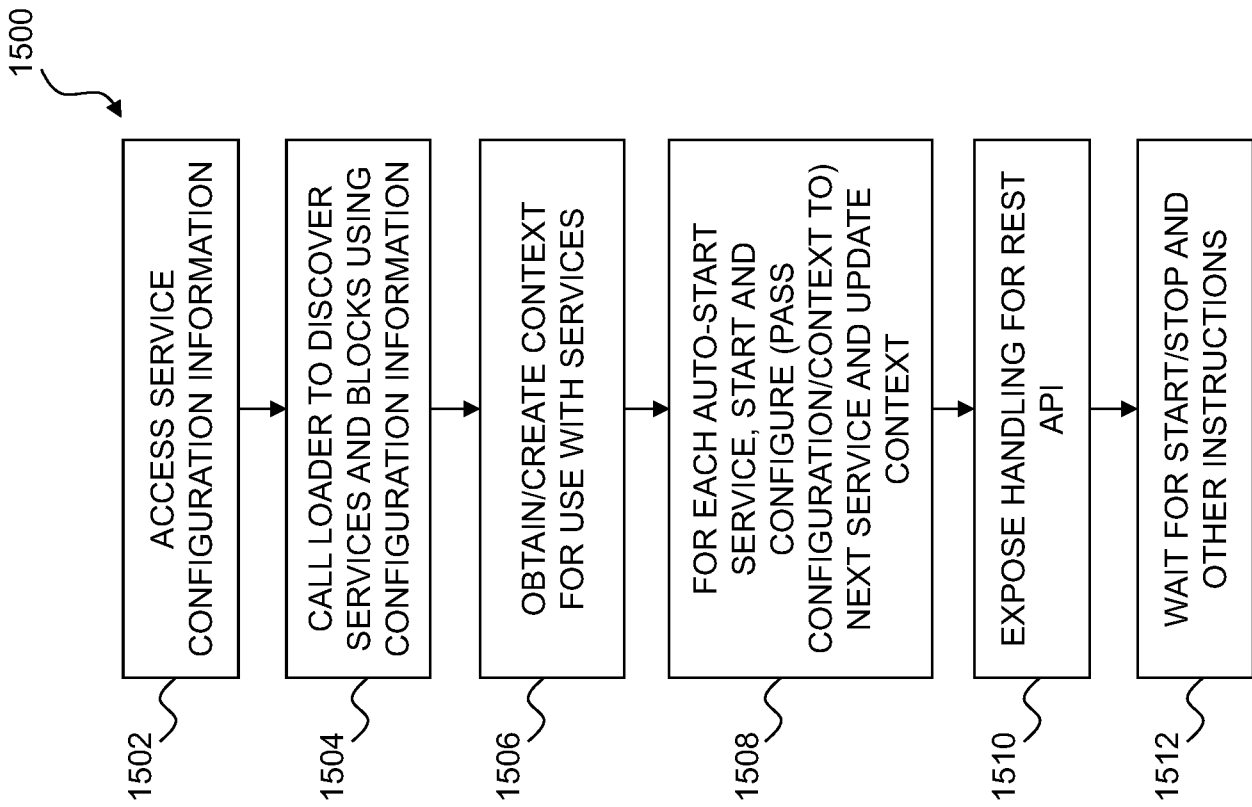


FIG. 15

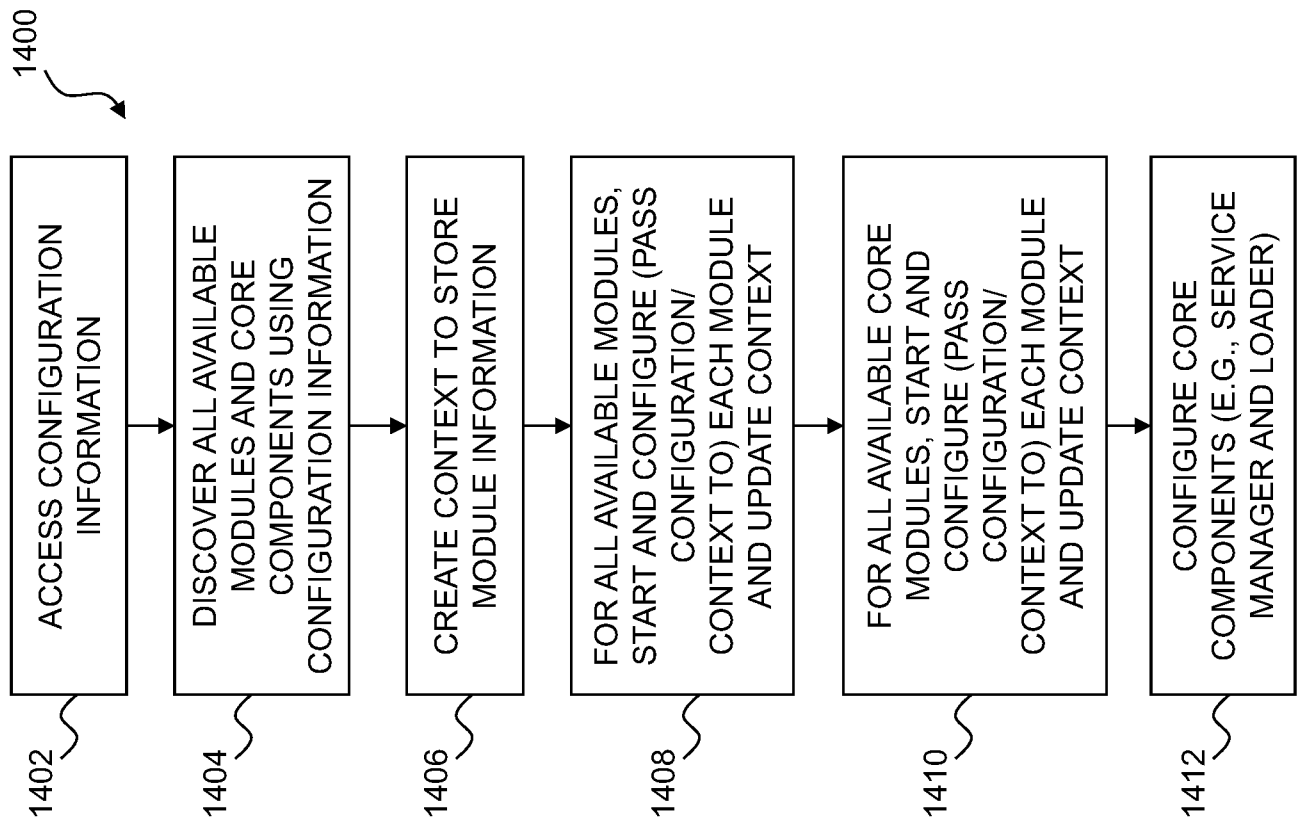


FIG. 14

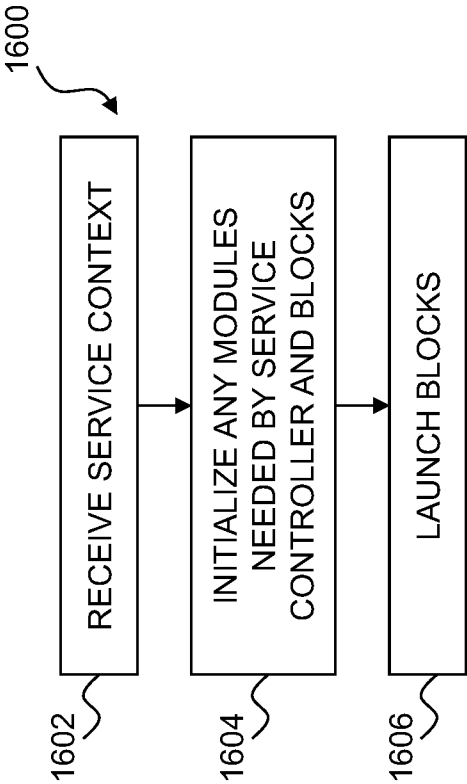


FIG. 16A

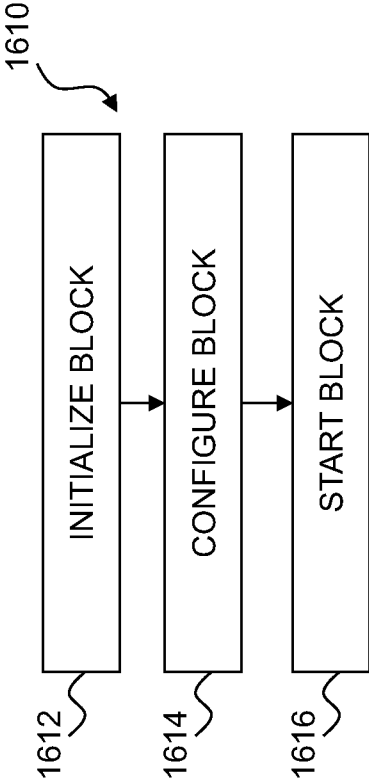


FIG. 16B

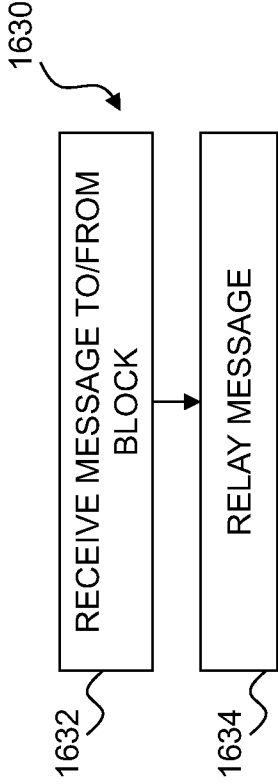


FIG. 16D

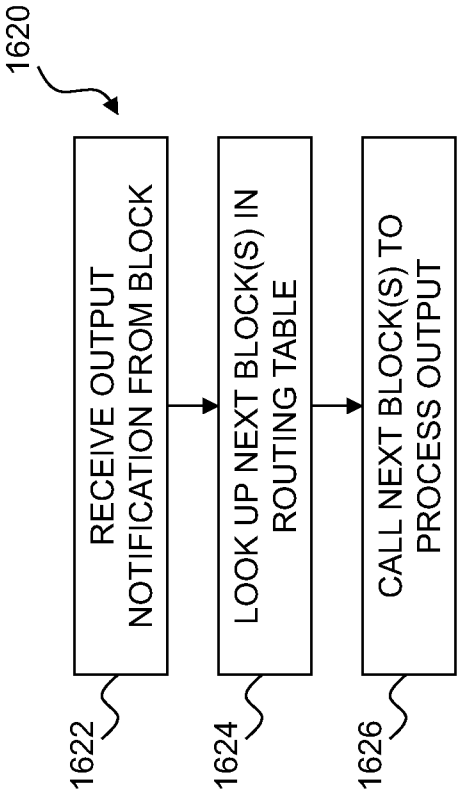


FIG. 16C

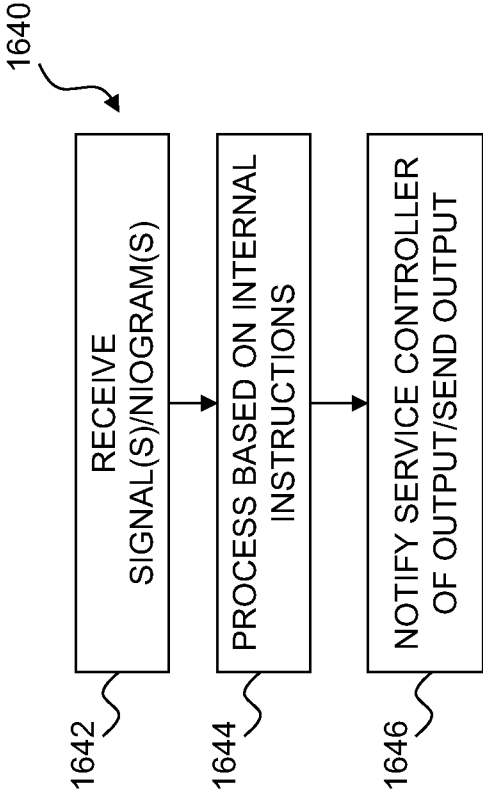


FIG. 16E

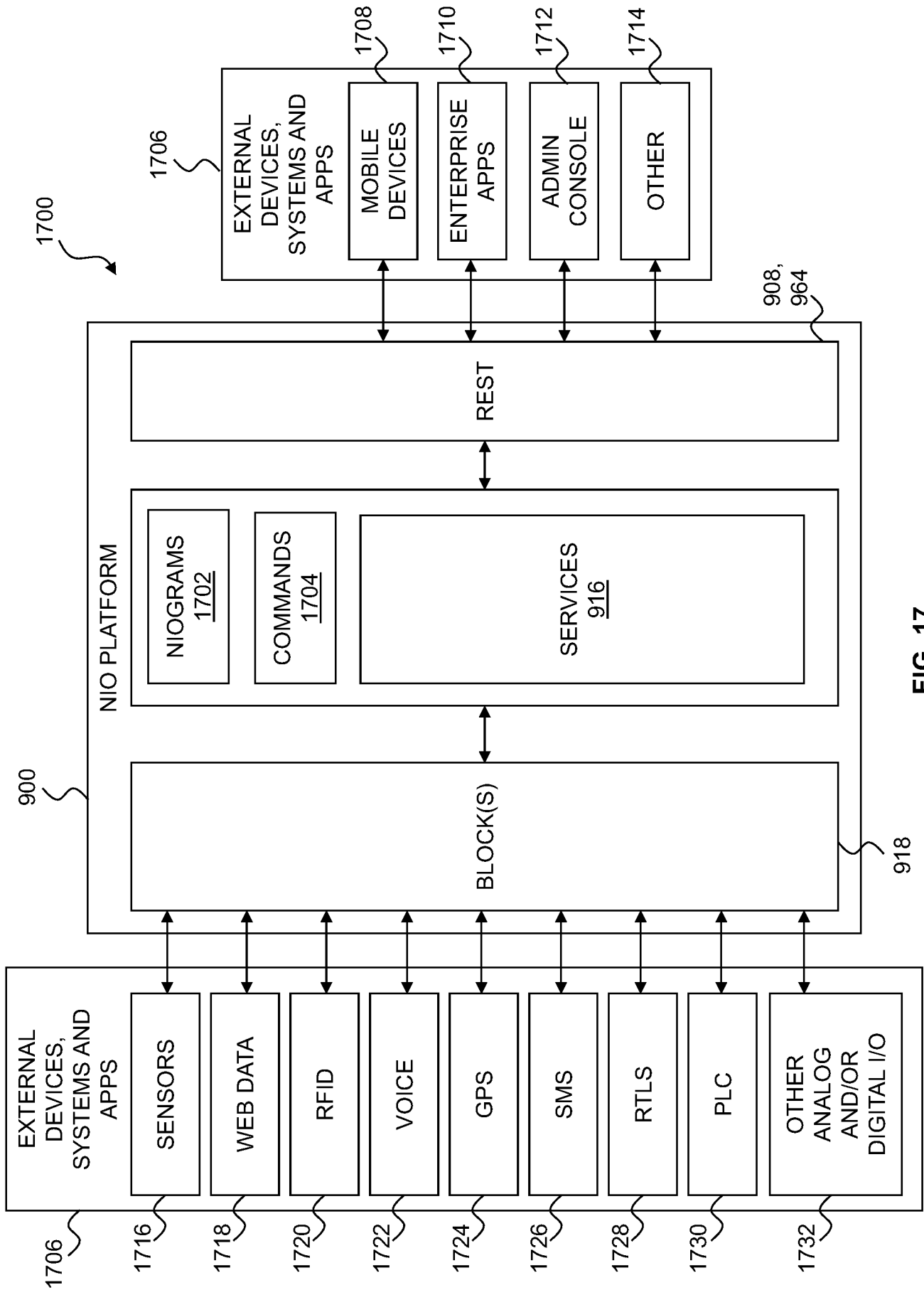


FIG. 17

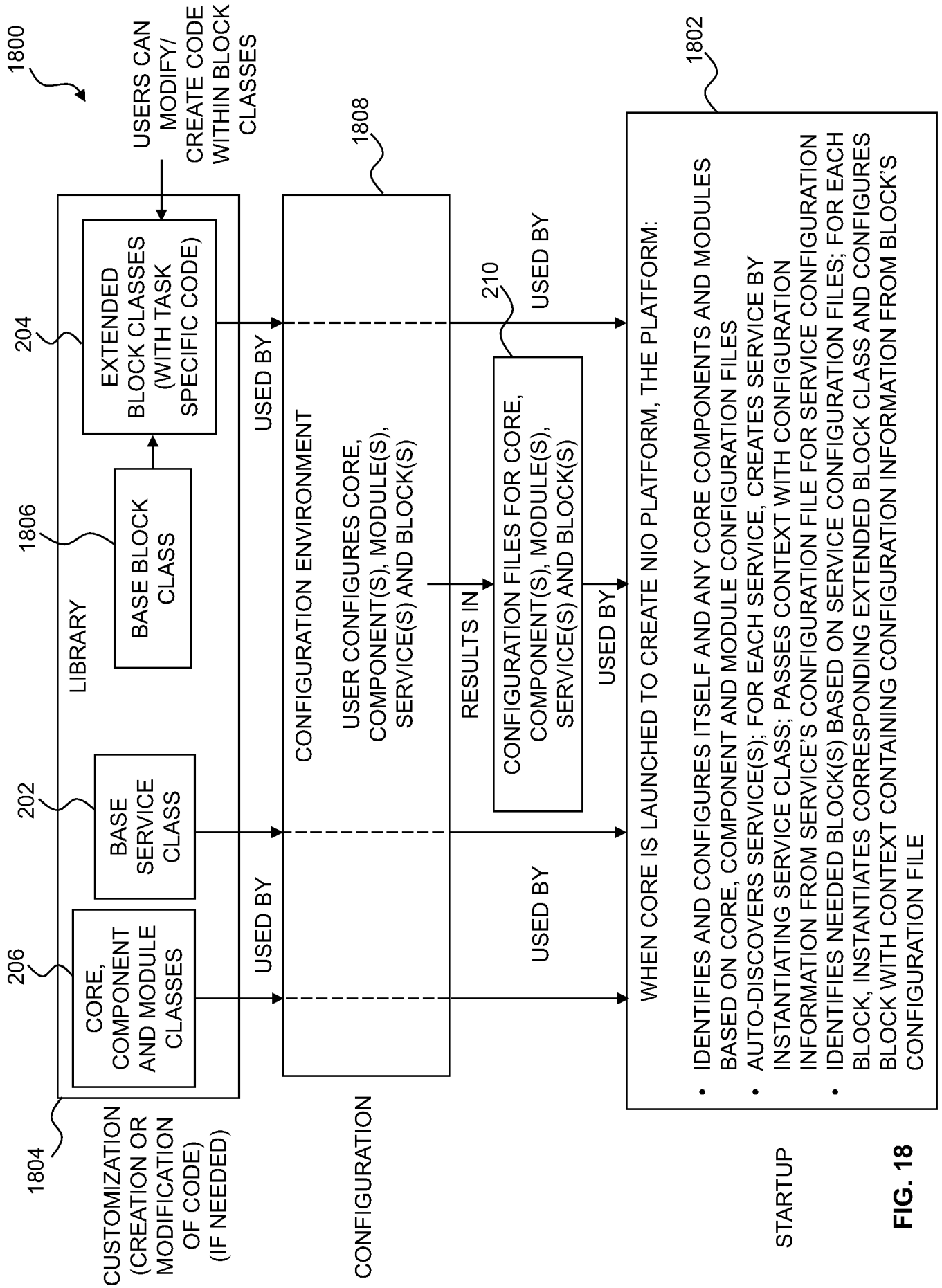


FIG. 18

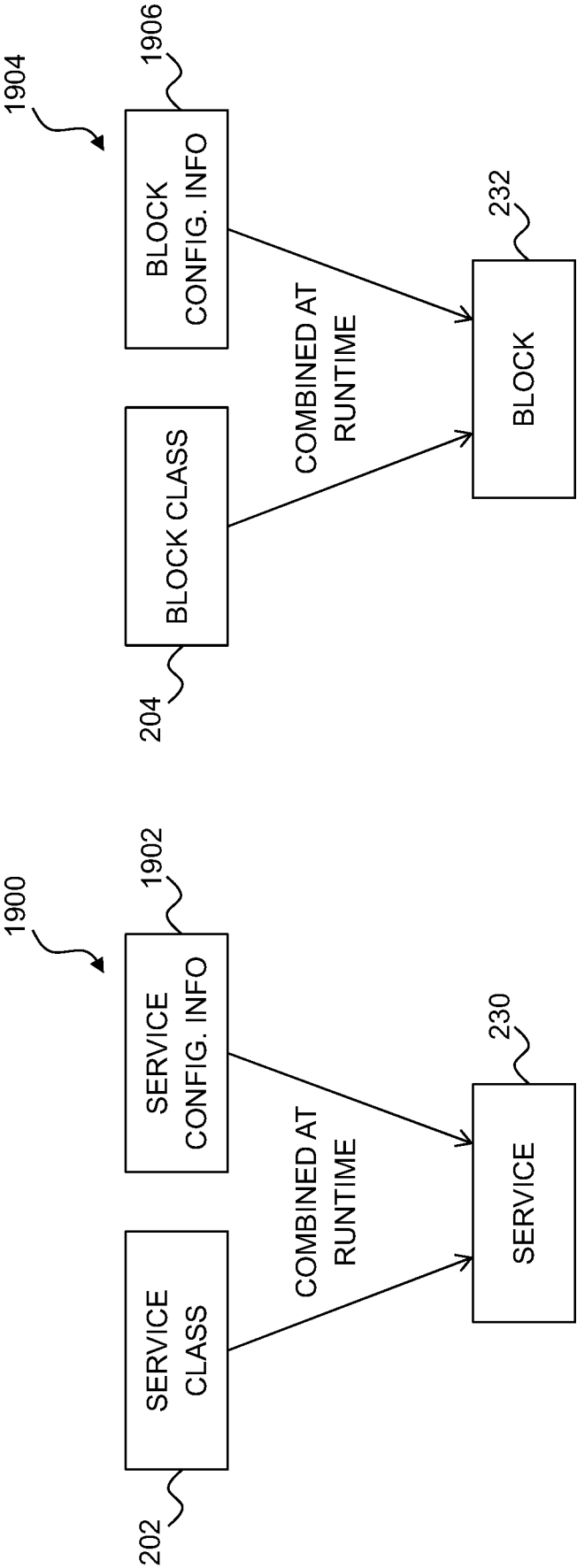


FIG. 19A

FIG. 19B

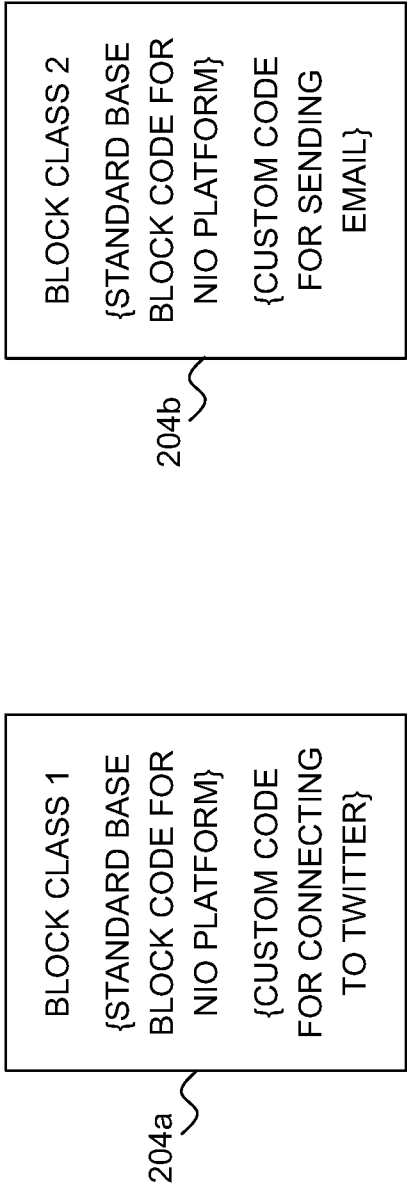


FIG. 20A

FIG. 20B

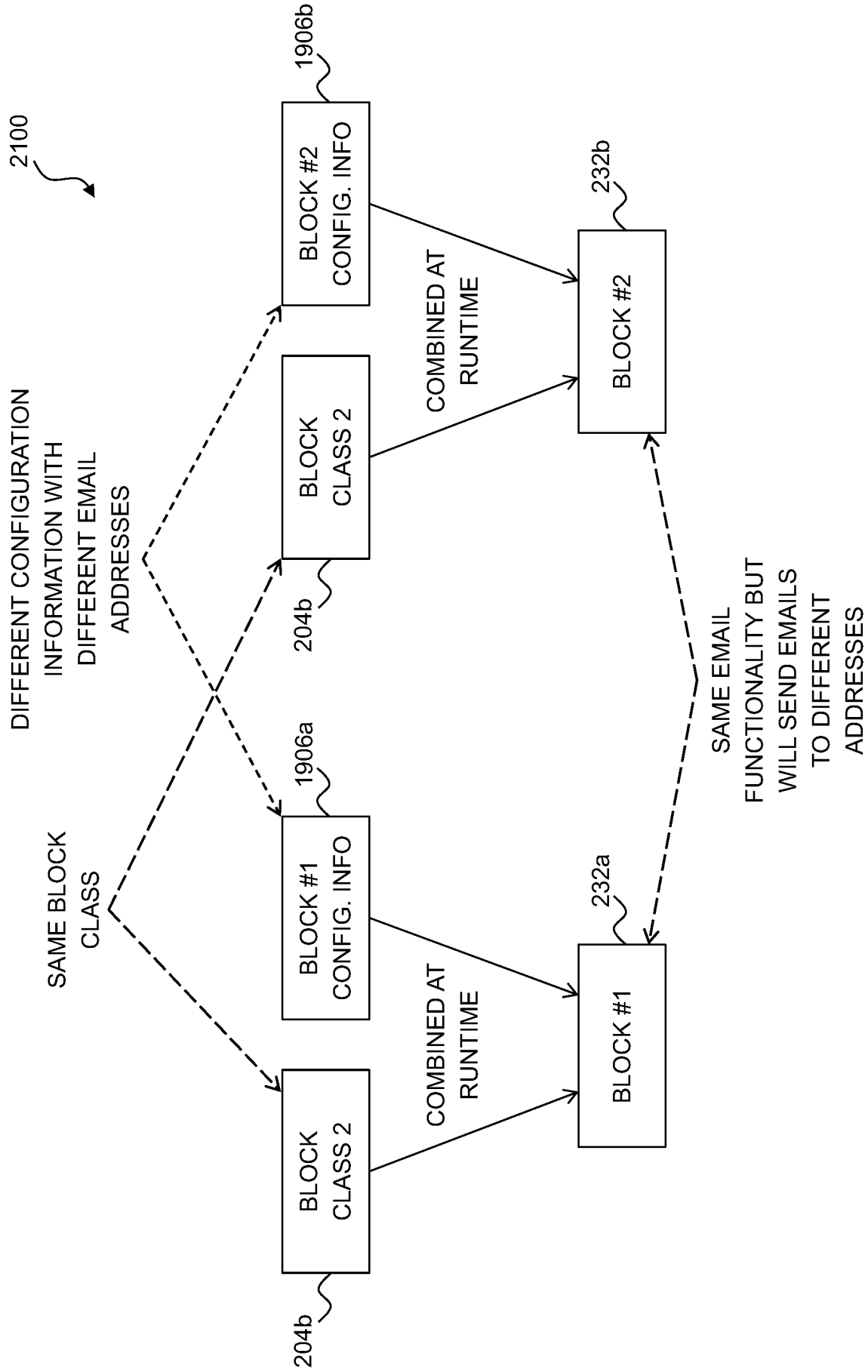


FIG. 21

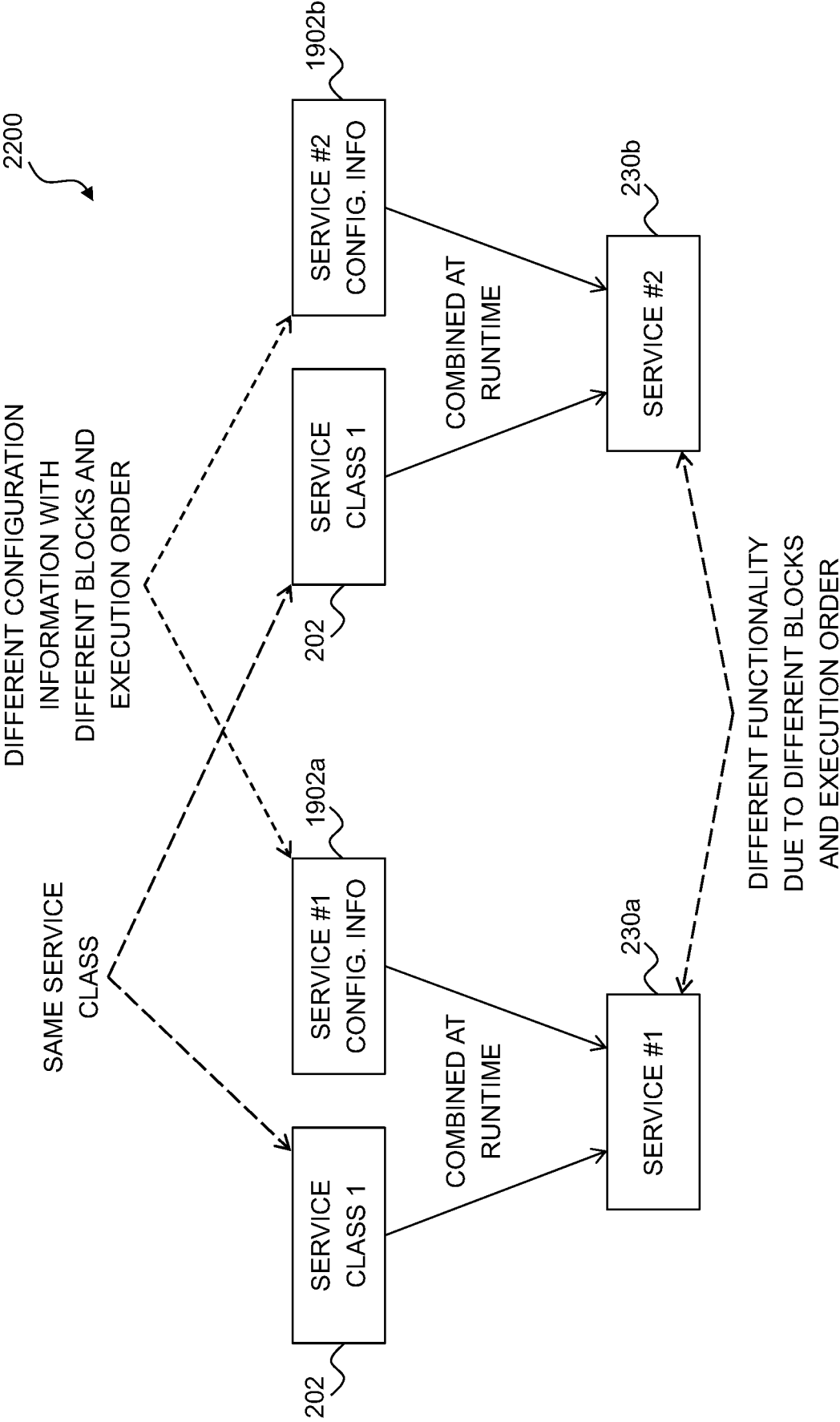


FIG. 22

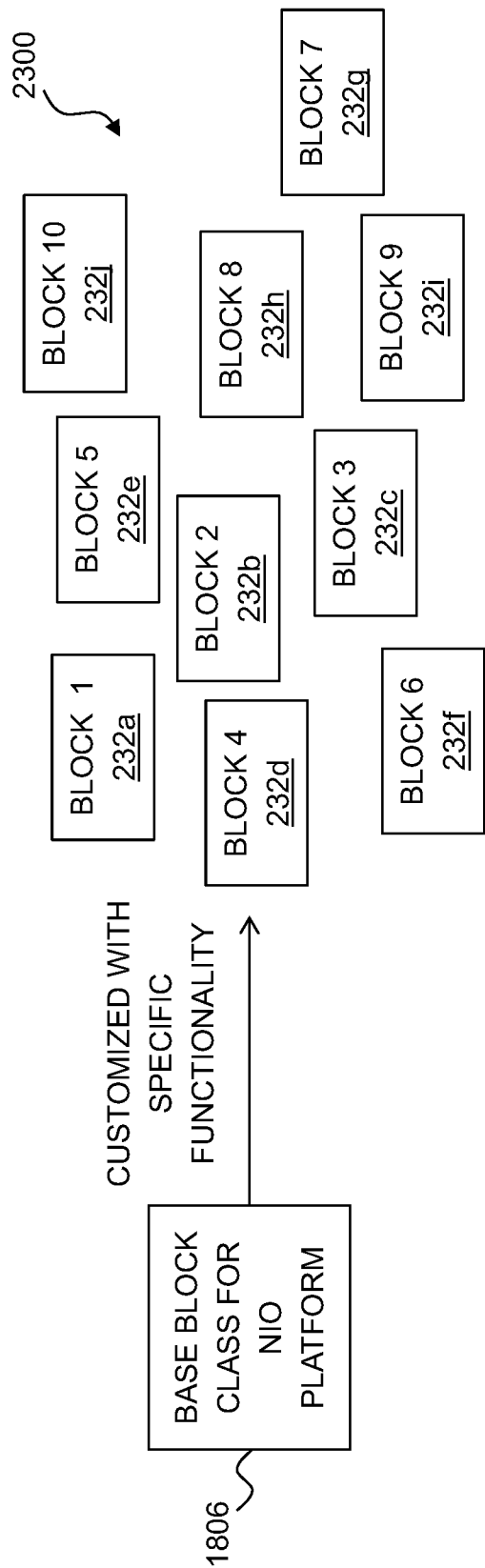


FIG. 23

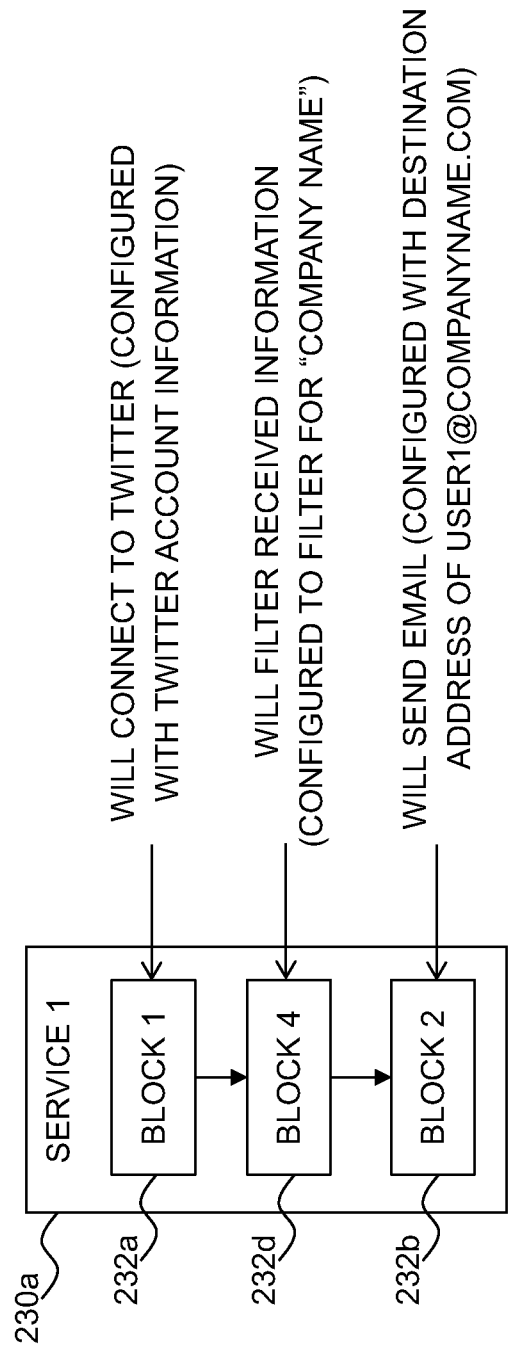


FIG. 24

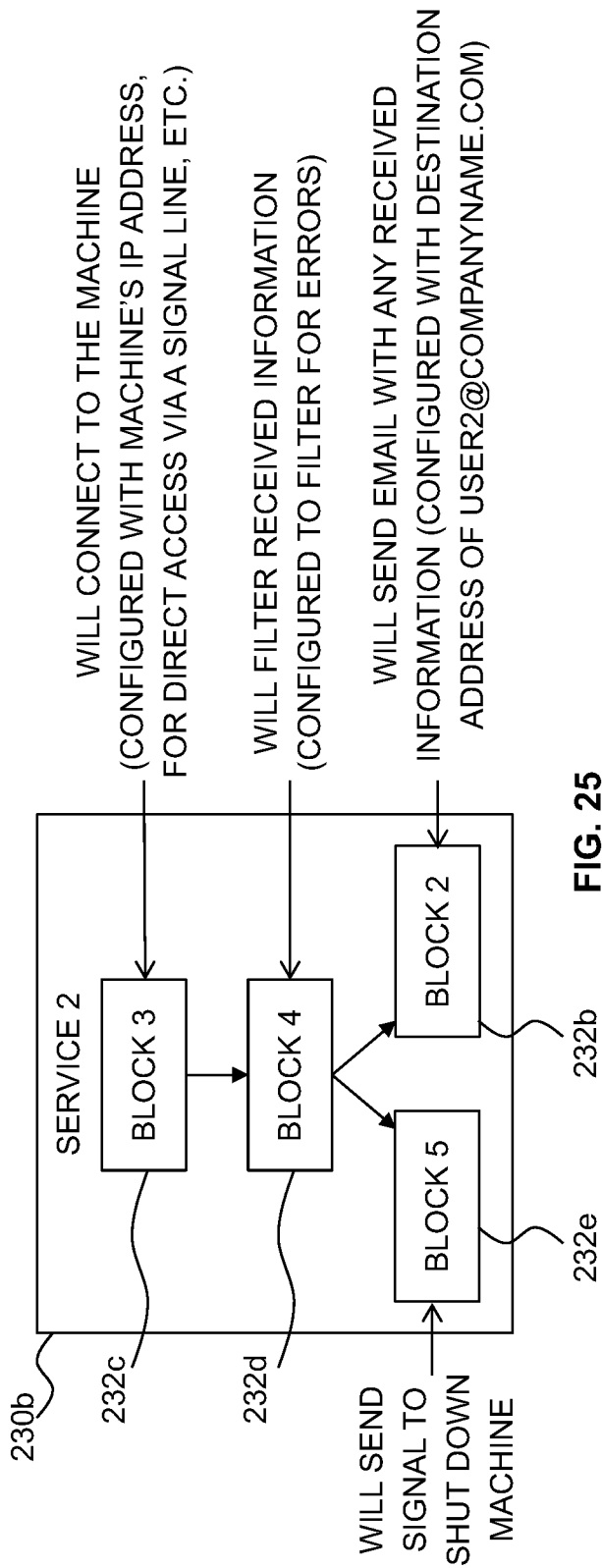


FIG. 25

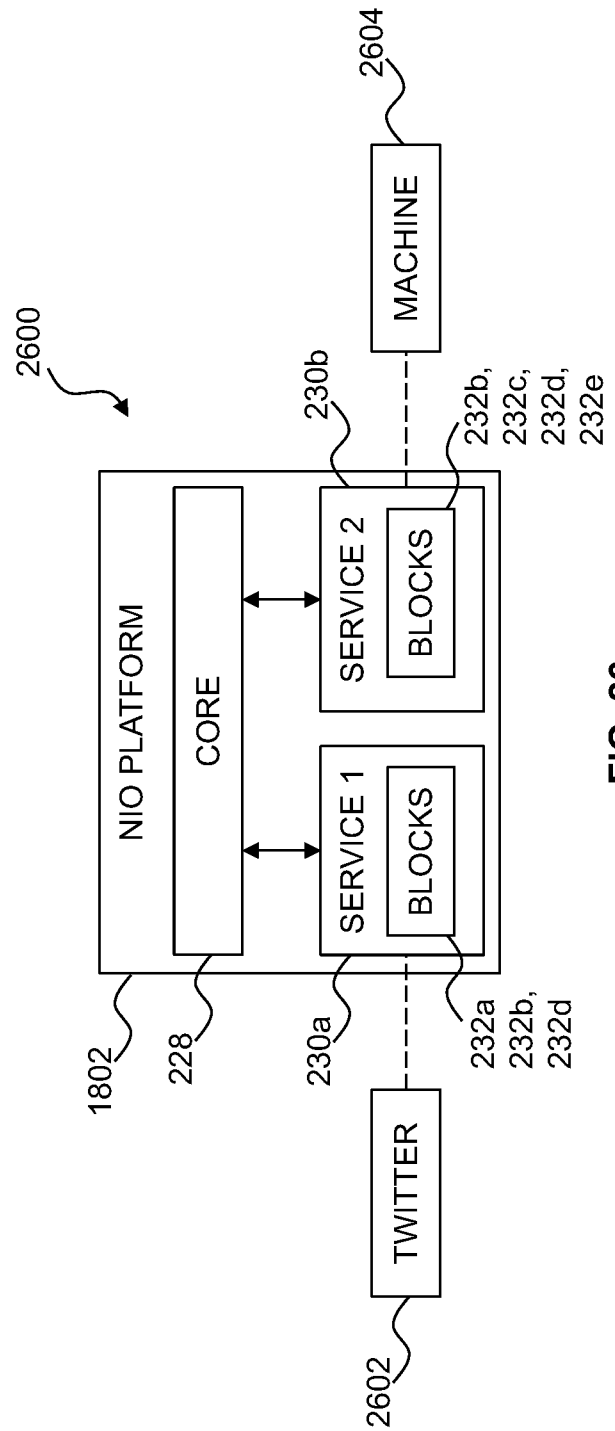


FIG. 26

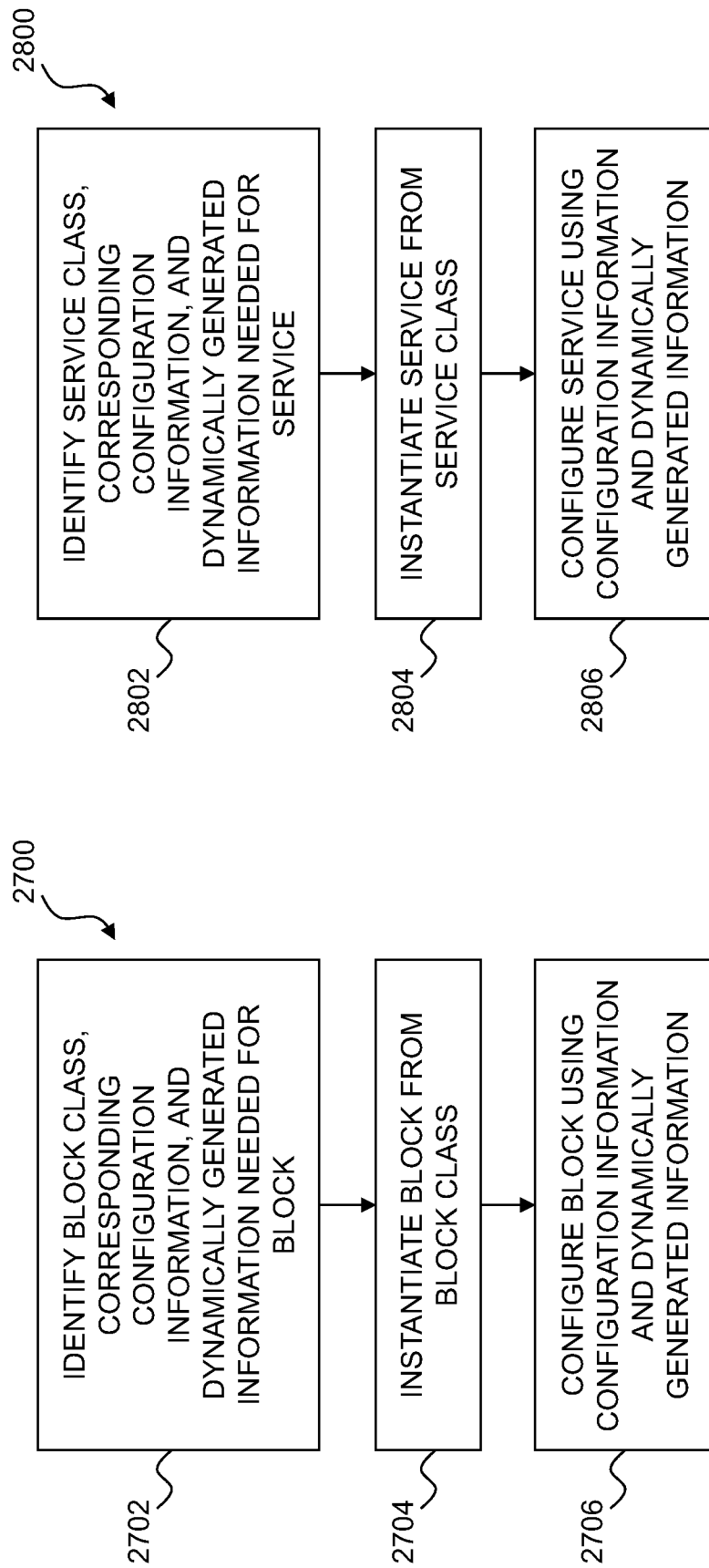


FIG. 27

FIG. 28

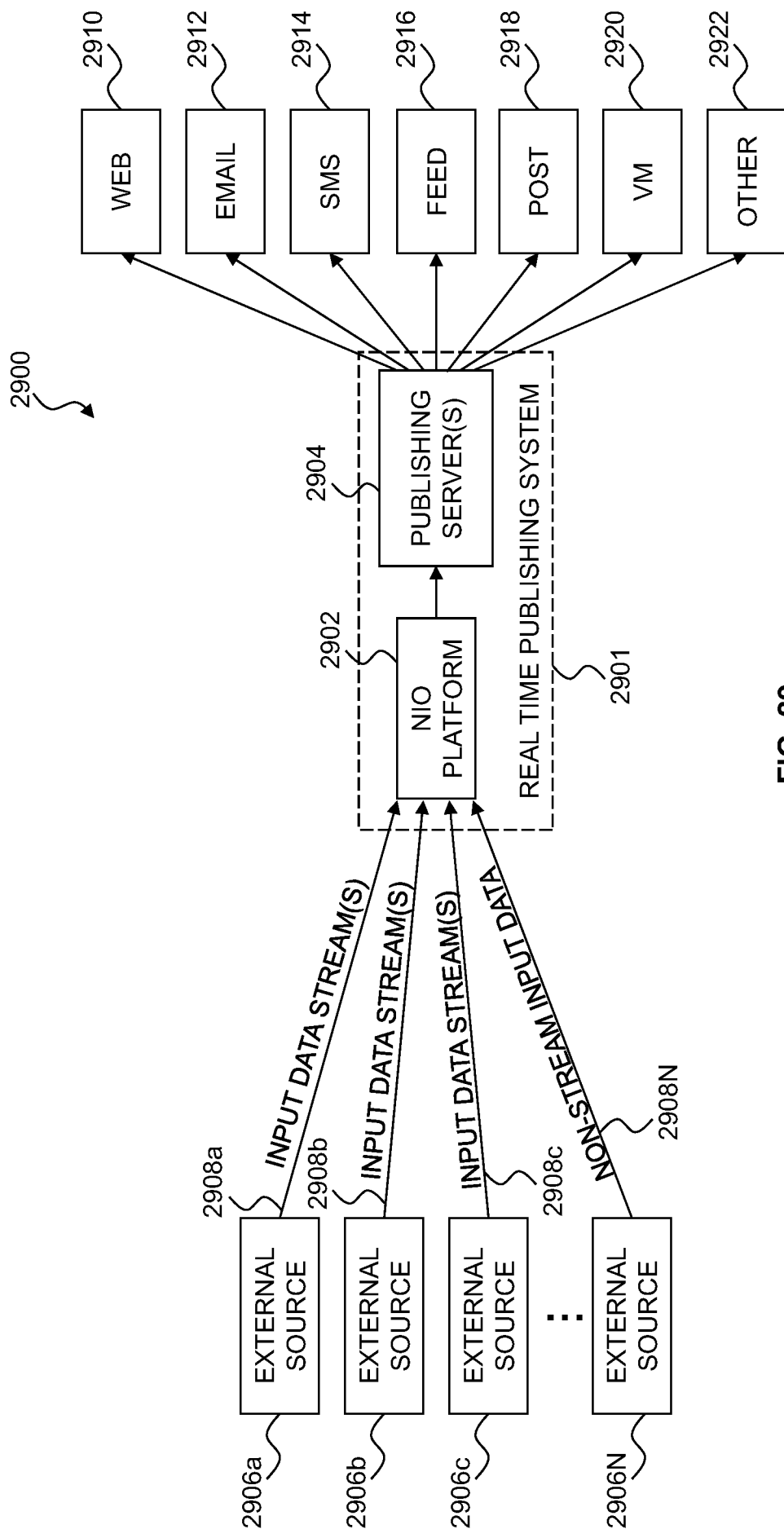


FIG. 29

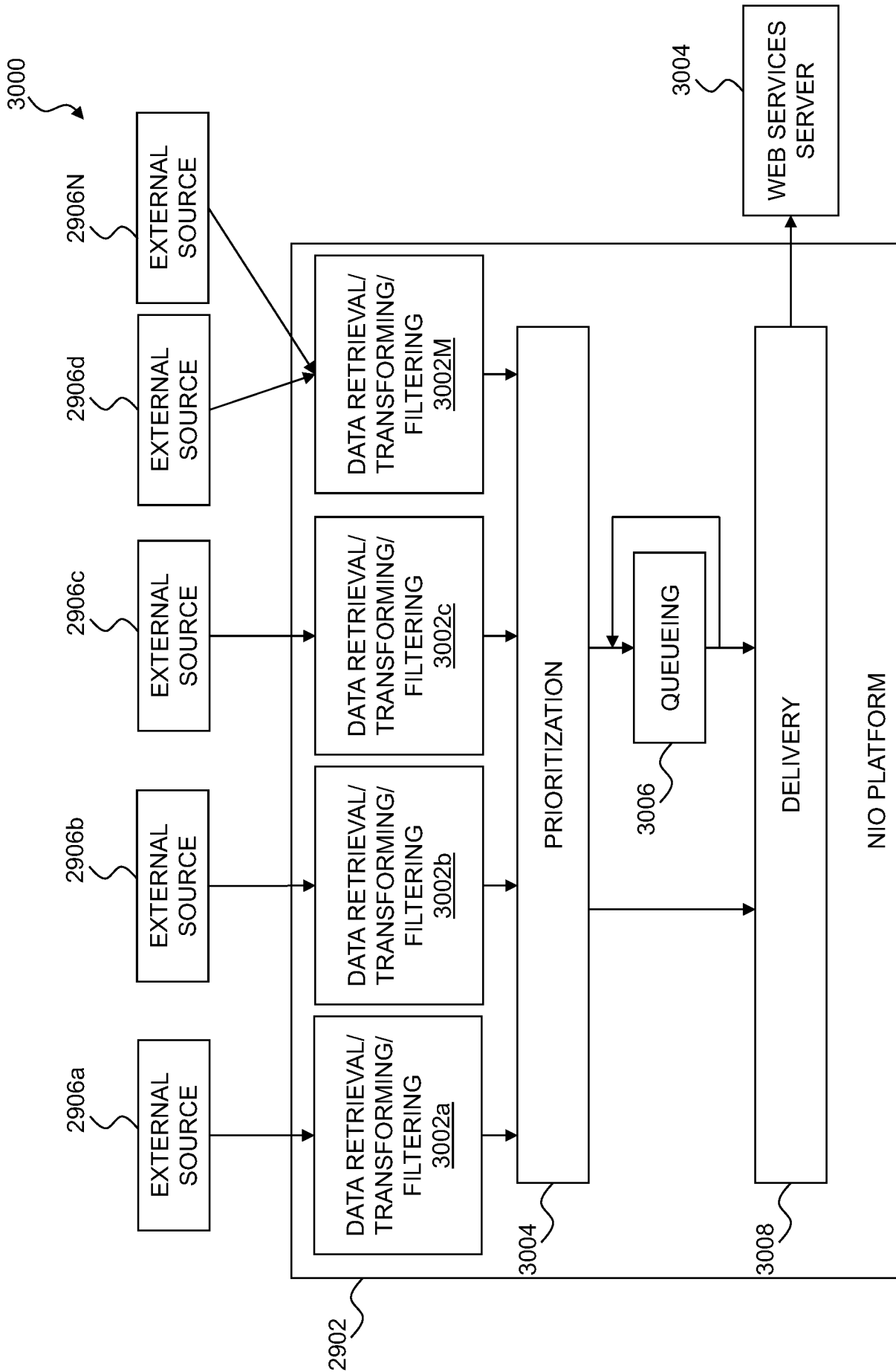


FIG. 30

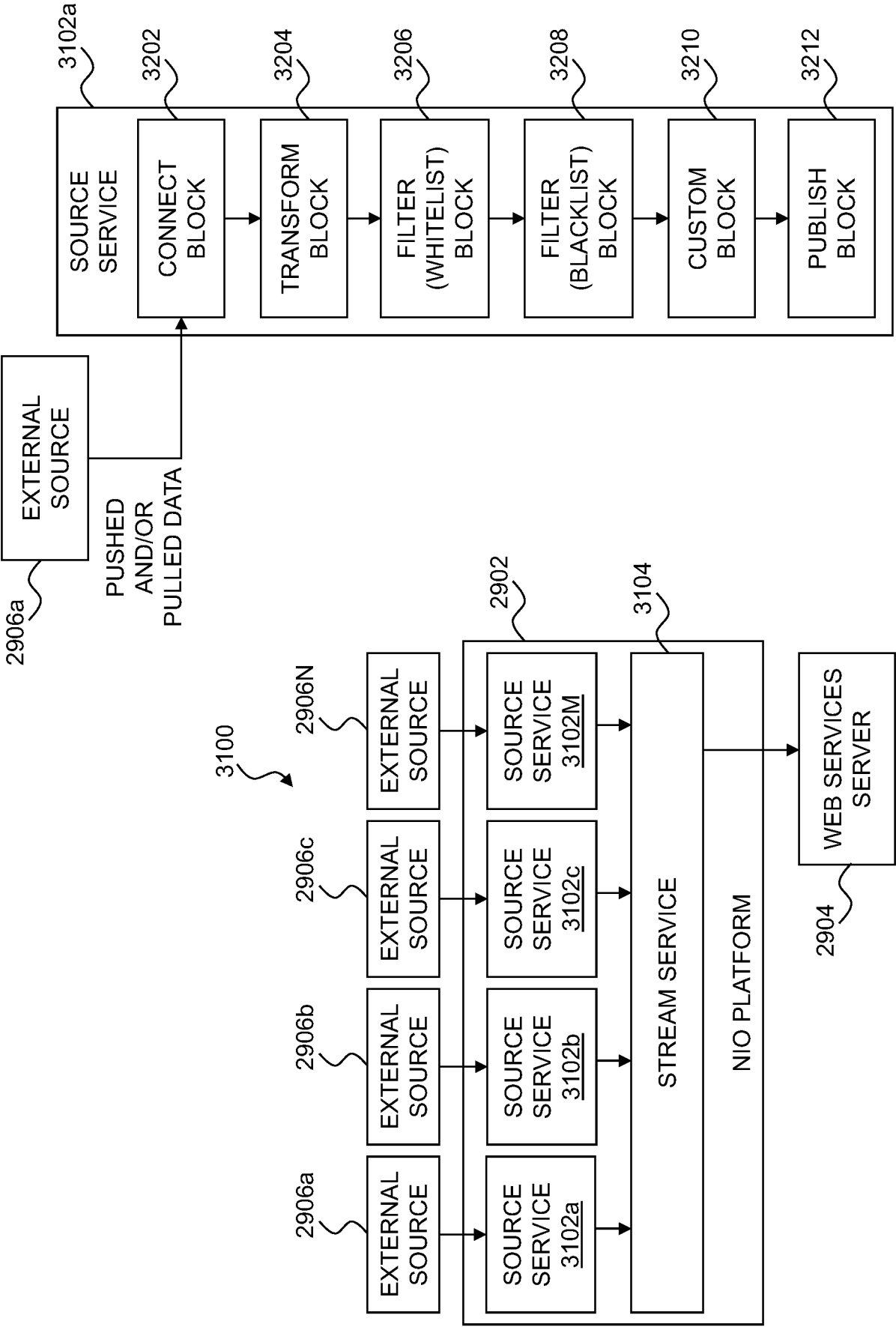


FIG. 32

FIG. 31

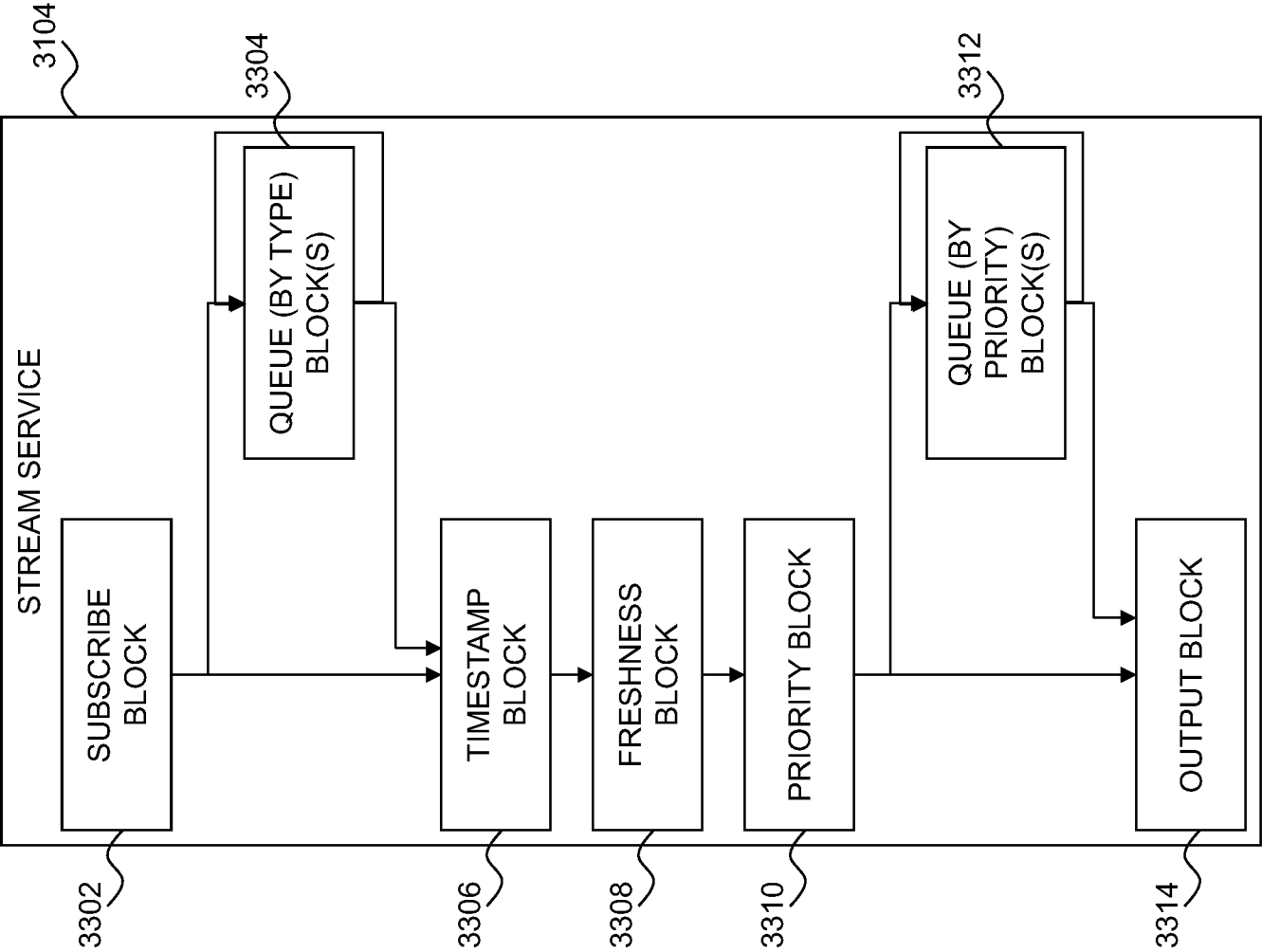


FIG. 33A

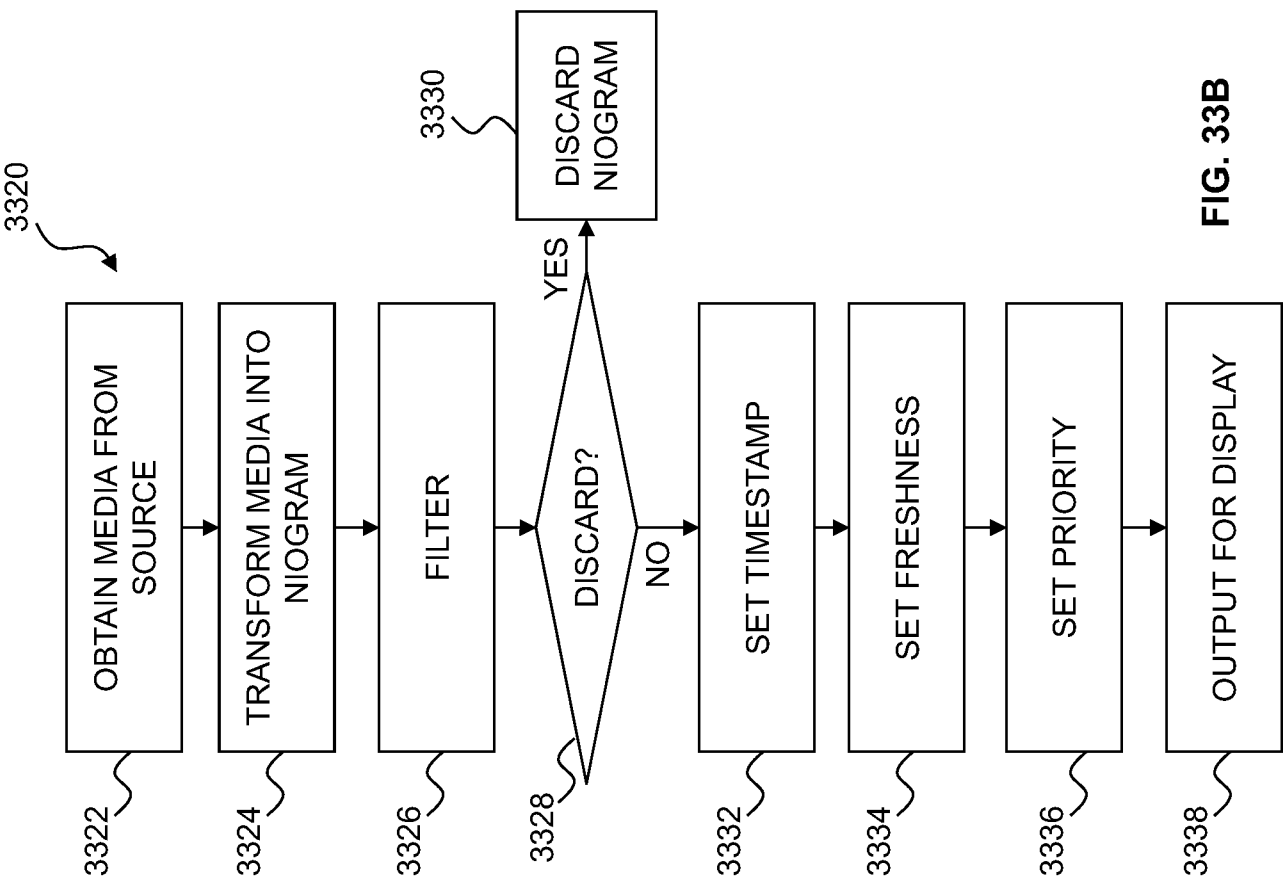


FIG. 33B

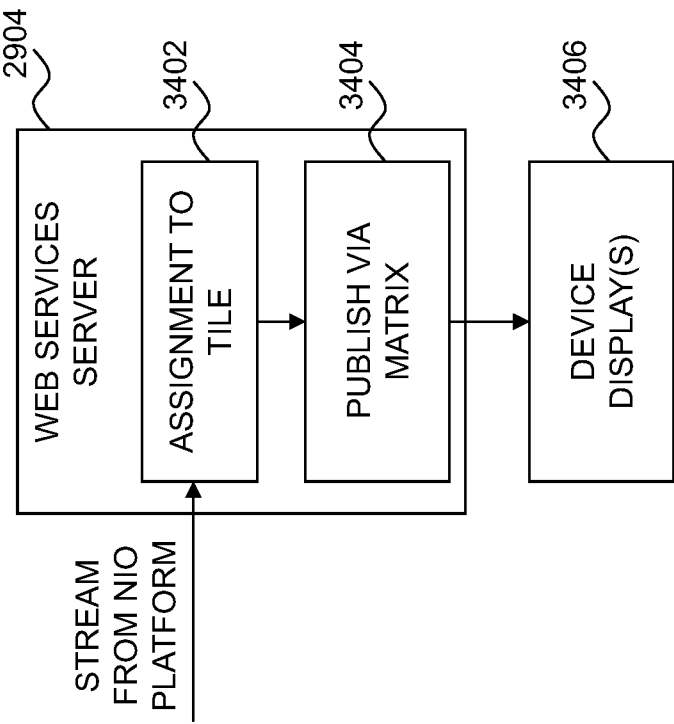
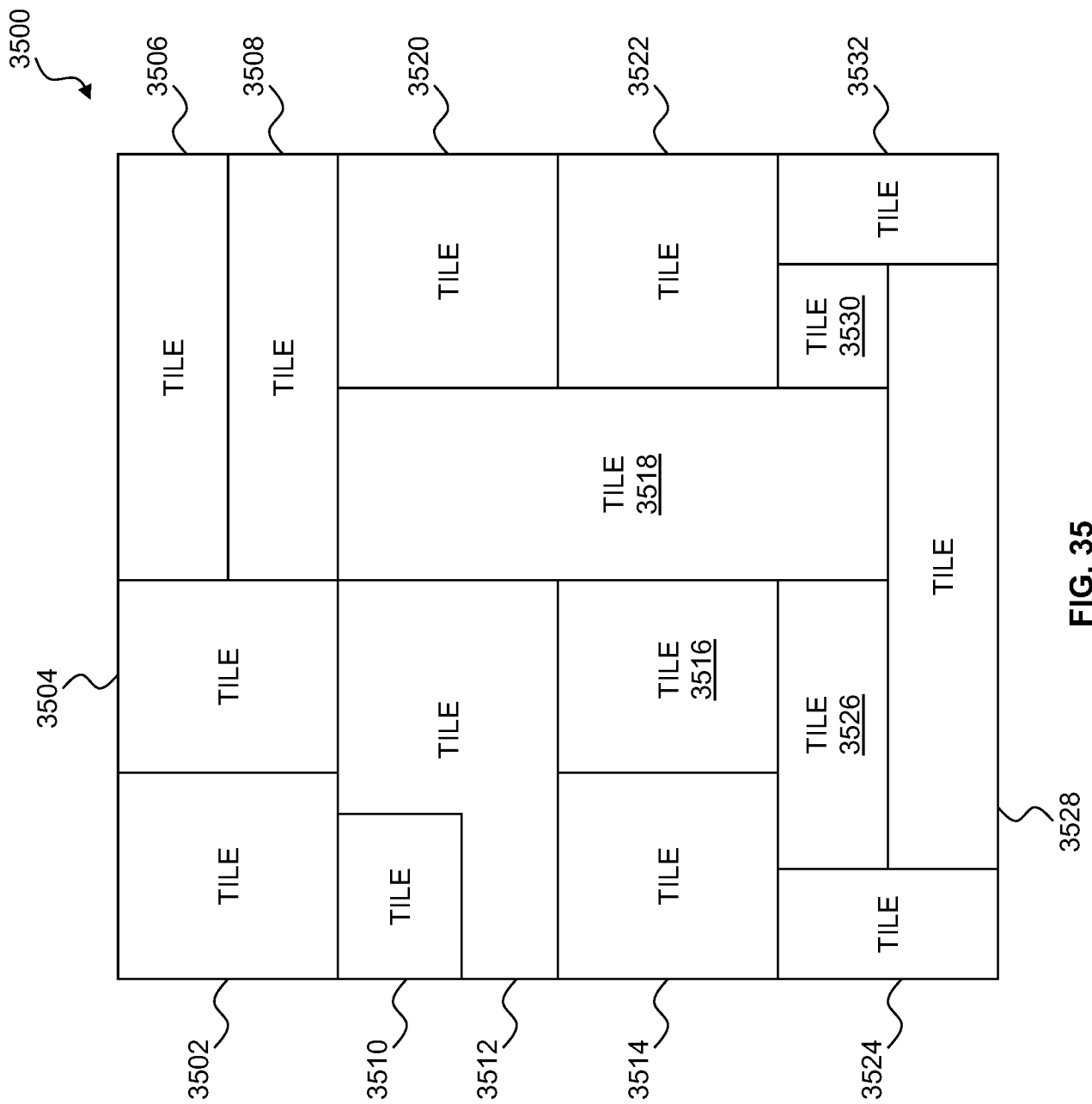


FIG. 34



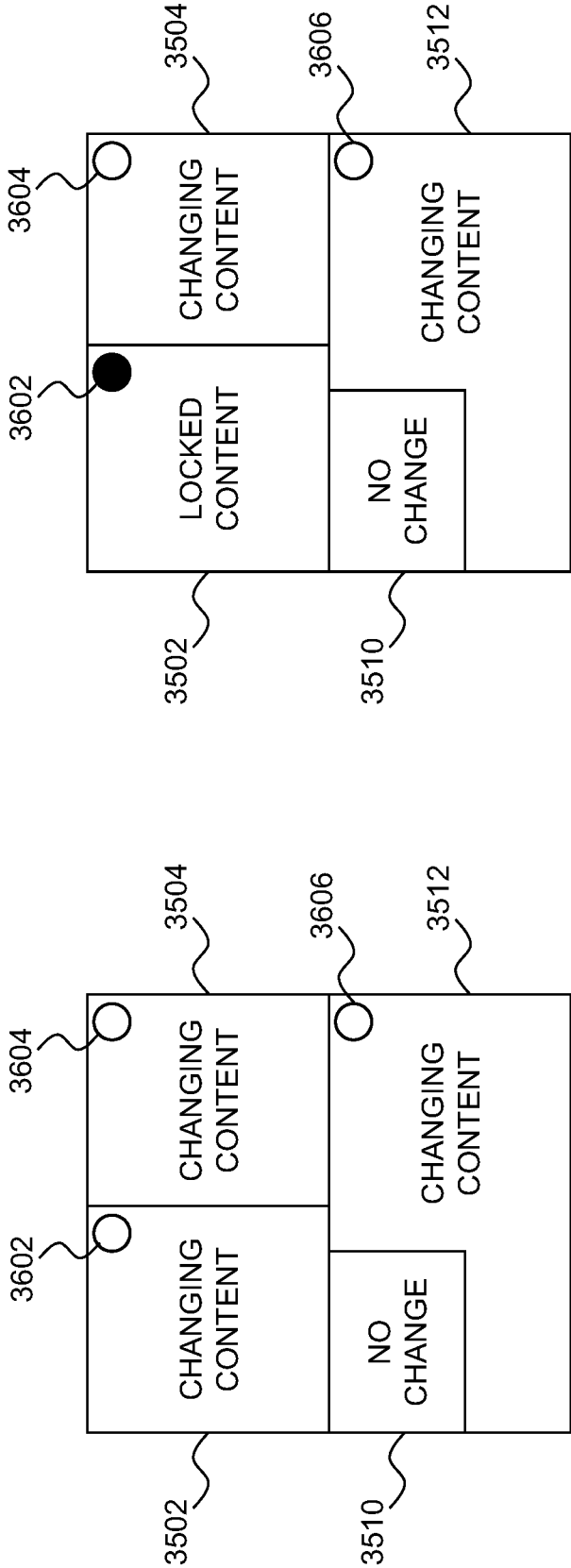


FIG. 36B

FIG. 36A

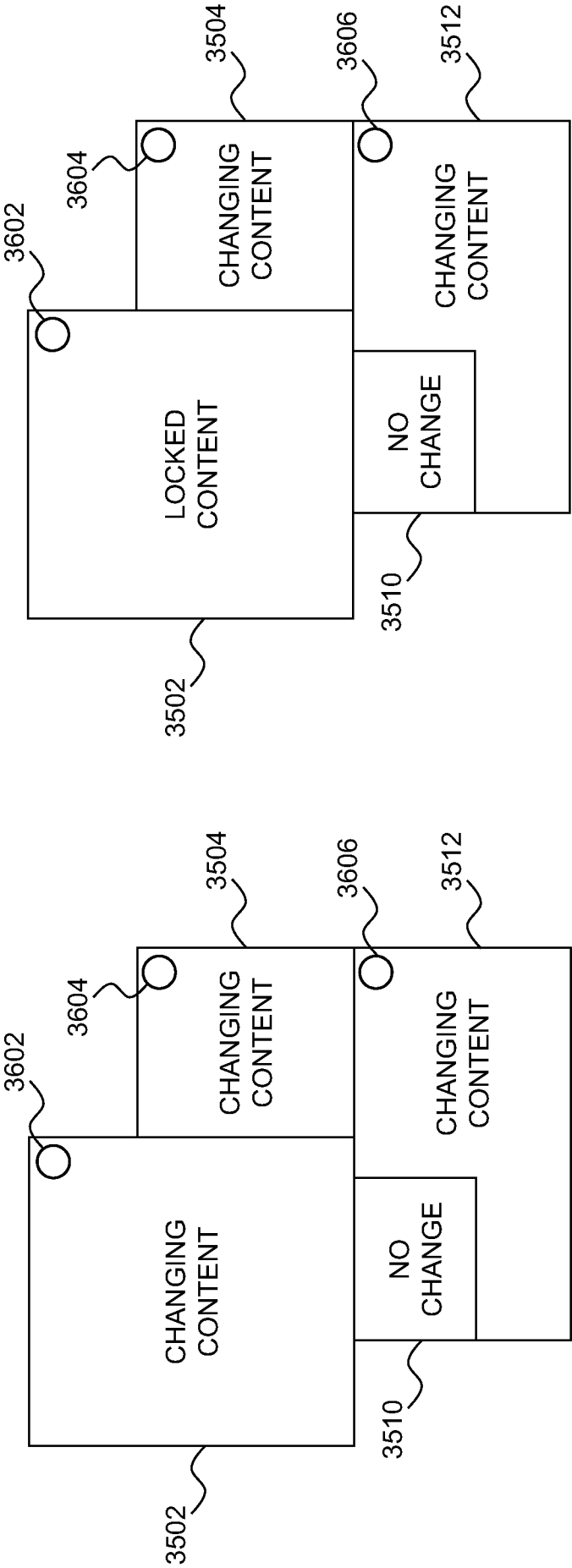


FIG. 36C

FIG. 36D

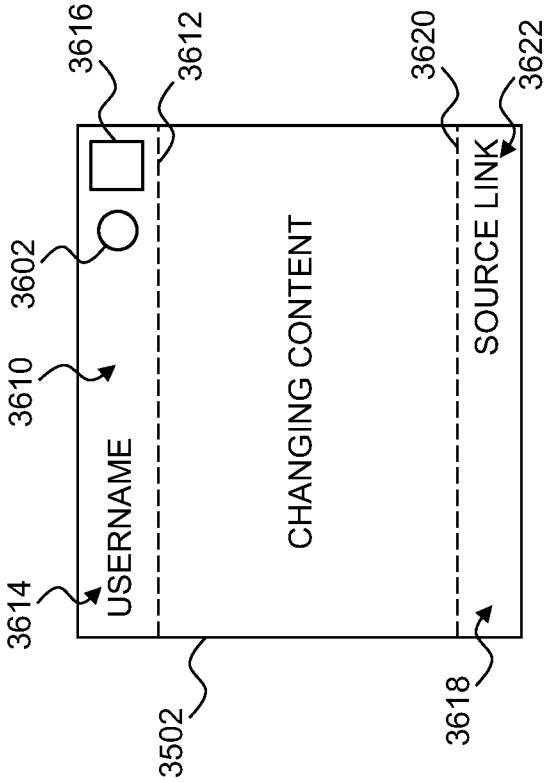


FIG. 36E

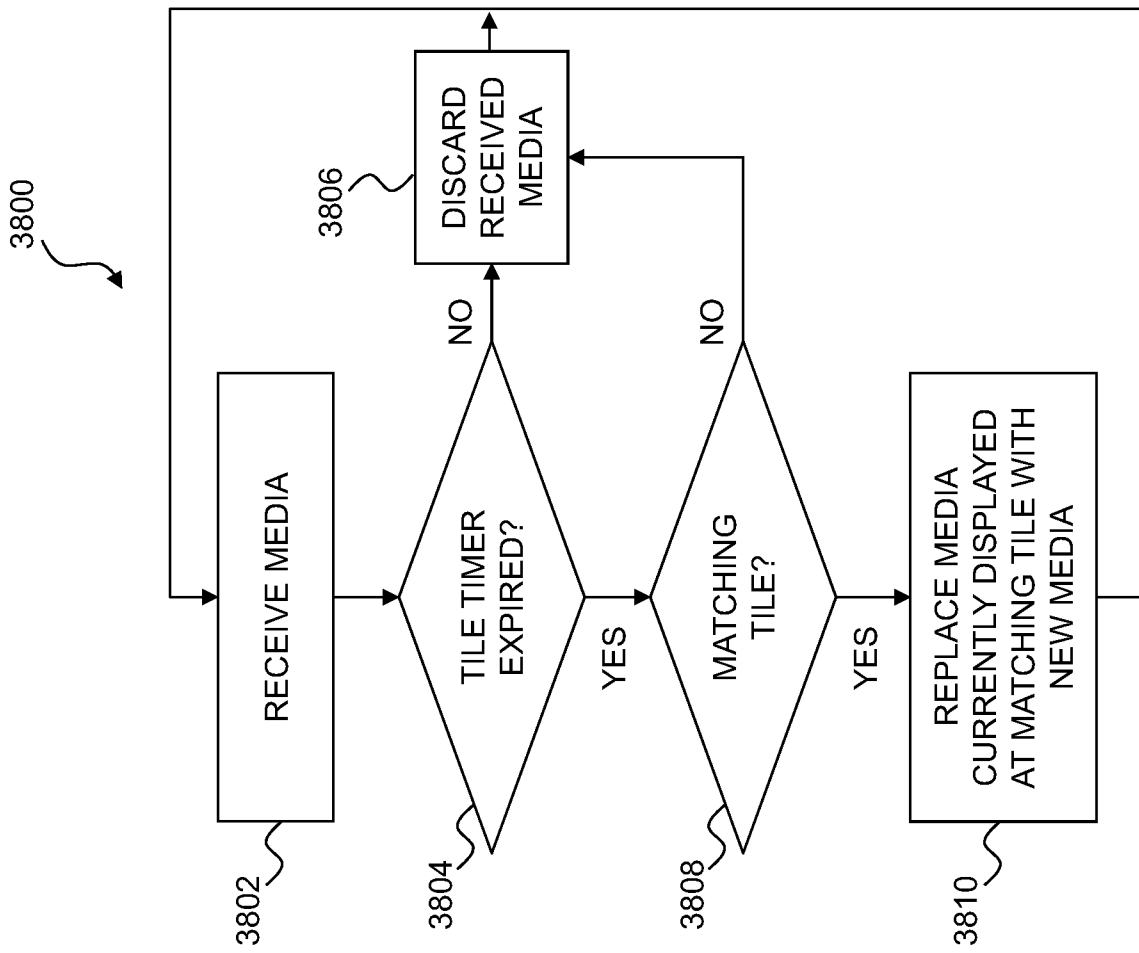


FIG. 38

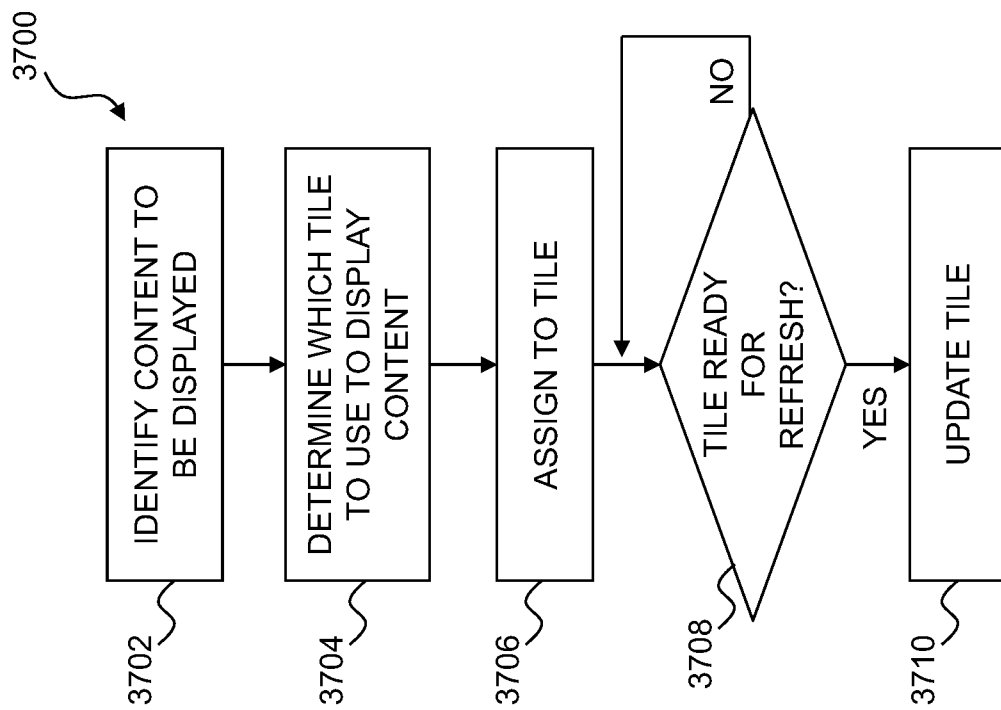


FIG. 37

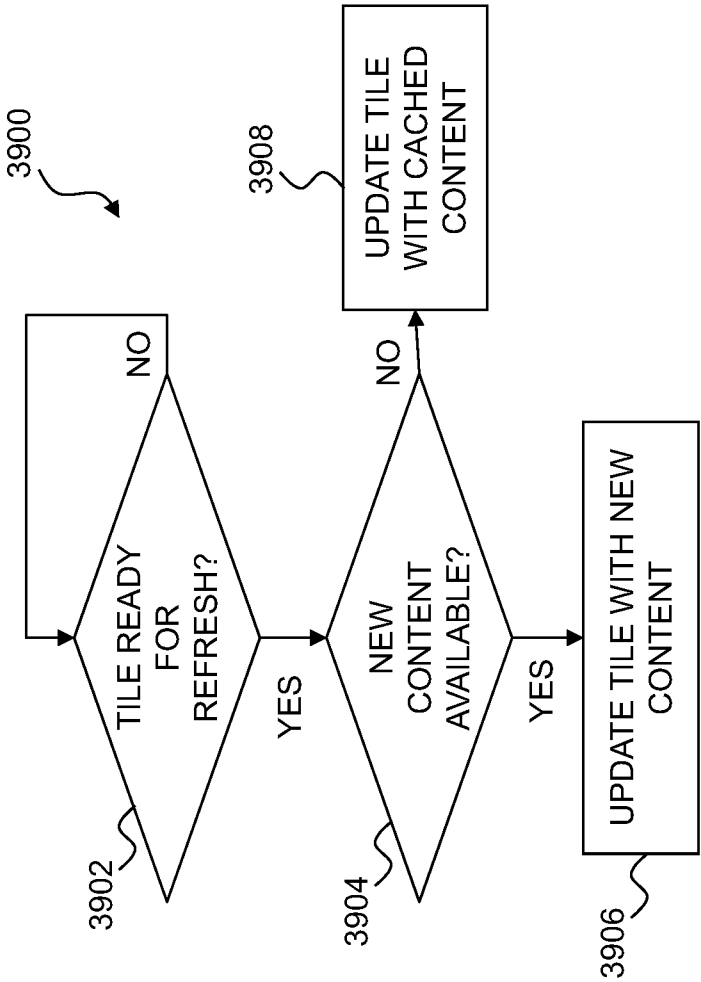


FIG. 39

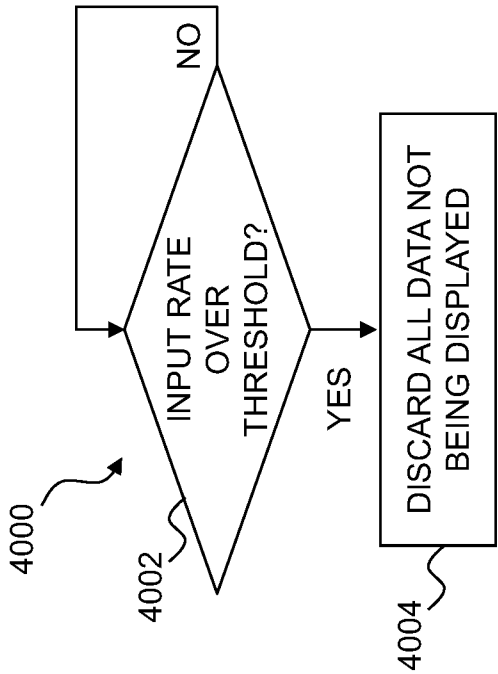


FIG. 40A

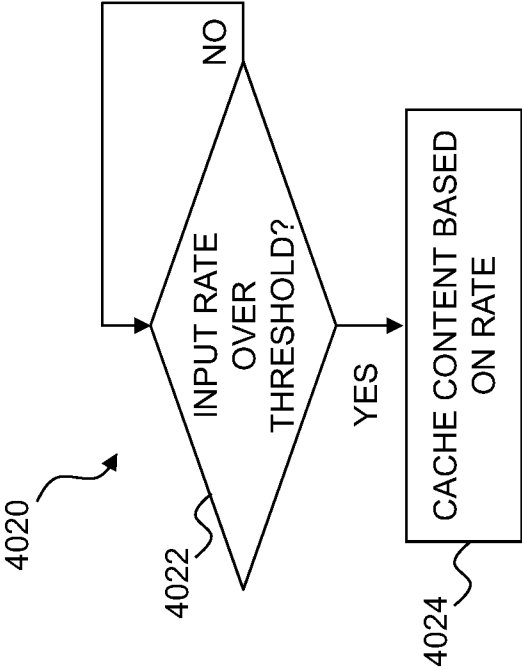


FIG. 40C

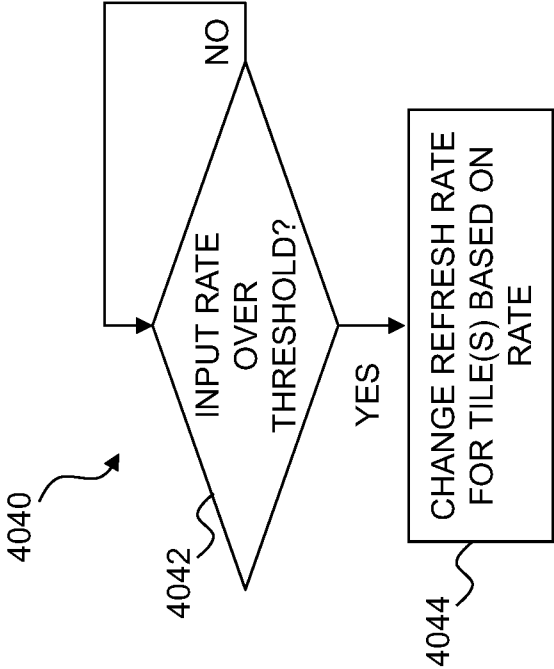


FIG. 40E

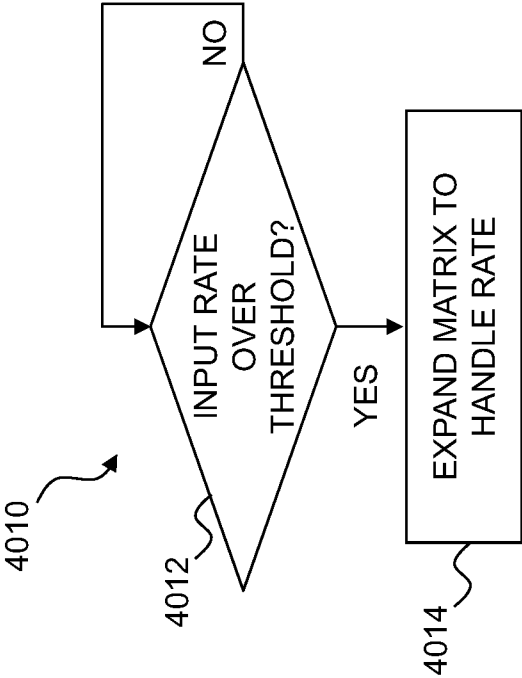


FIG. 40B

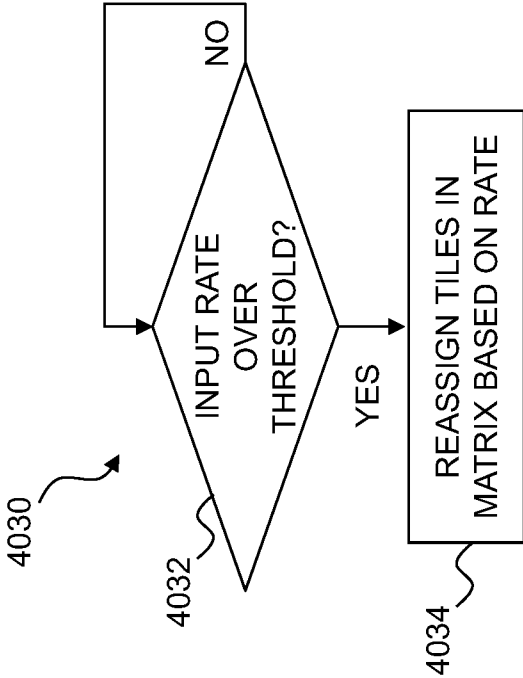


FIG. 40D

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB2015/001288

A. CLASSIFICATION OF SUBJECT MATTER

INV. G06F9/48 G06F9/50 G06F9/54 G06F9/445 G06F9/44
ADD.

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

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)

EP0-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	STEFANOV T ET AL: "System design using Khan process networks: the Compaan/Laura approach", DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION, 2004. PROCEEDINGS FEB. 16-20, 2004, PISCATAWAY, NJ, USA, IEEE, 16 February 2004 (2004-02-16), pages 340-345 Vol.1, XP032396460, DOI: 10.1109/DATE.2004.1268870 ISBN: 978-0-7695-2085-8 figures 3,4,6 page 1, left-hand column, line 1 - line 40 page 2, right-hand column page 4, right-hand column, line 18 - line 26 page 5, right-hand column, line 29 - line 31 ----- -/--	1-16



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

29 October 2015

Date of mailing of the international search report

10/11/2015

Name and mailing address of the ISA/

European Patent Office, P.B. 5818 Patentlaan 2
NL - 2280 HV Rijswijk
Tel. (+31-70) 340-2040,
Fax: (+31-70) 340-3016

Authorized officer

Balevic, Ana

INTERNATIONAL SEARCH REPORT

International application No
PCT/IB2015/001288

C(Continuation). DOCUMENTS CONSIDERED TO BE RELEVANT

Category*	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	Supun Kamburugamuve: "Survey of Distributed Stream Processing for Large Stream Sources", 14 December 2013 (2013-12-14), pages 1-16, XP055224158, Retrieved from the Internet: URL: http://grids.ucs.indiana.edu/ptliupages/publications/survey_stream_processing.pdf [retrieved on 2015-10-28] pages 3-5 pages 10,12,15	1-16
X	----- EP 1 372 071 A2 (MATSUSHITA ELECTRIC IND CO LTD [JP]) 17 December 2003 (2003-12-17) paragraphs [0021], [0035], [0045], [0054], [0093] -----	1-16

INTERNATIONAL SEARCH REPORT

Information on patent family members

International application No

PCT/IB2015/001288

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
EP 1372071	A2	17-12-2003	CN 1476232 A 18-02-2004
			DE 60318771 T2 22-01-2009
			EP 1372071 A2 17-12-2003
			JP 4086529 B2 14-05-2008
			JP 2003303102 A 24-10-2003
			US 2003212831 A1 13-11-2003
